On using network attached disks as shared memory

(Extended abstract)

Marcos K. Aguilera* Burkhard Englert† Eli Gafni‡

ABSTRACT

Recent advances in storage technology have enabled systems like Storage Area Networks, where disks are attached directly to the network, rather than being under the control of a single process. In such an environment there is no a priori bound on the number of processes that may access the network attached disks, and so *uniform* implementations are desirable, that is, implementations that do not rely on the number of processes. We investigate how to use network attached disks, where some disks may crash, as a shared communication medium. To do so, we model disk blocks as Multi-Writer Multi-Reader (MWMR) shared memory registers that may fail by crashing. We study whether a finite number of such fail-prone registers can be used to uniformly implement various types of fail-free target registers: wait-free atomic, atomic, and wait-free sequentially consistent. For each of these types, we determine the implementability of Multi-Writer Multi-Reader registers, Multi-Writer Single-Reader registers (MWSR), Single-Writer Multi-Reader registers (SWMR) and Single-Writer Single-Reader registers (SWSR). For example, we show that there is no uniform atomic implementation of a MWMR register using finitely many base registers, even if the implementation need not be wait-free. On the positive side we show that with infinitely many base registers then all types of registers can be implemented. This opens the question of how to translate uniform shared memory protocols that use MWMR registers to use network attached disks.

* HP Labs Systems Research Center, 1501 Page Mill Road, Mail Stop 1250, Palo Alto, CA 94304, aguilera@hpl.hp.com
† University of California at Los Angeles, Dept. of Mathematics, Los Angeles, CA 90095-1555, englert@math.ucla.edu
‡ University of California at Los Angeles, Dept. of Computer Science, Los Angeles, CA 90095-1555, eli@cs.ucla.edu

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

1. INTRODUCTION

Recent advances in storage technology have enabled systems like Storage Area Networks, which have network attached disks or NADs. A NAD is a simple device that just executes requests to read and write blocks of data. It can be accessed by any process in the system, so that the NADs effectively become a shared storage medium that can be used to solve distributed problems such as consensus, as in Disk Paxos [27]. Unlike message-passing systems, which typically require a majority of processes to be correct to avoid partitioning, NADs allow protocols that can withstand the crash of any number of processes. Therefore—like conventional shared-memory models—the model allows *uniform protocols* [13, 26, 30], i.e., protocols that do not require a priori knowledge of or an upper bound on the number of processes that may participate.

One difficulty of this model is that a NAD can fail by crashing and thereby become inaccessible. It is desirable to mask such failures and emulate the conventional shared-memory abstraction, in which memory does not crash. There are many existing shared memory algorithms, particularly uniform and adaptive [2, 3, 6, 7, 8, 9, 10, 11, 12, 14, 16, 17, 18, 19, 20, 22, 23, 28, 29, 33, 34, 35] ones, that use Multi-Writer Multi-Reader (MWMR) registers to solve known problems. As an example, consider Mutual-Exclusion (MX) [24]. Lamport's fast MX [32]—though not a uniform algorithm—uses MWMR registers in its 'fast-track', while Attiya and Bortnikov [15] propose a uniform MX protocol, which out of necessity resorts to MWMR registers. We therefore ask: Can we uniformly implement such registers with NADs? Such an implementation would allow an automatic translation of these MX algorithms, and many others, to use NADs.

We model each disk block of a NAD as a shared register that may fail, and the goal is to implement a shared register that does not fail. While [4, 5, 31] do exactly that in a wait-free manner, the given implementations are not uniform. The number of base registers required grows as the number of processes grows.

In this paper, we study the uniform implementability of fail-free shared registers in various settings. We consider three types of registers: wait-free atomic, atomic,

and wait-free sequentially consistent. For each of these types, we consider Multi-Writer Multi-Reader registers (MWMR), Multi-Writer Single-Reader registers (MWSR), Single-Writer Multi-Reader registers (SWMR) and Single-Writer Single-Reader registers (SWSR), and determine whether each one can be uniformly implemented with a finite number of fail-prone base registers.

For example, we show that no such implementation exists for a MWMR register, even if the implementation need not be wait-free. On the other hand, there *is* an implementation of a SWMR register (when processes are reliable). Therefore, one cannot use the standard technique of implementing a MWMR register by first implementing SWMR registers: doing so would blow up the space complexity. (This opens the question of whether there is an implementation that uses infinitely many registers with a constant number of steps per operation.) Thus, at the current state of affairs, in order to translate a uniform shared-memory algorithm (such as MX) to use NADs, one needs to “open the box”. This leaves another open question of whether one can identify larger “modules” in these algorithms, such that these modules—rather than registers—can be implemented with NADs. Alternatively, one might realize that these algorithms do not require the full power of atomicity, and implement a weaker register.

On the positive side, we show that the three types of MWMR, MWSR, SWMR and SWMR registers have a uniform implementation even in the infinite-arrival model [28], if the number of base registers is infinite. This implementation spreads registers across $2t + 1$ disks—each disk with infinitely many registers—such that all registers of up to t disks may crash.

In summary, the possibility and impossibility results in this paper are shown in Tables 1, 2, 3 and 4.

	Single-Reader	Multi-Reader
Single-Writer	Yes	No
Multi-Writer	No	No

TABLE 1. *Uniform wait-free implementability of atomic registers using finitely many base registers.*

	Single-Reader	Multi-Reader
Single-Writer	Yes	Yes
Multi-Writer	No	No

TABLE 2. *Uniform implementability of atomic registers using finitely many base registers when processes are reliable.*

	Single-Reader	Multi-Reader
Single-Writer	Yes	No
Multi-Writer	Yes	No

TABLE 3. *Uniform wait-free implementability of sequentially consistent registers using finitely many base registers.*

	Single-Reader	Multi-Reader
Single-Writer	Yes	Yes
Multi-Writer	Yes	Yes

TABLE 4. *Uniform wait-free implementability of atomic registers using infinitely many base registers.*

Related work

Uniform protocols, i.e., protocols that do not require a priori knowledge of the number of processes in the system, have been studied (e.g., [13, 30]), particularly in the context of ring protocols. Adaptive protocols, i.e. protocols whose step complexity is a function of the size of the participating set, have been studied in [6, 7, 8, 16, 23, 33]. Long-lived adaptive protocols that assume some huge upper bound N on the number of processes, but require the complexity of the algorithm to be a function of the concurrency have been studied in [2, 3, 9, 10, 11, 17, 18, 19, 20, 29, 34]. A shared object system where the base objects can fail was first investigated by [31], which studies the implementability of fault-tolerant objects from objects that can fail and it considers both registers and consensus objects. For registers, it shows how to implement a fault-tolerant wait-free MWMR atomic register, but the paper does not concern itself with uniformity and, in fact, the given implementation is not uniform.

Recently, Chockler and Malkhi [22] gave an implementation of Disk Paxos [27] on a Storage Area Network using *Active Disks* [1, 36] that supports unmediated concurrent data access by an unlimited number of processes. The implementation is based on the use of read-modify-write objects. These objects are readily available in Active Disks. Since, however, it is impossible to implement a reliable read-modify-write object from a collection of fail-prone ones [31], their implementation uses read-modify-write objects to emulate a new shared memory abstraction, which they call a *ranked register*. Such a ranked register, as they furthermore show, cannot be implemented in the given setting, using just read/write registers.

Our NAD model is different from the one in the Disk Paxos paper [27]. In that paper, the system is synchronous, so that if a process issues a write request and the disk does not respond within the expected delay, the process knows that the disk has crashed. Thus, either the request was executed before the crash, or it will never be executed. In our paper, we assume an *asynchronous* model, where disks may be arbitrarily slow in responding; while the disk does not respond, a pending write can take place at any time.

The proofs of our impossibility results use the idea of covering registers, first proposed in [21] to show some bounds on the number of registers necessary for mutual exclusion. However, our situation is harder than for mutual exclusion: with mutual exclusion, it is possible to directly get a contradiction by carefully running concurrent “get mutex” operations so as to cover base registers, in order to completely hide some “get mutex” operation; once such an operation is hidden, one immediately gets a contradiction by hav-

ing another process simultaneously enter the critical section. With registers, this is not the case: there is no contradiction when a write is completely hidden while there are concurrent writes because it is possible to linearize the writes in any order. Therefore, our proof arguments are significantly more complicated. Furthermore, in addition to covering writes—which are obtained by freezing a process in the midst of its operation when the process is about to write—we also need the notion of a *pending* write—which is a write left in the system after a process has terminated its operation. Such writes can occur in our model because a register may appear to have failed.

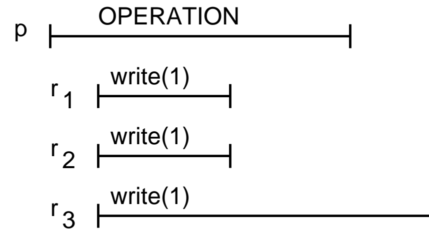

Figure 1: Process p may complete its operation while leaving a *pending* write on register r_3.

1.1 Roadmap

This paper is organized as follows. We explain our informal model of shared objects in Section 2. In Section 3 we consider wait-free implementations in systems where, in addition to registers, processes may also fail by crashing. Then, in Section 4 we consider the case when processes are reliable, and only the registers may fail. We consider sequentially consistent registers in Section 5. In Section 6 we consider the case of infinitely many base registers. Due to space limitations, in this paper the proofs are abbreviated or omitted. The full proofs can be found in the full version.

2. INFORMAL MODEL

We consider a distributed system where processes have shared access to a set of network attached disks.

Processes. Processes have unique id's, but they do not have any information about the number of processes in the system. In particular, the protocols cannot rely on any bounds on the maximum number of processes that participate. For the strongest results, for impossibility we assume the *finite-arrival model* [28], in which only a finite number of processes take steps in any given run, while for our algorithms we assume the *infinite-arrival model*.

We consider two cases regarding resiliency of processes: in the first case, processes may fail by crashing, and in the second case, processes are assumed to be reliable (fail-free). In the former case, we are interested in wait-free implementations, i.e., implementations that guarantee that a correct process terminates its operation in a finite number of steps regardless of the failure of other processes.

Network attached disks. Processes have access to a set of network attached disks, and each disk is divided into blocks. We model each block as a shared register, so that a disk is an array of shared registers. For some of our results, we view the system as a pool of shared registers, and it does not matter how they are separated into disks. The shared registers support two operations, read and write, which may be issued concurrently by many processes. The correctness condition for concurrent access is either atomicity (linearizability) or sequential consistency.

The system is *asynchronous*, meaning that there is no bound on the time it takes for registers to respond to operations. Registers are subject to failures. A *register crash* occurs when the register stops responding to its operations [31].[1] We say a *disk is faulty* if it has some crashed register. A *full disk crash* occurs when all the disk's registers crash. Our impossibility results hold even if a single register may crash[2], while our algorithms allow a bounded number of disks to be faulty, possibly experiencing full disk crashes, even when the crash of a disk affects the infinitely many registers of that disk.

To prevent processes from blocking when a register crashes, we allow processes to issue many nonblocking requests to different registers. For example, while running the implementation of some *OPERATION*, a process p may issue concurrent writes to registers r_1, r_2 and r_3, as shown in Figure 1. The process may decide to complete *OPERATION* even while the write to r_3 is still pending. In fact, the process may be required to do so, because r_3 may never respond. But r_3 may turn out to be just slow, and the write may take effect far in the future.

Without loss of generality, we assume that a process will not attempt to execute simultaneous operations on the *same* register. This assumption is not restrictive and does not affect any of our impossibility results or algorithms (if the process wishes to invoke multiple operations on the same object, it can simply issue one operation and "remember" the other operations in its state, issuing them at a later time. This of course implies that implementations may leave some code running in the background).

We would like to implement registers that do not fail using fail-prone registers. In order to differentiate the base registers from the desired target registers, we denote the operations on the former by read and write, and on the latter by $READ$ and $WRITE$. In addition, we also use capital letters to refer to the $READERS$ and $WRITERS$ of target registers, and lower capitals to refer to the readers and writes of base registers. To get the strongest possible impossibility results, we assume that our base registers are Multi-Writer Multi-Reader (MWMR registers), meaning that they

[1]This corresponds to the unresponsive mode in [31].

[2]Our impossibility results would also hold in a model where whole disks crash at once, that is, it is not legal for a register to crash without all registers of that disk crashing simultaneously.

can be read or written by multiple processes (in fact, by all processes in the system). We study the implementability of many types of target registers, including not only MWMR but also registers that support only a single designated writer but multiple readers (SWMR), a single designated reader but multiple writers (MWSR), or a single writer and single reader (SWSR).

3. WAIT-FREE ATOMIC REGISTERS

We now consider a system where both registers and processes may crash, and study the uniform wait-free implementability of *atomic* registers using a finite number of base registers, one of which may fail by crashing. We first show that it is impossible to implement a SWMR register. This result implies a fortiori that it is impossible to implement a MWMR register. It turns out it is also impossible to implement a MWSR register, and this result will follow directly from Theorem 2 in Section 4.1.

We then give a simple implementation of a SWSR register. Thus the implementability situation is the one shown in Table 1.

3.1 Impossibility result

In the full paper, we show the following theorem:

THEOREM 1. *Let S be a finite set of wait-free atomic MWMR registers and suppose that one of them may fail by crashing. Then there is no uniform wait-free implementation of an atomic SWMR register using S.*

3.2 Possibility result

It is very easy to implement a wait-free atomic SWSR register with only three base registers, by using sequence numbers in the obvious way: To $WRITE$, issue a write to all three registers[3] and wait for two of them to complete. To $READ$, read from two out of three and pick the largest sequence number among the read values and the largest sequence number ever seen before. This implementation works because (1) the sequence number makes it impossible to $READ$ values out of order, and (2) if a $WRITE$ completes, a subsequent $READ$ will either see the value $WRITTEN$ or a later value.

This implementation allows one base register to fail, but it is easy to extend it to allow t base registers to fail by using $2t + 1$ rather than three base registers.

4. ATOMIC REGISTERS

Even in a system where processes never fail, it is impossible to implement a MWMR register with finitely many base registers. In fact, we show that it is impossible to implement the weaker MWSR register. Curiously, it *is* possible to implement a SWMR register (with finitely many base registers), and we provide the implementation. This is in contrast to the case when processes may fail. Thus, the implementability situation for finitely many base registers is shown in Table 2. Of course, if infinitely many base registers are available, we can simply use the implementation of Section 6, since an implementation that tolerates process crashes works fine in the absence of such crashes.

4.1 Impossibility result

We show the following theorem:

THEOREM 2. *Let S be a finite set of atomic MWMR registers and suppose that one of them may fail by crashing. Even if processes never fail, there are no uniform implementations of an atomic MWSR atomic register using S.*

To prove this theorem, suppose that the system has s base registers, one of which may fail. The proof is by contradiction: assume that there is a uniform implementation of a Multi-Writer Single-Reader atomic REGISTER. We assume that the single $READER$ is different from the $WRITERS$ (otherwise, just forget about the fact that the $READER$ is allowed to $WRITER$—we get one less $WRITER$, but there are still infinitely many).

Throughout this proof, we consider runs in which each operation on a base register is linearized deterministically when the operation returns. This is a valid behavior for atomic registers, since operations can be linearized at any point between their invocation and return.

At any point in a run, a configuration $S = (S_{proc}, S_R, S_{pend})$ consists of the state S_{proc} of each process, the state S_R of each base register[4], and the set S_{pend} of pending operations on base registers, i.e., operations on base registers that have been requested but have not yet been executed. Note that a process cannot observe the whole state of the system: in fact, it does not know the state of other processes or the set of pending operations issued by other processes. Thus, as far as its execution is concerned, a process only sees its own state and the state of the registers.

Elements of $S_{pending}$ are of the form (p, op) meaning that p has issued operation op, where op could be either a read or a write for some value on some register. In this proof, we are only concerned about pending writes, and we will therefore ignore the pending reads in $S_{pending}$.

Note that a pending write is different from a covering write [21]: the latter is obtained by freezing a process that is about to issue a write, while the former occurs if a process issues a write but does not wait for it to complete.

DEFINITION 2.1. *We say that a configuration $S = (S_R, S_{proc}, S_{pending})$ is*

- ***possibly-no-pending*** *if there exists some configuration (of some run) that is identical to S except that any*

[3] If one of the registers has a pending write issued during a previous $WRITE$, then the $WRITER$ just forks a background task to issue the write as soon as all previous writes have finished.

[4] This makes sense since we consider runs in which base register operations are linearized deterministically.

subset of its pending operations have been dropped. In other words, for every $X \subseteq S_{pending}$ there exists $S' = (S_R, S'_{proc}, X)$, where the states in S'_{proc} are identical to S_{proc} except for processes with operations in $S_{pending} \setminus X$;[5]

- ***no-WRITE*** *if no processes are executing a $WRITE$ (according to S_{proc});*
- ***deceiving*** *if it is both possibly-no-pending and no-WRITE.*

Note that a no-WRITE configuration may contain pending operations, because a process may have finished its $WRITE$ while some operation is still pending. Also note that if a configuration is possibly-no-pending then a process p is required to complete its $READ$ or $WRITE$ even if a single base register never responds and no pending operations are flushed. In contrast, if a configuration is not possibly-no-pending, then another process q may have a pending operation on some register $r' \neq r$, and p might wait for either r to respond or for the pending operation on r' to be flushed, since p knows that it is illegal for both r and r' to crash.

A no-WRITE configuration has the notion of a "current value" of the REGISTER. This is a value that we could get if we ran a $READ$ from that configuration. This value need not be unique.

We now construct a run in which we progressively get into deceiving configurations that have an increasing number of pending writes. We first gather lots of pending writes to the same register, and then explain how to gather pending writes to more and more registers until all s of them have at least one pending write. Once we have done that, we get a contradiction as follows: we execute a solo $WRITE$ and then completely erase all the effects of the $WRITE$ by flushing all pending writes.

So here is the run construction. We will use the following:

LEMMA 2.1. *If S is deceiving then we can extend S to another configuration S' that is deceiving and contains one more pending operation than S.*

PROOF. Starting from configuration S, for each new process p (one that has not executed yet), consider a run continuation in which p attempts to $WRITE$ a "new" value (i.e., a value different from a "current value" of the REGISTER). In such a continuation, note that p needs to attempt to write to some register (else it is easy to get a contradiction) and let r_p be the register to which p tries to write first. Since there are only s base registers, we can find two distinct new processes p and q such that $r_p = r_q$. Now starting from S again, p executes the $WRITE$ until it covers r_q, i.e., until it is just about to write to r_q but we stop p before it issues the write. Now q executes the $WRITE$ to completion such that the write to r_p is left pending. Finally, we let p complete its $WRITE$ (i.e., p writes to r_p and then continues executing its WRITE code until completion). Let S' be the resulting configuration. Note that S' has a new pending write of process q to r_p. In addition, S' is no-WRITE (since S was no-WRITE and both p and q completed their $WRITES$) and possibly-no-pending (because it is possible in a different run that q flushed its pending write to r_p right before p wrote to it). Thus, S' is deceiving. □

We now construct a run R as follows. Start with the empty run, and notice that the initial configuration is deceiving. Now apply Lemma 2.1 sufficiently many times until there exists a single register r_1 with $bignum_1$ pending writes, where $bignum_1$ is a large number whose exact value is irrelevant now. This gets us to a deceiving configuration S^1. We now show how to get to a large number of pending operations on a different register.

LEMMA 2.2. *Suppose that S is any deceiving configuration that contains a pending write on some register r. If a new process executes a $WRITE$ of a new value starting from S then it must attempt to write to a register different than r.*

PROOF. If the $WRITE$ never writes or only writes to r then we can flush the write to r after the $WRITE$ completes and this would completely hide the $WRITE$ and would easily give us a contradiction. □

We now show how to extend S^1 to get a pending write to a register different from r_1.

LEMMA 2.3. *If S is deceiving and contains $i \geq s+1$ pending writes to some register r then we can extend S to another configuration S' that is deceiving and contains (a) a pending write to a register different than r and (b) $i-2$ pending writes to r.*

PROOF. Take a subset X of $s+1$ pending writes to r and for each $x \in X$ choose a distinct new process q_x. Consider the continuation from S in which we first flush x and then q_x executes a $WRITE$ of a new value. Now we can apply Lemma 2.2 because if we take any deceiving configuration and flush one of its pending writes we still get a deceiving configuration. So, by Lemma 2.2, q_x attempts to write to some register different from r. Let r_x be the first such a register. Since $|X| = s+1$ and there are only s base registers, we can find distinct $x, y \in X$ such that $r_x = r_y$. The rest of the proof is now quite similar to that of Lemma 2.1. More precisely, consider an extension of S in which we (1) flush x, (2) make q_x execute the $WRITE$ of the new value, but we suspend q_x when it covers r_x, (3) flush y, (4) make q_y execute the $WRITE$ of the new value to completion, while leaving the write to r_x pending and (5) let q_x complete its $WRITE$. Let S' be the resulting configuration. Then clearly S' contains (a) a pending write to $r_x \neq r$ and (b) two less pending writes to r than S. Moreover, S' is deceiving. □

[5] Note that when we say "dropped" we do **not** mean that the operations have been flushed, since flushing them would change the register state. We require S' to have exactly the same register state as S.

We now continue constructing our run R starting from S^1. We apply Lemma 2.3 sufficiently many times until we get $bignum_2$ writes pending on a single register r_2, where $bignum_2$ is a big number whose value is now irrelevant. Let S^2 be the resulting configuration. Note that S^2 is now a deceiving configuration with many pending writes to both r_1 and r_2.

We now explain how to proceed by induction. Suppose that S^k is a deceiving configuration with many pending writes to $r_1, \ldots, r_k$. We explain how to get to a deceiving configuration S^{k+1} with many pending writes to $r_1, \ldots, r_{k+1}$.

LEMMA 2.4. *Suppose S is any deceiving configuration that contains at least one pending operation on each register $r_1, \ldots, r_k$. If a new process executes a $WRITE$ of a new value starting from S then it must attempt to write to a register different than $r_1, \ldots, r_k$.*

PROOF. Similar to the proof of Lemma 2.2. □

LEMMA 2.5. *If S is deceiving and contains $n_1, \ldots, n_k \geq s+1$ pending writes to each of registers $r_1, \ldots, r_k$, respectively, then we can extend S to another configuration S' that is deceiving and contains (a) a pending write to a register different than $r_1, \ldots, r_k$ and (b) $n_1 - 2, \ldots, n_k - 2$ pending writes to $r_1, \ldots, r_k$, respectively.*

PROOF. Create a set X of $s+1$ vectors of distinct pending writes, where each vector has a write to each of the registers $r_1, \ldots, r_k$. For each vector $x \in X$ choose a distinct new process q_x. Consider the continuation from S in which we first flush all k writes in x and then q_x executes a $WRITE$ of a new value. Apply Lemma 2.4 and let r_x be the register different than $r_1, \ldots, r_k$ that q_x writes first. Since $|X| = s+1$ and there are only s base registers, we can find distinct $x, y \in X$ such that $r_x = r_y$. Consider an extension of S in which we (1) flush all writes in x, (2) make q_x execute the $WRITE$ of the new value, but we suspend q_x when it covers r_x, (3) flush all writes in y, (4) make q_y execute the $WRITE$ of the new value to completion, while leaving the write to r_x pending and (5) let q_x complete its $WRITE$. Let S' be the resulting configuration. Then clearly S' contains (a) a pending write to $r_x \neq r_1, \ldots, r_k$ and (b) two less pending writes to $r_1, \ldots, r_k$ than S. Moreover, S' is deceiving. □

We now apply Lemma 2.5 $(s-k)(bignum_{k+1} - 1) + 1$ times, so that we can find some register $r_{k+1} \neq r_1, \ldots, r_k$ with $bignum_{k+1}$ pending writes. We let S^{k+1} to be the resulting configuration, and note that S^{k+1} is a deceiving configuration with many pending writes to each of $r_1, \ldots, r_{k+1}$.

We apply this inductive construction until $k = s+1$ and we get our contradiction because there are not $s+1$ distinct registers in the system.

It remains for us to give a precise value for $bignum_j$ for $j = 1, \ldots, s+1$. We let $bignum_{s+1} = 1$ and $bignum_s = s+1$. For $1 \leq i \leq s-1$, we define it inductively backwards:

$$bignum_{i-1} = \left\{ \sum_{j=i}^{s+1} 2\left[(s-j-1)(bignum_j - 1) + 1\right] \right\}$$

where each term in the summand is the number of times we need to flush register r_{i-1} when creating $bignum_j$ pending writes on register r_j. Then $bignum_i = O(s^s)$ for every i.

4.2 Possibility result

We now provide a uniform implementation a SWMR register with only three SWMR registers, one of which may fail. Our implementation can be easily generalized to tolerate t register failures, by using $2t+1$ rather than three SWMR registers.

The $WRITER$ keeps a sequence number s. To $WRITE$ a value v, the $WRITER$ increases s, issues a write of (v, s) to all three base registers[6], and waits for the completion of two of them. The algorithm to $READ$ a value has two phases, which we call "choose-value" and "wait" phases. In the "choose-value" phase, the $READER$ issues a read of all three base registers and waits for two of them to complete. Let (v_0, s_0) be the value with largest sequence number. Then, in the "wait" phase the $READER$ keeps reading all three registers until two of them have a sequence number at least as big as s_0. The $READER$ then returns v_0 (the value chosen in the "choose-value" phase).

In the full paper, we show that this implementation is correct.

5. SEQUENTIALLY CONSISTENT REGISTERS

We now consider uniform wait-free implementations of a sequentially consistent register, rather than atomic. Sequentially consistent registers are easier to implement than atomic ones, but harder to show impossible to implement. This is because they do not require a $READ$ to return the latest $WRITE$ if it completes. For example, if a process $WRITES$ 0 and then $WRITES$ 1, and later on another process $READS$, the latter need not $READ$ 1: it is allowed to $READ$ 0. The only requirement is that there be an enumeration of the operations that preserves local order. In the example above, the enumeration is $WRITE$ 0, $READ$ 0, $WRITE$ 1.

We show that it is impossible to uniformly implement a SWMR register (with finitely many fail-prone base registers), while it is possible to implement a MWSR register. These results imply a fortiori that it is impossible to implement a MWMR register, and it is possible to implement a SWSR register, so that the implementability situation is the one in Table 3.

[6]As before, if one of the registers has a pending write issued during a previous $WRITE$, then the $WRITER$ just forks a background task to issue the write as soon as all previous writes have finished.

5.1 Sequential consistency

Roughly speaking, an *execution is sequentially consistent* if there exists an enumeration of its operations ($READS$ and $WRITES$) that (1) is consistent with the sequential specification of the object (e.g., for registers, the specification is that a $READ$ always returns the value written by the previous $WRITE$), and (2) respects local order (i.e., if the same process p executes OP_1 before OP_2 then OP_1 is before OP_2 in the enumeration). We call this enumeration a *serialization* of the operations. An *implementation of a sequentially consistent object* is one that always produces sequentially consistent executions.

We require the latter definition to hold even for executions with infinitely many operations. For if we required it only for finite executions, then there would exist a trivial and useless sequentially consistent implementation of a MWMR REGISTER without any shared registers, as follows: each process p has a local variable v_p that keeps the last value that p $WRITES$ to the REGISTER (initially, v_p is the initial value of the REGISTER). To $READ$ the REGISTER, process p simply returns the value in v_p.

This implementation is useless because a process never $READS$ a value $WRITTEN$ by another process. Yet, any of its finite executions can be easily serialized, by ordering initial $READS$ that return the initial value of the REGISTER at the beginning, and then, for the other operations, simply juxtaposing the local sequence of operations of each process.

Now consider the infinite execution in which p $WRITES$ 1 and q repeatedly forever $READS$ 0, the initial value of the REGISTER. Then there is no way to enumerate the operations in a way that is consistent with the sequential specification of registers: in any enumeration, the $WRITE$ needs to be in a certain finite position, say k, and so there are at most $k-1$ $READS$ that may return 0. Thus, the trivial implementation above is incorrect using the definition that requires infinite executions to be serializable. This definition requires the following liveness property to hold: in any execution with a finite number of $WRITES$ and an infinite number of $READS$, eventually all $READS$ must always return the value written by the last $WRITE$ to be serialized. Our proofs will frequently make use of this property.

5.2 Impossibility result

In the full paper, we show the following theorem:

THEOREM 3. *Let S be a finite set of wait-free atomic MWMR registers and suppose that one of them may fail by crashing. Then there is no uniform wait-free implementation of a sequentially consistent SWMR register using S.*

5.3 Possibility result

Figure 2 shows a uniform wait-free implementation of a sequentially consistent MWSR REGISTER that uses only three atomic MWMR registers r_1, r_2, and r_3, one of which

Initialization of shared variables:
 $r_1 \leftarrow (0,0,0)$, $r_2 \leftarrow (0,0,0)$, $r_3 \leftarrow (0,0,0)$

WRITER processes q:
 Initially: $seq_q \leftarrow 0$
 To $WRITE(v)$:
  $seq_q \leftarrow seq_q + 1$
  issue $write(q, seq_q, v)$ to r_1, r_2 and r_3
  wait for two of the writes to complete

READER process p:
 Initially: $lastv \leftarrow 0$, $seqs[r] \leftarrow 0$ for all processes r
 To $READ$:
  issue $read$ to r_1, r_2 and r_3
  wait for two of the reads to complete
  let (r^1, s^1, v^1) and (r^2, s^2, v^2) be the values read
  if $s^j > seqs[r^j]$ for some j **then**
   (* it does not matter which j we pick above *)
   $seqs[r^j] \leftarrow s^j$
   $lastv \leftarrow v^j$
  return $lastv$

Figure 2: Uniform wait-free implementation of a MWSR sequentially consistent register using three base registers.

may fail. It is easy to generalize this implementation to allow t register failures rather than one, by using $2t+1$ registers rather than three.

Each $WRITER$ q keeps a sequence number in its local variable seq_q. To $WRITE$ v, q increments seq_q, issues a write (q, seq_q, v) to r_1, r_2 and r_3, and waits for two of them to complete[7]. The $READER$ p keeps two local variables: $lastv$ and an array $seqs$ indexed by process id (this is an infinite array, but it is not a shared variable). To $READ$ a value, p issues a read to r_1, r_2 and r_3, waits for two of them to complete, and checks if it sees any read triple (r^j, s^j, v^j) is such that s^j is strictly greater than $seqs[r^j]$. If so, p picks one such a triple (it does not matter which), replaces $seqs[r^j]$ with s^j, and replaces $lastv$ with v^j. Then p returns $lastv$ as the value $READ$.

In the full paper, we show that this implementation is correct.

6. INFINITELY MANY BASE REGISTERS

Now suppose that there are infinitely many base registers, of which up to some known number t can fail. In this case, we show that there is a uniform implementation of any of the three register types (wait-free atomic, atomic, and wait-free sequentially consistent) for either SWSR, MWSR, SWMR, or MWMR registers. We only give an implementation for the strongest type (wait-free atomic MWMR register) since it implies the rest. So the implementability situation is the

[7] As before, if one of the registers has a pending write issued during a previous $WRITE$, then the $WRITER$ just forks a background task to issue the write as soon as all previous writes have finished.

one shown in Table 4.

For the strongest result, we consider the *infinite-arrival* model, where an infinite number of processes may take steps in any run, while at the same time the concurrency (the number of processes that take steps simultaneous) in any single state is finite. We say that the concurrency is unbounded, if in each prefix of a run R the concurrency is bounded, but there may be no finite bound over the infinite run R. Our algorithm works with unbounded concurrency. It relies on two building blocks, *one-shot register* and *name snapshot*, which we now explain and show how to implement.

One-shot register

Our algorithm uses a special type of Single-Writer Multi-Reader register that we call *one-shot* register. As it name implies, such a register allows a value to be written to it only once. Before a value is written, the register has its initial value, which is a constant.

It turns out to be very easy to implement such registers in our model. The implementation uses $2t + 1$ base registers located in different disks, where t is the number of disks that can be faulty. These registers are initialized with the initial value of the one-shot REGISTER. To $WRITE$ v the $WRITER$ writes v to all base registers and waits for $t + 1$ of them to complete. To $READ$, a $READER$ reads from $t + 1$ registers. If all reads return the initial value of the REGISTER, the $READ$ returns such a value. Else, let v be the value read different from the initial value (there can be only one such value v because there is at most one $WRITE$). The $READER$ then writes v to $2t + 1$ base registers and waits for $t + 1$ of them to complete. The $READER$ then returns v as the value $READ$.

Name snapshot

Roughly speaking, name snapshot [28] provides a list of processes that are participating in the algorithm; different processes may not exactly agree on this list, but different lists are always related by inclusion. More precisely, at any time a process i may start a snapshot, and when it terminates it outputs a set of processes S_i, called the *snapshot of* i, such that the following properties hold:

- *(Validity)* The snapshot of i contains itself, that is, $i \in S_i$
- *(Total Ordering)* Snapshots form an inclusion chain, that is, for any i and j either $S_i \subseteq S_j$ or $S_j \subseteq S_i$
- *(Integrity)* If j does not start by the time i terminates then j is not in the snapshot of i.

Note that Validity, Total Ordering and Integrity imply that if a process i terminates before j starts then the snapshot of j contains i.

An implementation of name snapshot for the infinite arrival model is given in [28]. We will not repeat this implementation here, but the important point is that it uses three infinite arrays of fail-free shared registers, $snap[1...\infty]$, $start[1...\infty]$, and $flag[1...\infty]$. We now argue that such arrays are implementable in our model. Indeed, $snap[i]$ and $start[i]$ are one-shot registers, which we have just shown how to implement. And $flag[i]$ is a boolean MWMR register that can be written to multiple times, but if it is written to more than once then all writes are for the same value. This type of register can be implemented with the same implementation given above for one-shot registers.

Therefore, by plugging in these implementations of $snap$, $start$, and $flag$, we get an implementation of name snapshot that works in our model, where up to t disks can be faulty. This implementation requires $d = 2t + 1$ disks, where each disk has an infinite number of registers.

Code for process p:

To $READ$:
  $S \leftarrow snapshot()$
  $T \leftarrow \{q \in S$ such that $v[q] \neq \emptyset\}$
  if $T = \emptyset$ **then return** initial value
  $q \leftarrow$ process in T with largest $v[q].snapshot$
  return $v[q].value$

To $WRITE(val)$:
  $S \leftarrow snapshot()$
  $tmp.value \leftarrow val$
  $tmp.snapshot \leftarrow S$
  $v[p] \leftarrow tmp$

Figure 3: Uniform wait-free implementation of an atomic MWMR register using infinitely many base registers.

MWMR register

The algorithm that implements a MWMR register is shown in Figure 3. To $WRITE$, we take a name snapshot (as explained above), and then store the snapshot and the value being written to a vector v of one-shot registers. To $READ$, we also take a snapshot and then remove those processes that have empty entries in the vector v (these are processes that have either previously $READ$ a value, or have started their $WRITE$ but have not yet completed it). Among the processes left, we then choose the one with the largest snapshot in inclusion order (we break ties in any deterministic fashion). We return the value written by such a process.

Note that this implementation allows a given process to $WRITE$ to the MWMR register only once. As in [28], it is easy to transform this implementation into one that allows a process to $WRITE$ multiple times, by assigning each process a name for each new operation. We do this by reserving infinitely many integer names for each process. Whenever a process performs another $READ$ or $WRITE$, it uses a new unused name.

THEOREM 4. *Let $D_1, \ldots, D_d$ be infinite sets of wait-free atomic MWMR registers and suppose that every register of*

up to t of these sets may fail by crashing, where $d \geq 2t+1$. Then there exists a uniform, wait-free implementation of an atomic MWMR register using $D_1, \ldots, D_d$.

We now show this theorem, by proving that the algorithm in Figure 3 implements an atomic (linearizable) register. To do so, we show how to assign each $READ$ and $WRITE$ operation to a point in time between the operation's invocation and response, in such a way that if we order operations according to their assigned time, we get a consistent history. In what follows, we say that a $WRITE$ completes when it writes a value to $v[p]$ [8], and we say that a $READ$ completes when it returns a value.

Assignment of WRITES: Consider an operation $OP = WRITE(val)$, and consider the time when OP completes. At this time, there are two cases to consider:

- *Case 1: no other $WRITES$ that have a larger snapshot have completed.* In this case, we assign OP to the time OP completes.
- *Case 2: there are $WRITES$ with larger snapshot values that have completed.* Among such $WRITES$, let OP_2 be the one with largest snapshot. We assign OP to the time right before OP_2 completes[9]. For this to make sense, we need to show that this time T is between the invocation of OP and its response. Suppose it is not. Then T must be earlier than OP's invocation. But OP_2 takes a snapshot before time T—and thus before OP takes a snapshot. Thus, the snapshot of OP_2 does not include p. This is a contradiction to the definition of OP_2. Thus, T is between the invocation of OP and its response.

Assignment of READS: Consider an $OP = READ$. Let S be the snapshot that OP obtains and let T be the set of processes in S from which OP reads a non-empty value for v. Note that all processes in T have previously performed a $WRITE$ operation, since only $WRITES$ assign values to v. Let p_2 be the process with the largest snapshot in T and let OP_2 be the $WRITE$ performed by p_2.

- *Case 1: OP_2's assigned time is before p starts OP.* In this case, we assign OP to the time it starts. We claim that no $WRITES$ are assigned between the assigned time of OP_2 and the invocation of OP. Indeed, in order to obtain a contradiction, suppose not. Then there is a $WRITE$ operation OP_3 by some process p_3, such that OP_3 has a larger snapshot than OP_2 and OP_3 completes between the assigned time of OP_2 and the invocation of OP. Then OP_3's snapshot does not include p (since p only takes a snapshot later), and so OP's snapshot includes p_3. Moreover, when OP reads v, OP_3 has already completed, so OP gets a non-empty value of $v[p_3]$. Thus, p_3 is in T. Since OP_3's snapshot is larger than OP_2's, that means that p cannot choose p_2 as the process in T with the largest snapshot—a contradiction that shows the claim.
- *Case 2: OP_2's assigned time is after p starts OP.* In this case, we assign OP to the time right after OP_2 is assigned. We claim that this time t is between the invocation and response of OP. Indeed, t is after the invocation by assumption. Moreover, if t were after OP's response then p_2 only writes to $v[p_2]$ after p completes OP. Thus, p cannot read a non-empty value from $v[p_2]$—a contradiction to the fact that p_2 is in T.

[8] In this terminology, the $WRITE$ may complete before the $WRITE$ returns since the process may block for arbitrarily long before returning.

[9] If other $WRITES$ have been assigned to the time "right before" OP_2 completes, it does not matter which one is assigned first.

7. CONCLUSION

In this paper we have studied the uniform implementability of objects from objects that may fail. We show than an object as simple as a Multi-Writer Multi-Reader register does not have a uniform self implementation using finitely many base registers. Such a result is applicable to systems like Storage Area Networks, where fail-prone disks are directly attached to the network so that the set of processes accessing the disk is unknown and possibly very large. An interesting research question is what problems can be solved in this setting. This paper has shed some light on this question.

8. ACKNOWLEDGMENTS

We are grateful to Leslie Lamport and Mark Lillibridge for helpful discussions on our impossibility results. We are also grateful to the PODC referees for useful comments.

9. REFERENCES

[1] A. Acharya, M. Uysal and J. Saltz. Active Disks: Programming model, algorithms and evaluation. *Proc. of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pp. 81-91, 1998.

[2] Y. Afek, H. Attiya, A. Fouren, G. Stupp and D. Touitou. Long-Lived Renaming made adaptive. *Proc. of 18th Annual ACM Symp. on Principles of Distributed Computing*: 91-103, May 1999.

[3] Y. Afek, H. Attiya, G. Stupp and D. Touitou. Adaptive long-lived renaming using bounded memory. *Proc. of the 40th IEEE Ann. Symp. on Foundations of Computer Science*, pages 262-272, October 1999.

[4] Y. Afek, D. S. Greenberg, M. Merritt and G. Taubenfeld. Computing with faulty shared memory. *in Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing*. (Vancouver, B.C., Canada). ACM New York, pp. 47-58, 1992.

[5] Y. Afek, D. S. Greenberg, M. Merritt and G. Taubenfeld. Computing with faulty shared objects. *J. ACM, 42*, 6, 1231-1274, 1995.

[6] Y. Afek, D. Dauber and D. Touitou. Wait-free made fast. *Proc. of the 27th Ann. ACM Symp. on Theory of Computing*:538-547, May 1995.

[7] Y. Afek and M. Merritt. Fast, wait-free $(2k-1)$-renaming. In *Proc. of the 18th Ann. ACM Symp. on Principles of Distributed Computing*, pages 105-112, May 1999.

[8] Y. Afek, M. Merritt, G. Taubenfeld and D. Touitou. Disentangling multi-object operations. In *Proc. of 16th Annual ACM Symp. on Principles of Distributed Computing*, pages 111-120, August 1997.

[9] Y. Afek, G. Stupp and D. Touitou. Long lived adaptive collect with applications. *Proc. of the 40th IEEE Ann. Symp. on Foundations of Computer Science*, pages 262-272, October 1999.

[10] Y. Afek, G. Stupp and D. Touitou. Long lived adaptive splitter and applications. Unpublished manuscript, December 1999.

[11] Y. Afek, G. Stupp and D. Touitou. Long lived and adaptive atomic snapshot and immediate snapshot. *Proc. of the 19th Ann. ACM Symp. on Principles of Distributed Computing*, pages 71-80, 2000.

[12] J. Anderson and Y.-J. Kim. Adaptive mutual exclusion with local spinning. *Proceedings of the 14th International Conference, DISC 2000*, pages 29-43, October 2000.

[13] D. Angluin. Local and global properties in networks of processors. In *Proceedings of the 12th ACM Symposium on Theory of Computing (STOC 1980)*, pages 82-93.

[14] J. Aspnes, G. Shah and J. Shah. Wait free consensus with infinite arrivals. *Proc. of the 34th Annual ACM Symposium on the Theory of Computing*, pages 524-533, May 2002.

[15] H. Attiya and V. Bortnikov. Adaptive and efficient mutual exclusion. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC 2000)*, pages 91-100, 2000.

[16] H. Attiya and E. Dagan. Universal operations: Unary versus binary. In *Proc. 15th Annual ACM Symp. on Principles of Distributed Computing*, pages 223-232, May 1996.

[17] H. Attiya and A. Fouren. Adaptive wait-free algorithms for lattice agreement and renaming. In *Proc. 17th Annual ACM Symp. on Principles of Distributed Computing*, pages 277-286, June 1998.

[18] H. Attiya and A. Fouren. Adaptive long-lived renaming with read and write operations. Technical Report 0956, Faculty of Computer Science, Technion, Haifa, 1999.

[19] H. Attiya and A. Fouren. Algorithms adaptive to point contention. *JACM*, 2001, submitted for publication.

[20] H. Attiya, A. Fouren and E. Gafni. An adaptive collect algorithm with applications. *Distributed Computing*, to appear.

[21] J. Burns and N. Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation* 107(2):171-184, December 1993.

[22] G. Chockler and D. Malkhi. Active Disk Paxos with infinitely many processes. *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing (PODC 2002)*, pages 78-87, 2002.

[23] M. Choy and A.K. Singh. Adaptive solutions to the mutual exclusion problem. *Distributed Computing*, 8(1), pages 1-17, 1994.

[24] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.

[25] A. Fekete, M. Frans Kaashoek and N. Lynch. Implementing sequentially consistent shared objects using broadcast and point-to-point communication. *Journal of the ACM*, 45(1), pp. 35-69, 1998.

[26] E. Gafni. A simple algorithmic characterization of uniform solvability. *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science (FOCS 2002)*, pp. 228-237, 2002.

[27] E. Gafni and L. Lamport. Disk Paxos.In M. Herlihy, editor, *Proceedings of the 14th International Conference on Distributed Computing (DISC 2000)*, volume 1914 of *Lecture Notes in Computer Science*, pages 330-344, 2000.

[28] E. Gafni, M. Merritt and G. Taubenfeld. The concurrency hierarchy and algorithms for unbounded concurrency.In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing (PODC 2001)*, pages 161-169, 2001.

[29] M. Inoue, S. Umetani, T. Masuzawa and H. Fujiwara. Adaptive long-lived $O(k^2)$ renaming with $O(k^2)$ steps. *Proceedings of the 15th International Conference on Distributed Computing (DISC 2001)*, pages 123-135, 2001.

[30] A. Itai and M. Rodeh. Symmetry breaking in distributed networks. *Information and Computation*, 88(1): 60-87, 1990.

[31] P. Jayanti, T. D. Chandra and S. Toeug. Fault-Tolerant wait-free shared objects. *Journal of the ACM*, Vol. 45, No. 3, pp. 451-500, 1998.

[32] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1-11, 1987.

[33] M. Merritt and G. Taubenfeld. Speeding Lamport's fast mutual exclusion algorithm. *Information Processing Letters*, 45:137-142, 1993.

[34] M. Moir and J. H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Sci. Comput. Programming*, 25(1):1-39, October 1995.

[35] G.L. Peterson. Time efficient adaptive mutual exclusion algorithms. Unpublished manuscript, 2001.

[36] E. Riedel, C. Faloutsos, G. A. Gibson and D. Nagle. Active Disks for large scale data processing. *IEEE Computer*, June 2001.