Byzantine Disk Paxos:
Optimal Resilience with Byzantine Shared Memory*

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

September 2, 2005

Abstract

We present Byzantine Disk Paxos, an asynchronous shared-memory consensus algorithm that uses a collection of $n > 3t$ disks, t of which may fail by becoming non-responsive or arbitrarily corrupted. We give two constructions of this algorithm; that is, we construct two different t-tolerant (i.e., tolerating up to t disk failures) building blocks, each of which can be used, along with a leader oracle, to solve consensus. One building block is a t-tolerant *wait-free* shared safe register. The second building block is a t-tolerant regular register that satisfies a weaker termination (liveness) condition than wait freedom: its write operations are wait-free, whereas its read operations are guaranteed to return only in executions with a finite number of writes. We call this termination condition *finite writes (FW)*, and show that wait-free consensus is solvable with FW-terminating registers and a leader oracle. We construct each of these t-tolerant registers from $n > 3t$ base registers, t of which can be non-responsive or Byzantine. All the previous t-tolerant wait-free constructions in this model used at least $4t + 1$ fault-prone registers, and we are not familiar with any prior FW-terminating constructions in this model.

We further show tight lower bounds on the number of invocation rounds required for optimal resilience reliable register constructions, or more generally, constructions that use less than $4t + 1$ fault-prone registers. Our lower bounds show that such constructions are inherently more costly than constructions that use $4t + 1$ registers, and that our constructions have optimal round complexity. Furthermore, our wait-free construction is *early-stopping*, and it achieves the optimal round complexity with any number of actual failures.

Keywords: shared-memory emulations, t-tolerant object implementations, Byzantine failures, wait freedom, consensus, lower bounds.

[†]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.

*Accepted for publication in Distributed Computing. Preliminary version appears in Proceedings of the 23rd ACM Symposium on Principles of Distributed Computing (PODC '04), July 2004, pages 226–235.

1 Introduction

We consider an asynchronous system with multiple processes accessing fault-prone shared memory objects [AMT93, AGM+95, JCT98]. We study implementations of reliable objects from *base objects* that may fail by being non-responsive [AMT93, JCT98] or by returning arbitrary values [AGM+95, JCT98] (i.e., by being Byzantine); this failure model is called *non-responsive arbitrary (NR-Arbitrary) faults* [JCT98]. We focus on t-tolerant implementations [JCT98], that is, implementations that are correct as long as up to t base objects fail [JCT98]. In addition to memory failures, we assume that any number of the processes accessing the shared objects may crash.

This model is important in capturing a fair amount of recent work on Byzantine fault-tolerant "data centric" distributed storage, where data is stored at widely-dispersed servers, and is accessed by a large collection of ephemeral clients. Note that running a Byzantine fault-tolerant message-passing algorithm among all clients is not a viable option in such settings due to the ephemeral nature of clients. Data centric replication arises in three different application domains: (i) scalable client-server systems, e.g., Fleet [MR00], SBQ-L [MAD02], Agile Store [LAV03], Coca [ZSvR02], and [Baz00], in which servers store information on behalf of clients and all (or most) communication is between clients and servers, allowing servers to be highly scalable; (ii) Byzantine fault-tolerant peer-to-peer storage systems, e.g., Rosebud [RL04] and [LQLZ04]; and (iii) storage area networks, e.g., PASIS [GWGR04], where shared disks are used for data sharing as well as for synchronization. Since each server is entrusted with the data of many clients, modeling servers as prone to Byzantine faults protects clients from server intrusion. On the other hand, servers can use access control to ensure that clients access their own data only; tolerating Byzantine client failures is therefore meaningless, since a Byzantine client can corrupt its own data regardless of any measure taken by the protocol. Since our work is motivated by settings where processes access the shared memory over a network, our primary complexity metric is the number of *memory access rounds*, that is, the largest number memory operations invoked on a single base object.

Our goal is to enhance this fruitful line of work into a survivable distributed storage system that tolerates arbitrary corruption and unresponsiveness (i.e., NR-Arbitrary faults) in up to a third of its disks (or servers) as well as process crashes. (Tolerating NR-Arbitrary faults in a third or more of the disks is impossible [MAD02]). Although a number of projects have set out to tackle this problem (e.g., E-vault [GGJR00], Fleet [MR00], Agile Store [LAV03], SBQ-L [MAD02], Coca [ZSvR02], PASIS [GWGR04], as well as [ABO03] and [Baz00]), to date, this goal has not been achieved in our setting.

Consensus is a fundamental building block that may be used to realize such distributed storage systems. For solving consensus with shared disks, we turn to shared memory failure-detector based consensus algorithms [LH94, GL03], and in particular, the shared-memory version of Disk Paxos [GL03], which employs shared *wait-free single-writer multi-reader (SWMR) regular registers*[1] and a leader oracle; a leader oracle is a failure detector of class Ω, which outputs one trusted process at each process, and guarantees that eventually a single correct process is permanently trusted by all correct processes. Thus, the problem of solving consensus (assuming a leader oracle) can be reduced to implementing a wait-free SWMR regular register. When disks are subject to unavailability faults only, a reliable wait-free register is easily implemented from a collection of $2t+1$ crash-prone registers, each stored on one disk, by reading and writing from/to a majority of disks [GL03].

Coping with NR-Arbitrary faults is more challenging. Since the introduction of this failure model, researchers have constructed t-tolerant wait-free shared registers using $4t+1$ [MR98, GWGR04], $5t+1$ [JCT98], or even $6t+1$ [CMR01] fault-prone base objects. Several works have achieved better resilience by weakening the model in different ways – by adding synchrony [Baz00]; by storing signed self-verifying data [MR98, MAD02]; or by providing solutions that may block indefinitely if processes fail [MAD02, ABO03]. However, $t < n/4$ is the best resilience achieved thus far by t-tolerant wait-free constructions for the model considered herein.

In contrast, the literature is abundant with message-passing consensus algorithms that tolerate Byzantine failures of less than a third of the processes. Therefore, an appealing way to go about searching for a more resilient solution would be to try and adapt the techniques used in those algorithms to our model. (Since our model does not incorporate digital signatures, we restrict our

[1] A *wait-free object* is one that is live in the presence of any number of process failures. 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.

attention to consensus algorithms that do not use authentication). We observe that this resilience is achieved by means of echoing (e.g., [BT85, DLS88]). Unfortunately, echoing cannot help us address the challenge we have set out to solve in this paper. Indeed, if a correct process can correctly echo information to all other processes, this is essentially like having a wait-free register through which the process conveys the information to the other processes. And implementing such a register from fault-prone ones is exactly what we seek to do in this paper.

Having ruled out the use of standard techniques to improve the resilience threshold, we proceed to examine whether there exist inherent limitations that prevent algorithms in our model from achieving better resilience. We observe that all existing algorithms for fault-prone shared memory models (e.g., [MR98, MAD02, Baz00, JCT98, ABO03, GWGR04]) implement (emulate) write operations in a single round; that is, they invoke one write operation on each base object. Moreover, all the solutions that assume $t < n/4$, also implement read operations in a single round (e.g., [MR98, Baz00, JCT98, GWGR04]). In Section 7, we show that such good performance is not attainable by optimal-resilience solutions. Specifically, Section 7.1 proves that if $t \geq n/4$, then it is impossible to emulate the write operations of a t-tolerant wait-free register by invoking a single round of operations on base objects. Our proof applies to emulations of all meaningful register types, as it is proven for a binary single-writer single-reader *safe register*[2] [Lam86]. We further show, in Section 7.2, that if $n = 3t + 1$, then every algorithm in which the reader does not modify the base objects' states has at least one execution in which at least $t + 1$ rounds of read operations are invoked in order to emulate a single read operation of a t-tolerant single-writer single-reader safe register. More generally, for any $0 \leq f \leq t$, there is an execution in which f objects are Byzantine faulty in which the algorithm invokes $\min(t+1, f+2)$ rounds of base object operations. We further conjecture that even if readers can modify the base objects, then it still holds that either the read or the write emulation must take $\min(t+1, f+2)$ rounds. Our bounds explain why previous algorithms, which did not perform more than one round of operations, and did not have readers modify base objects, could not have achieved optimal resilience.

[2] A safe register guarantees that every read operation that does not overlap any write operation 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.

In Section 5, we present a t-tolerant wait-free SWMR register construction that uses as little as $3t + 1$ base registers that may suffer arbitrary corruption. As dictated by the lower bounds of Section 7, we implement (emulate) write operations in two rounds, that is, our emulation invokes two operations on some of the base objects. Our read emulation is *early-stopping*, and it incurs the optimal $\min(f + 2, t + 1)$ rounds, where f is the actual number of Byzantine faults in the given execution. Specifically, we construct a t-tolerant wait-free SWMR safe register; multi-writer registers can be constructed from single-writer ones [VA86]. Using known reductions from regular to safe registers (see e.g., [Lam86] and a survey in [HV02]), we can thus achieve a t-tolerant wait-free regular register, which in turn, can be used to solve consensus with a leader oracle [LH94, GL03]. Nevertheless, it is worth noting that implementing a regular register using safe ones requires additional space (as multiple safe registers are used to construct a single regular one), induces additional rounds of memory access, and adds additional complexity, as most existing constructions are fairly elaborate.

We therefore further seek a simpler, self-contained, and efficient implementation of a regular register. The key to such a solution is a very simple yet surprisingly powerful shift of paradigm: we weaken the termination condition the register is required to satisfy. Specifically, we define a new termination condition called *finite writes termination (FW-termination)*, which guarantees progress only in executions with a finite number of writes. In other words, write operations always terminate, whereas read operations are guaranteed to terminate whenever they occur in parallel with a finite number of writes. In order for FW-terminating registers to guarantee progress for the readers, a contention management mechanism is required, so as to limit the number of writes occurring in parallel with read operations. Nevertheless, we observe that in the context of consensus, this can be provided by a leader oracle, which is necessary for consensus anyway [LH94, DFG02, CHT96].

Indeed, this leads us to implement a t-tolerant FW-terminating reliable *regular* register out of ones that can suffer NR-Arbitrary faults, and to use such registers for implementing consensus. The result is a simple, efficient, and self-contained adaptation of Disk Paxos, which tolerates NR-Arbitrary faults of up to a third of the disks. As our FW-terminating construction is simpler than the wait-free one, we present it first in the paper, in Section 4.

From a formal perspective, solving wait-free consensus with shared objects that are not wait-free is in itself a contribution. In [LH94, GL03, DFG02], it was shown that wait-free consensus is possible with wait-free read/write registers and a leader oracle. We show, for the first time, that registers satisfying a weaker progress condition (i.e., have more allowable behaviors) suffice. This approach integrates well with the Paxos general philosophy, which decomposes consensus into a safety building block (called Synod in [Lam98]) and a progress component (leader election). In shared memory this deconstruction was substantiated in [BDFG03, CM02], where coarse-grained shared objects encapsulating the Synod algorithm were identified. In this paper, the approach of separating safety requirements from liveness ones is applied right down to the lowest level objects of which Paxos is constructed: the read/write registers.

Contributions and road map. Section 2 presents the formal system model and defines the register types considered in this paper. Section 3 provides informal intuition for the formal results presented later in the paper: it discusses previous register emulations in the NR-Arbitrary fault model, illustrates the challenges in working with optimal resilience, and exemplifies our results (lower bounds and upper bounds). We then turn to introduce Byzantine Disk Paxos, a shared-memory consensus algorithm that tolerates NR-Arbitrary faults of up to a third of the system. We present two constructions of Byzantine Disk Paxos. In Section 4, we present a direct construction of a t-tolerant FW-terminating regular register from Byzantine shared memory. Section 6 identifies such registers as building blocks for consensus. In Section 5, we present an emulation of a t-tolerant wait-free register out of $3t+1$ corruptible ones. In Section 7, we prove tight lower bounds on the round complexity of emulations that use less than $4t+1$ base objects, showing that the construction in Section 5 has optimal round complexity as well as resilience, and that the write emulation of the FW-terminating register in Section 4 is also optimal. Our complexity lower bounds are proven for implementations of the weakest meaningful register type, namely a binary SWSR safe register, from atomic objects of arbitrary type. Our constructions employ regular SWMR base registers; the fact that these constructions achieve optimal complexity implies that using atomic objects would provide no added value. Our constructions do not work with safe base registers; although it is possible to emulate regular registers from safe ones, this would induce additional complexity.

2 The System Model

We consider an asynchronous shared memory system consisting of a collection of processes interacting with a finite collection of objects. Objects and processes are modeled as I/O automata [LT89]. An I/O automaton's state transitions are triggered by *actions*. Actions are classified as *input*, *output*, and *internal*. The automaton's interface is determined by its input and output actions, which are collectively called *external* actions. An action π of an automaton A is said to be *enabled* in state s if A has a state transition of the form (s, π, s'). The transitions triggered by input actions are always enabled, whereas those triggered by output and internal actions, (collectively called *locally controlled* actions), depend solely on the automaton's current state.

Let A be an I/O automaton. An execution α of A is a (finite or infinite) sequence of alternating states and actions $s_0, \pi_1, s_1, \ldots$, where s_0 is A's initial state, and each triple (s_{i-1}, π_i, s_i) is a state transition of A. The trace of an execution α of A is the subsequence of α consisting of the external actions in α. An infinite execution α of A is *fair* if every locally controlled action of A either occurs infinitely often in α or is disabled infinitely often in α. A finite execution α of A is *fair* if no locally controlled action of A is enabled at the end of α. A *fair trace* of A is the trace of a fair execution of A. An automaton's external behavior is specified in terms of the properties of its traces. Liveness properties are required to hold only in fair traces.

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* $\times$ $Vals$, specifying the object's semantics in sequential executions.

An asynchronous shared memory system is a composition of a (possibly infinite) collection of process automata $P_1, P_2, \ldots$ and a finite collection of object automata $O_1, O_2, \ldots O_n$. Let O_j be an object of type $\mathcal{T}$, and a (b) be an invocation (resp. response) of $\mathcal{T}$. Process P_i interacts with O_j using actions of the form a_i (resp. b_i), where a_i is an output of P_i and an input of O_j (resp. b_i is an output of O_j and an input of P_i).

We say that the interaction between a process and an object is *well-formed* if it consists of alternating invocations and responses, starting from an invocation. In this paper, we only consider systems in which the interaction between P_i and O_j is well-formed for all i and j. Well-formedness allows an invocation occurring in an execution α to be paired with a unique response (when such exist). If an invocation has a response in α, the invocation is said to be *complete*; otherwise, it is *incomplete*. If two invocations are incomplete after some prefix of α, then they are said to be *overlapping* in α. 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.

Objects may suffer NR-Arbitrary failures [JCT98], i.e., may fail to respond to an invocation, or may respond with an arbitrary value. We consider t-tolerant implementations [JCT98], which remain correct (in the sense that the emulated object satisfies its specification) whenever at most t base objects suffer NR-Arbitrary failures.

Any number of the processes may fail by stopping. The failure of a process P_i is modeled using a special external event $stop_i$. Once $stop_i$ occurs, all locally controlled actions of P_i become disabled indefinitely. A process that does not fail in an execution is *correct* in that execution.

2.1 Registers

A *read/write register* (or simply, register) type supports an arbitrary set of values, $Vals$, with an arbitrary initial value $v_0 \in Vals$. Its invocations are read and write(v), $v \in Vals$. Its responses are $v \in Vals$ and *ack*. Its sequential specification, f, requires that every write overwrites the last value written and returns ack (i.e., $f(\text{write}(v), w) = (ack, v)$), and every read returns the last value written (i.e., $f(\text{read}, v) = (v, v)$). In this paper, we consider only single-writer registers, i.e., registers that can be written by a single pre-designated process. Registers can be either *single-writer multi-reader (SWMR)*, meaning that they can be read by any number of processes, or *single-writer single-reader (SWSR)*, in which case only one designated processes can read from them.

We now define several register properties. Let σ be a (well-formed) sequence of invocations and responses of reads and writes.

Safe register. σ is *safe* [Lam86] 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. σ is *regular* [Lam86] 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.

Wait Freedom. σ satisfies *wait freedom* if every invocation by a correct process in σ is complete. A register is *wait-free* if all its fair traces satisfy wait freedom.

FW-termination. σ satisfies *FW-termination* if every write invocation by a correct process in σ is complete, and moreover, either every read invocation by a correct process is complete, or infinitely many writes are invoked in σ. A register is *FW-terminating* if all its fair traces satisfy FW-termination.

Note that our liveness definitions (wait freedom and FW-termination) require operations by correct process to complete regardless of the number of process failures, since failed processes are not required to take any steps in fair executions (as their locally controlled actions are all disabled).

We now examine the relationship of FW-termination with previously suggested termination conditions by comparing the sets of behaviors they allow. We observe that the set of FW-terminating traces is a strict subset of both lock freedom (sometimes called non-blocking) and obstruction freedom [HLM03] (or deterministic solo termination). An FW-terminating trace is lock-free since progress is always guaranteed: if there is a write invocation, it is guaranteed to complete, and if there is none, all read invocations are guaranteed to complete. An FW-terminating trace is also obstruction free, since when a read invocation is allowed to take steps by itself, other processes cannot initiate infinitely many write invocations in parallel with the read, and hence FW-termination ensures that the read completes, as required by obstruction-freedom. FW-termination is a strict superset of wait freedom, since a read operation that is concurrent with infinitely many writes is not required to complete. This relationship is illustrated in Figure 1.

We note that FW-termination is similar to giving priority to the writers in the readers-writers problem, but differs from it in that if a writer fails while accessing the shared object, FW-termination guarantees progress to the readers, whereas in the readers-writers problem, a single write operation that does not relinquish the shared object keeps all readers are blocked.

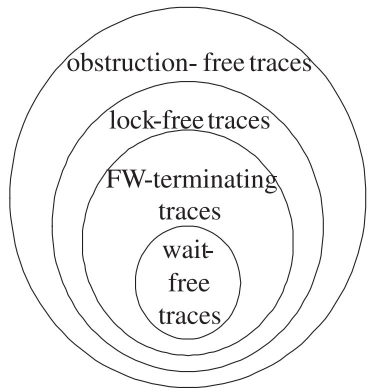


Figure 1: Relationship among different termination conditions.

Finally, although we have defined FW-termination above specifically for read/write registers, we note that our definition may be extended to model any single-writer multi-reader data structure.

3 Exemplifying the Results

This section provides informal intuition for the lower bounds and algorithms presented in this paper. Section 3.1 describes previously suggested register emulations from Byzantine storage, and uses them in order to illustrate the challenge in achieving optimal resilience. Section 3.2 then intuitively describes the techniques we use in our algorithms.

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).

3.1 Previous Solutions and Remaining Challenges

Traditionally, in asynchronous algorithms, one waits for at most $n - t$ responses to each request, since waiting for more objects may violate liveness. Thus, an emulated WRITE(v) operation issues write requests to all base objects, and returns once the lower-level write operations on $n - t$ of the n base objects return. WRITE operations are implemented exactly in this manner in all previously suggested constructions tolerating NR-Arbitrary memory faults, e.g., [MR98, MAD02, JCT98, ABO03, GWGR04]. Note that of the $n-t$ base objects that respond, t may be faulty, whereas the t that have not responded may be simply slow. Thus, when a WRITE operation completes, there can be t correct base objects that do not store the written value. Likewise, a READ invocation typically sends read requests to all base objects and waits for $n-t$ responses. If as in [MR98, GWGR04], $n > 4t$, then the set of $n-t$ read objects includes at least $2t+1$ correct objects, of which at least $t+1$ were updated by every complete WRITE. Since no incorrect value can be read from $t+1$ objects, if written values are associated with monotonically increasing timestamps, READ can safely return the highest timestamped value read from $t+1$ base objects [MR98, GWGR04]. Jayanti et al [JCT98] eliminate the need for timestamps by using a resilience threshold of $n > 5t$ and returning values read from $2t+1$ base objects.

However, working with $n \leq 4t$ is more challenging. Let us examine the special case that $t = 1$ and resilience is optimal, i.e., $n = 4$, and only safe semantics are required. In this case, after WRITE(v) completes, it is possible that only 2 correct base objects have stored v. Safety mandates that if a READ operation is invoked after WRITE(v) returns, and no further WRITE operations are invoked, then the READ must return v. The READ operation probes the base objects and waits for responses from $n-t = 3$ of them (in order to ensure liveness). The responses may be as follows: one correct object that did not store v returns an old value s_0, one incorrect object also returns s_0, and only one correct object returns v. (This scenario is illustrated in Figure 8(c) in Section 7.1, where we formally prove our lower bound on WRITE emulations). The reader has no way of distinguishing this situation from one where v was never written and is returned by a faulty object. Therefore, the reader cannot return.

One may attempt to overcome this situation by simply waiting for more responses. In the above scenario, the reader can in fact wait for an additional response, since it has already heard from the only faulty object, and the fourth object is correct and will eventually respond as well. However, the reader cannot distinguish the scenario above from a situation in which WRITE(v) is in progress during the READ, and all three responses are from correct objects. In this case, the reader cannot wait for a response from the fourth object, which may be faulty. Since the reader can neither safely return any value nor wait for an additional response, it must invoke another round of base object read operations. More generally, in Section 7.2, we formally prove that if the reader does not write to the base objects, there are executions in which READ must invoke at least $t+1$ rounds of read operations on base objects.

If one assumes that processes cannot fail, then the WRITE(v) operation (implemented by writing to 3 base objects, as described above) eventually completes. If no further WRITEs occur and READ continues to initiate additional read rounds, then eventually, 2 correct objects return v in some read round and another object returns a value with a smaller timestamp, and READ can return safely v. This is the approach taken by Attiya and Bar-Or [ABO03], where READ operations are guaranteed to eventually terminate assuming that processes do not fail and a finite number of WRITEs are invoked. A similar approach is implemented by Martin et al. [MAD02], where the shared objects reside on servers that implement a subscription model and push all register updates to the subscribed clients instead of having the clients continuously issue read rounds. (Note that this model is different from the shared memory model we consider in this paper.) By allowing this additional functionality at the servers, Martin et al. guarantee termination even when there are infinitely many WRITEs.

Unfortunately, if processes can fail, the above approach violates liveness. (Martin et al. [MAD02] overcome this liveness problem by allowing servers to communicate with each other, which is impossible in the shared memory model). If a writer fails (crashes) in the course of the WRITE(v) invocation, then the system can permanently remain in a state where exactly one correct object has stored v (see Figure 8(b)). In this situation, even if the reader continues to initiate read rounds and no other processes take steps, the reader will not be able to complete the READ. In Section 7.1 we formally prove that, indeed, in order to tolerate process failures, WRITE must invoke two operations on some base object.

3.2 Intuitive Description of Our Algorithms

We now illustrate the general idea behind our optimal-resilience algorithms for the special case that $n = 4$ and $t = 1$.

As dictated by the lower bound described above, our algorithms emulate WRITE by invoking two rounds of operations on base objects. Each base object (register) stores two values. The emulation of WRITE(v) first performs a *pre-write* phase, in which v is written to the base registers' first field, pw. After getting acks from $n-t$ base objects that have stored v, the *write* phase writes v to $n-t$ registers' second field, w. This solves the problem described above, since if the writer fails before finishing the pre-write phase, v does not appear in any register's w field, and eventually, all 3 correct registers will attest to the fact that v was never fully written, whereas if the writer fails after completing the pre-write phase, then v is stored in 2 correct registers' pw fields, and the reader can therefore know that WRITE(v) was indeed invoked. Finally, if the reader reads two values, v and s_0, each from 2 objects, then READ must return the later one. To this end, each value is written along with a monotonically increasing timestamp. Our FW-terminating READ emulation (presented in the next section) continuously invokes read rounds until there is a value that appears in the pw fields of 2 registers (or more generally, $t+1$ registers), and for every higher timestamped read value v', there are at least 3 registers (more generally, $2t+1$) that do not return v' in their w fields. The number of rounds initiated by this algorithm is unbounded, by design, but the algorithm is guaranteed to terminate in executions with a finite number of WRITEs, even if the writer fails.

In Section 5, we present a t-tolerant wait-free safe register construction that bounds the number of read rounds. In the special case that $t = 1$, our t-tolerant wait-free READ emulation invokes at most two read rounds. In the first read round, READ reads from 3 registers, and collects candidate return values— these are the values that are read from w fields. It then issues a second read round. Since this algorithm implements a safe register, it can return an arbitrary value when a WRITE overlaps the READ. If no WRITE overlaps the READ, then the latest written value is stored at at least 2 correct registers' w fields throughout the READ. Therefore, if there is any candidate value v such that 3 registers respond without v in their w fields, then v can be removed from the set of candidates. If the set of candidates is empty, there must be a WRITE overlapping the READ, and any value can be returned. Otherwise, consider the highest timestamped candidate v. If 2 registers respond with v or higher timestamped values, then v is a valid return value. This is because at least one of these registers is correct, implying that either v was indeed written, or WRITE(v') occurred for some higher timestamped value v', which is not a candidate. In the latter case, since v' is not a candidate, WRITE(v') overlaps the READ. Thus, either way, v can be returned.

In order for READ to return, we thus need to ensure that eventually, each candidate is either missing from 3 responses, or there are 2 responses that include v or a higher timestamped value. When $t = 1$, two read rounds suffice to ensure this. To see why, consider a candidate value v:

1. First, if v was concocted by a faulty register in the first round, then there is an additional correct register that did not yet respond in the first round, and will eventually respond. Once this register responds, there will be 4 responses in the first read round, ensuring that every value either occurs in 2 of them or does not occur in 3 of them.

2. Otherwise, v was read from the w field of a correct register in the first round, implying that its pre-write phase has completed before it was read. The pre-write could have taken place concurrently with the first read round, but since the second read round causally follows the first, the pre-write must have completed and stored v at 2 correct registers before the second read round. Thus, at least 2 correct registers will eventually respond to the second round with either v or a later value in their pw field.

As noted above, when $t > 1$, two read rounds do not always suffice, and the general algorithm is quite a bit more complex.

4 t-Tolerant FW-Terminating Regular Register Emulation

In this section, we construct a t-tolerant FW-terminating SWMR regular register in a shared memory system consisting of an arbitrary number of processes and $n > 3t$ SWMR fault-prone FW-terminating regular registers, $x_1 \ldots x_n$. The register emulation is presented in Section 4.1. Its correctness is proven in Section 4.2, and its efficiency is discussed in Section 4.3.

4.1 Register Emulation

Each base register, x_i, stores a pair of values, each associated with a timestamp, taken from a totally ordered set TS, with the minimum element ts_0. The shared registers are defined in Figure 2.

Types:
$TSVals = TS \times Vals$, with selectors ts, val;

Shared regular registers $x_i \in TSVals \times TSVals$, $1 \leq i \leq n$, with selectors pw, w,
initially, $\langle\langle ts_0, v_0\rangle, \langle ts_0, v_0\rangle\rangle$ for all $1 \leq i \leq n$;

Figure 2: Base registers used in register constructions.

The emulation of the FW-terminating register's WRITE operation appears in Figure 3. As dictated by the lower bound of Section 7.1, the WRITE emulation consists of two rounds: First, the *pre-write phase* writes the value to the base registers' *pw* fields, and then, the *write* phase writes the new value to both the registers' fields. Each value is written together with a monotonically increasing timestamp. Since the underlying registers can be non-responsive, the process must invoke operations to different registers in parallel in separate threads, so as to avoid blocking forever when waiting for a faulty register to respond. Each phase (write and pre-write) is complete once $n - t$ of the registers (threads) respond. Threads that do not respond by the time WRITE is complete remain active (pending) after it returns.

In order to ensure well-formedness, (i.e., that at any instant, the writer has at most one incomplete invocation on each register), subsequent instances of WRITE must refrain from invoking new operations on registers whose threads are still pending. To this end, we track the status of each base register x_i using two bits. The *pending[i]* bit indicates whether an invocation on x_i is in progress. To initiate an invocation on x_i, the main thread sets *enabled[i]* to true, and then repeatedly calls the procedure CHECK in order to invoke write operations on the base registers for which an invocation is enabled and none is pending, and to check when these invocations are complete. When an operation on x_i is invoked, CHECK sets *enabled[i]* to false. Thus, a repeat-CHECK-until loop never invokes more than one operation on each base objects. Once invocations on $n - t$ registers have responded (i.e., are neither enabled nor pending), the phase is complete.

The notation INVOKE write(x_i, v), (resp., INVOKE $tmp \leftarrow$ read(x_i)) means that a new thread is spawned to perform a write(v) on register x_i (resp., 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 on register x_i has completed its execution. Note that since we maintain

Local variables:
    Boolean arrays $enabled[n],\ pending[n]$, initially false for all $1 \leq i \leq n$;
    $pw,\ w \in TSVals$, initially $\langle ts_0, v_0 \rangle$;
    $ts \in TS$;

WRITE(v):
    choose $ts \in TS$ larger than previously used;

    / Pre-write phase */*
    $pw \leftarrow \langle ts, v \rangle$;
    for $1 \leq i \leq n,\ enabled[i] \leftarrow$ true;
    repeat
        CHECK;
    until $|\{i\ :\ \neg enabled[i] \wedge \neg pending[i]\}| \geq n - t$;

    / Write phase */*
    $w \leftarrow \langle ts, v \rangle$;
    for $1 \leq i \leq n,\ enabled[i] \leftarrow$ true;
    repeat
        CHECK;
    until $|\{i\ :\ \neg enabled[i] \wedge \neg pending[i]\}| \geq n - t$;

    return ack;

CHECK:
    for $1 \leq i \leq n$
        if $(enabled[i] \wedge \neg pending[i])$ **then**
            $\langle enabled[i], pending[i] \rangle \leftarrow \langle$false,true$\rangle$;
            INVOKE write($x_i, \langle pw, w \rangle$);
        if (x_i RESPONDED) **then**
            $pending[i] \leftarrow$ false;

Figure 3: Emulation of WRITE operations.

well-formedness, at any given instant of time, there is at most one incomplete operation invoked on register x_i. Hence, the notation x_i RESPONDED is well-defined.

The READ emulation appears in Figure 4. It repeatedly invokes rounds of read operations on base registers, until it finds a value that it can safely return. For each register x_i, $w[i]$ and $pw[i]$ hold the latest value read from $x_i.w$ and $x_i.pw$, resp. Like the WRITE emulation, READ uses the $pending[i]$ and $enabled[i]$ bits in order to track the status of active invocations to x_i and ensure well-formedness. Each read round is invoked by setting all the *enabled* bits to true (line 4), and repeatedly calling the procedure CHECK until $n - t$ registers respond (lines 5–7). As in the WRITE emulation, CHECK invokes enabled base register operations and checks the status of pending ones. Recall that threads invoked in one instance of READ may remain active after that instance returns. In this situation, subsequent READs must ignore the return values of such old threads, so as not to violate safety. To this end, the *old[i]* bit is used. When READ is invoked (line 1), this bit is set for registers that have pending invocations from previous READs. When old threads return, their data is discarded.

READ defines a number of predicates in order to determine which value is safe to return. The $\mathsf{readFrom}(c, i)$ predicate is true if the value-timestamp pair $c \in TSVals$ was read from x_i, either from the w or from the pw field, by the latest read operation invoked on x_i. In order to ensure that READ does not return a value concocted by faulty registers, the return value must be read from at least $t + 1$ registers. This condition is captured by the predicate safe.

In order to ensure regularity, READ must not return old values written before the last WRITE that precedes the READ. Enforcing this condition is more subtle: although returning the highest timestamped read value would ensure this, this value cannot be returned unless it is safe. Simply waiting for the highest timestamped value to become safe may violate liveness, because this value may come from a faulty register. To overcome this difficulty, we introduce the predicate invalid. This predicate ascertains that a given value-timestamp pair was *not* written before READ was invoked, and can therefore be safely excluded from the set of potential return values. A value-timestamp pair is deemed invalid if $2t+1$ of the registers either return values with lower timestamps or return a different value with the same timestamp. The algorithm then waits for the highest timestamp-value

Local variables:

Boolean arrays $enabled[n],\ pending[n],\ old[n]$, initially false for all $1 \le i \le n$;
Arrays $pw[n],\ w[n],\ tmpPW[n],\ tmpW[n]$ with elements in $TSVals \cup \{\bot\}$;
$C \subseteq TSVals$;

Predicate definitions:

$\mathsf{readFrom}(c,i) \triangleq c \in TSVals \wedge (pw[i] = c \vee w[i] = c)$
$\mathsf{safe}(c) \triangleq |\{i\ :\ \mathsf{readFrom}(c,i)\}| \ge t+1$
$\mathsf{invalid}(c) \triangleq |\{i\ :\ \exists c' : \mathsf{readFrom}(c',i) \wedge c'.ts < c.ts \vee (c'.ts = c.ts \wedge c'.v \ne c.v)\}| \ge 2t+1$
$\mathsf{highestValid}(c) \triangleq \forall c' \forall i : (\mathsf{readFrom}(c',i) \wedge c'.ts \ge c.ts \wedge c' \ne c) \rightarrow \mathsf{invalid}(c')$

READ():
for $1 \le i \le n,$ **if**$(pending[i])$ **then** $old[i] \leftarrow$ true;
for $1 \le i \le n,\ pw[i], w[i] \leftarrow \bot$;
repeat
  for $1 \le i \le n,\ enabled[i] \leftarrow$ true;
  repeat
    CHECK;
  until $|\{i\ :\ \neg enabled[i] \wedge \neg pending[i]\}| \ge n-t$;
  $C \leftarrow \{c\ :\ \mathsf{safe}(c) \wedge \mathsf{highestValid}(c)\}$;
until $(C \ne \emptyset)$;
return $c.val : c \in C$;

CHECK:
for $1 \le i \le n$
  if $(enabled[i] \wedge \neg pending[i])$ **then**
    $\langle enabled[i], pending[i]\rangle \leftarrow \langle$false,true$\rangle$;
    INVOKE $\langle tmpPW[i], tmpW[i]\rangle \leftarrow \mathrm{read}(x_i)$;
  if (x_i RESPONDED) **then**
    if $(\neg old[i])$ **then**
      $pw[i] \leftarrow tmpPW[i]$;
      $w[i] \leftarrow tmpW[i]$;
    $pending[i] \leftarrow$ false;
    $old[i] \leftarrow$ false;

Figure 4: Emulation of READ operations of the t-tolerant FW-terminating regular register.

pair that is not invalid to become safe, and returns this value. The predicate $\mathsf{highestValid}(c)$ holds for a timestamp-value pair c if all the other read values with a timestamp greater than or equal to c's are invalid.

The set C holds value-timestamp pairs that are safe, and for which all the pairs with a higher timestamp or with the same timestamp and a different value are invalid (line 8). Once $C \neq \emptyset$, READ terminates (line 9) and returns some value in C (line 10). This guarantees regularity, as proven in Lemma 4.1 below. The emulation is also FW-terminating, since once no more WRITE invocations occur, the latest written value eventually becomes safe, and higher timestamped values from faulty registers are eventually invalidated. This is proven in Lemma 4.2 below.

Note that even if we use wait-free base registers, the register implementation in Figures 3 and 4 is not wait-free. Even with infinitely many WRITE invocations, a reader may never return even in a fault-free execution. This is because in each read round, one new (concurrently-written) candidate value may be observed in both the pw and w fields of one base register, while the other correct registers respond with older values due to asynchrony. The new candidate value is neither safe nor invalid.

4.2 Correctness

Lemma 4.1 (Regularity). *The register whose* WRITE *emulation appears in Figure 3 and whose* READ *emulation appears in Figure 4 is regular.*

Proof. We prove that the algorithm has only regular traces. First, observe that if READ returns a value $c.val$, then $\mathsf{safe}(c)$ holds. Thus, at least $t+1$ registers respond with c, and at least one of these is correct. Therefore, c has either been written by WRITE($c.val$) or is $\langle ts_0, v_0\rangle$. It is left to show that READ does not return older values than the one written by the latest complete WRITE before the READ.

If no WRITE completes before READ is invoked, then we are done. Otherwise, let R be a READ invocation and W = WRITE(v) be the last WRITE that completes before R is invoked. Let ts be the timestamp written with v. We need to show that R does not return an older value, i.e., that any return value $c.val$ is not associated with a timestamp $c.ts < ts$ (as timestamps are monotonically increasing). That is, we need to show that if $c.val$ is returned, then $c.ts \geq ts$.

Since the write phase of W completes before R is invoked, $\langle ts, v\rangle$ is written to the pw and w fields of at least $t+1$ correct registers before the read. Since the base registers are regular, each of these $t+1$ correct registers responds to each read operation of R with a pair $\langle pw, w\rangle$ such that $pw.ts \geq ts \wedge w.ts \geq ts$. Consider the reader's state after line 8 (in any iteration) of R. As $n-t$ responses are awaited in line 7, at least one of the responders is among the correct registers updated by W. Denote this register as x_i. Hence, $pw[i].ts \geq ts \wedge w[i].ts \geq ts$. Let c be the smallest timestamped pair returned by a correct register x_k (either in its pw or w field) for which $c.ts \geq ts$. We prove that c is not $\mathsf{invalid}$. Assume the contrary. By definition of $\mathsf{invalid}$, at least $2t+1$ registers must have responded with values c' (in either their pw or w fields) such that $c'.ts < c.ts \vee (c'.ts = c.ts \wedge c'.val \neq c.val)$. Thus, at least one of these responses must be from a correct register x_j that was updated by W. Therefore, $pw[j].ts \geq ts \wedge w[j].ts \geq ts$. By choice of c, either $pw[j].ts = c.ts \wedge pw[j].val \neq c.val$, or $w[j].ts = c.ts \wedge w[j].val \neq c.val$. Since x_j and x_k are both correct, two different values were written with the same timestamp, which by the WRITE code is impossible. A contradiction.

We have proven that whenever line 8 is executed, there is a timestamp-value c such that $\mathsf{readFrom}(c, k)$ and $c.ts \geq ts$ and $\neg\mathsf{invalid}(c)$. Therefore, no c' such that $c'.ts < ts$ can satisfy the $\mathsf{highestValid}$ predicate, and no value with a timestamp smaller than ts can be included in C in line 8 or returned in line 10. Hence, regularity is satisfied. □

Lemma 4.2 (FW-Termination). *The register emulated by the* WRITE *code in Figure 3 and the* READ *code in Figure 4 is FW-terminating.*

Proof. Since at most t registers are faulty, by FW-termination, in every fair execution, at least $n-t$ correct registers respond to every write invocation. Since no more than $n-t$ responses are awaited in either phase of any WRITE invocation, every WRITE invoked by a correct process in a fair execution completes.

Let α be a fair execution in which a finite number of WRITE invocations occur. We now prove that every READ invocation by a correct process in α completes. Since as argued above, the WRITE invocations all complete, and since the READ emulation does not invoke write operations on base registers, there is a point in α after which no write operations on base registers are invoked, and by FW-termination of the base registers, a later point τ, by which all write operations invoked on correct registers are complete.

We first note that READ is not stuck forever in the repeat-until loop in lines (5–7), since $n-t$ responses are awaited in each iteration, at most t registers can be non-responsive, and the correct registers all respond by FW-termination (since a finite number of writes are invoked on base registers in α). Therefore, READ continues to issue new read rounds as long as it does not return a value. Assume by contradiction that READ never returns, then it invokes read operations on all correct registers after time τ. Let $\tau' > \tau$ be a point in α by which every correct register has responded to at least one read invocation that was initiated after point τ.

Let $\langle ts, v\rangle$ be the value-timestamp pair written in the last complete WRITE invocation in α, or $\langle ts_0, v_0\rangle$ if there is none. We consider two cases: First, if no later (incomplete) WRITE completes the pre-write phase (either no later WRITE is invoked, or one is invoked but the writer fails before completing the pre-write phase), then from point τ' onward, (1) $\langle ts, v\rangle$ appears at least $t+1$ times in $w[*]$, and is therefore safe; and (2) there are at least $2t+1$ responses in $w[*]$ (from the correct registers) with either $\langle ts, v\rangle$ or with a timestamp smaller than ts. Therefore, every value-timestamp pair c such that $c.ts > ts \vee (c.ts = ts \wedge c.val \neq v)$ is $\mathsf{invalid}$. Thus, at the end of the next iteration of line 8, $\langle ts, v\rangle \in C$. Hence, the termination condition in line 9 is satisfied and READ returns, a contradiction.

Second, suppose that an incomplete WRITE invocation $\mathrm{W}' = $ WRITE$(\langle ts', v'\rangle)$ occurs, and the pre-write phase of W' completes. Then after τ', $\langle ts', v'\rangle$ appears at least $t+1$ times in $pw[*]$, and is therefore safe. Moreover, since no value-timestamp pair c such that $c.ts > ts' \vee (c.ts = ts' \wedge c.val \neq v')$ is ever written, there are at least $2t+1$ responses with either $\langle ts', v'\rangle$ or a smaller timestamp than ts'. Thus, $\mathsf{highestValid}(\langle ts', v'\rangle)$ holds, and after the next iteration of line 8, $\langle ts', v'\rangle \in C$. Hence, the termination condition in line 9 is satisfied and, again, READ returns. A contradiction. □

We have proven the following:

Theorem 4.3 (FW-Terminating Register Emulation). *The algorithm consisting of the* WRITE *emulation in Figure 3 and the* READ *emulation in Figure 4 implements a t-tolerant SWMR FW-*

terminating regular register using $n > 3t$ SWMR FW-terminating regular registers up to t of which can suffer NR-arbitrary failures.

4.3 Efficiency

Note that although we have proven that the READ emulation terminates in a finite number of rounds whenever there is a finite number of WRITEs, there is no upper bound on the number of rounds it can take in asynchronous executions. The proof of Lemma 4.2 implies that in executions with a finite number of WRITEs, READ always terminates once it gathers responses that follow the last WRITE from all correct registers. In particular, in an execution without any WRITE invocations, READ terminates once all correct registers respond to its first round of invocations. However, since a new read round is initiated whenever $n - t$ responses for the previous round arrive, faulty registers responding much faster than slow correct ones may cause the READ emulation to invoke an unbounded number of rounds of operations on the fast base registers before all the correct registers respond to the first round of invocations.

Nevertheless, we informally observe that in a *synchronous* execution without a concurrent WRITE, READ always takes a single round. In order to formally make such a claim, one needs to consider an eventually synchronous (or timed-asynchronous) model [DLS88, CF99]. In such models, there is an expected response time Δ, such that in periods when the system is synchronous (stable), responses from correct registers always arrive within Δ time. In order to exploit eventual synchrony, the reader should wait for responses in each round at least Δ time before moving to the next round, even after $n - t$ responses are gathered. That is, the pseudo-code in Figure 4 should be changed as follows: before line 5, a timer should be set to expire Δ time later, and the waiting condition in line 7 should be changed as follows:

7: **until** $|\{i \ : \ \neg enabled[i] \wedge \neg pending[i]\}| \geq n - t \wedge timer\ expired$;

In synchronous (stable) executions, responses from correct registers always arrive by this timeout, and READ thus invokes a single round of operations on base registers whenever there is no overlapping WRITE. Moreover, in synchronous executions with overlapping WRITEs, once all the lower level write operations invoked by these WRITEs on correct registers are complete, it takes at most one round for all the pending lower level read operations to return, and then one more round of read operations for READ to complete. Examining the number of rounds invoked in synchronous executions is significant, since synchronous executions are common in practice, and therefore achieving good performance in such executions is important.

At the same time, it is important to note that eventual synchrony is *not* a condition for liveness: even in asynchronous executions, the algorithm satisfies FW-termination, as proven above.

5 t-Tolerant Wait-free Safe Register Emulation

We now proceed to construct a t-tolerant SWMR wait-free safe register out of $n \geq 3t + 1$ wait-free regular base registers. Note that the register constructed in this section only provides safe semantics, and is not regular. In order to obtain a regular register, as required, e.g., for consensus algorithms, standard constructions from the literature can be used (see e.g., [Lam86] and a survey in [HV02]). The register emulation is presented in Section 5.1. Its correctness is proven in Section 5.2, and its round complexity is discussed in Section 5.3.

5.1 Register Emulation

The t-tolerant wait-free safe register's implementation uses the same base registers as the FW-terminating register implementation (see Figure 2), except that it requires wait-free base registers rather than FW-terminating ones. The WRITE operation is emulated exactly the same way as that of the FW-terminating register (see Figure 3). The READ implementation is presented in Figure 5.

The partial function $ReadW$ ($ReadPW$) maps every read timestamp-value pair to all the registers from which this pair was read from the w (resp. pw) field in the current invocation of READ. $prevReadW$ holds a copy of $ReadW$ from the end of the previous read round (line 9). The macro Responded returns the set of registers that responded to read requests thus far.

The algorithm first invokes a round of read operations on all base registers, and awaits $n - t$ responses (lines 4–6). It then invokes additional rounds of base register operations (lines 8–14), until it finds a value that it can safely return, as will be explained shortly.

The set C holds candidate return values. After the first round, C consists of values c that were read from base registers' w fields such that at most $2t$ registers responded without c (line 7). Any value that is not included in C is missing from $2t+1$ registers' w fields, which means that it was either not completely written before the READ began (its WRITE could have begun but could not have completed), or was already over-written (the write phase of a subsequent WRITE has begun). In subsequent rounds, if for some candidate $c \in C$, there are $2t+1$ registers that responded but never with c in their w field, then c is removed from C (line 14) for the same reason.

We now explain the termination condition of the while loop (line 8). First, if $C = \emptyset$, then it must be the case that a WRITE invocation overlaps the READ, since there is no value that appears in the w fields of more than t registers throughout the READ, and every WRITE always writes to $t+1$ correct registers. In this case, every return value is safe. The algorithm then breaks from the loop and returns an arbitrary value, v_0 (line 17).

If C is not empty, then the leading candidate to return is the one associated with the highest timestamp, because if several values were completely written before READ began, then the latest one should be returned. This is captured by the predicate $\mathsf{highCand}(c)$.

Let c be a candidate for which $\mathsf{highCand}(c)$ is true. It is safe to return c if $t+1$ registers have responded either with c or with later values (in either field, pw or w). This is because at least one of these registers must be correct. If that register responds with c, then c was indeed written, and is a correct return value. Otherwise, the correct register returns a value, c', which was written later than c. Since $\mathsf{highCand}(c)$ is true, c' is not in C, implying that its WRITE did not complete before the READ. In other words, a WRITE operation (of c') is occurring concurrently with the READ, in which case a safe register is allowed to return any value. This condition is captured by the predicate safe. Once there is a candidate $c \in C$ such that $\mathsf{highCand}(c) \wedge \mathsf{safe}(c)$, READ breaks from the while loop (line 8) and returns $c.val$ (line 16).

Unlike the FW-terminating implementation, the number of read rounds invoked by the algorithm is bounded, even in the presence of an unbounded number of concurrent WRITEs. We now explain how we limit the number of rounds executed by the algorithm. In order for a return value to become safe, the algorithm needs to gather responses from the $t+1$ correct registers at which the

Local variables:

Boolean arrays $enabled[n],\ pending[n],\ old[n]$, initially false for all $1 \le i \le n$;
Arrays $pw[n], w[n]$ with elements in $TSVals$;
$ReadW,\ ReadPW, prevReadW$, partial functions from $TSvals$ to $\mathcal{P}(\{1 \ldots n\})$, initially $\emptyset$;
$C \subseteq TSVals$, initially $\emptyset$ /* *Candidate values to return* */

Predicate and macro definitions:

$\mathsf{Responded} \triangleq \{i \ :\ \exists\langle w, i\rangle \in ReadW\}$
$\mathsf{highCand}(\langle ts, v\rangle) \triangleq \langle ts, v\rangle \in C \wedge (ts = max\{ts' \ :\ \langle ts', v'\rangle \in C\})$
$\mathsf{safe}(c) \triangleq |ReadW(c) \cup ReadPW(c) \cup \bigcup_{c'.ts > c.ts}(ReadW(c') \cup ReadPW(c'))| \ge t+1$

READ():

for $1 \le i \le n$, **if**$(pending[i])$ then $old[i] \leftarrow$ true;
$ReadPW, ReadW \leftarrow \emptyset$;
/* *Round 1* */
for $1 \le i \le n,\ enabled[i] \leftarrow$ true;
repeat
    CHECK;
until $|\{i \ :\ \neg enabled[i] \wedge \neg pending[i]\}| \ge n - t$;
$C \leftarrow \{w[i] \ :\ |\mathsf{Responded} \setminus ReadW(w[i])| < 2t+1\}$;
/* *Rounds 2 ...* */
while $(C \ne \emptyset \wedge (\neg\exists c \in C : \mathsf{highCand}(c) \wedge \mathsf{safe}(c)))$ **do**
    $prevReadW \leftarrow ReadW$;
    for $1 \le i \le n,\ enabled[i] \leftarrow$ true;
    repeat
        CHECK;
    until $|\{i \ :\ \neg enabled[i] \wedge \neg pending[i]\}| \ge n - t \ \wedge$
        $\forall c \in C :\ (\mathsf{safe}(c) \vee |\mathsf{Responded} \setminus prevReadW(c)| \ge n - t)$;
    $C \leftarrow \{c \in C \ :\ |\mathsf{Responded} \setminus ReadW(c)| < 2t+1\}$;
if $(C \ne \emptyset)$ **then**
    return $c.val : \mathsf{highCand}(c) \wedge \mathsf{safe}(c)$;
return v_0;

CHECK:

for $1 \le i \le n$
    if $(enabled[i] \wedge \neg pending[i])$ **then**
        $\langle enabled[i], pending[i]\rangle \leftarrow \langle$false,true$\rangle$;
        INVOKE $\langle pw[i], w[i]\rangle \leftarrow$ read(x_i);
    if $(x_i$ RESPONDED$)$ **then**
        if $(\neg old[i])$ **then**
            $ReadPW(pw[i]) \leftarrow ReadPW(pw[i]) \cup \{i\}$;
            $ReadW(w[i]) \leftarrow ReadW(w[i]) \cup \{i\}$;
        $pending[i] \leftarrow$ false;
        $old[i] \leftarrow$ false;

Figure 5: t-tolerant wait-free safe register READ emulation.

value was written. Although these registers eventually respond to every read round, their responses may be arbitrarily slow. As explained in Section 4.3 above, if exactly $n - t$ responses are awaited in each round, then in asynchronous executions, the faulty registers may respond much faster than t slow correct ones, causing the algorithm to initiate an unbounded number of rounds before the required $t + 1$ correct ones respond. The key to limiting the number of rounds the algorithm takes is therefore *waiting for more than* $n - t$ *responses*. Of course, this must be done with care, because up to t faulty registers may never respond.

In addition to waiting for $n - t$ responses, the repeat loop in lines 11–13 continues to await responses until for every candidate c in C, either c becomes safe or there are $n - t$ responses from registers that did not return c in previous rounds. This does not violate progress, because if c was read from a correct register in previous rounds, it will eventually become safe, and otherwise, there are $n - t$ correct registers that did not respond with c in previous rounds. (This is formally proven in Lemma 5.2 below). This mechanism ensures that while there are candidates that are not safe, every read round gathers responses from at least one additional register, and thus, after at most $t + 1$ rounds, responses from all the registers are gathered, and every candidate value either becomes safe or is removed from C (see Lemma 5.3).

5.2 Correctness

We begin by proving safety.

Lemma 5.1 (Safety). *The register whose* WRITE *emulation appears in Figure 3 and whose* READ *emulation appears in Figure 5 is safe.*

Proof. We prove that all the traces of the algorithm are safe. Let R be a READ invocation. If some WRITE operation overlaps R, then R is allowed to return any value, and the lemma vacuously holds. We therefore assume that no WRITE overlaps R. Let $\langle ts, v \rangle$ be the value-timestamp pair written in the latest WRITE invocation that returns before R is invoked, or $\langle ts_0, v_0 \rangle$ if no WRITE completes before R. We need to show that R does not return a value other than v.

If $ts = ts_0$, then $\langle ts_0, v_0 \rangle$ appears in both the w and pw fields of all correct registers throughout the duration of R. Otherwise, by the WRITE implementation, $\langle ts, v \rangle$ is both pre-written and written to at least $n - t$ base registers before WRITE returns. Therefore, both of the following conditions hold throughout the duration of R: (1) there are at least $t+1$ correct registers that have $\langle ts, v\rangle$ in their w and pw fields; and (2) there are at most $2t$ base registers (t of which were not updated by the last preceding write and the remaining t of which are Byzantine) that can respond without $\langle ts, v\rangle$. By the READ code, responses from at least $n - t$ registers are awaited in lines 6 and 13. Therefore, by (1), $\langle ts, v\rangle$ is included in C in line 7. Moreover, by (2), $\langle ts, v\rangle$ is never excluded from C in line 14. Hence, $C \neq \emptyset$ from the first time line 7 is executed onward, and the algorithm does not return in line 17. Finally, observe that no $\langle ts', v'\rangle \neq \langle ts, v\rangle$ can be $\mathsf{highCand}$ and safe, because no correct register returns a value with $ts' > ts$ or $ts' = ts \land v' \neq v$. Hence, no value other than v is returned in line 16. □

We now turn to discuss liveness. We first show that every read round eventually terminates. That is, the algorithm is never stuck forever in a repeat-until loop (lines 4–6 and lines 11–13).

Lemma 5.2 (Non-Blocking). *The* READ *emulation never remains indefinitely in the repeat-until loops in lines 4–6 and 11–13.*

Proof. First note that in each round, at least $n - t$ responses are awaited (lines 6 and 13). This is always eventually satisfied, since there are at least $n - t$ wait-free correct base registers. Therefore, the algorithm eventually exits from the loop in lines 4–6. We next show that the extra waiting condition in line 13 does not violate liveness.

Consider an iteration of the while loop (lines 8–14), and the execution of the repeat-until loop in lines 11–13 during that iteration. Since the while loop is still being executed, $C \neq \emptyset$. Consider a candidate $c \in C$. Since c is a candidate, $prevReadW(c) \neq \emptyset$ after line 9. There are two cases for c:

1. *At least one register in prevReadW(c) is correct.* In this case, the pre-write phase of WRITE(c) must have completed on at least $t+1$ correct registers before the current round is initiated (because c has been read from the w field of a correct register in round 1 before line 7 is executed). Each correct register eventually responds in the current round with either c or a higher timestamped value, and $\mathsf{safe}(c)$ eventually holds.

2. *All the registers in prevReadW(c) are faulty.* In this case, there are at least $n - t$ correct registers that did not previously respond with c. Since these registers eventually respond, and since none of them is included in $prevReadW(c)$ during the loop in lines 11–13, eventually, $|\mathsf{Responded} \setminus prevReadW(c)| \geq n - t$ becomes true.

Line 13 waits for one of the above to hold, and therefore eventually terminates. □

To see why READ invokes at most $t + 1$ rounds, observe that once every candidate in C is safe, the termination condition of the loop in line 8 is satisfied. If C is empty, we are done. Otherwise, consider $c \in C$. Lines 11–13 are executed until either c becomes safe or there are $n - t$ responses from registers that did not previously respond with c. Thus, if c does not become safe in line 13, then either $ReadW(c)$ grows, or c is removed from C in line 14, because $n - t \geq 2t + 1$ registers respond without c. After j read rounds, for every $c \in C$ that is not safe at the end of line 14, $ReadW(c)$ has grown j times, and therefore includes at least j elements. Once $ReadW(c)$ includes $t + 1$ elements, c is safe. We conclude that the algorithm is wait-free:

Lemma 5.3 (Wait Freedom). *The safe register emulation satisfies wait freedom.*

Proof. Every WRITE operation invoked by a correct process returns, as argued in Lemma 4.2. Consider a READ operation R. By Lemma 5.2, R exits from every repeat-until loop. Moreover, as argued above, at most $t+1$ rounds of READ operations are invoked before the termination condition of the loop in line 8 is satisfied, at which point READ completes. □

We have proven the following:

Theorem 5.4 (t-Tolerant Wait-Free Register Emulation). *The algorithm consisting of the* WRITE *emulation in Figure 3 and the* READ *emulation in Figure 5 implements a t-tolerant wait-free SWMR safe register from $n > 3t$ wait-free SWMR regular registers, up to t of which can be NR-arbitrary faulty.*

5.3 Round Complexity

The algorithm's early-stopping property is more subtle, and is proven in the following lemma.

Lemma 5.5 (Early-Stopping). *In every execution in which at most f registers exhibit Byzantine behavior, (i.e., return incorrect values), the* READ *emulation invokes at most* $\min(t+1, f+2)$ *rounds of read operations on base registers.*

Proof. We will show that if READ initiates a jth read round, then $j \leq t+1$ and also $j \leq f+2$.

For $j > 1$, consider the initiation of the jth read round, occurring in lines 11–13 (during the $(j-1)$th iteration in the loop of lines 8–14). Since the loop is executed, $C \neq \emptyset$. Let c be the highest timestamped candidate in C, i.e., $\mathsf{highCand}(c)$ holds. Then c is not safe at the beginning of this iteration (otherwise, the termination condition in line 8 holds). Since c was not removed from C in line 14 during previous iterations, it was returned in every round before j, and each time by a new register (i.e., $|ReadW(c)|$ has increased $j-1$ times). Since c is not safe, we know that $|ReadW(c)| < t+1$, and therefore $j-1 < t+1$, that is $j \leq t+1$.

If c was never returned by a correct register, then $ReadW(c)$ includes at most f elements, and $j \leq f+1$. Otherwise, let $k < j$ be the first round during which c is read from the w field of a correct register. Then c was sent by at least $k-1$ Byzantine faulty registers before round k. Consider the set S of registers that respond to rounds $k+1 \ldots j-1$. In round $k+1$, at least $2t+1$ registers that did not previously respond with c are read (and are therefore included in S). Moreover, since $ReadW(c)$ continues to increase in each round, S includes at least $2t+j-k-1$ registers excluding the $k-1$ Byzantine registers that sent c before round k. Since $k-1$ Byzantine faulty registers send c before round k, at most $f-k+1$ members of S are Byzantine faulty, and S includes at least $2t+j-k-1-(f-k+1) = 2t+j-f-2$ correct registers.

Finally, since the pre-write phase of WRITE(c) is complete before round $k+1$ is initiated, at most t correct registers respond to rounds $k+1 \ldots j-1$ with values older than c in their pw field. Therefore, if S would include $2t+1$ correct registers, then at least $t+1$ of them would have either c or a higher value in their pw field, and c would be safe. But c is not safe. We get that $2t+j-f-2 \leq 2t$, that is, $j \leq f+2$. □

The algorithm's round complexity is optimal for optimal-resilience algorithms in which read operations do not modify the base objects. For such algorithms, $\min(t+1, f+2)$ rounds is a tight lower bound on the number of rounds for read, by the lemma above and Corollary 7.7 below. Note also that only Byzantine failures cause READ to take more rounds; benign (i.e., crash) failures do not slow the algorithm down. Our next lemma shows that the algorithm takes the optimal number of rounds for *any* resilience threshold, matching the lower bound of Theorem 7.6.

Lemma 5.6 (Optimality For Arbitrary Resilience). *Let $n = 3t + k$, $k > 0$. The* READ *emulation invokes at most $\lfloor t/k \rfloor + 1$ rounds of read operations on base registers.*

Proof. First, consider the case that $k \geq t + 1$, i.e., $n \geq 4t + 1$. Since at least $3t + 1$ registers are read in the first round, every read value either appears in at least $t + 1$ responses, in which case it is safe, or is missing from at least $2t + 1$ responses, in which case it is not in C. Therefore, the algorithm never enters the while loop of lines 8–14, and only one read round is invoked.

Next, assume that $k < t + 1$. If the while loop in lines 8–14 is entered, then there is at least one candidate $c \in C$ that is not safe. Since c is a candidate, it is missing from at most $2t$ of the responses gathered in earlier rounds. Since at least $n - t = 2t + k$ responses are awaited in each round, c appears in at least k responses of previous rounds. Therefore, if c does not become safe in this iteration of the loop, at line 14, at least k responses are awaited from objects that did not previously respond in any read round. Therefore, if a $(j + 1)$th read round is invoked, then at least $2t + jk$ objects have responded in previous rounds. Since once $3t + 1$ objects respond every read value is either safe or removed from C, we get that $2t + jk < 3t + 1$, i.e., $j < (t + 1)/k$. Since j, t, and k are integers, $j < (t + 1)/k \leq \lfloor t/k \rfloor + (k - 1)/k + 1/k$. Hence, $j < \lfloor t/k \rfloor + 1$ as needed. □

Our next lemma shows that in invocations of READ that do not overlap any WRITE invocation, READ invokes at most $f + 1$ rounds. This is also a tight lower bound, by Theorem 7.8. In particular, in the common case that none of the base registers returns faulty values and no overlapping WRITE occurs, READ invokes a single round of read operations.

Lemma 5.7 (Early-Stopping Without Concurrent Writes). *In every execution in which f registers exhibit Byzantine behavior and no* WRITE *operations overlap the* READ*, the* READ *emulation invokes at most $f + 1$ rounds of read operations on base registers.*

Proof. Since no WRITE overlaps the READ, at least $t + 1$ correct registers return the latest written value in the every read round, and this value is safe throughout the execution of the loop in lines

8–14. Whenever an iteration of the loop begins, this value is not the highest timestamped candidate (otherwise the loop's termination condition is satisfied), i.e., there is higher timestamped candidate c', which was never written. For c' to remain a candidate after read round k, it has to be read in every round $1 \dots k$. Since c' was never written, at most f faulty registers return c', and hence after $f + 1$ read rounds c' is removed from C. $\square$

Finally, we observe that in *synchronous* executions, READ always terminates in two rounds. In order to take advantage of the system synchrony, the reader's wait statements should be augmented with timeouts as explained in Section 4.3.

6 Wait-free Consensus with FW-Terminating Regular Registers and Ω

In this section, we show that FW-terminating registers can be used, along with a leader oracle, Ω, to solve consensus in shared memory. In a *consensus* problem, each process has an input and may decide on an output, so that the following conditions are satisfied: (1) *wait freedom*: each correct process decides; (2) *agreement*: every two correct processes that decide decide on the same value; and (3) *validity*: every decision is the input of some process. A shared memory consensus object has a single invocation, *decide(v)*, which takes the invoking process' input value as a parameter and returns the decision value. We assume that each process invokes *decide* at most once.

We present a shared memory consensus algorithm based on those of [LH94, GL03], and prove that it works correctly with FW-terminating regular registers. Since the algorithm closely resembles ones in the literature, the contribution of this section is in observing that it works correctly with FW-terminating registers.

The algorithm is presented in Figure 6. It solves consensus among m processes $P_1, \dots, P_m$ using m FW-terminating SWMR regular registers $x_1, \dots, x_m$, where x_i is writable by P_i and readable by all processes. It employs a distributed leader oracle $\mathcal{L}$, which is a failure detector of class Ω [CHT96], the weakest for consensus [LH94, DFG02, CHT96]. Each process P_i accesses $\mathcal{L}$ via its local module $\mathcal{L}_i$, whose output at any given time is the index of the process that is currently considered to be

Types: $X = \mathbb{N} \times Vals \times \{\perp, pc, c\}$, with selectors bal, val, $stat$;

Shared FW-terminating regular registers $x_i \in X$, $1 \leq i \leq m$, initially $\langle 0, \perp, \perp \rangle$;
Each x_i is writable by P_i and readable by all processes.

Algorithm for process i:

Local variables: $val \in Vals$, $bal \in \mathbb{N}$, $\ell \in \mathbb{N}$, $a_j \in X$ for $1 \leq j \leq m$;

DECIDE$_i$(inp):
1: $bal \leftarrow i$;
2: $val \leftarrow inp$;
3: **while** (true) **do**
4:     $\ell \leftarrow \mathcal{L}_i$;
5:     **if** ($\ell = i$) **then**; /* Leader case */
6:         write($x_i, \langle bal, \perp, \perp \rangle$);
7:         $a_j \leftarrow$ read(x_j), for each j, $1 \leq j \leq m$;
8:         **if** ($\max\{a_j.bal : 1 \leq j \leq m\} \leq bal$) **then**
9:             **if** ($\exists j : a_j.val \neq \perp$) **then**
10:                 $val \leftarrow a_k.val$: $1 \leq k \leq m \wedge a_k.bal = \max\{a_j.bal : 1 \leq j \leq m \wedge a_j.val \neq \perp\}$;
11:             write($x_i, \langle bal, val, pc \rangle$);
12:             $a_j \leftarrow$ read(x_j), for each j, $1 \leq j \leq m$;
13:             **if** ($\max\{a_j.bal : 1 \leq j \leq m\} \leq bal$) **then**
14:                 write($x_i, \langle bal, val, c \rangle$);
15:                 **return** val;
16:         $bal \leftarrow bal + m$;
17:     **else** /* Non-leader case */
18:         $a_\ell \leftarrow$ read(x_ℓ);
19:         **if** ($a_\ell.stat = c$) **then**
20:             **return** $a_\ell.val$;

Figure 6: Wait-free consensus with FW-terminating regular registers and a leader oracle.

trusted by P_i. A failure detector of class Ω guarantees that there is a time after which a single correct process is permanently trusted by all correct processes.

The algorithm is leader-based. A process ℓ that trusts itself (i.e., believes itself to be the leader), decides upon a value and writes it in x_l with the tag c (line 14), whereas other processes continuously read x_ℓ until they find a decision value there (lines 18–20). Before P_ℓ decides, it *proposes* a decision value (line 11), by writing it in x_i with the tag pc. Each proposed value is associated with a unique ballot bal. We say that a process ℓ *proposes* (resp. *decides*) value v at ballot b if ℓ completes line 11 (resp. 14) with $val = v$ and $bal = b$. To propose a value, P_ℓ chooses the value previously proposed with the highest ballot number, or its own initial value if there is none (lines 7–10). The leader then reads the other processes' registers in order to check whether a concurrent leader has written a higher value. Recall that Ω only eventually guarantees that the leader is unique; initially, multiple concurrent leaders may exist. The leader's proposal succeeds if no higher ballot is read (lines 12–13).

The key to guaranteeing wait freedom despite the use of FW-terminating registers is the fact that once a unique leader ℓ emerges (as guaranteed by Ω), P_ℓ is the only process that invokes write on *any* register. Moreover, the ballot numbers stop increasing, and therefore P_ℓ invokes a finite number of writes. Therefore, by FW-termination, all the read operations terminate. We now formally prove that the algorithm satisfies wait freedom.

Lemma 6.1 (Wait Freedom). *In any fair execution, all non-faulty processes eventually decide.*

Proof. Let α be a fair execution of the algorithm. Since $\mathcal{L} \in \Omega$, each correct process i permanently trusts the same correct process ℓ after some finite prefix α_1 of α. We first show that P_ℓ decides in α. Assume by contradiction that this never happens. By the code, after α_1, each process $i \neq \ell$ writes x_i at most twice (in lines 11 and 14), and then proceeds to read x_ℓ (line 18) in a loop without invoking any other shared memory operations. Therefore, by FW-termination, process ℓ eventually completes all the read operations it invokes on all registers. As long as P_ℓ does not decide, it repeatedly executes lines 6–16. Since bal_ℓ is increased every time P_ℓ executes line 16, and no other processes j increases bal_j after α_1, bal_ℓ eventually becomes the highest ballot ever written. Once this happens, the **if** statements in lines 8 and 13 are evaluated to true, and process ℓ writes a decision value (with $stat = c$) and returns. A contradiction. We conclude that there exists a prefix α_2 of α after which P_ℓ no longer participates in consensus.

Next, we show that all correct processes decide in α. Assume the contrary. Let $\alpha_3 = \max(\alpha_1, \alpha_2)$. Consider a process $i \neq \ell$ which is still undecided after α_3. Since after α_3, P_i never trusts itself as a leader, P_i is either blocked in one of the read operations in lines 7, 12 or 18, or loops in lines 18–19. Since after α_1 all processes $j \neq \ell$ write their registers at most twice, by FW-termination, after α_3, P_i can only be blocked in a read operation from x_ℓ. However, P_ℓ never writes x_ℓ after α_3. Thus, by

FW-termination, there exists a prefix $\alpha_4 \geq \alpha_3$ of α after which P_i is looping in lines 18–19 without being blocked in the read in line 18. Since after α_4, P_ℓ has already completed write$(x_\ell, \langle *, *, c\rangle)_\ell$, by regularity, the next invocation of read(x_ℓ) by P_i will respond with a_ℓ such that $a_\ell.stat = c$. Hence, P_i decides. A contradiction. □

We next prove the algorithm's safety properties, namely agreement and validity.

Lemma 6.2 (Agreement). *All decision values are identical.*

Proof. Let b_1 be the lowest ballot at which some process decides, and assume that process i decides v_1 in this ballot. Suppose that a process k proposes a value v_2 at a ballot $b_2 \geq b_1$. We show that $v_2 = v_1$, which implies agreement, since a value v_2 decided in a ballot b_2 is first proposed in that same ballot. The proof is by induction on ballot numbers $b \geq b_1$. The base case $b = b_1$ is trivially true, since ballot numbers are unique.

Inductive step: Suppose that the result holds for all b, $b_1 \leq b < b_2$, and consider a process k proposing v_2 in ballot b_2. Since P_i decides v_1 at b_1, it must have proposed v_1 at b_1. Moreover, since P_i decided in line 14, the condition in line 13 evaluated to true, which means that for all register values a_j that P_i read in line 12, $a_j.bal \leq b_1$.

Before proposing any value at ballot b_2, process k must perform write$(x_k, \langle b_2, \perp, \perp\rangle)_k$ in line 6. This write must return after the read(x_k) by i in line 12 has been invoked, because otherwise, by regularity of x_k and because ballots are monotonically increasing at each process, read(x_k) by i must respond with $\langle b', \perp, \perp\rangle$ such that $b' \geq b_2 > b_1$ contradicting the fact that $a_j.bal \leq b_1$ in line 13. Thus, the read $R =$ read(x_i) by k in line 7 is invoked after write$(x_i, \langle b_1, v_1, pc\rangle)_i$ is complete. Since i returns after deciding, it does not overwrite x_i after ballot b_1. Hence, by regularity of x_i, R returns $\langle b_1, v_1, *\rangle$.

Consequently, when k completes line 7, $a_i = \langle b_1, v_1, *\rangle$ and therefore, both of the following hold: (1) the test in line 9 is true; and (2) the value v' chosen in line 10 was written with a ballot $b' \geq b_1$. Furthermore, since the condition in line 8 is true, $b' \leq b_2$. By line 6, $a_k.val = \perp$. Thus, we get that v' must have been written at ballot b', such that $b_1 \leq b' < b_2$. Finally, since for any value $v \neq \perp$ such that for some j, $x_j.val = v$, v must have been either proposed or decided by j, and because the value decided at any ballot must be equal to the value proposed at this ballot, v' must have been proposed with ballot b', $b_1 \leq b' < b_2$. By the induction hypothesis, $v' = v_1$. Therefore, k proposes v_1 at b_2. □

Lemma 6.3 (Validity). *Every decision value is the initial value of some process.*

Proof. Immediately follows from the fact that every proposed value is either the proposer's initial value or a previously proposed value. □

We have proven the following:

Theorem 6.4 (Wait-free Consensus with FW-Terminating Regular Registers). *The pseudo-code in Figure 6 solves m-process wait-free consensus using m SWMR FW-terminating regular registers in an asynchronous shared memory system augmented with a failure detector of class Ω.*

Like any Ω-based asynchronous consensus algorithm, the algorithm's running time is unbounded. This is inevitable, since the leader oracle's output may be arbitrary for an unbounded length of time, and while it is, consensus is not solvable. Thus, a discussion of such algorithms' worst-case performance is meaningless. One way to reason about the efficiency of algorithms in this model is to focus on runs in which the oracle's output is accurate from the outset [KR01], that is, on executions throughout which the oracles at all processes output the same correct process ℓ. We observe that in such executions, only ℓ writes to the register, and it decides after one iteration of the loop in lines 3–20, which involves writing three times and reading twice. (The last write phase is not necessary, but we have added it in order to simplify the algorithm and its proof.) The remaining processes decide after reading ℓ's last written value.

We can now combine the wait-free consensus algorithm with our register constructions from the previous sections in order to obtain Byzantine Disk Paxos. Byzantine Disk Paxos uses the notion of a *disk*, which is a collection of base objects that share their fates. That is, all the base objects stored on a given disk D are faulty if and only if the disk D is faulty; each base object pertains to a single disk.

Given a system with $3t+1$ disks, t of which can be arbitrarily corrupted or non-responsive, we use the construction in Section 4 in order to emulate m t-tolerant FW-terminating registers from $n = 3t+1$ base registers, each stored on a different disk, and use them to solve t-tolerant wait-free consensus, as illustrated in Figure 7. Thus, we have the following corollary:

Corollary 6.5 (Byzantine Disk Paxos). *There is a solution for t-tolerant wait-free consensus using $3t+1$ disks, t of which can be arbitrarily corrupted or non-responsive, and a failure detector of class Ω.*

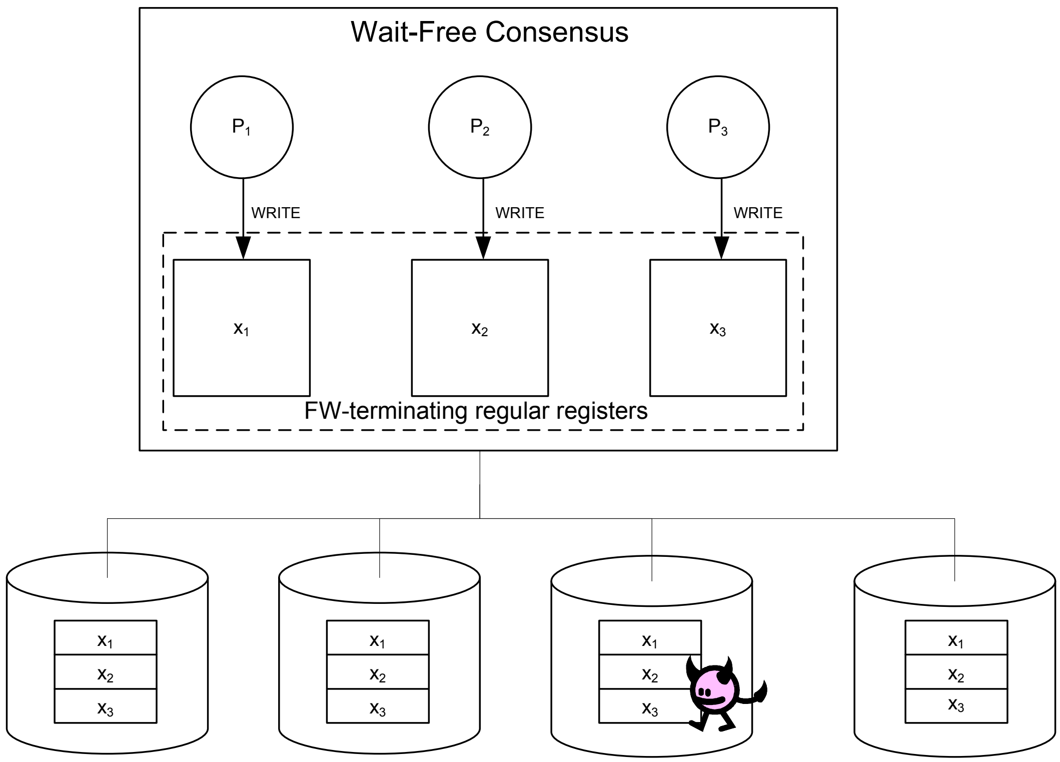


Figure 7: Byzantine Disk Paxos example: First, four disks of which one can be corrupt are used to emulate three 1-tolerant SWMR FW-terminating registers. The emulated registers are then used to solve 1-tolerant wait-free consensus among three processes.

7 Lower Bounds on Register Emulations

We now prove lower bounds on memory emulations of reliable objects from ones that can suffer NR-Arbitrary faults. Obviously, at least $3t+1$ base objects are required in order to emulate a reliable one in this model (see [MAD02]). Our lower bounds focus on emulations that use less than $4t+1$ base objects, since using $4t+1$ base objects, one can emulate both READ and WRITE operations in a single round of base object invocations. In Section 7.1 we prove a lower bound of two rounds for WRITE emulations, and in Section 7.2 we prove lower bounds on the number of rounds for READ emulations in which the READ emulation does not write to the base objects.

To strengthen our lower bounds, we prove them for emulations of the weakest meaningful register type: a SWSR safe register [Lam86] with a binary value domain. Without loss of generality, we assume that the emulated register's initial value is 0. We allow for atomic base objects of *any* type. We prove the lower bounds for emulations of FW-terminating registers. We note that our lower bounds apply to obstruction free registers as well, but since obstruction freedom is not the focus of our paper, we do not make formal claims regarding obstruction free emulations.

Since we are not seeking space lower bounds, we can assume a model in which all base objects have the same types and initial states. A concurrent system consisting of base objects $O_1, \ldots O_n$ of different types $T_1, \ldots, T_n$ can be emulated in this model by replacing each base object O_i with a tuple O'_i of type $T_1 \times \cdots \times T_n$, and initializing all base registers to the same initial value $s_0 = \langle s_0^1, \ldots, s_0^n \rangle$ where s_0^i is the initial state of O_i. We thus henceforth assume that all base objects are initialized to the same (arbitrary-type) value, s_0.

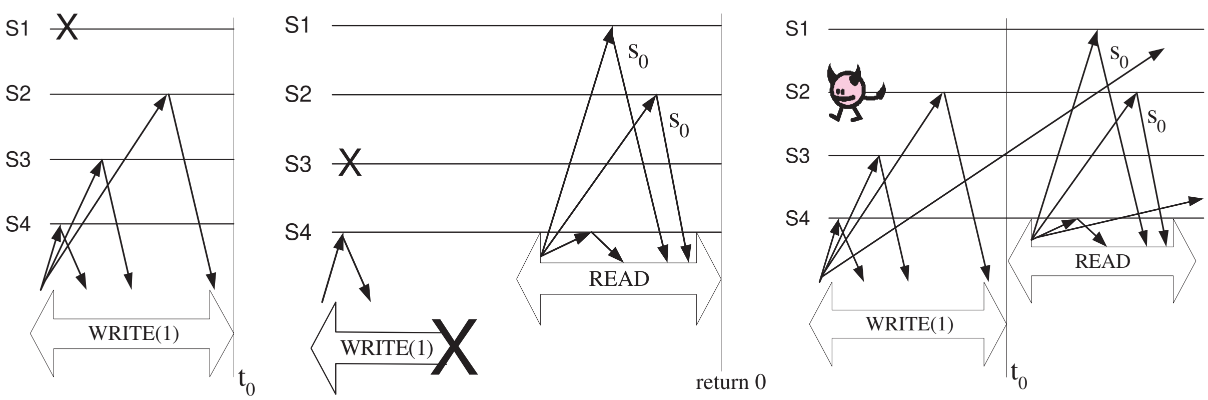
7.1 Lower Bound on WRITE Emulations

The following simple lemma shows that any algorithm implementing an FW-terminating SWSR safe register has executions where the WRITE implementation invokes at least one complete invocation request on some base object.

Lemma 7.1 (One Write Round). *Consider a concurrent system C implementing a t-tolerant FW-terminating SWSR safe register out of $n > 0$ base objects. Consider a fair execution α where a correct writer invokes* WRITE(1) *and no other operations are invoked. Then α consists of at least one complete invocation request to some correct base object.*

Proof. By FW-termination and fairness, WRITE(1) must terminate and return ack at some point τ in α. Assume by contradiction that α does not include any complete invocation request to a correct base object. Since the writer returns in α without seeing any responses from correct base objects, all invocation requests that were issued to the correct base objects (if any) were invoked in separate threads of control, which did not return. We construct an execution α', which starts with all the activity of α except that no invocation request events to correct objects occur in α' until point τ (that is, in α' the threads that handle correct object invocations in α are slowed down so that all the invocation requests issued to the correct objects are postponed). We then extend α' with a complete READ invocation. Note that by FW-termination READ must complete, because the reader is correct and no WRITE is in progress. We construct α' so that any faulty object that received an invocation request by WRITE(1), does not change its state, and responds to the READ the same as in an execution in which no WRITE ever occurs. Since α' is indistinguishable to the reader from an execution where no WRITE operations were ever invoked, the READ response must be 0. However, since WRITE(1) completes before READ is invoked, safety requires READ to return 1. A contradiction. □

We now prove our main lower bound on WRITE emulations:



(a) Execution α_1. (b) Execution α_2. (c) Execution α_3.

Figure 8: Illustrating the lower bound on WRITE emulations.

Theorem 7.2 (Write Lower Bound). *Let C be a concurrent system implementing a t-tolerant FW-terminating SWSR binary safe register out of $0 < n \leq 4t$ base objects. Then, for every $0 \leq f \leq t$, there exists an execution of C in which f base objects are faulty, and which includes a complete invocation of* WRITE(1) *and no other invocations, such that during* WRITE(1) *at least two* *invocation requests are completed on some base object.*

Proof. We partition the n base objects into four disjoint sets, S_1, S_2, S_3, and S_4, such that $|S_i| \leq t$, for each $i \geq 1$. Assume by contradiction that in all executions consisting of a complete WRITE(1) invocation and no other invocations, less than two invocations complete on each base object. By Lemma 7.1, some correct base objects are written by WRITE(1). Without loss of generality, assume that the first base objects to which WRITE(1) writes are those in S_4 (if it writes to fewer objects, then it writes to a subset of S_4). Let α_1 be an execution of C with a single WRITE(1) invocation and no READ invocations, in which all base objects in S_1 are crashed. By FW-termination, WRITE(1) completes by some point t_0 in α_1 (see Figure 8(a)).

Next, we construct an execution α_2 where all the objects in S_3 are initially crashed. Execution α_2 begins with a WRITE(1) invocation, the invocations it issues to objects in S_4, and their responses as they occur in α_1. No invocations on other objects occur. This is a valid execution of C since it represents the situation in which the writer fails after receiving responses from some objects in the set S_4. Moreover, since WRITE does not invoke more than one operation on any correct base object, the objects in S_4 are exactly in the same states as after the complete WRITE in α_1.

We then extend α_2 with a READ invocation and assume that the reader is correct. All the objects in S_1 and S_2 are in state s_0 when accessed by the READ. The objects in S_3 are crashed, and hence do not respond. Since the reader runs by itself, by FW-termination, READ completes in α_2. Since the reader sees at most t objects (those in S_4) in states different from s_0, α_2 is indistinguishable to the reader from an execution in which WRITE(1) is never invoked and all objects in states other than s_0 are faulty. Therefore, READ returns 0 in α_2, as illustrated in Figure 8(b).

We next construct an execution α_3 in which all the objects in S_2 are Byzantine faulty and the remaining objects are correct. Execution α_3 starts with all the activity of α_1 except that the invocation requests targeted to the objects in S_1 do not occur in α_3 until t_0 (i.e., in α_3, the threads that handle invocations on objects in S_1 are slowed down so that all the invocation requests issued in α_1 are postponed). Since until t_0, α_3 is indistinguishable to the writer from α_1, WRITE(1) also terminates in α_3 after completing at most one invocation request at each base object.

We then extend α_3 with the segment of α_2 that starts with the READ invocation request and ends with its corresponding response. Note that α_3 is a valid execution of C because (i) the objects in S_2 are Byzantine faulty, and are therefore allowed to respond as if their state is s_0 even after WRITE(1) terminates; (ii) no invocations by the writer occur at objects in S_1 in α_3, and therefore these objects' states are s_0 when READ occurs; (iii) the responses of objects in S_3 are delayed until after the READ returns; and (iv) the objects in S_4 are in the same states as in α_2. This scenario is depicted in Figure 8(c). By construction, α_3 is indistinguishable to the reader from α_2, and therefore, the READ must return 0 in α_3. But since in α_3, the READ follows WRITE(1) and does not overlap any WRITE, by safety, READ must return 1. A contradiction. $\square$

7.2 Lower Bound on READ Emulations

In this section we show a lower bound on a number of rounds of base object invocations required to emulate READ operations of a binary t-tolerant FW-terminating SWSR safe register. We consider a system with $n = 3t + k$ base objects, t of which can fail. The special case where $k = 1$ represents an optimal resilience algorithm.

Since our complexity metric is the number of rounds, we can assume that operations are invoked in rounds, and each round attempts to invoke operations on all base objects; if on some base object there is a pending invocation, then the new invocation awaits the completion of the pending one.

Our lower bound results only apply to algorithms in which the reader does not modify the base objects' states. The significance of this assumption is that the reader cannot communicate to the writer that a READ is in progress, and hence WRITE must behave the same way regardless of whether or not there is a READ in progress. We conjecture that even if readers can modify the base objects, it still holds that either the READ or the WRITE emulation must take $\min(t+1, f+2)$ rounds. We discuss this conjecture at the end of this section. We note that in all register emulations suggested thus far in this model, (e.g., [MR98, Baz00, JCT98, GWGR04]), readers do not modify the base objects, and therefore the lower bound is of interest regardless of whether our conjecture holds.

To derive the lower bound, we focus on a subset of executions in which at most one WRITE is invoked, and if invoked, it WRITEs 1. Given our assumption that the reader does not modify the base objects, we get that WRITE behaves the same way in all such executions.

Since there is a single process modifying the base objects, we can assume, without loss of generality, that the base objects are read/write registers. Moreover, as we are not seeking a space lower bound, we can assume a full information model in which the writer attempts to write the same value to all base objects in each round. This does not limit the generality, since if the writer intends to write values $v_1, \ldots, v_k$ to objects $o_1, \ldots, o_k$, resp., it can simply write the tuple $\langle v_1, \ldots, v_k \rangle$ to all base objects, and the reader can ignore the irrelevant elements of each tuple.

We further assume, without loss of generality, that WRITE does not return before $2t + k$ base objects respond to its last round of write invocations. Note that waiting for $2t + k$ responses does not violate liveness since at least $2t + k$ correct base objects are guaranteed to respond. This assumption implies that when WRITE(1) completes, there exist $t + k$ correct base objects whose states are equal to the last value written by WRITE. As noted above, this value is the same in all the executions we consider. We denote this state by s_1. As before, we denote the base objects' initial state as s_0.

We begin by proving the following simple lemma:

Lemma 7.3. *For any $0 \leq f \leq t$, there is a finite execution that includes a single complete* WRITE *operation, in which f objects fail, and at the end of which the states of t correct base objects are s_0.*

Proof. Consider an execution σ in which t (faulty) objects, $o_1, \ldots, o_t$ crash at the beginning of the execution, and WRITE(1) is invoked. By FW-termination, WRITE must return without hearing from $o_1, \ldots, o_t$. Let τ be the point in σ at which WRITE returns. We construct an execution σ' that until point τ looks to the writer exactly like σ, but in which $o_1, \ldots, o_t$ are correct, and the requests sent to these objects are delayed until after τ. The remaining $n - t$ objects (f of which are faulty) all abide by the protocol, and respond exactly as they do in σ. Since until τ, σ' is indistinguishable to the writer from σ, WRITE returns at τ, before the delayed requests reach $o_1, \ldots, o_t$. Hence, when WRITE returns, t correct objects' states are still s_0. □

We now prove lower bounds on the number of rounds in READ emulations. Our lower bounds will be derived from the following key lemma, which inductively constructs executions in which the READ emulation is forced to invoke more and more read rounds. The executions constructed in the lemma below are illustrated in Figure 9 for the special case that $k = 1$; the responses to write invocations are omitted, whereas for read invocations, only responses are shown. Incomplete invocations of any kind are not shown.

Lemma 7.4 (Read Lower Bound). *Assume that $k \leq t$. For $1 \leq i \leq t/k$, there exist three finite executions α_i, β_i, and γ_i, in each of which a READ emulation has issued $i+1$ rounds of read invocations, such that:*

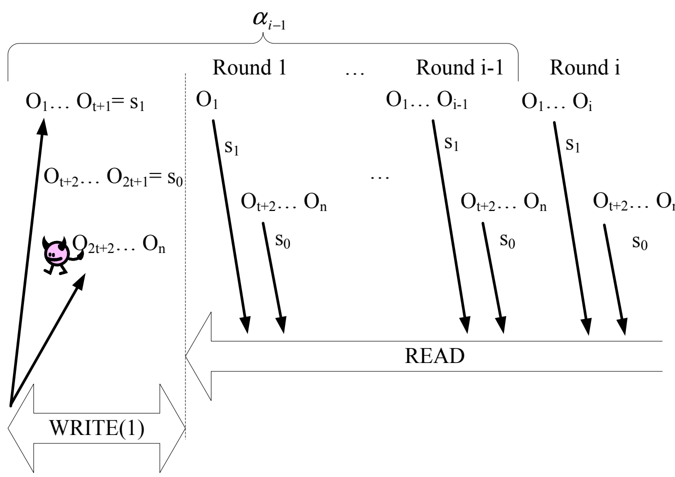
1. *In α_i WRITE(1) completes before READ is invoked; objects $o_1, \ldots, o_{t+k}$ are correct and their state is s_1 from a time before the beginning of the READ onward; objects $o_{t+k+1}, \ldots, o_{2t+k}$ are correct and their state is s_0 throughout α_i; and the remaining t objects are faulty.*

2. *In β_i no WRITE ever occurs; objects $o_1, \ldots, o_{ik}$ are faulty; and objects $o_{t+k+1}, \ldots, o_n$, are correct and their state is s_0 throughout β_i. All the invocations directed to objects $o_{ik+1}, \ldots, o_{t+k}$ are delayed, i.e., are not invoked in β_i. For these objects, we do not specify whether they are faulty or not since they do not participate in the execution yet. Note that at least k of them must be correct, but we do not specify which ones.*

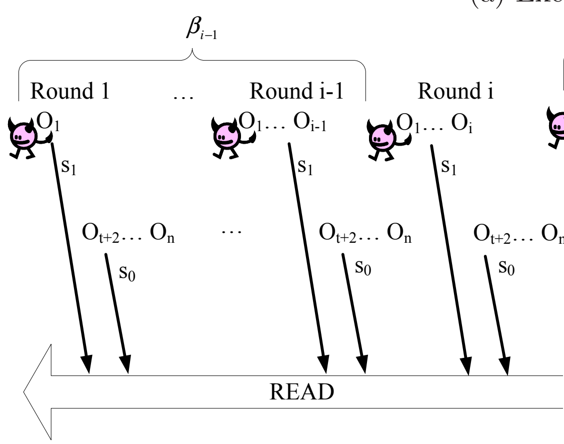
3. *In γ_1, all objects are correct. For $i > 1$, objects $o_1, \ldots, o_{(i-1)k}$ are faulty in γ_i and the remaining objects are correct.*

4. *The responses that the reader receives in all three executions in rounds $1, \ldots, i$ are as follows: In response to every round j read, $1 \leq j \leq i$, objects $o_{t+k+1}, \ldots, o_n$ return s_0 in round j, objects $o_1, \ldots, o_{jk}$ return s_1 in round j, and each object o_ℓ for $(j+1)k \leq \ell \leq ik$ returns s_1, but this response only arrives in round ℓ. No other objects respond.*

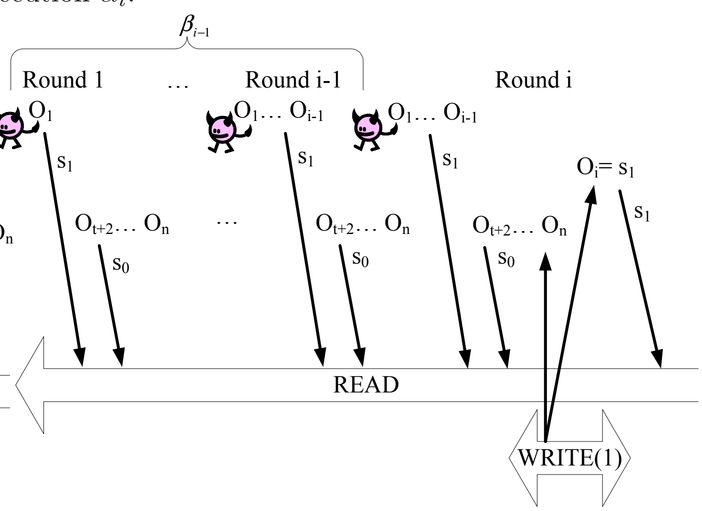
5. *The last events of each of these executions are round $i+1$ read invocations on all base registers that responded to all the previously issued read invocations.*



(a) Execution α_i.



(b) Execution β_i.



(c) Execution γ_i.

Figure 9: Illustrating the lower bound on READ emulations for the special case that $k = 1$.

Proof. **Base case**

A READ emulation begins by invoking read operations on all base objects, and waiting for responses. This is the first round. We construct three different executions that are indistinguishable to the reader after receiving $2t + k$ responses in the first round.

1. In execution α_1, a WRITE(1) operation completes before READ is invoked. Therefore, READ must return 1. When READ is invoked, at least $t+k$ correct objects' states are s_1. Without loss of generality, these are objects $o_1, \ldots, o_{t+k}$. Moreover, t correct objects are in state s_0 (this is possible by Lemma 7.3). Without loss of generality, these are objects $o_{t+k+1}, \ldots, o_{2t+k}$. In response to the first read round, k correct objects, $o_1, \ldots, o_k$, return s_1 (if $k = 1$ then only o_1 returns s_1), t correct objects, $o_{t+k+1}, \ldots, o_{2t+k}$ return s_0, and t objects $o_{2t+k+1}, \ldots, o_n$ are faulty and also return s_0. The remaining objects do not respond.

2. In execution β_1, no WRITE ever occurs. Therefore, READ must return 0 (the register's initial value), and all the correct objects are in state s_0. In this execution, objects $o_1, \ldots, o_k$ are faulty and return s_1 in the first read round. $2t$ objects, $o_{t+k+1}, \ldots, o_n$, are correct and return s_0. The read invocations on the remaining objects do not take place.

3. In execution γ_1, a WRITE(1) operation occurs concurrently with the READ. In this execution, all objects are correct. The first read request reaches objects $o_{t+k+1}, \ldots, o_n$ before WRITE is invoked, and they therefore respond with s_0. The first read request sent to objects $o_1, \ldots, o_k$ is delayed while the WRITE is executing. By FW-termination, the WRITE eventually completes and changes the states of objects $o_1, \ldots, o_k$ to s_1. The read invocations on objects $o_1, \ldots, o_k$ then take place, and they respond with s_1.

In all three executions, the reader receives the same responses from base objects. In α_1, it is not allowed to return 0. In β_1, it is not allowed to return 1. In γ_1, it is not allowed to wait for more round 1 responses, because it already heard from $2t + k$ correct objects, and the remaining t may be faulty, and since the WRITE completes, and no new WRITEs are invoked, FW-termination mandates that READ complete as well. Therefore, a second round of base object invocations is initiated at the end of each of the three executions.

Inductive step

Assume that $1 < i \leq t/k$. We use our inductive hypothesis for α_{i-1} and β_{i-1}, and show how to construct α_i, β_i, and γ_i from α_{i-1} and β_{i-1}. Note that $(i+1)k \leq t+k$ (since $i \leq t/k$), and thus, there are at least $2k$ objects have not yet responded to any of the read rounds in α_{i-1} and β_{i-1}: $O_{(i-1)k+1}, \ldots, O_{(i+1)k+1}$. We construct the three executions by having k additional objects respond to all the read rounds, as follows:

1. Execution α_i extends α_{i-1} by having the correct objects $o_{(i-1)k+1}, \ldots, o_{ik}$ respond to all the read rounds with their state s_1; and having all the objects that have responded in previous rounds respond the same way in round i.

2. Execution β_i extends β_{i-1} by having the objects $o_{(i-1)k+1}, \ldots, o_{ik}$ become faulty and respond to all the read rounds with s_1; and having all the objects that have responded in previous rounds respond the same way in round i.

3. Execution γ_i extends β_{i-1} as follows: first, all the objects that have responded in previous rounds respond the same way in round i. Meanwhile, the invocations to all remaining objects continue to be delayed. A WRITE(1) operation is then invoked and, by FW-termination, completes and changes the states of objects $o_{(i-1)k+1}, \ldots, o_{ik}$ to s_1. Subsequently, the reader's delayed threads are resumed, and read invocations take place at objects $o_{(i-1)k+1}, \ldots, o_{ik}$. These objects are all correct in γ_i, and they respond to all read rounds with s_1.

Again, we have a situation in which in all three executions, the reader receives exactly the same responses. That is, these executions are indistinguishable to the reader. In execution α_i, the reader is not allowed to return 0. In execution β_i, it is not allowed to return 1. In execution γ_i, it hears from $2t+k$ correct objects in all the rounds, and is therefore not allowed to wait for more responses in any round, because the remaining objects may be faulty. By FW-termination, READ must complete. Thus, round $i+1$ must be initiated. We add the initiation of round $i+1$ at the end of α_i, β_i, and γ_i, and the lemma follows. □

From Lemma 7.4, we derive the following theorems:

Theorem 7.5 (Read Lower Bound). *For every algorithm A emulating a t-tolerant SWSR safe register in a system with $n = 3t + k$ base objects, t of which can suffer NR-Arbitrary failures, and in which the reader does not modify the base objects' states, there is an execution of A in which the* READ *emulation invokes $\lfloor t/k \rfloor + 1$ rounds of base object operations.*

Proof. For $k > t$ the theorem trivially holds, because obviously at least one round is required in order to read a value from the register. When $k \leq t$, by Lemma 7.4, there exists an execution, $\alpha_{\lfloor t/k \rfloor}$, in which $\lfloor t/k \rfloor + 1$ rounds are invoked. $\square$

Next, we observe that in execution γ_i, $(i-1)k$ objects are faulty. Therefore, Lemma 7.4 also implies the following lower bound for adaptive (early-stopping) algorithms:

Theorem 7.6 (Early-Stopping Read Lower Bound). *Consider an algorithm A emulating a t-tolerant SWSR safe register in a system with $n = 3t + k$ base objects, t of which can suffer NR-Arbitrary failures, and in which the reader does not modify the base objects' states. For every $1 \leq i \leq \lfloor t/k \rfloor$, there is an execution of A in which $(i-1)k$ objects fail and the* READ *emulation invokes $i + 1$ rounds of base object operations.*

For the special optimal resilience case, (i.e., $k = 1$), we get a lower bound of $f + 2$ rounds in executions with $f < t$ failures by substituting f for $i - 1$ in Theorem 7.6; the $t + 1$ lower bound follows from Theorem 7.5. We get the following corollary:

Corollary 7.7. *Consider an algorithm A emulating a t-tolerant SWSR safe register in a system with $n = 3t + 1$ base objects, t of which can suffer NR-Arbitrary failures, and in which the reader does not modify the base objects' states. For every $0 \leq f \leq t$, there is an execution of A in which f objects fail and the* READ *emulation invokes $\min(t + 1, f + 2)$ rounds of base object operations.*

Finally, observe that in execution β_i, ik objects are faulty and no READ operation overlaps any WRITE operation. We thus get the following lower bound for READs that do not overlap any WRITE:

Theorem 7.8 (Read Lower Bound without Concurrent Writes). *Consider an algorithm A emulating a SWSR safe register in a system with $n = 3t + k$ base objects, t of which can suffer NR-Arbitrary failures, and in which the reader does not modify the base objects' states. For every*

$1 \leq i \leq \lfloor t/k \rfloor$, *there is an execution of A in which ik objects fail and a* READ *emulation that does not overlap any* WRITE *invokes $i+1$ rounds of base object operations.*

When $k = 1$, this yields a lower bound of $f + 1$ rounds on READ emulations that do not overlap any WRITE. The t-tolerant wait-free algorithm presented in Section 5 shows that all the bounds proven in this section are tight.

7.2.1 Allowing Readers to Modify Objects

The lower bounds above assume that the reader does not modify the base objects. We now revisit this assumption. Consider an algorithm in which the reader modifies the base objects and the writer reads information from them. How can such an algorithm be more efficient than an algorithm in which the reader is not allowed to modify the base objects? Conceivably, the reader may be able to signal to the writer that a read is in progress, and the writer could conceivably use this signal in order to refrain from writing to base objects while the reader is reading them. Observe that indeed, our lower bound proof makes use of the fact that WRITE can occur concurrently with the READ.

Whether expediting the READ emulation by allowing readers to write and writers to read is possible or not remains an open problem. However, we believe that in order to allow some form of meaningful communication from the reader to the writer, one would need the abstraction of a SWSR safe register, where the READ emulation is the writer and the WRITE emulation is the reader. Intuitively, a safe register is needed in order for the reader to be able to signal to the writer that read is in progress, and for the writer to be able to distinguish the case that the reader never signaled that read is in progress from the case that the reader did signal so before the WRITE began. We conjecture that no form of communication weaker than a safe register can help reduce the cost of a READ emulation. Therefore, we believe that a safe register in one direction must be emulated at the "full cost" before a safe register in the other direction can be emulated faster. We therefore conjecture that if it is possible to expedite the read in this manner, then the WRITE emulation needs to invoke at least $\lfloor t/k \rfloor + 1$ rounds of read operations on base objects:

Conjecture 7.1. *For every algorithm A emulating a t-tolerant SWSR safe register in a system with $n = 3t + k$ base objects, t of which can suffer NR-Arbitrary failures, there is an execution of A*

in which either the READ *emulation or the* WRITE *emulation invokes* $\lfloor t/k \rfloor + 1$ *operation rounds.*

8 Conclusions

We have studied asynchronous implementations of wait-free shared memory objects from base objects that can suffer NR-Arbitrary faults, focusing on the number of rounds of base object invocations as our primary complexity metric. This failure model and performance metric are important in capturing much recent work on scalable widely-distributed systems that are based on either lightweight replicated servers (e.g., Fleet [MR00] and Agile Store [LAV03]) or on the emerging technology of Storage Area Networks (e.g., PASIS [GWGR04]).

We have addressed a previously open question – whether it possible to construct t-tolerant wait-free shared registers in this model using as little $3t + 1$ base objects, t of which can fail. We have shown that such constructions are indeed possible, but also inherently more costly than constructions that use $4t + 1$ or more fault-prone base registers: First, when $n \leq 4t$, emulating WRITE operations requires two rounds of base object write invocations. Second, we have shown a lower bound of $\min(t+1, f+2)$ rounds for emulating READ operations in executions with f failures in systems where the reader does not modify the base objects. Since in all known constructions for the NR-Arbitrary fault model readers do not modify the base objects, the lower bound has broad applicability. Whether this lower bound still holds when readers are allowed to modify the base objects remains an open problem. However, we have conjectured that even if readers can modify the base objects, it still holds that either the READ or the WRITE emulation must take $\min(t+1, f+2)$ rounds.

We have presented, for the first time, an optimal resilience t-tolerant wait-free construction (i.e., using $3t + 1$ base objects, t of which can fail) of a safe register in the shared memory model. Our safe register construction is early-stopping, and its round complexity is optimal, as we prove in the Section 7. Based on known reductions from safe registers to regular ones, our construction yields a Byzantine version of the Disk Paxos consensus algorithm, which employs as little as $3t + 1$ disks, t of which can be arbitrarily corrupted or non-responsive, and a leader oracle.

Nevertheless, emulating a regular register from safe ones incurs additional rounds of operation invocations. Moreover, our safe register construction is quite elaborate, as are known efficient reductions from safe registers to regular ones. Therefore, from a practical perspective, it is desirable to derive simpler solutions, directly constructing regular registers.

We have addressed this challenge by defining a weaker termination condition called FW-termination, which allows read operations not to terminate if infinitely many writes are invoked. We have presented a simple and elegant construction of an t-tolerant FW-terminating regular register, which we have shown, suffices for solving consensus with a leader oracle. Note that by design, the number of rounds executed by the READ emulation of the t-tolerant FW-terminating register can be unbounded. Nevertheless, in synchronous executions, which are most common in practice, the READ operations of the t-tolerant FW-terminating register always terminate in two rounds. Our t-tolerant FW-terminating construction is therefore quite practical – it is simple, direct, and likely to perform well in a real system.

Acknowledgments

We are thankful to Partha Dutta, Rachid Guerraoui, Maurice Herlihy, Petr Kouznetsov, Victor Luchangco, Nancy Lynch, Mark Moir, and Nir Shavit for many interesting discussions and insightful comments.

References

[ABO03] 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.

[AGM+95] Y. Afek, D.S. Greenberg, M. Merritt, , and G. Taubenfeld. Computing with faulty shared objects. *Journal of the ACM*, 42(6):1231–1274, November 1995.

[AMT93] Y. Afek, M. Merritt, and G. Taubenfeld. Benign failures models for shared memory. In *Proceedings of the 7th International Workshop on Distributed Algorithms*, pages 69–83. Springer Verlag, September 1993. In: *LNCS 725*.

[Baz00] R. Bazzi. Synchronous byzantine quorum systems. *Distributed Computing*, 13(1):45–52, 2000.

[BDFG03] R. Boichat, P. Dutta, S. Frolund, and R. Guerraoui. Deconstructing paxos. *Distributed computing column of the ACM SIGACT News*, 34(1):47–67, 2003.

[BT85] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824–840, October 1985.

[CF99] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, pages 642–657, June 1999.

[CHT96] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, 1996.

[CM02] G. Chockler and D. Malkhi. Active disk paxos with infinitely many processes. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing (PODC'02)*, 2002.

[CMR01] G. Chockler, D. Malkhi, and M. K. Reiter. Backoff protocols for distributed mutual exclusion and ordering. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, pages 11–20, 2001.

[DFG02] C. Delporte, H. Fauconnier, and R. Guerraoui. Failure detection lower bounds on registers and consensus. In *Proceedings of the 16th International Symposium on Distributed Computing (DISC)*, October 2002.

[DLS88] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[GGJR00] Juan A. Garay, Rosario Gennaro, Charanjit Jutla, and Tal Rabin. Secure distributed storage and retrieval. *Theoretical Computer Science*, 243(1–2):363–389, 2000.

[GL03] E. Gafni and L. Lamport. Disk paxos. *Distributed Computing*, 16(1):1–20, 2003.

[GWGR04] G. Goodson, J. Wylie, G.Ganger, and M. Reiter. Efficient byzantine-tolerant erasure-coded storage. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN-2004)*, June 2004.

[HLM03] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS)*, page 522. IEEE Computer Society, 2003.

[HV02] Sibsankar Haldar and Paul Vitanyi. Bounded concurrent timestamp systems using vector clocks. *J. ACM*, 49(1):101–126, 2002.

[JCT98] P. Jayanti, T. Chandra, , and S. Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM*, 45(3):451–500, 1998.

[KR01] I. Keidar and S. Rajsbaum. On the cost of fault-tolerant consensus when there are no faults – a tutorial. Technical Report MIT-LCS-TR-821, MIT Laboratory for Computer Science, May 2001. Preliminary version in SIGACT News 32(2), pages 45–63, June 2001 (published May 15th 2001).

[Lam86] L. Lamport. On interprocess communication – part ii: Algorithms. *Distributed Computing*, 1(2):86–101, 1986.

[Lam98] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[LAV03] S. Lakshmanan, M. Ahamad, and H. Venkateswaran. Responsive security for stored data. *IEEE Trans. on Parallel and Distributed Systems*, 14(19):818–828, September 2003.

[LH94] W. K. Lo and V. Hadzilacos. Using failure detectors to solve consensus in asynchronous shared-memory systems. In *Proceedings of the 8th International Workshop on Distributed Algorithms (WDAG)*, pages 280–295. Springer-Verlag, 1994. In: *LNCS 857*.

[LQLZ04] S. Lin, M. Chen Q. Lian, 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.

[LT89] N. A. Lynch and M.R. Tuttle. An introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, 1989.

[MAD02] J-P. Martin, L. Alvisi, and M. Dahlin. Minimal byzantine storage. In *Proceedings of the 16th International Symposium on Distributed Computing (DISC)*, October 2002.

[MR98] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[MR00] D. Malkhi and M. Reiter. An architecture for survivable coordination in large distributed systems. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):187–202, 2000.

[RL04] 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.

[VA86] P. Vitanyi and B. Awerbuch. Atomic shared register access by asynchronous hardware. In *27th IEEE Symp. Found. Comput. Sci.*, pages 233–243, 1986.

[ZSvR02] L. Zhou, F. B. Schneider, and R. van Renesse. Coca: A secure distributed on-line certification authority. *ACM Transactions on Computer Systems*, 20(4):329–368, 2002.