Byzantine Disk Paxos

Optimal Resilience with Byzantine Shared Memory

Ittai Abraham
Hebrew University
ittaia@cs.huji.ac.il

Gregory V. Chockler[*]
CSAIL/MIT
grishac@theory.lcs.mit.edu

Idit Keidar
Technion

Dahlia Malkhi
Hebrew University
dalia@cs.huji.ac.il

ABSTRACT

We present Byzantine Disk Paxos, an asynchronous shared-memory consensus protocol 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 protocol; that is, we construct two different building blocks, each of which can be used, along with a leader oracle, to solve consensus. One building block is a shared *wait-free* safe register. The second building block is a 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 consensus is solvable with FW-terminating registers and a leader oracle. We construct each of these reliable registers from $n > 3t$ base registers, t of which can be non-responsive or Byzantine. All the previous 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.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles—*shared memory*; D.4.5 [**Operating Systems**]: Reliability—*fault-tolerance*; I.1.2 [**Symbolic and Algebraic Manipulations**]: Algorithms—*analysis of algorithms*; D.4.2 [**Operating Systems**]: Storage Management—*secondary storage, distributed memories*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*distributed systems*

[*]Research supported by NSF grant #CCR-0121277, NSF-Texas Engineering Experiment Station grant #64961-CS, Air Force Aerospace Research-OSR contract #F49620-00-1-0097, MURI AFOSR contract #F49620-00-1-0327, and DARPA award #F33615-01-C-1896.

General Terms

Algorithms, reliability

Keywords

Shared-memory emulations, Byzantine failures, termination conditions, consensus

1. INTRODUCTION

We consider an asynchronous system with multiple processes accessing fault-prone shared memory objects [1, 2, 17]. We assume that a threshold t of the memory objects may fail by being non-responsive [2, 17] or by returning arbitrary values [1, 17] (i.e., by being Byzantine); this failure model is called *non-responsive arbitrary (NR-Arbitrary) faults* [17]. In addition to memory failures, we assume that any number of the processes accessing the shared objects may crash.

This model captures a fair amount of recent work on "data centric" replication, which comes in three main flavors:

1. One flavor is message-passing client-server systems in which servers store information on behalf of clients and the only communication is between clients and servers. Scalability is achieved by making servers as light as possible. Thus, the servers can be modeled as storage components. Examples of systems built using this approach include Fleet [24], SBQ-L [25], Agile Store [18], Coca [29], and [4].

2. The second flavor is given by today's peer-to-peer systems. These systems consist of a collection of nodes spread all over the Internet that store data objects. Naturally, due to their Internet-wide deployment, the storage nodes are prone to malicious attacks. This motivates adopting a Byzantine failure model for the storage nodes. Examples of peer-to-peer systems that adopt storage-centric replication to support availability in face of Byzantine failures include Rosebud [26] and [27].

3. The third flavor directly expresses an emerging network technology, the Storage Area Network (SAN). SAN allows clients to access disks directly over the

network so that the file server bottleneck is eliminated. Examples of SAN-based systems that use disks for information sharing and coordination include Compaq's Petal [21] and Frangipani [28], Disk Paxos [13], and Active Disk Paxos [8].

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 [25]). Although a number of projects have set out to tackle this problem (e.g., E-vault [14], Fleet [24], Agile Store [18], SBQ-L [25], Coca [29], [3], and [4]), 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 [13, 22], and in particular, the shared-memory version of Disk Paxos [13], which employs shared *wait-free single-writer multi-reader (SWMR) regular registers*[1]. Thus, the problem of solving consensus (assuming a leader oracle) can be reduced to implementing an SWMR regular register. When disks are subject to unavailability faults only, such a register is implemented from a collection of fail-prone registers, each stored on one disk, by reading and writing from/to a majority of disks [13].

But coping with NR-Arbitrary faults is more challenging. Since the introduction of the NR-Arbitrary failure model, researchers have constructed wait-free shared registers using $4t + 1$ [24], $5t + 1$ [17], or even $6t + 1$ [9] fault-prone base objects. Several works have achieved better resilience by weakening the model in different ways – by adding synchrony [4]; by storing signed self-verifying data [24, 25]; or by assuming that processes never fail and providing solutions that may block indefinitely if they do fail [3, 25]. However, $t < n/4$ is the best resilience achieved thus far for wait-free constructions in 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 attention to consensus algorithms that do not use authentication). We observe that this resilience is achieved by means of echoing (e.g., [6, 12]). Unfortunately, echoing cannot help us to 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.

We now explain the challenge in working with $n = 3t + 1$. 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 returns once the lower-level write operations on $n - t$ base objects return. But of the $n - t$ base objects that return, t may be faulty, whereas the t that have not responded may be simply slow. In this case, only $n - 2t = t + 1$ correct base objects have stored the value V. 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 $n - t$ of them (again, in order to ensure liveness). The set of $n - t$ responders may overlap the set of $t + 1$ correct objects that have V by as little as one object. The reader has no way of distinguishing this value, which is returned by one correct object, from an arbitrary value V' concocted by a corrupt object.

We present, for the first time, a wait-free SWMR safe register[2] construction using as little as $3t + 1$ base registers, out of which t may suffer arbitrary corruption. Using known reductions from regular to safe registers (see e.g., [19] and a survey in [15]), we can thus achieve a wait-free regular register, which in turn, can be used to solve consensus with a leader oracle. Our construction differs from previous constructions of reliable registers in the NR-Arbitrary failure model in that we implement the *write* operation in *two* rounds, whereas previous constructions required only one. In the full paper, we prove that this cost is required for achieving the optimal $3t + 1$ resilience. The *read* operation in our register implementation is *early-stopping*; it incurs $\min(f + 2, t + 1)$ rounds, where f is the actual number of faults in the execution. We further prove in the full paper that this complexity is also tight. For completeness, these lower bounds are stated in Section 6.

Although we cannot hope to improve on the complexity of this solution, we further seek simpler solutions from an engineering perspective. The key to such solutions 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. Of course, in order for FW-terminating registers to provide a useful service, a contention management mechanism is required. However, we observe that in the context of consensus, this is provided by a leader oracle, which is necessary for consensus anyway [22, 11, 7]. It is therefore both natural and efficient for us to construct registers that guarantee progress only when a unique leader emerges.

Indeed, this leads us to implement an 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 an efficient and simple adaptation of Disk Paxos, which tolerates NR-Arbitrary faults of up to a third of the disks. We enjoy the simplicity of this construction so much that we present it first in the paper.

From a formal perspective, solving consensus with shared objects that are not wait-free is in itself a contribution. In [22, 13, 11], it was shown that wait-free consensus is pos-

[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*.

[2]A safe register guarantees that every *read* operation that does not overlap any *write* returns the latest written value, or the initial value if no value was written; the result of a *read* operation that does overlap a *write* operation may be arbitrary.

sible with wait-free read/write registers and a leader oracle. In this paper, we show for the first time that registers satisfying a weaker (in the sense of allowable behaviors) progress condition suffice. This approach integrates well with the Paxos general philosophy, which decomposes consensus into a safety building block (called Synod in [20]) and a progress component (leader election). In shared memory this deconstruction was substantiated in [5, 8] where coarse-grained shared objects encapsulating the Synod algorithm were identified. In this paper, the approach of separating safety requirements from liveness is applied right down to the lowest level objects of which Paxos is constructed: the read/write registers.

Contributions. In summary, our contributions are as follows. We introduce Byzantine Disk Paxos, the first shared-memory consensus algorithm to tolerate NR-Arbitrary faults of up to a third of the system. We present two constructions of Byzantine Disk Paxos. In Section 3, we present a construction of an FW-terminating regular register from Byzantine shared memory, which enjoys engineering simplicity. Section 5 identifies such registers as building blocks for consensus. In Section 4, we present an emulation of a wait-free register out of $3t + 1$ corruptible ones, which was never before achieved. The construction is tight in both resilience and round complexity, by the lower bounds we prove in the full version of this paper and state in Section 6.

2. THE SYSTEM MODEL

We model both processes and objects (registers) as I/O automata [23]. 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. The transitions triggered by the input actions are always enabled, whereas those triggered by the output and internal actions (collectively called *locally controlled* actions) depend solely on the current automaton's state.

Let A be an I/O automaton. An execution α of A is a sequence (finite or infinite) of alternating states and actions $s_0, \pi_1, s_1, \ldots$, where s_0 is the initial state of A and each triple (s_{i-1}, π_i, s_i) is a transition of A. The trace of an execution α of A is the subsequence of α consisting of all the external actions. An execution α of A is *fair* if every locally controlled action of A either occurs infinitely often in α or is enabled only a finite number of times in α. A trace is a *fair trace* of A if it is the trace of fair execution of A.

An object *type* 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* $\times$ $Vals$. An asynchronous shared memory system is a composition of a (possibly infinite) collection of process automata $P_1, P_2, \ldots$ and 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 input of P_i and an output of O_j).

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 the following, we only consider systems where interactions 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 *complete*; otherwise, it is *incomplete*. Note that well-formedness does not rule out concurrent operation invocations on the same object by different processes. Nor does it rule out parallel invocations by the same process on different objects, which can be performed in separate threads of control.

A threshold t of the objects may suffer NR-Arbitrary failures [17], i.e., may fail to respond to an invocation, or may respond with an arbitrary value. Any number of the processes may fail by stopping. 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.

2.1 Registers

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

We now define several register properties. Fix x to be a single-writer/multi-reader (SWMR) register, and let σ be a sequence of invocations and responses of x.

Safe register. σ is *safe* [19] 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* [19] 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 *write*s 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. Register x 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, every *read* invocation by a correct process either completes, or infinitely many *write*s are invoked. Register x is *FW-terminating* if all its fair traces satisfy FW-termination.

Note that operations by correct process are required to complete regardless of the number of process failures, since failed processes are not required to take any steps in fair executions.

We now examine the relationship of FW-termination with previously suggested termination conditions by comparing their corresponding sets of allowable behaviors: We observe that FW-termination is strictly stronger than lock freedom as well as obstruction-free termination as defined in [16], since with FW-termination, write operations are wait-free, and *all* the read operations by correct processes that do not overlap any write operation are required to complete. FW-termination is strictly weaker than wait freedom, since a read operation that is concurrent with infinitely many writes is not required to complete.

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

3. FW-TERMINATING REGULAR REGISTER CONSTRUCTION

We construct an FW-terminating SWMR regular register by a shared memory system consisting of any number of processes and $n > 3t$ SWMR fault-prone regular registers, $x_1 \ldots x_n$. Each register stores a pair of values, and each of these values is associated with a timestamp, taken from a totally ordered set TS, with the minimum element ts_0. The shared registers are defined in Figure 1. To distinguish between the emulated register's interface and that of the underlying base registers, we denote the emulated read (resp. write) operation as READ (resp. WRITE). The emulation of the FW-terminating register's WRITE operation appears in Figure 2, and the READ emulation appears in Figure 3.

Since the underlying registers can be non-responsive, processes 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. The notation INVOKE $write(x_i, v)$, (respectively, INVOKE $tmp \leftarrow read(x_i,))$ means that a new thread is spawned that performs a *write* on register x_i with value v (respectively, a *read* of register x_i whose response will be stored in local variable *tmp*). The notation x_i RESPONDED means that the last thread created by an INVOKE operation has completed its execution on register x_i. Note that this is well defined because we maintain well formedness (i.e., that at any instant, each register has at most one incomplete invocation). We track the status of each base register x_i using three 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. When an invocation is enabled and is not pending, a thread is spawned to initiate an invocation on x_i. In order to avoid using values that were read in previous invocations of READ, the *old[i]* bit is set (first line of READ) for registers that have pending invocations from previous READs. Data read from old threads is discarded.

As dictated by the lower bound shown in the full version of this paper, the WRITE emulation consists of two rounds: First, the *pre-write phase* writes the value to the base registers' *pw* components, and then, the *write* phase writes to both the registers' components. Each value is written together with a monotonically increasing timestamp.

READ 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. 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. Moreover, to ensure regularity, READ must not return old values written before the last WRITE that precedes the READ. Enforcing this condition is more subtle: 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 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 different value are invalid (line 6). Once $C \neq \emptyset$, READ terminates and returns some value in C. This guarantees regularity, as proven below:

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

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

Figure 1: Base registers used in constructions.

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

WRITE($v$):
    choose $ts \in TS$ larger than previously used;
    $pw \leftarrow \langle ts, v\rangle$;
    for $1 \le i \le n$, $enabled[i] \leftarrow true$;
    repeat
        CHECK();
    until $|\{i : \neg enabled[i] \wedge \neg pending[i]\}| \ge n - t$;
    $w \leftarrow \langle ts, v\rangle$;
    for $1 \le i \le n$, $enabled[i] \leftarrow true$;
    repeat
        CHECK();
    until $|\{i : \neg enabled[i] \wedge \neg pending[i]\}| \ge n - t$;
    return $ack$;

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

Figure 2: The WRITE code executed by the register emulations.

LEMMA 1 (REGULARITY). *The algorithm consisting of the* WRITE *emulation in Figure 2 and the* READ *emulation*

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

Predicate and macro definitions:
    $\mathsf{safe}(c) \triangleq |\{i : pw[i] = c \vee w[i] = c\}| \ge t+1$
    $\mathsf{invalid}(\langle ts, v\rangle) \triangleq$
        $|\{i : ((pw[i] = \langle v', ts'\rangle) \vee (w[i] = \langle v', ts'\rangle)) \wedge$
         $((ts' < ts) \vee (ts' = ts \wedge v' \neq v))\}| \ge 2t+1$
    $\mathsf{higherValid}(c) \triangleq \exists c' \in TSVals$ :
              $(pw[i] = c' \vee w[i] = c') \wedge$
              $c'.ts \ge c.ts \wedge c' \neq c \wedge \neg\mathsf{invalid}(c')$

READ():
1:  for $1 \le i \le n$, if($pending[i]$) then $old[i] \leftarrow true$;
2:  for $1 \le i \le n$, $pw[i], w[i] \leftarrow \perp$;
3:  repeat
4:      for $1 \le i \le n$, $enabled[i] \leftarrow true$;
        repeat
            CHECK();
5:      until $|\{i : \neg enabled[i] \wedge \neg pending[i]\}| \ge n-t$;
6:      $C \leftarrow \{c \in TSVals \mid \mathsf{safe}(c) \wedge \neg\mathsf{higherValid}(c)\}$;
7:  until $(C \neq \emptyset)$;
8:  return $c.val$: $c \in C$;

CHECK()
    if $(\exists i : enabled[i] \wedge \neg pending[i])$ then
        $\langle enabled[i], pending[i]\rangle \leftarrow \langle false, true\rangle$;
        INVOKE $\langle tmpPW[i], tmpW[i]\rangle \leftarrow read(x_i)$;
    if $(\exists i : x_i$ 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 3: FW-terminating regular register READ emulation.

in Figure 3 has only regular traces.

PROOF. 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 $\langle ts_0, v_0\rangle$. If no WRITE completes before READ is invoked, then c is a valid return value, and we are done.

Otherwise, let R_i be a READ invocation and $W =$ WRITE(v) be the last WRITE that completes before R_i occurs. Let ts be the timestamp written with v. We need to show that R_i does not return an older value, i.e., that the return value $c.val$ is not associated with a timestamp $c.ts < ts$ (as timestamps are monotonically increasing). Since the write phase of W is complete, $\langle ts, v\rangle$ has been written to the w field of at least $n - 2t \ge t+1$ correct registers. Since the base registers are regular, each of these $t+1$ correct registers responds to each *read* operation of R_i with a pair $\langle pw, w\rangle$ such that $pw.ts \ge ts \wedge w.ts \ge ts$.

Consider the reader's state after line 6 (in any iteration) of R_i. As $n-t$ responses were read, at least one of them is from a correct register, x_i, updated by W. Hence, $pw[i].ts \ge ts \wedge w[i].ts \ge ts$. Let c be the smallest timestamped pair returned by a correct register x_k (either in its pw or w field) for which $pw[k].ts \ge ts \vee w[k].ts \ge ts$. 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 pairs $\langle pw', w'\rangle$ such that $pw' = \langle ts', v'\rangle \vee w' = \langle ts', v'\rangle$ and $((ts' < c.ts) \vee (ts' = c.ts \wedge v' \neq c.val))$. Thus, at least one of these responses must be from a correct register x_j that was updated by W. Therefore, $pw[j].ts \ge ts \wedge w[j].ts \ge ts$. 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 in line 6, there is a response $c = w[k]$ or $c = pw[k]$ such that $c.ts \ge ts$ and $\neg\mathsf{invalid}(c)$. Therefore, $\forall c' \in C$, $c'.ts \ge ts$, and no value with a timestamp smaller than ts can be returned. ☐

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.

LEMMA 2 (FW-TERMINATION). *The register emulated by the* WRITE *code in Figure 2 and the* READ *code in Figure 3 is FW-terminating.*

PROOF. Let α be a fair execution of the register emulation algorithm. Since no more than $n-t$ responses are awaited at either phase of WRITE, and at most t registers are faulty, a WRITE invoked by a correct process in α completes. We next prove that if there is a finite number of WRITE invocations, then a READ invocation by a correct process completes.

Note that READ never blocks at the wait statement in line 5, since $n-t$ responses are awaited. Therefore, READ continues to issue new *read* rounds as long as it does not return a value. Assume by contradiction that READ never returns. Let t be a point in α after which no *write* operations on base registers are invoked, and by which all *write* operations invoked on correct registers have completed. Let $t' > t$ be a point by which every correct register has responded to at least one *read* invocation that was initiated after t.

Let $\langle ts, v\rangle$ be the value-timestamp pair written in the last complete WRITE invocation W, or $\langle ts_0, v_0\rangle$ if there is none. We consider two cases: First, if no WRITE later than W completes the pre-write phase (either none is invoked, or one is invoked but the writer fails before completing the pre-write phase), then from point t' 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, by the end of line 6, $\langle ts, v\rangle \in C$. Hence, the termination condition in line 7 is satisfied and READ returns.

Second, suppose that an incomplete WRITE invocation $W' =$ WRITE($\langle ts', v'\rangle$) occurs after W, and the pre-write phase of W′ completes. Then after t', $\langle ts', v'\rangle$ appears at least $t+1$ times in $pw[]$, and is 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,

$\neg$higherValid$(\langle ts', v' \rangle)$ holds, and after line 6, $\langle ts', v' \rangle \in C$. Hence, the termination condition in line 7 is satisfied and READ returns. □

We have proven the following:

THEOREM 1. *The algorithm consisting of the* WRITE *emulation in Figure 2 and the* READ *emulation in Figure 3 implements an SWMR FW-terminating regular register using $n > 3t$ SWMR regular registers up to t of which can suffer NR-arbitrary failures.*

Note that since a new read round is initiated whenever $n - t$ responses for the previous round arrive, faulty registers responding much faster than correct ones may cause the READ emulation to take an unbounded number of rounds, even if no concurrent WRITE occurs. Nevertheless, we observe that in *synchronous* runs without a concurrent WRITE, READ always terminates in two rounds. In order to formally make such a claim, we need to consider a partially synchronous (or timed-asynchronous) model [12, 10]. In such models, it is possible to wait for messages until a certain timeout. In periods when the system is synchronous, messages from correct processes always arrive by this timeout. Achieving good performance in synchronous runs of an asynchronous system is important, as such runs are common in practice.

4. WAIT-FREE SAFE REGISTER CONSTRUCTION

We now proceed to construct an SWMR wait-free safe register out of $n \geq 3t+1$ regular base registers. The implementation uses the same base registers as the FW-terminating register implementation (see Figure 1), and the WRITE operation is emulated exactly the same way as that of the FW-terminating register (see Figure 2). The READ implementation is presented in Figure 4. As in the FW-terminating register, the reader invokes multiple rounds of *read* operations to the base registers, until it finds a value that is safe to return. However, unlike the FW-terminating implementation, the number of *read* rounds is bounded. We begin by explaining why the implementation is safe; we shall later discuss its liveness and complexity.

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; *prevReadW* holds a copy of *ReadW* from the end of the previous read round (line 7). Responded is the set of registers that responded to *read* requests thus far.

The set C includes candidate return values. After the first *read* round, C consists of values that were read from w fields (line 5). Since at least $n - t \geq 2t + 1$ base registers are read (line 4), any value that is not included in C was either not completely written before the READ began (its WRITE could have begun but could not have completed), or was already over-written (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 10). If $C = \emptyset$, it must be the case that a WRITE invocation overlaps the READ, and every return value is correct. If C is not empty, the leading candidate to return is the one associated with the highest timestamp, because if several values were completely written before the READ began, the latest one should be returned. This is captured by the predicate highCand(c).

```
Local variables:
    Boolean arrays $enabled[n]$, $pending[n]$, $old[n]$,
        initially $pending[i] = enabled[i] = old[i] = false$
        for all $1 \leq i \leq n$;
    Arrays $pw[n], w[n]$ with elements in $TSVals$;
    $ReadW$, $ReadPW, prevReadW$ :
        partial functions from $TSvals$ to $\mathcal{P}(\{1 \ldots n\})$;
    $C \subseteq TSVals$, initially $C = \emptyset$
Predicate and macro definitions:
    Responded $\triangleq \{i : \exists \langle w, i \rangle \in ReadW\}$
    highCand$(\langle ts, v \rangle) \triangleq \langle ts, v \rangle \in C \land (ts = max\{ts' | \langle ts', v' \rangle \in C\})$
    safe$(c) \triangleq |ReadW(c) \cup ReadPW(c) \cup$
                 $\bigcup_{c'.ts > c.ts}(ReadW(c') \cup ReadPW(c'))| \geq t + 1$

READ():
1:  for $1 \leq i \leq n$, if$(pending[i])$ then $old[i] \leftarrow true$;
2:  $ReadPW, ReadW \leftarrow \emptyset$;
3:  for $1 \leq i \leq n$, $enabled[i] \leftarrow true$;
4:  repeat
        CHECK();
    until $|\{i : \neg enabled[i] \land \neg pending[i]\}| \geq n - t$;
5:  $C \leftarrow \{w[i] : i \in$ Responded$\}$;
6:  while $(C \neq \emptyset \land (\neg \exists c \in C :$ highCand$(c) \land$ safe$(c)))$ do
7:      $prevReadW \leftarrow ReadW$;
8:      for $1 \leq i \leq n$, $enabled[i] \leftarrow true$;
9:      repeat
            CHECK();
        until $|\{i : \neg enabled[i] \land \neg pending[i]\}| \geq n - t \land$
              $\forall c \in C$: (safe$(c)$ $\lor$
                            $|$Responded $\setminus prevReadW(c)| \geq n - t)$;
10:     $C \leftarrow C \setminus \{c \in C : |$Responded $\setminus ReadW(c)| \geq 2t + 1\}$;
11: if $(C \neq \emptyset)$ then
12:     return $c.val$ : highCand$(c)$;
13: return $v_0$;

CHECK()
    if $(\exists i : enabled[i] \land \neg pending[i])$ then
        $\langle enabled[i], pending[i] \rangle \leftarrow \langle false, true \rangle$;
        INVOKE $\langle pw[i], w[i] \rangle \leftarrow read(x_i)$;
    if $(\exists i : x_i$ 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 4: Wait-free safe register READ emulation.

Let c be a candidate for which highCand(c) is true. It is safe to return c if $t+1$ registers have responded either with c or with later values in their pw field. This is because at least one of these registers must be correct, implying that either c was indeed written, or that a WRITE operation 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. The correctness of this condition is proven by the following lemma.

LEMMA 3 (SAFETY). *The algorithm consisting of the* WRITE *emulation in Figure 2 and the* READ *emulation in Figure 4 has only safe traces.*

PROOF. Let $\langle ts, v\rangle$ be the value-timestamp pair written in the latest WRITE invocation that returns before $R =$ READ is invoked, or $\langle ts_0, v_0\rangle$ if no WRITE has completed. Assume further that no WRITE overlaps the READ. We show that R does not return a value other than v. 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, there are at least $t + 1$ correct registers that have $\langle ts, v\rangle$ in their w and pw fields throughout the duration of R. By the READ code, responses from at least $n-t$ registers are awaited in lines 4 and 9. Therefore, $\langle ts, v\rangle$ is included in C in line 5. Moreover $\langle ts, v\rangle$ is never excluded from C in line 10. Hence, $C \neq \emptyset$ from the first time line 6 is executed onward, and the algorithm does not return in line 13. 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 $ts' = ts \wedge v' \neq v$. Hence, no value other than v is returned in line 12. □

We now turn to discuss liveness. In each round, at least $n - t$ responses are awaited (line 9). This does not lead to blocking, since there are at least $n - t$ correct base registers. In order to limit the number of initiated rounds, line 9 includes an extra waiting condition that must be satisfied before an additional round can be initiated. We now explain why this condition does not violate liveness. Consider a candidate $c \in C$. Note that $prevReadW(c) \neq \emptyset$ in line 9. There are two cases:

1. *At least one register in prevReadW(c) is correct.* Therefore, 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 an earlier round). Each correct register eventually responds in the current round with either c or a higher timestamped value, and $\mathsf{safe}(c)$ 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 send c. Since these registers eventually respond, and since none of them is included in $prevReadW(c)$ during the wait, eventually $|\mathsf{Responded} \setminus prevReadW(c)| \geq n - t$ becomes true.

Line 9 waits for one of the above to hold, and therefore does not block. We have proven the following:

LEMMA 4 (NON-BLOCKING). *The* READ *emulation never blocks indefinitely at a wait statement.*

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 6 is satisfied. If C is empty, we are done. Otherwise, consider $c \in C$. Line 9 waits 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 9, then either $ReadW(c)$ grows, or c is removed from C in line 10 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 10, $ReadW(c)$ 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 (WAIT FREEDOM). *The safe register emulation satisfies wait freedom.*

PROOF. Every WRITE operation invoked by a correct process returns, as argued in Lemma 2. Consider a READ operation R. By Lemma 4, R does not block in any wait statement. Moreover, as argued above, at most $t+1$ rounds of READ operations are invoked before the termination condition of the loop in lines 6–10 is satisfied, at which point READ completes. □

We have proven the following:

THEOREM 2. *The algorithm consisting of the* WRITE *emulation in Figure 2 and the* READ *emulation in Figure 4 implements a wait-free SWMR safe register from $n > 3t$ SWMR regular registers, up to t of which can suffer NR-arbitrary failures.*

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

LEMMA 6 (EARLY-STOPPING). *In every run in which f registers exhibit Byzantine behavior, the* READ *emulation invokes at most $\min(t+1, f+2)$ rounds of read operations on base registers.*

PROOF (SKETCH). For $j > 1$, consider the initiation of the jth *read* round occurring in line 8 (during the $j-1$st iteration in the loop of lines 6–10). 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. Since c was not removed from C in line 10 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, and there are at most $f - k + 1$ Byzantine faulty registers that are not in $ReadW(c)$. Consider the set S of registers that respond to rounds $k+1 \dots j-1$. 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 at most $f-k+1$ members of S are Byzantine faulty, 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) has completed before round $k + 1$ was initiated, at most t correct registers respond to rounds $k+1 \dots j-1$ with values older than c in their pw field. Therefore, if S includes $2t + 1$ correct registers, then at least $t + 1$ of them have either c or a higher value in their pw field, and c is safe. But we assumed that c is not safe. We get that $2t + j - f - 2 \leq 2t$, that is, $j \leq f + 2$. □

Note that only Byzantine failures cause READ to take more rounds; benign (i.e., crash) failures do not slow the algorithm down. In invocations of READ that do not overlap any WRITE invocation, READ invokes at most $f + 1$ rounds. For space limitations, we do not prove this here, but it can be proven similarly to Lemma 6. Moreover, we observe that in *synchronous* runs, READ always terminates in two rounds.

5. CONSENSUS WITH FW-TERMINATING REGISTERS AND Ω

In a *consensus* problem, each process has an input and may decide on an output, so that the following conditions are satisfied: (1) *termination*: 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. We present a shared memory consensus algorithm based on those of [22, 13], 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 5. It solves consensus among n processes $P_1, \ldots, P_n$ using n FW-terminating SWMR regular registers $x_1, \ldots, x_n$, 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 [7], the weakest for consensus [22, 11, 7]. 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 trusted by P_i. A failure detector of class Ω guarantees that, eventually, a single correct process is permanently trusted by all correct processes.

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

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

Algorithm for process $i$:

Local variables:
        $val \in Vals$, $bal \in \mathbb{N}$, $\ell \in \mathbb{N}$, $a_i \in X$ for $1 \le i \le n$;

1:      $bal \leftarrow i$;
2:      $val \leftarrow inp_i$;        // $inp_i$ is the initial value of $i$
3:      while ($true$) do
4:          $\ell \leftarrow \mathcal{L}_i$;
5:          if ($\ell = i$) then
6:              $write(x_i, \langle bal, \bot, \bot \rangle)$;
7:              $a_j \leftarrow read(x_j)$, for each $j$, $1 \le j \le n$;
8:              if ($\max\{a_j.bal : 1 \le j \le n\} \le bal$) then
9:                  if ($\exists j : a_j.val \ne \bot$) then
10:                     $val \leftarrow a_k.val$: $1 \le k \le n \wedge a_k.bal =$
                            $\max\{a_j.bal : 1 \le j \le n \wedge a_j.val \ne \bot\}$;
11:                 $write(x_i, \langle bal, val, pc \rangle)$;
12:                 $a_j \leftarrow read(x_j)$, for each $j$, $1 \le j \le n$;
13:                 if ($\max\{a_j.bal : 1 \le j \le n\} \le bal$) then
14:                     $write(x_i, \langle bal, val, c \rangle)$;
15:                     decide $val$ and halt;
16:             $bal \leftarrow bal + n$;
17:         else
18:             $a_\ell \leftarrow read(x_\ell)$;
19:             if ($a_\ell.stat = c$) then
20:                 decide $a_\ell.val$ and halt;
```

Figure 5: The consensus algorithm.

The algorithm is leader-based. A process ℓ that trusts itself 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 (line 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, ℓ chooses the value previously proposed with the highest ballot number, or its own initial value if there is none (lines 7–10). A proposal succeeds if no higher ballot is read (lines 12–13).

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

LEMMA 7. *Suppose that a process ℓ decides, and let i be a process that permanently trusts ℓ. If i reaches line 18, then i eventually decides.*

PROOF. Suppose that process ℓ decides, and let i be a process that permanently trusts ℓ. Suppose that i has reached line 18. Since i continues to trust ℓ, it will proceed to read x_l (line 18) in a loop. By the code, once a process decides, it halts. Thus, no more writes to x_ℓ will be ever invoked. By FW-termination, every $read(x_\ell)$ invoked by any process i will eventually return. Since each process i continues to invoke $read(x_\ell)$ in a loop, eventually, some $read(x_\ell)_i$ will be invoked after $write(x_\ell, \langle *, *, c \rangle)_\ell$ returns. Once this happens, by regularity of x_ℓ, the response of $read(x_\ell)_i$ stored in a_ℓ will have $a_\ell.stat = c$, and i will decide. □

LEMMA 8 (TERMINATION). *In any fair execution, all non-faulty processes eventually decide.*

PROOF. Since $\mathcal{L} \in \Omega$, every fair execution of the algorithm eventually reaches a point after which no more failures occur and each correct process i permanently trusts the same correct process ℓ. By the code, after this point, each process $i \ne \ell$ writes x_i at most twice (in lines 11 and 14), and then proceeds to read x_l (line 18) in a loop. If the process ℓ has already decided, then by Lemma 7, each process i will decide as well.

Otherwise, by FW-termination, process ℓ will eventually be able to complete any read operation from any register. Consequently, ℓ will be able to execute lines 6–16 sufficiently many times for its ballot to become the highest ballot ever written. Once this happens, process ℓ will decide and halt. By Lemma 7, all other processes waiting for ℓ's decision will decide and halt as well. □

We next prove the algorithm's safety properties (agreement and validity).

LEMMA 9 (AGREEMENT). *All decision values are identical.*

PROOF. Let b_1 be the lowest ballot at which some process i decides value v_1. Suppose that a process k proposes value v_2 at ballot $b_2 \ge b_1$. We show that $v_2 = v_1$, which implies agreement. The proof is by induction on ballot numbers $b \ge b_1$. Since the base case is trivially true, we proceed directly to the induction step. Suppose that the result holds for all b, $b_1 \le b < b_2$, and consider $b = b_2$. Since i decides

v_1 at b_1, it must have proposed v_1 at b_1. Moreover, for all register values a_j read in line 12, $a_j.bal \leq b_1$. Therefore, the write in line 6 by j that writes $b_2 > b_1$ to x_j, must return after the *read*(x_j) by i in line 12 has been invoked. For otherwise, by regularity of x_j and because ballots are monotonically increasing at each process, *read*(x_j) by i must respond with $\langle b', \perp, \perp \rangle$ such that $b' \geq b_2 > b_1$ contradicting the fact that $a_j.bal \leq b_1$. Thus, the read $R = read(x_i)$ by j in line 7 is invoked after $write(x_i, \langle b_1, v_1, pc \rangle)_i$ is completed. Since i halts 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, upon completion of 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, by line 8, $b' \leq b_2$. Since j has not yet proposed any value, by line 6, $a_j.val = \perp$. Thus, we receive that v' must have been written at ballot b', $b_1 \leq b' < b_2$. Finally, since for any value $v \neq \perp$ such that $x_k.val = v$, v must have been either proposed or decided by k, 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, j will propose v_1 at b_2 as needed. □

LEMMA 10 (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 3. *The pseudo-code in Figure 5 is a solution for the consensus problem.*

COROLLARY 1. *n SWMR FW-terminating regular registers are sufficient to solve consensus among n processes in an asynchronous shared memory system augmented with a failure detector of class Ω.*

Given a system with $3t + 1$ disks, t of which can be arbitrarily corrupted or non-responsive, we use the construction in Section 3 in order to emulate each of the n FW-terminating registers required for consensus from $3t+1$ base registers, each stored on a different disk. Thus, we have the following corollary:

COROLLARY 2. *Consensus can be solved using $3t+1$ disks, t of which can be arbitrarily corrupted or non-responsive, and a failure detector of class Ω.*

6. LOWER BOUNDS ON OPTIMAL RESILIENCE FW-TERMINATING EMULATIONS

In the full paper, we also prove lower bounds on memory emulations of reliable objects from ones that can suffer NR-Arbitrary faults. Here, we merely state these bounds without proof.

Obviously, at least $3t + 1$ base objects are required in order to emulate a reliable one in this model (see [25]). We focus on optimal-resilience emulations, that is, emulations using $3t + 1$ base objects. The results are proven for FW-terminating registers, and hence apply to wait-free ones as well. To strengthen our bounds, we prove the lower bounds for emulations of the weakest meaningful register type: *a single-writer single-reader (SWSR)* safe register [19] with a binary value domain; and we allow for atomic base objects of any type.

We define a *round of invocations* to be a collection of operations that are invoked *concurrently* on a number of base objects (at most once on each base object). We consider a concurrent system C implementing an FW-terminating SWSR safe register out of $n > 0$ base objects.
The following theorem shows that emulating write operations requires two rounds:

THEOREM 4. *Suppose $n \leq 4t$. Then, for every $0 \leq f \leq t$, there exists a run of C with f base object failures that includes a complete invocation of write(1) and no other invocations on the emulated SWSR register, such that at least two invocation requests are completed on some correct base object.*

The following two theorems prove lower bounds on the number of invocations required for emulating a read operation:

THEOREM 5. *Suppose $n = 3t + k$, and assume that the reader does not modify the base Suppose $n = 3t + k$, and assume that the reader does not modify the base objects' states. For every $1 \leq i \leq \lfloor t/k \rfloor$, there is a run of C in which ik objects fail and a read emulation invokes $i + 1$ rounds of base object operations.*

THEOREM 6. *Suppose $n = 3t + k$, and assume that the reader does not modify the base objects' states. For every $1 \leq i \leq \lfloor t/k \rfloor$, there is a run of C in which $(i - 1)k$ objects fail, and a read emulation that does not overlap any write operations invokes $i + 1$ rounds of base object operations.*

When $k = 1$, we get that for $0 \leq f \leq t$, the lower bound on the number of rounds required to emulate read in runs with f failures is $min(t + 1, f + 2)$. If a read invocation does not overlap any write operation, then the lower bound is $f + 1$. Our algorithm shows that these bounds are tight.

7. DISCUSSION

We have presented asynchronous implementations of reliable shared memory objects from base objects that can suffer NR-Arbitrary faults. We gave the first optimal resilience construction of a wait-free safe register in this model. 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.

Our safe register construction is early-stopping, and its round complexity is optimal, as we prove in the full version of this paper. 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 an engineering perspective, it is desirable to derive simpler solutions.

We have addressed this challenge by defining a weaker termination condition called FW-termination. We have presented a simple and elegant construction of an FW-terminating regular register, which we have shown, suffices for solving consensus with a leader oracle. In contrast to our wait-free register, the number of rounds executed by the read operation of the FW-terminating register is unbounded, even in the absence of contention. Thus, there is a tradeoff between our two constructions. Nevertheless, in synchronous runs, the read operations of the FW-terminating construction always terminate in two rounds. Since such runs are common in practice, this construction is likely to perform well in a real system.

8. ACKNOWLEDGMENTS

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

9. REFERENCES

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

[2] 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*.

[3] H. Attiya and A. Bar-Or. Sharing memory with semi-Byzantine clients and faulty storage servers. In *SRDS*, 2003.

[4] R. Bazzi. Synchronous Byzantine Quorum Systems. *Distributed Computing*, 13(1):45–52, 2000.

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

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

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

[8] 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.

[9] 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.

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

[11] 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.

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

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

[14] J. A. Garay, R. Gennaro, C. Jutla, and T. Rabin. Secure distributed storage and retrieval. *Theoretical Computer Science*, 243(1–2):363–389, 2000.

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

[16] 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.

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

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

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

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

[21] E. K. Lee and C. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 84–92, 1996.

[22] 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*.

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

[24] 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.

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

[26] 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.

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

[28] C. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 224–237, 1997.

[29] 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.