Wait-free Regular Storage from Byzantine Components
Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi
Wait-Free Regular Storage from Byzantine Components

Ittai Abraham*  Gregory Chockler†  Idit Keidar‡  Dahlia Malkhi*

April 4, 2005

Abstract

We present a simple, efficient, and self-contained construction of a wait-free regular register from Byzantine storage components. Our construction utilizes a novel building block, called 1-regular register, which can be implemented from Byzantine fault-prone components with the same round complexity as a safe register, and with only a slight increase in storage space.

Keywords: shared-memory emulations, Byzantine failures, wait freedom.

*School of Computer Science and Engineering, The Hebrew University of Jerusalem. Email: {ittaia,dalia}@cs.huji.ac.il.
†Lab for Computer Science and Artificial Intelligence. Massachusetts Institute of Technology. Email: grishac@theory.lcs.mit.edu.
‡Department of Electrical Engineering, The Technion - Israel Institute of Technology. Email: idish@ee.technion.ac.il.
1 Introduction

In this paper we consider a problem of constructing wait-free distributed storage from Byzantine fault prone components in asynchronous settings. Our specific objective is to devise a solution that will be simple, efficient, and feasible in practice, yet providing meaningful semantics for higher level applications. Roughly speaking, a wait-free object is one that always guarantees the liveness of shared memory operations, including in the presence of any number of process (client) failures.

Constructing efficient storage solutions from Byzantine components received considerable attention recently, as such solutions are useful in a number of emerging application domains. Originally, such solutions have been introduced in the context of scalable client-server systems. Such systems achieve more scalability than traditional state-machine replication approaches by removing the direct communication among the servers, and thus reducing the load that each client request (or transaction) imposes on the servers. This approach was pioneered in the Fleet [17] system, and adopted in many others, e.g., SBQ-L [18], Agile Store [11], Coca [28], and [5]. Similar solutions have also been employed in the setting of Storage Area Networks (SANs). SAN technology allows clients to access disks directly over the network so that the file server bottleneck is eliminated. Examples of SAN-based systems that use disks for information sharing and coordination include Compaq's Petal [13] and Frangipani [23], Disk Paxos [7], Active Disk Paxos [6], and Byzantine Disk Paxos [1]. More recently, solutions of this nature have been adopted in peer-to-peer systems, which consist of a collection of widely spread nodes storing data objects. Naturally, due to their Internet-wide deployment, the storage nodes are prone to malicious attacks, which motivates adopting a Byzantine failure model for the storage nodes. Examples of peer-to-peer systems that adopt storage-centric replication to support availability in face of Byzantine failures include Rosebud [21] and [14].

Fault-prone storage systems as mentioned above can be formally modeled as an asynchronous shared memory system where a threshold \( t \) of the memory objects may fail by being non-responsive [3, 10] or by returning arbitrary values [2, 10] (i.e., by being Byzantine); this failure model is called non-responsive arbitrary (NR-Arbitrary) faults [10]. In this paper, we assume that less than one fourth of the memory objects can fail. In [1] we show that this assumption is necessary for achieving wait-free implementations as efficient as those presented in this paper; that is, every construction that uses less than \( 4t + 1 \) objects must have a higher latency.

All existing wait-free Byzantine-resilient storage constructions provide safe register semantics\(^1\) [10, 16, 1]. The only previous direct constructions of objects with stronger (regular or atomic) semantics from Byzantine storage satisfy weaker (non-wait-free) termination conditions [18, 1]. In this paper we focus on wait-free constructions. Safe register semantics, 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 from wait-free safe registers to stronger ones [19, 27, 8, 12, 24, 25, 26, 9]. However, this approach results in constructions that are not self-contained, and as we argue below, are not tailored to the requirements of a distributed storage system. In this paper, we do not use these reductions as black boxes, but instead capitalize on their techniques in order to derive a new self-contained wait-free regular register construction that is simple, efficient, and feasible in distributed storage environments.

Most existing constructions of strong memory objects are fairly elaborate. We believe that a

\(^1\)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 does overlap a write operation may be arbitrary.
major reason for this complexity is the fact that they aim to achieve an atomic register. However, recent studies indicate that storage with regular semantics is sufficient in most cases [7, 22, 1]. A regular register is weaker than an atomic one; roughly speaking, a regular register guarantees that every read operation returns the value that was written by a write operation invoked not earlier than the last write operation that returns before the read is invoked, or the initial value if no value is written before the read. We therefore focus on constructing regular registers in this paper.

Existing constructions of strong (regular or atomic) wait-free objects from weaker ones were not designed with distributed storage in mind. In particular, such constructions have typically focused 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, we believe that a practical construction for the model we consider herein should focus on simplicity and reducing communication costs, even at the cost of using unbounded counters. This is precisely the approach we take in this paper. We give an algorithm that uses unbounded counters, (which is acceptable in practice), achieves better latency than all existing regular register implementations, and is very simple to understand and implement.

Our approach further differs from existing constructions in the basic building blocks employed. Traditional shared memory constructions often use a collection of safe single-bit registers, which are mathematical models of flip-flops. In contrast, in a distributed storage setting, we can assume that each storage unit (representing a disk or a server) can support stronger objects. In this paper we introduce a novel intermediate building block, called 1-regular register. We show that a 1-regular register can be implemented from Byzantine fault-prone components with the same round complexity as a safe register, and with only a slight increase in storage space. We then give a simple and efficient implementation of a wait-free regular register using 1-regular ones.

Outline: The rest of this paper is organized as follows: In Section 3, we describe the formal computation model used throughout the paper and give the definitions of various register types. In Section 4, we construct a wait-free 1-regular register from \( n > 4t \) base registers up to \( t \) of which can incur Byzantine faults. Finally, in Section 5, we show how to use 1-regular registers in order to construct a wait-free regular register. We note that all but one of the 1-regular registers employed in this construction can be replaced with (weaker) safe ones. Therefore, for completeness, we present a direct safe register construction in the appendix. As noted above, using safe registers instead of 1-regular ones does not reduce the latency, but it slightly reduces the space requirements.

2 Related Work

Wait-free shared register constructions have been an actively researched area for several decades [19, 27, 8, 12, 24, 25, 26, 9]. Most of the constructions found in the literature aim to implement atomic registers from safe bits. One notable exception is the construction by Lamport in [12], which implements an \( n \)-valued regular register using \( O(n) \) safe bits. The construction of [27] was purported to be an atomic register construction, but in fact, only provides regular semantics.

Peterson [19] and Tromp [24] present constructions of atomic registers from safe bit tracks and atomic control bits. It appears that these constructions can be easily adapted to implement regular registers by replacing the atomic control bits with regular ones, which in turn can be easily obtained from safe bits as shown in [12]; nevertheless, this was neither claimed nor proven in those
papers. Although both of these constructions have logarithmic space complexity, the number of shared memory accesses they employ is rather high. Therefore, they are not directly applicable in a distributed setting. Nonetheless, our work benefits from several techniques and ideas underlying these constructions, e.g., using separate tracks to store copies of the register value, and using a handshake mechanism to coordinate between the reader and the writer.

All the existing wait-free Byzantine-fault-tolerant register constructions for distributed storage setting only provide safe semantics [16, 10, 1]. Other constructions achieve stronger semantics at the cost of weaker termination guarantees: the protocols by [18] and [4] implement atomic and regular registers, respectively, but do not guarantee termination in the face of client crashes; and the algorithm of [1] implements a regular register where the read operation is guaranteed to terminate only if it eventually runs in isolation for sufficiently long.

Our 1-regular register notion is a generalization of the pseudo-regular register of [20]. Whereas a read operation on our 1-regular register can return ⊥ if it is concurrent with more than one write, a read from the pseudo-regular register is allowed to return ⊥ if it is concurrent with any number of writes. Hence, the guarantees it provides corresponds to 0-regularity in our terminology.

3 The System Model

We consider asynchronous shared memory systems consisting of a collection of processes interacting with a finite collection of objects. Objects and processes are modeled as I/O automata [15]; for space constraints, we do not repeat the details of the I/O model here.

An object automaton’s interface is determined by its type, which is a tuple consisting of the following components: (1) a set $Vals$ of values; (2) a set of invocations; (3) a set of responses; and (4) a sequential specification, which is a function from invocations $\times Vals$ to responses $\times Vals$. In a shared memory system consisting of processes $P_1, P_2, \ldots$, an object of type $T$ interacts with a process $P_i$ by means of input actions of the form $a_i$, where $a$ is an invocation of $T$, and output actions of the form $b_i$, where $b$ is a response of $T$. An object’s external behavior is specified in terms of the properties of its traces (i.e., executions consisting of external actions only). Liveness properties are required to hold only in fair executions, i.e., executions where each output and internal action has infinitely many opportunities to occur.

The interaction between a process and an object is well-formed if it consists of alternating invocations and responses, starting from an invocation. We only consider systems where interactions between processes and objects are well-formed. Well-formedness allows an invocation occurring in an execution $\alpha$ to be paired with a unique response (when such exist). If an invocation has a response in $\alpha$, the invocation is complete; otherwise, it is incomplete. Note that well-formedness does not rule out concurrent operation invocations on the same object by different processes. Nor does it rule out parallel invocations by the same process on different objects, which can be performed in separate threads of control.

A threshold $t$ of the objects may suffer NR-Arbitrary failures [10], 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.
3.1 Registers

A read/write register (or simply, register) type supports an arbitrary set $V$ of values with an arbitrary initial value $v_0 \in V$. Its invocations are read and write$(v)$, $v \in V$. Its responses are $v \in V$ and ack. Its sequential specification, $f$, requires that every write overwrites the last value written and returns ack (i.e., $f(\text{write}(v), w) = (\text{ack}, w)$); and every read returns the last value written (i.e., $f(\text{read}, v) = (v, v)$). In a shared memory system consisting of processes $P_1, P_2, \ldots$, a process $P_i$ interacts with a shared register by means of input actions of the form read$_i$ and write$(v)_i$, and output actions of the form $v_i$ and ack$_i$. A read/write register is called $k$-reader/$m$-writer if only $k$ ($m$) processes are allowed to read (resp. write) the register. We use the term multi-reader when the particular number of readers is not important.

We now define several register properties that will be used throughout the paper. Fix $x$ to be a single-writer/multi-reader (SWMR) or single-writer/single-reader (SWSR) register, and let $\sigma$ be a sequence of invocations and responses of $x$.

**Safe register.** $\sigma$ is safe [12] if every complete read operation that does not overlap any write operation returns the register's value when read was invoked (i.e., the latest written value or the initial value $v_0$ if no value was written). A register is called safe if it has only safe traces.

**Regular register.** $\sigma$ is regular [12] if it is safe, and in addition, a read operation that does overlap some write operations returns either one of the values written by overlapping writes or the register's value before the first overlapping write is invoked. A register is regular if it has only regular traces.

**1-regular register.** $\sigma$ is regular if it is safe, and in addition, a read operation that overlaps at most one write operation returns either the value written by overlapping write or the register's value before the overlapping write is invoked. Otherwise, a read operation may return in addition a special \bot value. A register is 1-regular if it has only 1-regular traces.

**Wait Freedom.** Register $x$ is wait-free if in any fair execution of any shared memory system that includes $x$, every invocation of $x$ by a correct process is complete.

4 Wait-free 1-Regular Register Construction

The implementation of a wait-free SWMR 1-regular register from $n > 4t$ wait-free SWMR regular registers up to $t$ of which can incur NR-arbitrary failures is depicted in Figure 1. The notation invoke write$(x_i, v)$, (respectively, invoke tmp $\leftarrow$ read$(x_i)$) means that a new thread is started that performs a write on register $x_i$ with value $v$ (respectively, a read of register $x_i$ whose response will be stored in local variable tmp). The notation $x_i$ responded means that the last thread created by an invoke operation has completed its execution on register $x_i$. Note that this is well defined because we maintain well formedness using control flags pending and enabled in such a manner that there is at most one pending thread for each register. Each of the $n$ base registers $x_i$ consists of a cyclic two-value buffer whose elements are denoted $x_i[0]$ and $x_i[1]$ respectively. Each WRITE operation WRITE$(v)$ first chooses a unique monotonically increasing timestamp ts and then writes the pair $(ts, v)$ to the base registers $x_i$, provided that the $k$th WRITE operation updates the $k$ mod 2th part of $x_i$. This mechanism ensures that every READ operation that is overlapped by at
most one WRITE operation will always be able to recover a good value (i.e., the value that upholds regularity) from the values read from either part of the registers as explained below.

The READ implementation reads the values from at least \( n - t \) base registers and stores the values read from the \( x_i[0] \) and \( x_i[1] \) components in the arrays \( u0[1\ldots n] \) and \( w1[1\ldots n] \) respectively. The predicate \( \text{safe}(c, w) \), where \( c \in \text{TSVals} \) (i.e., a timestamp-value pair) and \( w \) is a vector of \( \text{TSVals} \), evaluates to true if \( c \) appears in at least \( t + 1 \) elements of \( w \). A value \( c \) which is safe in either \( u0 \) or \( w1 \) is known to be returned by at least one correct register \( x_i \) and therefore, was written by some previously invoked WRITE. However, the \( \text{safe} \) predicate by itself is insufficient to ensure regularity since some old values could still be returned by \( t + 1 \) registers (e.g., if there are \( t \) registers which are not up to date and 1 which is faulty). Hence, in order for a safe value \( c \) to be returned, we must ensure that all values with a timestamp higher than \( c \) are \( \text{invalid} \), i.e., could not have been written by a later complete WRITE operation.

The validity check is accomplished by the predicate \( \text{invalid}(c, w) \), which returns true if and only if for at least \( 2t + 1 \) elements \( w[j] \), \( w[j].ts < c.ts \) or \( w[j].ts = ts \wedge w[j].val \neq c.val \). Since every completely written value can only be overwritten by a higher timestamped value, a value \( c \) is invalid in \( u0 \) (resp. \( w1 \)) if and only if it was either not written to at least \( n - t \) components \( x_i[0] \) (resp. \( x_i[1] \)), or was not written at all. Hence, every value \( c \) which is both safe in \( u0 \) (resp. \( w1 \)) and for which all the higher timestamped values are invalid, is known to be not older than the value written by the last WRITE operation that writes \( x_i[0] \) (resp. \( x_i[1] \)) and completes before READ is invoked. Therefore, the highest timestamped such value is guaranteed to preserve regularity.

The informal discussion above is formalized in the following two lemmas:

**Lemma 1.** If READ returns \( v \neq \bot \), then \( v \) is either a value that was written by an overlapping WRITE, or the register’s value before the first overlapping WRITE was invoked.

**Proof.** Let \( R \) be a READ operation that completes. Let \( v \) be the value written by the last WRITE operation \( W \) that completes before \( R \) is invoked. By the WRITE implementation, there exists a timestamp \( ts \) such that \( \langle ts, v \rangle \) is written to either \( x_i[0] \) or \( x_i[1] \) component of at least \( n - t \) base registers \( x_i \). Hence, \( v \) was written to either one of two components, say to \( x_i[0] \). Since READ returns a value \( \neq \bot \), by the READ code, \( C0 \cup C1 \neq \emptyset \). There are two cases to consider:

First, suppose that \( C0 = \emptyset \) and \( C1 \neq \emptyset \). By line 5, this implies that either (1) \( \neg \text{safe}(c, u0) \) for all \( c \in \text{TSVals} \), or (2) for every \( c \in \text{TSVals} \) such that \( \text{safe}(c, u0) \) holds, \( \text{higherValid}(c, u0) \) is also true. In both these cases, READ overlaps at least one complete WRITE operation that writes \( \langle ts', v' \rangle \) with \( ts' > ts \) to the \( x_i[1] \) component of the base registers \( x_i \). Hence, there are at most \( 2t \) base registers \( x_i \) such that \( x_i[1].ts < ts \) or \( x_i[1].ts = ts \wedge x_i[1].val \neq v \). Thus, for any \( c' \in \text{TSVals} \) such that \( c'.ts < ts \), \( \text{higherValid}(c', w1) \) is true. Therefore, for each \( c1 \in C1 \), \( c1 \) was returned by at least one correct register \( \text{safe}(c1) \), and \( c1.ts \geq ts \). Hence, \( c1.val \) was written by a WRITE operation that follows \( W \). Since \( C1 \neq \emptyset \), the return value \( val \) is equal to \( c1.val \) for some \( c1 \in C1 \). Hence, the regularity is maintained.

Finally, suppose that \( C0 \neq \emptyset \). In this case, there are at most \( 2t \) base registers \( x_i \) such that \( x_i[0].ts < ts \) or \( x_i[0].ts = ts \wedge x_i[0].val \neq v \). Thus, for any \( c' \in \text{TSVals} \) such that \( c'.ts < ts \), \( \text{higherValid}(c', u0) \) is true. Since \( C0 \neq \emptyset \), then for each \( c0 \in C0 \), \( c0 \) was returned by at least one correct register \( \text{safe}(c0) \), and \( c0.ts \geq ts \). Furthermore, by line 8, the value returned must have been written with a timestamp which is at least as high as \( c0.ts \). Hence, the return value upholds regularity. \[\square\]
WRITE Emulation

Local variables:

\( enabled[1 \ldots n], pending[1 \ldots n] \in \text{Boolean} \)

\( \forall i \in [1 \ldots n], enabled[i] = pending[i] = \text{false} \)

\( w[2] \in \text{TSVals}, w[0] = w[1] = \langle ts_0, v_0 \rangle \)

\( turn \in \{0, 1\}, turn = 0 \)

\( ts \in \text{TS} \)

WRITE(w):

choose \( ts \in \text{TS} \) larger than previously used;

\( w[\text{turn}] \leftarrow \langle ts, v \rangle \)

for \( 1 \leq i \leq n \), enabled[i] \leftarrow true;

repeat

CHECK

until \( \left\{ i : \neg enabled[i] \land \neg pending[i] \right\} \geq n - t \)

\( turn \leftarrow \neg turn \)

return ack

CHECK:

if \( \exists i : enabled[i] \land pending[i] \) then

\( \langle enabled[i], pending[i] \rangle \leftarrow \langle \text{false}, \text{true} \rangle \)

\( \text{INVOCATE write}\langle x_i, \langle w[0], w[1] \rangle \rangle \)

if \( \exists i : x_i \text{ responded} \) then

pending[i] \leftarrow false

READ Emulation

Local variables:

\( enabled[1 \ldots n], pending[1 \ldots n], old[1 \ldots n] \in \text{Boolean} \)

\( \forall i \in [1 \ldots n], enabled[i] = pending[i] = \text{false} \)

\( w[0][1 \ldots n], w[1][1 \ldots n], tmp0[1 \ldots n], tmp1[1 \ldots n] \in \text{TSVals} \)

Predicate and macro definitions:

\( \text{invalid}(\langle ts, v \rangle, w) \triangleq \left\{ i : w[i] = \langle ts', v' \rangle \land \right. \left. \left( ts' < ts \lor (ts' = ts \land v \neq v') \right) \right\} \geq 2t + 1 \)

\( \text{safe}(c, w) \triangleq \left\{ \left\{ i : w[i] = c \right\} \right\} \geq t + 1 \)

\( \text{higherValid}(c, w) \triangleq \exists i : w[i] = c \land ts \geq c.ts \land \neg \text{invalid}(c', w) \)

READ:

1. for \( 1 \leq i \leq n \), if \( pending[i] \) then \( old[i] \leftarrow true \)

2. for \( 1 \leq i \leq n \), enabled[i] \leftarrow true;

3. for \( 1 \leq i \leq n \), \( w[i] \leftarrow \bot, w[1][i] \leftarrow \bot \)

4. repeat

CHECK

until \( \left\{ i : \neg enabled[i] \land \neg pending[i] \right\} \geq n - t \)

5. \( C0 \leftarrow \{ o \in \text{TSVals} : \text{safe}(c, w) \land \neg \text{higherValid}(c, w) \} \)

6. \( C1 \leftarrow \{ c \in \text{TSVals} : \text{safe}(c, w) \land \neg \text{higherValid}(c, w) \} \)

7. if \( C0 \cup C1 \neq \emptyset \) then

\( \text{return cval} : \text{cval} = \max(c'.ts : c' \in C0 \cup C1) \)

\( \text{return } \bot \)

CHECK:

if \( \exists i : enabled[i] \land pending[i] \) then

\( \langle enabled[i], pending[i] \rangle \leftarrow \langle \text{false}, \text{true} \rangle \)

\( \text{INVOCATE read}\langle x_i \rangle \)

if \( \exists i : x_i \text{ responded} \) then

if \( \neg \text{old}[i] \) then

\( \langle w[0][i], w[1][i] \rangle \leftarrow \langle \text{tmp0}[i], \text{tmp1}[i] \rangle \)

pending[i] \leftarrow \text{false}; old[i] \leftarrow \text{false}

Lemma 2. If \( R = \text{READ} \) returns \( \bot \), then \( R \) is concurrent with at least two WRITE operations.

Proof. Let \( W = \text{WRITE}(v) \) be the last WRITE operation that completes before \( R \) is invoked, and \( ts \) be the timestamp used to write \( v \) to the base registers. Suppose w.l.o.g. that \( \langle ts, v \rangle \) was written to the \( x_i[0] \) component of the base registers \( x_i \). Assume by contradiction that READ overlaps at most one WRITE operation \( W' = \text{WRITE}(v') \) and let \( ts' \) be the timestamp used to write \( v' \) to the base registers. Since the base registers updated by \( W \) and \( W' \) intersect the base registers read by \( R \) by at least \( t + 1 \) correct registers, either \( \text{safe}(\langle ts, v \rangle, w_0) \) or \( \text{safe}(\langle ts', v' \rangle, w_1) \) (or both) are true. Moreover, there are at most \( 2t \) registers \( x_i \) such that \( x_i[0], ts < ts \lor (x_i[0], ts = ts \land x_i[0].val \neq v) \) or \( x_i[1], ts < ts' \lor (x_i[1], ts = ts' \land x_i[1].val \neq v') \). Hence, either \( \neg \text{higherValid}(\langle ts, v \rangle, w_0) \) or
-\text{higherValid}(ts',v'),w1) is true. Therefore, C0 \cup C1 \neq \emptyset so that \bot cannot be returned. A contradiction. 

The two lemmas above imply the following

**Lemma 3 (1-Regularity).** All the traces of the algorithm in Figure 1 are 1-regular.

Finally, both WRITE and READ are wait-free because neither one ever awaits more than $n - t$ replies and at least $n - t$ base registers are responsive. We proved the following

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

## 5 Wait-free Regular Register Construction

We now present a regular register construction from 1-regular and safe ones. We note that only one of the registers in this construction needs to be 1-regular; it suffices for the remaining regulars to satisfy the weaker safe semantics. For completeness, we show in Appendix A a direct construction of a wait-free safe register from $n > 4t$ regular base registers up to $t$ of which can be Byzantine faulty, using a technique similar to that of [16].

The algorithm in Figure 2 uses one wait-free 1-writer/m-reader 1-regular registers, $m$ 1-reader/1-writer safe registers writeable by the writer, and $m$ 1-reader/1-writer safe registers writeable by the reader to construct a wait-free 1-writer/m-reader regular register.

The read and write operations to the shared safe and 1-regular registers are denoted read and write. We construct WRITE and READ operations that maintain regularity.

The two writer's registers are called $P$ and $B$ for primary and backup respectively. These registers are used as buffers (or tracks) to store the values written by the writer. Each reader process $i$, $1 \leq i \leq m$ uses a 1-writer/1-reader multi-valued safe register $RL_i$ to signal the writer about a possible concurrent read in progress.

The construction works as follows: The writer starts by writing the primary register $P$ (line 1). It then examines the reader registers $RL_i$ to see whether the value of some of these registers has changed since the last time it was read by the writer. If so, it saves the last written value in the backup registers $B_j$ for every register $RL_j$ where a change was observed (lines 4–6) before returning ack (line 7).

The reader $i$ starts by writing the register $RL_i$ (line 2) and then proceeds to read the primary track $P$ (line 3). We consider the following two cases:

1. The read of the primary track by the reader (line 3) returns a value $\neq \bot$. In this case, 1-regularity of $P$ ensures that the return value preserves regularity;

2. The read of the primary track by the reader (line 3) returns $\bot$. In this case, 1-regularity implies that \textit{read}(P) by $i$ is concurrent with at least two writes to $P$ by the writer. Therefore, the writer's code is executed in whole at least once since \textit{read}(P) has been invoked. This implies that the writer observes the change in the value of the register $RL_i$, and therefore, writes the $i$'s backup track exactly one time before \textit{read}(P) completes. Therefore, when reader $i$ eventually returns from \textit{read}(P), it finds $B_i$ already written, and therefore, returns a correct value.
Shared objects:
1-writer/1-reader safe registers $RL_i \in \text{Integers}$, writeable by the reader $i$, $1 \leq i \leq m$, and readable by the writer; initially 0;
1-writer/m-reader 1-regular register $P \in \text{Vals} \cup \bot$ writeable by the writer and readable by the readers; initially $v_0$;
1-writer/1-reader safe register $B_i \in \text{Vals}$ writeable by the writer and readable by the reader $i$; initially $v_0$;

Local to the writer:
\[ \text{Static } \, w[1 \ldots m] \in \text{Integers} \]
\[ \text{initially } \forall i \, w[i] = 0; \]
\[ a_i \in \text{Integers}, \text{ for } 1 \leq i \leq m; \]

\[ \text{WRITE}(v); \]
\[ 1: \quad \text{write}(P, v); \]
\[ 2: \quad a_i \leftarrow \text{read}(RL_i), \text{ for each } i, 1 \leq i \leq m; \]
\[ 3: \quad \text{if } (\exists i : a_i \neq w[i]) \text{ then} \]
\[ 4: \quad \text{for each } i, a_i \neq w[i]: \]
\[ 5: \quad w[i] \leftarrow a_i; \]
\[ 6: \quad \text{write}(B_i, v); \]
\[ 7: \quad \text{return } \text{ack}; \]

Local to the reader $i$, $1 \leq i \leq m$:
\[ \text{Static } r[i] \in \text{Integers}, \text{ initially } 0; \]
\[ x \in \text{Vals} \cup \{\bot\}; \]

\[ \text{READ}_i, 1 \leq i \leq m: \]
\[ 1: \quad r[i] \leftarrow r[i] + 1; \]
\[ 2: \quad \text{write}(RL_i, r[i]); \]
\[ 3: \quad x \leftarrow \text{read}(P); \]
\[ 4: \quad \text{if } (x = \bot) \text{ then} \]
\[ 5: \quad x \leftarrow \text{read}(B_i); \]
\[ 6: \quad \text{return } x; \]

Figure 2: The 1-writer/m-reader Wait-Free Regular Register Construction.

We observe that all the writes to the backup registers (line 6) can be executed in parallel. Hence in a distributed implementation, updating all $B_i$’s would incur only the single round-trip message latency. This could be optimized even further by combining the writes targeted to the same destinations within a single message.

```
R_i
\[ R_i,\text{write}(RL_i) \quad \text{R_i,read}(P) \quad \text{R_i,read}(B_i) \]
W_1 \[ W_1,\text{write}(P) \]
W_2 \[ W_2,\text{read}(RL_i) \]
W_3 \[ \]
```

Figure 3: Several write operations overlapping read.

The above intuition is formalized by the following lemma:

**Lemma 4.** Let read overlap one or more write operations. Then read returns one of the values written in an overlapping write, or the value written in the latest write operation preceding the read.

**Proof.** (We refer the reader to Figure 3 for intuition on this proof.)
Let $R_i$ be a read operation by process $i$. Let $R_i$ consist of a write $R_i, write(RL_i)$ to $RL_i$; a read $R_i, read(P)$ of $P$; and potentially a read $R_i, read(B_i)$ of $B_i$.

For any write operation $W$, we refer to specific operations within write as follows. We denote by $W, write(P)$ the first write operation (to $P$). We denote by $W, read(RL_i)$ the read from $RL_i$, $W, a_i$ the value returned from $W, read(RL_i)$; and $W, w_l$ the value of $w_l$ at the beginning of $W$. We let $W, write(B_i)$ denote the write to $B_i$, if exists.

Let $W_i$ denote the latest write such that $W_i, write(P)$ terminates before $R_i, write(RL_i)$ completes. Let the two write operations succeeding $W_i$ be denoted by $W_2$ and $W_3$, respectively.

By choice of $W_1, write(P)$, $W_2, read(RL_i)$ strictly succeeds $R_i, write(RL_i)$ (see Figure 2). Hence, if no write operation before $W_2$ sees the value written in $R_i, write(RL_i)$, then $W_2$ must have $W_2, w_l = W_2, a_i$, and $W_2, write(B_i)$ exists. Otherwise, some write preceding $W_2$ already sees $R_i, write(RL_i)$ and writes $B_i$. In any case, at the latest when $W_2$ completes, a write to $B_i$ completes. Moreover, $B_i$ is written at most once during $R_i, write(RL_i)$. The value stored in $B_i$ in this write is the most up-to-date value written in a write preceding or concurrent with $R_i$.

Now there are two possible cases. The first case is when $R_i, read(P)$ returns a value other than $\bot$. Then by 1-regularity, it returns a good value from $P$ (i.e., it returns a non-$\bot$ value of an overlapping write or the latest write preceding the read).

The second case is when $R_i, read(P)$ returns $\bot$. Then by 1-regularity, $R_i, read(P)$ overlaps two write operations. Since $W_2$ strictly precedes $R_i, read(P)$, we have that $W_2$ and $W_3$ overlap $R_i, read(P)$. Hence, $W_2$ completes before $R_i, read(P)$ terminates. As we show above, at the latest, $W_2$ is a write that sees $R_i, write(RL_i)$ and updates $B_i$. Since $R_i, write(RL_i)$ completes before $R_i, read(B_i)$ starts, and because no new write to $B_i$ is invoked until $R_i, read(B_i)$ returns, by safety of $B_i$, $R_i, read(B_i)$ returns a non-$\bot$ value that upholds regularity.

Finally, if a read operation $R_i$ by a process $i$ is not concurrent with any write, then by 1-regularity of $P$, if there exists a write operation preceding $R_i$, then $R_i$ returns the value written to $P$ by the latest such write. Otherwise, $R_i$ returns $v_0$, the initial value of the register. Also, the algorithm is obviously wait-free since in a fair execution, it could never block forever in any statement of the pseudocode. Hence, we proved the following:

**Theorem 2.** The algorithm in Figure 2 is an implementation of a 1-writer/in-reader wait-free regular register from one 1-writer/in-reader wait-free 1-regular registers and 2m 1-writer/1-reader safe registers.

6 Conclusions

We have presented a simple, efficient, and self-contained construction of a wait-free regular register from Byzantine components. This yields a practical building block for distributed storage applications tolerating Byzantine faults.

References


A Wait-Free Safe Register Construction

In this section we present a wait-free SWMR safe register construction from $n > 4t$ regular registers up to $t$ of which can be Byzantine faulty (see Figure 4). The implementation is based on techniques similar to those of [16].

We now prove that the algorithm in Figure 4 is a safe register construction:

**Lemma 5 (Safety).** All traces of the algorithm in Figure 4 are safe.

**Proof.** Let $R$ be a READ operation that returns, and $W = \text{write}(v)$ be a WRITE operation that returns before $R$ is invoked, and assume that for no $W'$ operation that follows $W$, $W'$ is invoked before $R$ returns. We prove that $R$ returns $v$, thereby upholding safety. Indeed, by the WRITE code, $\text{write}((ts,v))$ terminates on at least $n-t>3t+1$ base registers before $R$ is invoked. Since READ awaits responses from at least $n-t$ registers, by regularity of $x_i$, there are at least $t+1$ correct registers that respond with $(ts,v)$ to the base object reads issued during $R$. Therefore, upon completion of line 4, $\text{safe}((ts,v))$ is true. Thus, $(ts,v) \in C$ after line 5. Finally, since no WRITE operation that follows $W$ is invoked before $R$ returns, there might be at most $t$ (faulty) base registers that return $(ts',v')$ with $ts' > ts$ or $ts' = ts \land v' \neq v$. Hence, $(ts,v)$ is the only highest timestamped value in $C$ which is safe. By line 8, $v$ must be the $R$'s return value in this case.  

Finally, the construction is trivially wait-free since neither WRITE nor READ are ever awaiting more than $n-t$ responses and at least $n-t$ registers are correct. Hence, we proved the following:

**Theorem 3.** The algorithm in Figure 4 is an implementation of a SWMR safe register from $n > 4t$ base regular register up to $t$ of which can be Byzantine faulty.
Shared objects
SWMR regular registers $x_i \in TSVals, 1 \leq i \leq n$; initially $x_i = \langle ts_0, u_0 \rangle$;

WRITE Emulation

Local variables:
\- enabled[1\ldots n], pending[1\ldots n] \in \text{Boolean}
\- initially $\forall i \text{ enabled}[i] = \text{pending}[i] = \text{false}$
\- $w \in TSVals; ts \in TS$

WRITE(v):
\- choose $ts \in TS$ larger than previously used;
\- $w \leftarrow \langle ts, v \rangle$
\- for $1 \leq i \leq n$, enabled[i] $\leftarrow$ true;
\- repeat
\- \hspace{1em} \text{CHECK;}
\- \hspace{2em} until $\{i : \neg \text{enabled}[i] \land \neg \text{pending}[i]\} \geq n - t$;
\- return ack;

CHECK:
\- if ($\exists i : \text{enabled}[i] \land \neg \text{pending}[i]$) then\hspace{1em} (enabled[i], pending[i]) $\leftarrow$ (false, true);
\- \text{INVOKE write($x_i, w$)};
\- if ($\exists i : x_i \text{ RESPONDED}$) then\hspace{1em} pending[i] $\leftarrow$ false;

READ Emulation

Local variables:
\- enabled[1\ldots n], pending[1\ldots n], old[1\ldots n] \in \text{Boolean}
\- initially $\forall i \text{ enabled}[i] = \text{pending}[i] = \text{false}$
\- $w[1\ldots n], tmp[1\ldots n] \in TSVals$;

Predicate and macro definitions:
\- safe(c) $\triangleq \{i : w[i] = c\} \geq t + 1$;

READ:
\- for $1 \leq i \leq n$, if(pending[i]) then old[i] $\leftarrow$ true;
\- for $1 \leq i \leq n$, enabled[i] $\leftarrow$ true;
\- for $1 \leq i \leq n$: $w[i] \leftarrow \perp$;
\- repeat
\- \hspace{1em} \text{CHECK;}
\- \hspace{2em} until $\{i : \neg \text{enabled}[i] \land \neg \text{pending}[i]\} \geq n - t$;
\- $C \leftarrow \{c \in TSVals : \text{safe}(c, w)\}$;
\- if ($C = \emptyset$) then
\- \hspace{1em} \text{return $\text{val: } cts = \max\{c' : c' \in C\}$};
\- \hspace{1em} return $u_0$;

CHECK:
\- if ($\exists i : \text{enabled}[i] \land \neg \text{pending}[i]$) then\hspace{1em} (enabled[i], pending[i]) $\leftarrow$ (false, true);
\- \text{INVOKE tmp[i] $\leftarrow$ read($x_i$)};
\- if ($\exists i : x_i \text{ RESPONDED}$) then
\- \hspace{1em} if ($\neg \text{old[i]}$) then\hspace{1em} $w[i] \leftarrow \text{tmp[i]}$
\- \hspace{2em} pending[i] $\leftarrow$ false; old[i] $\leftarrow$ false;

Figure 4: The SWMR safe register emulation.