Active Disk Paxos

with infinitely many processes *

Gregory Chockler
School of Computer Science and Engineering
The Hebrew University of Jerusalem
Jerusalem, Israel 91904
grishac@cs.huji.ac.il

Dahlia Malkhi
School of Computer Science and Engineering
The Hebrew University of Jerusalem
Jerusalem, Israel 91904
dalia@cs.huji.ac.il

ABSTRACT

We present an improvement to the Disk Paxos protocol by Gafni and Lamport which utilizes extended functionality and flexibility provided by *Active Disks* and supports unmediated concurrent data access by an unlimited number of processes. The solution facilitates coordination by an infinite number of clients using finite shared memory. It is based on a collection of read-modify-write objects with faults, that emulate a new, reliable shared memory abstraction called a *ranked register*. The required read-modify-write objects are readily available in Active Disks and in Object Storage Device controllers, making our solution suitable for state-of-the-art Storage Area Network (SAN) environments.

1. INTRODUCTION

In recent years, advances in hardware technology have made possible a new approach for storage sharing, in which clients access disks directly over a *storage area network* (SAN). In a SAN, disks are directly attached to high speed networks that are accessible to clients. The clients access raw disk data, which is mediated by disk controllers with limited memory and CPU capabilities. Clients run file system services and name servers on top of raw I/O. Since clients (or a group of designated SAN servers) need to coordinate and secure their accesses to disks, they need to implement distributed access control and locking for the disks. However, once a client obtains access to a file, it accesses data directly through the SAN, thus eliminating the slowdown bottleneck at the file system server. IBM's *Storage Tank* [7] is an example of a commercially available SAN system that solves many of the coordination, sharing and security issues involved with SANs (More examples are given in Section 2). In this paper, we tackle the issue of scaling the number of clients that are served by a SAN.

As in many other distributed settings, a fundamental enabler in this environment for clients to coordinate their actions is an agreement protocol. It is well known that in order to solve agreement in a non-blocking manner three phases are needed [43, 44]. This leads to the usage of the Paxos protocol [31, 32, 15, 34] and its variants, as is done, e.g., in Petal [35] and Frangipani [46]. Briefly, the Paxos protocol is a 3-phase commit protocol that uses the 1st phase to determine a proposition value, the 2nd phase to fix a decision value, and the 3rd phase to commit to it. The Paxos protocol was recently adapted for utilization by SAN clients in the Disk Paxos protocol [20]. Both the original Paxos protocol and its Disk variant are geared toward a fixed and known number of clients. In particular, in Disk Paxos, each client must use a pre-designated area on disk to write values, and must read the values written by all other potential clients. Consequently, adding new clients to the system is a costly operation that involves real-time locking [20]. Also, the complexity of memory (disk) operations does not scale with the number of clients.

In contrast, we provide an adaptation of Paxos that supports infinitely many clients. Our solution builds on a strengthening of the disk model which is driven by current technological advances in the storage area. Our use of strong memory objects is further justified by the impossibility result of Section 5.3, that shows that even in failure-free runs, finite read/write memory is insufficient for solving agreement among infinitely many processes[1]. Hence, to provide a solution which is realistic in practice, we employ stronger memory objects. This approach is motivated by recent development in controller logic that enhances the functionality of disks for SAN and provide for *Active Disks*, capable of supporting stronger semantics objects (see, e.g., [22]). In particular, specialized functions that require specific semantics not normally provided by drives can be provided by remote functions on Active Disks. Examples include a *read-modify-write* operation, or an atomic *create* that both creates a new file object and updates the corresponding directory object. Such advanced operations are already used for optimization of higher-level file systems such as NFS on NASD [23].

The existence of strong shared memory objects does not ob-

*This work was supported in part by the Israeli Ministry of Science grant #1230-3-01

PODC '02 Monterrey, California USA

[1] Section 5.3 actually provides a stronger result, proving impossibility of constructing a different type of object than a consensus object. By the universality of the consensus object[27], this a fortiori implies impossibility of constructing agreement.

viate the need for an agreement protocol. Admittedly, if we had even one reliable disk with read-modify-write operations, we could leverage coordination off it to solve agreement, as shown by Herlihy in [27]. However, in a scalable SAN, disks will frequently become unavailable. Unfortunately, it is impossible to use a collection of fail-prone read-modify-write objects to emulate a reliable one [28]. Hence, our construction is necessarily more involved. It should be noted that using a farm of disks also has the benefit of distributing client accesses among multiple disks in order not to introduce unnecessary contention. Hence, our solution provides for both high availability, and for load sharing among disks.

Our solution first breaks the Paxos protocol using an abstraction of a shared object called a *ranked register*, which is driven by a recent deconstruction of Paxos by Boichat et al. in [6]. Briefly, a ranked register supports rr-$read$ and rr-$write$ operations that are both parameterized by an integer (the rank/ballot). The main property of this object is that a rr-$read$ with rank r_1 is guaranteed to "see" any completed rr-$write$ whose rank r_2 satisfies $r_1 > r_2$. In order for this property to be satisfied, some lower ranked rr-$write$ operations that are invoked after a rr-$read$ has returned must *abort*. Armed with this abstract shared object, we show the following two constructions:

1. We provide a simple implementation of Paxos-like agreement using the abstraction of one reliable shared ranked register that supports infinitely many clients. Briefly, in this implementation (see Figure 1) a participating client chooses a (unique) rank, rr-$read$'s the ranked register with it, and then writes the ranked register either with the value it read (if exists) or with its own input. If the rr-$write$ operation succeeds (i.e., it does not abort), then the process decides on the written value. Else, it retries with a higher rank.

2. The reliable shared ranked register abstraction cannot be supported for an unbounded number of clients using only finite read/write memory (proof is provided in Section 5.3). Furthermore, no single fail-prone disk with even stronger semantics object may implement it. Therefore, we provide an implementation of a ranked register shared among an unbounded number of clients. The implementation employs a farm of disks, each of which supports one read-modify-write register, of which a threshold may experience non-responsive crash faults. The fault tolerant emulation performs each rr-$read$ or rr-$write$ operation on a majority of the disks, and takes the maximally ranked result as the response from an operation. The number of participating disks required for the emulation is determined only by the level of desired fault tolerance, and the memory on each one is constant, regardless of the number of participating clients.

Our approach is readily implementable is SAN with Active disks. To this extent, it may serve as an important specification of the kind of functionality that is desired by SAN clients and that disk manufacturers may choose to provide.

Additionally, our approach faithfully represents another realistic setting, the classic client-server model, with a potentially very large and dynamic set of clients. This is the setting for which scalable systems like the Fleet object repository [37] were designed. In this setting, a highly available service is implemented by a replicated set of servers, a threshold of which may be faulty. Clearly, each server is capable of implementing stronger-semantics objects, e.g., a single shared ranked register, that is accessible by any number of clients. Thus, our paradigm provides for coordination and information sharing among transient clients through the group of servers. It does not require servers to interact among themselves, and it avoids the complexity of failure monitoring and reconfiguration which is manifested, e.g., in group communication middlewares [41, 11].

2. RELATED WORK

Our work deals with solving the Consensus problem [33], one of the most fundamental problems in distributed computing. Consensus is the building block for replication paradigms such as state machine replication [30, 45], group membership (see [41, 11] for survey), virtual synchrony [5], atomic broadcast [10], total ordering of messages [29, 19], etc. Consensus is known to be unsolvable in most realistic models such as asynchronous message passing systems [18] and asynchronous shared memory with read/write registers [36, 27, 16] if even a single process can fail by crashing. While it is usually straightforward to guarantee the consistency of a consensus decision alone (safety), the difficulty is in guaranteeing progress in face of uncertainty regarding process failures. The usual approaches to circumventing Consensus impossibility include strengthening the basic model by assuming different degrees of synchrony (see e.g., [16, 17, 14]), augmenting the system with unreliable failure detectors [10], and employing randomization (see a survey in [13]). Specifically, our solution uses one of the most widely deployed implementations of the state machine replication [30, 45], the Paxos algorithm [31, 32, 15, 34]. At the core of Paxos is a consensus algorithm called *Synod*. The Synod protocol deals with the Consensus impossibility by guaranteeing progress only when the system is stable so that an accurate leader election is possible. This assumption is equivalent to assuming the Ω failure detector of [9] which was shown in [9] to be the weakest failure detector that can be used to solve Consensus.

As shown below in Section 5.3 though, when an infinite number of processes is present, even non-faulty ones, agreement is impossible to achieve using only a finite number of atomic read/write registers. Not surprisingly, the Paxos protocol is in fact designed with built-in knowledge of all of the participants. The focus of our work is on guaranteeing safety of the consensus decision in the presence of an infinite number of processes. Other results in this model and a classification based on levels of simultaneity can be found in [38, 21]. As for liveness, we can use standard approaches as above to circumvent impossibility, and we leave it outside the scope of this work.

Our usage of shared-access SAN disks as shared memory is greatly influenced by the recent Disk Paxos protocol of Gafni and Lamport [20]. In Disk Paxos, the protocol state is replicated at network attached disks some of which can crash or become inaccessible. The participating processes access the state replicas directly over a SAN. Disk Paxos assumes simple commodity disks which support only primitive read and write operations. It supports a bounded and known number of clients, and uses disk memory proportional to their number. In contrast, we stipulate Active Disks that are capable of serving higher semantics objects, which provide us with the strength needed to guarantee safe decisions in face of an unbounded number of clients. The amount of memory we utilize per disk is fixed regardless of the number of participating clients.

The environment model that faithfully reflects our setting is an asynchronous shared memory system where processes interact by means of a finite collection of shared objects some of which can be faulty [1, 28]. Similarly, the Consensus protocol of Disk Paxos, called Disk Synod, is in fact an implementation of Consensus in an asynchronous shared memory system with atomic read/write registers which can incur non-responsive crash failures. It should be noted that in [28], Jayanty, Chandra and Toueg prove that it is impossible to implement wait-free Consensus in such an environment if at most one shared object can stop responding forever. This result holds regardless of the number, size and type of the shared objects used by the implementation. Hence, merely by stipulating stronger disks we would still be unable to circumvent the impossibility. Nevertheless, we show that the ranked register is sufficient for implementing non-fault-tolerant Consensus with unbounded number of participants. A remarkable feature of the ranked register is that it allows for wait-free implementation in a shared memory system with non-responsive crash faults and therefore, can be used as a building block for implementing fault-tolerant Disk Paxos with unbounded number of processes. As before, the way to guarantee progress despite the impossibility result is by augmenting the system with a leader election primitive which is required to be eventually accurate in order for the protocol to be live.

Our ranked register abstraction was largely inspired by work of Boichat et al. [6] on deconstructing the Paxos protocol. This paper proposes a modular decomposition of Paxos based on a simple shared memory register called *round-based register*. Intuitively, both the round-based register and the ranked register encapsulate the notion of Paxos *ballots*[2] which are used by the protocol to ensure value consistency in presence of concurrent updates. While being in line with the general deconstruction idea of Boichat et al., our ranked register nevertheless provides much weaker guarantees and supports a slighter different interface.

2.1 SAN technology

Our work was motivated by advances in storage technology and the SAN paradigm. A storage area network enables cost-effective bandwidth scaling by allowing the data to be transferred directly from network attached disks to clients so that the file server bottleneck is eliminated. The Network Attached Secure Disks (NASD) [22] of CMU is perhaps the most comprehensive joint academy-industry project which laid the technological foundation of network attached storage systems. NASD introduced the notion of an *object storage device (OSD)* which is a network attached disk that exports variable length "objects" instead of fixed size blocks. This move was enabled by recent advances in the Application Specific Integrated Circuit (ASIC) technology that allows for integration sophisticated special-purpose functionality into the disk controllers. The NASD project also addresses other aspects of the network attached disk technology such as file system support [23], security [24] and network protocols [22].

Active Disks [42, 2] is a logical extension of the OSD concept which allows arbitrary application code to be downloaded and executed on disks. One of the applications of the active disks technology is enhancing disk functionality with specialized methods, such as atomic read-modify-write, that can be used for optimization and concurrency control of higher-level file systems.

Issues concerned with data management in SAN based file systems, such as synchronization, fault tolerance and security, are investigated in [7] in the context of IBM Storage Tank project.

Other work which addresses scalability and performance issues of network storage systems (not necessarily concerned with network attached disks) include NSIC's Network-Attached Storage Device project [40], the Netstation project [25] and the Swarm Scalable Storage System [26]. Petal [35] is a project to research highly scalable block-level storage systems. Frangipani [46] is a scalable distributed file system built using Petal. xFS: Serverless Network File Service [4] attempts to provide low latency, high bandwidth access to file system data by distributing the functionality of the server (e.g. cache coherence, locating data, and servicing disk requests) among the clients.

Concurrency control was identified as one of the critical issues in the network attached storage technology because of inherent lack of a central point of coordination [3]. The concurrency control in the Petal [35] virtual disk storage system and the Frangipani [46] file system is achieved using replicated lock servers which utilize Paxos for consistency. Consequently, Disk Paxos is a natural candidate for enabling lock management in network attached storage systems. In this paper we show that by enhancing network attached disk functionality with two simple read-modify-write operations, which are realistic to support with the OSD and Active Disk technologies, it is possible both to adapt Disk Paxos to support an unbounded number of clients and to reduce its communication cost.

3. SYSTEM MODEL

We consider an asynchronous shared memory system consisting of a countable collection of client processes interacting with each other by means of a finite collection of shared objects. The processes are designated by numbers $1, 2, \ldots$. Client may fail by stopping (crashing). The implementation should be wait-free in the sense that the progress of each non-faulty client should not be prevented by other clients concurrently accessing the memory as well as by failures incurred by other clients.

Operations on memory objects have non-zero duration, commencing with an invocation request and ending with a response. We assume that the sequence of requests produced by a process i is *well-formed* in the sense that i never initiates a new request before it has received a response to its previously invoked request. The shared memory objects themselves may fail by crash, i.e., stop responding. As in [28], we call these *non-responsive crash faults*.

According to [28], wait-free consensus is impossible in such a setting. Therefore, similar to the Paxos approach, we over-

[2] Ballots roughly correspond to *rounds* and to *ranks* in the round-based and the ranked register respectively.

come this impossibility by assuming the existence of a separate leader election oracle. The oracle guarantees the eventual emergence of a unique non-faulty leader, though when this happens in unknown to the clients themselves.

4. PAXOS WITH INFINITELY MANY PROCESSES

In this section we present the implementation of a Paxos protocol that supports infinitely many clients. Our protocol employs a special type of shared memory register, called a *ranked register*, which for now we assume is failure-free. Later, we show how to implement a fault tolerant ranked register in our environment.

Intuitively, the ranked register encapsulates the notion of *ballots* which are used by the Paxos protocol to ensure value consistency in presence of concurrent updates. The idea of modeling the Paxos protocol this way is due to [6]. However, while the ranked register interface bears similarities to the *round-based register* of [6], its specification is weaker than that of [6]. The register provides a clean isolation of the essential properties of Paxos into a well-defined building block, thus simplifying reasoning about the protocol behavior.

4.1 The ranked register

We first define ranked register as a building block of Paxos. Let *Ranks* be a totally ordered set of ranks with a distinguished initial rank r_0, and *Vals* be a set of values. We also consider the set of pairs denoted $RVals$ which is $Ranks \times Vals$ with selectors $rank$ and $value$. A ranked register is a multi-reader, multi-writer shared memory register with two operations: $\mathsf{rr}\text{-}read(r)_i$ by process i, $r \in Ranks$, whose corresponding reply is $value(V)_i$, where $V \in RVals \cup \{\langle r_0, \bot\rangle\}$. And $\mathsf{rr}\text{-}write(V)_i$ by process i, $V \in RVals$, whose reply is either $commit_i$ or $abort_i$. Note that in contrast to a standard read/write register interface, both $\mathsf{rr}\text{-}read$ and $\mathsf{rr}\text{-}write$ operations on a ranked register take a rank as an additional argument; and its $\mathsf{rr}\text{-}write$ operation might abort, whereas the $write$ operation on a standard read/write register always commits (i.e., returns ack).

In the following discussion we often say that a $\mathsf{rr}\text{-}read$ operation R *returns* a value V meaning that the register responds with $value(V)$ in response to R. We also say that a $\mathsf{rr}\text{-}write$ operation W *commits* (*aborts*) if the register responds with $commit$ ($abort$) in response to W.

We will restrict our attention to runs in which invocations of $\mathsf{rr}\text{-}write$ on a ranked register use unique ranks. More formally, we will henceforth assume that all runs satisfy the following:

DEFINITION 1. *We say that a run satisfies* rank uniqueness *if for every rank $r \in Ranks$, there exists at most one $v \in Vals$ and one process i such that $\mathsf{rr}\text{-}write(\langle r, v\rangle)_i$ is invoked in the run.*

In practice, rank uniqueness can be easily ensured by choosing ranks based on unique process identifier and a sequence number. The main reason we use this restriction is to simplify establishing the correspondence between the values written with specific ranks and the values returned by the $\mathsf{rr}\text{-}read$ operation.

We now give a formal specification of the ranked register. We start by introducing the following definition:

DEFINITION 2. *We say that a $\mathsf{rr}\text{-}read$ operation $R = \mathsf{rr}\text{-}read(r_2)_i$* sees *a $\mathsf{rr}\text{-}write$ operation $W = \mathsf{rr}\text{-}write(\langle r_1, v\rangle)_j$ if R returns $\langle r', v'\rangle$ where $r' \geq r_1$.*

The ranked register is required to satisfy the following three properties:

PROPERTY 1 (SAFETY). *Every $\mathsf{rr}\text{-}read$ operation returns a value and rank that was written in some $\mathsf{rr}\text{-}write$ invocation or $\langle r_0, \bot\rangle$. Additionally, let $W = \mathsf{rr}\text{-}write(\langle r_1, v\rangle)_i$ be a $\mathsf{rr}\text{-}write$ operation that commits, and let $R = \mathsf{rr}\text{-}read(r_2)_j$, such that $r_2 > r_1$. Then R sees W.*

PROPERTY 2 (NON-TRIVIALITY). *If a $\mathsf{rr}\text{-}write$ operation W invoked with the rank r_1 aborts, then there exists a $\mathsf{rr}\text{-}read$ ($\mathsf{rr}\text{-}write$) operation with rank $r_2 > r_1$ which returns before W is invoked, or is concurrent to W.*

PROPERTY 3 (LIVENESS). *If an operation ($\mathsf{rr}\text{-}read$ or $\mathsf{rr}\text{-}write$) is invoked by a non-faulty process, then it eventually returns.*

Note that if the $\mathsf{rr}\text{-}write$ operation would not have been allowed to abort sometimes, then it would be impossible to satisfy all the three properties above, since once a $\mathsf{rr}\text{-}read$ operation with a rank r returns a value written by a $\mathsf{rr}\text{-}write$ operation with a rank $r' < r$, there is no way it could see the value written by a subsequent $\mathsf{rr}\text{-}write$ with a rank $r' < r'' < r$.

Also note that our ranked register specification is very weak: In particular, it allows in some situations for $\mathsf{rr}\text{-}write$ operation to commit even though there exists another previously committed $\mathsf{rr}\text{-}write$ with a higher rank. The reason for that not being a problem stems from the way the ranked register is used by the Consensus implementation in Section 4.2. In particular, each process in our Consensus implementation invokes $\mathsf{rr}\text{-}write$ only after it invokes $\mathsf{rr}\text{-}read$ with the same rank and this $\mathsf{rr}\text{-}read$ returns. Thus, the ranked register Safety property ensures that in every finite execution prefix, each value written by a committed $\mathsf{rr}\text{-}write$ must be returned by one of the $\mathsf{rr}\text{-}read$ operations with a higher rank if such exist. Consequently, in each run of the Consensus implementation, any $\mathsf{rr}\text{-}write$ operation, which is invoked after $\mathsf{rr}\text{-}read$ with a higher rank has returned, would necessarily abort.

4.2 Agreement using a ranked register

We now outline an agreement protocol which employs a shared ranked register. This formulation of the agreement protocol is identical to that of Boichat et al. in [6], and is repeated here for completeness. However, since the specification of our ranked register differs significantly from the properties of the *round-based register* of [6], we provide a different and necessary proof of correctness for the protocol employing our ranked register. In particular, we address the correctness in a setting with an unbounded number of clients.

The *decide* routine depicted in Figure 1 is executed by every client that tries to form agreement. It takes as arguments an initial value and a monotonically increasing unique rank value. It returns the agreement value or aborts. The *decide* routine is guaranteed to return an agreement value at the latest when a non-faulty leader has been elected and allowed to force a decision.

We now outline the correctness argument of the agreement algorithm. Due to space limitation, we provide only a sketch of a proof.

```
Shared: ranked register rr, read/write register decision ∈ RVals,
initially decision = ⟨r0, ⊥⟩
Local: V ∈ RVals

Process i:

Procedure DECIDE(inp), inp ∈ RVals:
    if (decision ≠ ⟨r0, ⊥⟩)
        return decision;
    V ← rr.rr-read(inp.rank)_i;
    if (V = ⟨r0, ⊥⟩) then
        V.value ← inp.value;
    V.rank ← inp.rank
    if (rr.rr-write(V)_i = commit) then
        decision ← V;
        return decision;
    else
        return abort;
    fi
```

Figure 1: Paxos using a ranked register

LEMMA 1. *For any finite execution α, let $W_0 = rr.\mathsf{rr}\text{-}write(\langle r_0, v_0\rangle)$ be the lowest ranked $\mathsf{rr}\text{-}write$ invocation which commits in α. Then, in any extension of α in which $W = rr.\mathsf{rr}\text{-}write(\langle r, v\rangle)$, $r > r_0$, is invoked, $v = v_0$.*

PROOF. Our proof strategy is to build a chain of $\mathsf{rr}\text{-}write$'s from W_0 to W, such that each W writes the value that it reads from the preceding $\mathsf{rr}\text{-}write$ in the chain. We then show that the same value is written in all of these $\mathsf{rr}\text{-}write$'s by induction on the length of such chains.

Indeed, let $R = rr.\mathsf{rr}\text{-}read(r)$ be the $\mathsf{rr}\text{-}read$ corresponding to W that is executed before W is invoked. By safety, R returns the pair $\langle r_0, w_0\rangle$ or a higher ranking pair $\langle r_k, w_0\rangle$ that was written in some $W_k = \mathsf{rr}\text{-}write(r_k, *)$. Since $r_k > r_0$, again the corresponding $rr.\mathsf{rr}\text{-}read(r_k)$ returns $\langle r_0, w_0\rangle$ or a higher ranked written value. And so on. Eventually, we obtain a unique chain $W_0, W_1, ..., W_k, W$, such that for each of $W_1, .., W_k, W$, the corresponding $\mathsf{rr}\text{-}read$ returns the value/rank pair written by the preceding $\mathsf{rr}\text{-}write$ in the chain.

We now show by induction on the length k of the chain that W writes v_0. If $k = 0$, then R returns v_0 and by the agreement protocol W writes v_0.

Otherwise, suppose for all chains of length $< k$ it holds that the last $\mathsf{rr}\text{-}write$ writes v_0, and consider the chain above of length k. For W_k, the (unique) chain from W_0 is $W_0, W_1, ..., W_k$. By the induction hypothesis, W_k writes v_0. Hence, again R reads v_0 and according to the protocol, W writes v_0. □

The following theorem immediately follows from Lemma 1:

THEOREM 1. *The algorithm in Figure 1 guarantees that for any two processes i and j such that $decide(v)_i$ returns V and $decide(v')_j$ returns V' , $V.value = V'.value$; and the decision value is equal to v.value, where v is the argument of some decide operation which was invoked in the run.*

4.3 Atomic object emulation using a ranked register

Ultimately, the purpose of forming coordination is to support data sharing among clients consistently. Many protocols leverage atomic data emulation off of the consensus building block we already have. In this section we show how a ranked register can be used *directly* to construct an atomic object of an arbitrary type $\mathcal{T}$. This yields a one-tier, practical construction.

We start by defining the notion of the object type. An object type $\mathcal{T}$ consists of the following: (1) a set V of values; (2) an initial value $v_0 \in V$; (3) a set of $invocations$; (4) a set of $responses$; and (5) a function $f : invocations \times V \rightarrow responses \times V$.

The atomic object emulation pseudocode appears in Figure 2. The operation $submit$ takes as a parameter the invocation to execute, and returns the invocation response. We assume that each invocation $a \in invocations$ can be submitted at most once throughout the run. We assume that the *chooseRank* routine returns unique and monotonically increasing ranks. The ranked register is used to build a unique invocation order that ensures that the returned responses are consistent with the object type $\mathcal{T}$.

The protocol employs a data type, called *ObjectStates*, whose elements are called *object states*. An object state encapsulates the current object value $v \in V$, and the set of invocations which were applied to obtain v along with their corresponding responses. More formally, *ObjectStates* is defined to be a set of pairs $V \times \{invocations \times responses\}$ with selectors val and $resp$. We use the set $RStates = Ranks \times ObjectStates$ with selectors $rank$, $state$ to represent the set of values written/read to/from the ranked register.

Let us fix $\sigma \in ObjectStates$ and an invocation a. We define the following shortcuts to query and modify the σ's components: (1) a function $response$: $ObjectStates \times invocations \rightarrow responses$, such that $response(\sigma, a) = \rho$ iff $\langle a, \rho\rangle \in \sigma.resp$; (2) a predicate $reflects$: $ObjectStates \times invocations \rightarrow \{true, false\}$ such that $reflects(\sigma, a) = true$ iff $\exists \rho \in responses : \langle a, \rho\rangle \in \sigma.response$; (3) an operator $apply : ObjectStates \times invocations \rightarrow ObjectStates$, such that $apply(\sigma, a) = \sigma'$ iff $\sigma'.val = v'$ and $\sigma'.resp = \sigma.resp \cup \{\langle a, \rho\rangle\}$, where $\langle v', \rho\rangle = f(a, \sigma.val)$.

Due to the lack of space we will not give a correctness proof of the atomic object emulation. Below, we outline the proof strategy and state the main results. The proof is based on the same idea as the correctness proof of the agreement implementation: We consider a chain of $\mathsf{rr}\text{-}write$ invocations starting from the initial $\mathsf{rr}\text{-}write$ (which commits with $\langle r_0, v_0, \emptyset\rangle$) such that each $\mathsf{rr}\text{-}write$ in the chain writes the value that it reads from the preceding $\mathsf{rr}\text{-}write$. We prove that the object values written by each committed $\mathsf{rr}\text{-}write$ in this chain are consistent with a sequence of object values obtained by applying operations submitted in the run according to the function f of type $\mathcal{T}$. This result directly implies the following:

THEOREM 2 (ATOMICITY). *The algorithm in Figure 2 emulates an atomic object of type $\mathcal{T}$.*

Finally, the next theorem asserts the liveness:

THEOREM 3 (LIVENESS). *There exists a number N such that if a non-faulty process i invoking $submit(a)$ becomes the exclusive leader and remains to be the exclusive leader for at most N iterations of the loop in lines 2.3–15, then $submit(a)$ eventually returns.*

Shared: A read/write register $\sigma \in ObjectStates$;
initially $\sigma = \{v_0, \emptyset\}$;
a ranked register rr with values in $RStates$
initialized by $rr.\mathsf{rr}\text{-}write(\langle r_0, v_0, \emptyset\rangle)$ which commits.
Local: $r \in Ranks$, initially $r = r_0$; $V \in RStates$;

Process i:

submit$(a)_i$: $invocations \rightarrow responses$:
(1) while(*true*) do
(2)   wait to become the leader;
(3)   while($isLeader()$) do
(4)   if ($reflects(\sigma, a)$)
(5)   return $response(\sigma, a)$;
(6) $r \leftarrow chooseRank(r)$;
(7) $V \leftarrow rr.\mathsf{rr}\text{-}read(r)$;
(8)   if ($\neg reflects(V.state, a)$)
(9) $V.state \leftarrow apply(V.state, a)$;
(10) $V.rank \leftarrow r$;
(11)   if ($rr.\mathsf{rr}\text{-}write(V) = commit$) then
(12) $\sigma \leftarrow V.state$;
(13)   return $response(V.state, a)$;
(14)   fi
(15)   od
(16) od

Figure 2: Emulating an arbitrary atomic object

PROOF. Since i becomes an exclusive leader for the first time, $chooseRank$ ensures that there exists a number n that after n iterations of the loop in lines 2.3–15, its rank r becomes higher than ranks of all other $\mathsf{rr}\text{-}write$ invocations in the run. At this point, non-triviality implies that $\mathsf{rr}\text{-}write$ in line 2.11 must commit. The committed object state σ depends on the object state value σ' returned by the preceding $\mathsf{rr}\text{-}read$: If σ' was derived using a, then $\sigma' = \sigma$. Otherwise, σ' is obtained by applying a to σ'. Thus, $submit(a)$ returns at the end of this iteration so that $N = n$. □

4.4 Providing liveness

As in the original Paxos protocol, we guarantee liveness through a separate leader election service module. The leader election service does not need to be always safe, and may allow multiple leaders to exist at times. However, in order to guarantee progress, it must eventually and for a sufficiently long time provide an exclusive leader. For many years, the distributed computing community identified various building blocks that guarantee such progress. The semi-synchronous (likewise the timed-asynchronous) model [14] does this by stipulating that the system goes through stability periods in which the system is synchronous, and that are long enough to elect a leader. The failure-detectors approach initiated by Chandra and Toueg in [10] formally models the minimal conditions that guarantee that (eventually) a unique leader emerges using a failure-suspicion oracle or a leader oracle [10, 39]. Chockler et al. [12] provide an explicit construction of a mutual exclusion primitive that guarantees probabilistically the eventual emergence of unique leader. And randomized algorithms introduce randomization steps that probabilistically guarantee that a decision value is converged on by a majority (see the survey in [13]).

Adapting these approaches to a setting with infinitely many processes poses an interesting challenge: Intuitively, a desirable leader election oracle should be powerful enough to solve Consensus, and at the same time be implementable under some reasonable system assumptions (e.g., partial synchrony). But even during system stability periods, it is unrealistic to require a failure detector to output an exclusive leader forever, unless some bounds are assumed on the maximum number of clients that can potentially or concurrently contend for becoming a leader. For example, the probabilistic mutual exclusion primitive of [12] guarantees eventual emergence of an exclusive leader if the number of concurrently contending processes is bounded (but unknown). Other examples of such restricting assumptions can be found in [38]. The exact specification and implementation of a leader election module is the subject of the ongoing work and is not pursued further here.

5. IMPLEMENTING A RANKED REGISTER

In this section, we deal with the problem of implementing a wait-free shared ranked register. First, in Section 5.1, we specify how a single ranked register is implemented from a read-modify-write object. Second, in Section 5.2, we build a wait-free fault tolerant ranked register from a collection of fail-prone ones. We complete with a proof of impossibility of constructing a ranked register out of finitely many read/write registers in Section 5.3.

5.1 A single ranked register

Our shared memory model assumes the existence of atomic shared objects such as read-modify-write registers. By this, we capture the assumption that each "disk" is capable of accepting from clients subroutines with I/O operations for execution, and indivisibly performing them. The disk itself may become unavailable, and hence, the shared memory objects it provides may suffer non-responsive crash faults. For this reason, no single read-modify-write object suffices for solving agreement on its own (as in Herlihy's consensus hierarchy, see [27]). Rather, we first use each read-modify-write object to construct a ranked-register (which may also incur a non-responsive crash fault), and then, use a collection of ranked registers to construct a non-faulty ranked-register, from which agreement is built.

Let $X = (Ranks \times Ranks \times Vals) \cup \{\langle r_0, r_0, \perp\rangle\}$ with selectors rR, wR and val. The implementation of a ranked register uses a single read-modify-write shared object $x \in X$ of unbounded size whose field $x.rR$ holds the maximum rank with which a $\mathsf{rr}\text{-}read$ operation has been invoked; $x.wR$ holds the maximum rank with which a $\mathsf{rr}\text{-}write$ operation has been invoked; and $x.val$ holds the current register value. The implementation pseudocode is depicted in Figure 3. For clarity, invocations of read-modify-write operations $\mathsf{rmw}\text{-}read$ and $\mathsf{rmw}\text{-}write$ are enclosed within "lock" and "unlock" statements, to indicate that they execute indivisibly.

LEMMA 2. *The pseudocode in Figure 3 satisfies Safety.*

PROOF. That a $\mathsf{rr}\text{-}read$ operation can only return a valid value that was actually used in a $\mathsf{rr}\text{-}write$ operation or $\langle r_0, \perp\rangle$ is obvious from the code. Now consider a $\mathsf{rr}\text{-}write$ operation $W_1 = \mathsf{rr}\text{-}write(\langle r_1, v_1\rangle)_i$ that commits and let $R_2 =$

```
Types: X = (Ranks × Ranks × Vals) ∪ {⟨r0, r0, ⊥⟩}
with selectors rR, wR and val
Shared: x ∈ X.
Initially x = ⟨r0, r0, ⊥⟩

Local: V ∈ RVals, status ∈ {ack, nack}.

Process i:

rr-read(r)_i:                              Read-modify-write procedures:
    lock x:                                rmw-read(r):
        V ← rmw-read(r)                        if (x.rR < r)
    unlock x                                       x.rR ← r
    return V                                   return ⟨x.wR, x.val⟩


rr-write(⟨r, v⟩)_i:                        rmw-write(r, v):
    lock x:                                    if (x.rR ≤ r ∧ x.wR < r)
        status ← rmw-write(r, v)                   x.wR ← r
    unlock x                                       x.val ← v
    if (status = ack)                              return ack
        return commit                          return nack
    return abort
```

Figure 3: An implementation of a single ranked register

rr-$read(r_2)_j$, $r_2 > r_1$ be a rr-$read$ operation which returns $\langle r, v\rangle$. Let mw_1 denote the rmw-$write()$ procedure called from within W_1 and mr_2 the rmw-$read()$ procedure invoked within R_2. Since the read-modify-write semantics of x ensures sequential access, mr_2 must be sequenced after mw_1. For otherwise, $x.rR \geq r_2 > r_1$ so that mw_1 returns $nack$ and W_1 aborts. Thus, R_2 returns the tuple written by a rmw-$write$ procedure mw' which is either mw_1 or some rmw-$write$ procedure sequenced after mw_1. Let r', v' be the arguments passed to mw'. Then, $r' \geq r_1$, since otherwise, $x.wR \geq r_1 > r'$ so that the value of x remains unchanged. Moreover, by the rank-uniqueness assumption, $r' = r_1$ implies that $mw' = mw_1$. Therefore, $\langle r, v\rangle = \langle r', v'\rangle$ and either $\langle r', v'\rangle = \langle r_1, v_1\rangle$, or $r' > r_1$ as needed. □

LEMMA 3. *The pseudocode in Figure 3 satisfies Non-Triviality.*

PROOF. According to the pseudocode, a rr-$write$ operation W with rank r aborts if the rmw-$write()$ procedure w called within W returns $nack$. This happens if w sees $x.rR > r$ or $x.wR \geq r$. This is only possible if some rmw-$write()$ procedure with rank $r' \geq r$, or a rmw-$read()$ procedure with rank $r' > r$ is sequenced before w. This could happen only as a result of some previously returned or concurrent rr-$read$ (rr-$write$) with rank $r' > r$ ($r' \geq r$). By the rank-uniqueness assumption, no two rr-$write$ operations are ever invoked with the same rank. Therefore, W can abort only due to some previously returned or concurrent rr-$read$ or rr-$write$ with rank $r' > r$ as needed. □

LEMMA 4. *The pseudocode in Figure 3 satisfies Liveness.*

PROOF. Liveness trivially holds since both rr-$read$ and rr-$write$ always return something (i.e., the implementation is *wait-free*). □

We have proven the following theorem:

THEOREM 4. *The pseudocode in Figure 3 is an implementation of a ranked register.*

5.2 A fault-tolerant construction of a ranked register

In this section we present a wait-free implementation of a ranked register from ranked registers that may experience non-responsive crash faults. The register supports an unbounded number of clients. Our construction utilizes n shared ranked registers up to $\lfloor (n-1)/2 \rfloor$ of which can incur non-responsive crash. The pseudocode appears in Figure 4.

LEMMA 5. *The pseudocode in Figure 4 satisfies Safety.*

PROOF. That a rr-$read$ operation can only return a valid value that was actually used in a rr-$write$ operation or $\langle r_0, \perp\rangle$ is obvious from the code. Now consider a rr-$write$ operation W_1 = rr-$write(\langle r_1, v_1\rangle)_i$ that commits and let R_2 = rr-$read(r_2)_j$, $r_2 > r_1$ be a rr-$read$ operation which returns $\langle r, v\rangle$. Since both W_1 and R_2 access at least $\lceil (n+1)/2 \rceil$ ranked registers, there exists a single register rr_k accessed by both W_1 and R_2. Moreover, the Safety of rr_k ensures that the tuple $\langle r', v'\rangle$ returned by rr_k.rr-$read(r_2)_i$ must satisfy $r' \geq r_1$. Since R_2 returns the tuple with maximum rank, $r \geq r' \geq r_1$ as needed. □

LEMMA 6. *The pseudocode in Figure 4 satisfies Non-Triviality.*

PROOF. According to the protocol, a rr-$write$ operation W = rr-$write(\langle r, v\rangle)_i$ aborts if there exists k such that rr_k.rr-$write(\langle r, v\rangle)_i$ aborts. By the Non-Triviality of rr_k, this can happen only if some invocation rr_k.rr-$write(\langle r', v'\rangle)_j$ (rr_k.rr-$read(r')_j$) with $r' > r$ occur before or concurrently to rr_k.rr-$write(\langle r, v\rangle)_i$. This can only be the case if some rr-$write$ or rr-$read$ operation with rank r' has been completed before or is concurrent to W. □

```
Shared: Ranked registers rr_j, 1 ≤ j ≤ n

Local: S_1 ⊆ RVals, S_2 ⊆ {commit, abort}.

Process i:

rr-read(r)_i:
    S_1 ← ∅
    Invoke in parallel for each 1 ≤ j ≤ n:
        S_1 ← S_1 ∪ {rr_j.rr-read(r)_i}
    wait until |S_1| ≥ ⌈(n + 1)/2⌉
    ⟨r, v⟩ ← ⟨r′, v′⟩ : ⟨r′, v′⟩ ∈ S_1 ∧ r′ = max_{⟨r″,v″⟩∈S_1} r″
    return ⟨r, v⟩


rr-write(⟨r, v⟩)_i:
    S_2 ← ∅
    Invoke in parallel for each 1 ≤ j ≤ n:
        S_2 ← S_2 ∪ {rr_j.rr-write(⟨r, v⟩)_i}
    wait until |S_2| ≥ ⌈(n + 1)/2⌉
    if (abort ∈ S_2)
        return abort
    return commit
```

Figure 4: A wait-free construction of the ranked register out of n ranked registers

LEMMA 7. *The pseudocode in Figure 4 satisfies Liveness.*

PROOF. Each rr-$write$ or rr-$read$ operation is guaranteed to terminate since at most $\lceil (n+1)/2 \rceil$ ranked registers are required to respond, no more than $\lfloor (n-1)/2 \rfloor$ ranked registers can incur non-responsive crash, and each individual non-faulty ranked register is wait-free. □

We have proven the following theorem:

THEOREM 5. *The pseudocode in Figure 4 is a wait-free construction of a ranked register out of n ranked registers such that at most $\lfloor (n-1)/2 \rfloor$ can incur non-responsive crash faults.*

5.3 Impossibility of constructing ranked-register from read/write registers

In this section we prove that a ranked register cannot be implemented using a bounded number of atomic read/write registers (of unbounded size) in the presence of unbounded number of clients. This proves that stronger types of shared memory objects (such as the read-modify-write registers) are indeed necessary even for our weak ranked-register building block. The main result of this section is expressed in Theorem 6 below. It shows that any algorithm that implements the ranked register specification in a shared memory system with n processes must use at least n atomic read/write registers. It then follows that if the number of processes is not bounded, the number of shared read/write registers needed to implement the ranked register is also unbounded.

In order to prove this result, we utilize the technique of [8] to prove lower bounds on the number of atomic registers needed to solve mutual exclusion[3]. We start with some definitions.

[3]We believe that this technique is general enough to be applied for proving lower bounds for many other shared memory problems in settings with infinitely many processes.

We say that two system states s and s' are indistinguishable to process i, denoted $s \overset{i}{\sim} s'$, if the state of process i and the values of all shared variables are the same in s and s'. We say that process i *covers* shared variable x in system state s if i is about to write on x in s.

LEMMA 8. *Suppose that there exists an algorithm that implements a ranked register using only shared atomic read/write registers. Let s be a reachable system state in which r is the highest rank that appears in any operation. Then a* rr-$write$ *operation $W =$ rr-$write(\langle r', v' \rangle)_i$ by process i with $r' > r$ must write some shared variable which is not covered in s.*

PROOF. Assume in contradiction that no non-covered shared variable is written by i in the course of W. We construct a system execution which violates the Safety property of the ranked register as follows:

We first run from s each process which covers some shared variable exactly one step so that they write the shared variables they cover. Let s' be the resulting system state.

Next, we construct an execution fragment α_1 starting in s' and not involving i by invoking a rr-$read(r'')$ operation R at some process $j \neq i$ whose rank r'' satisfies $r'' > r'$. By the Liveness and the Safety properties of the ranked register, R must return a value written by some rr-$write$ operation with rank at most r.

We now construct another execution fragment α_2 which starts from s as follows: We run i solo until W commits; since no higher rank appears in s, by the Non-Triviality property W must indeed commit. By assumption, it writes only shared variables that are covered in s. From the resulting state, we run each process which covers some shared variable exactly one step so they overwrite everything written by i in its solo run. Let s'' be the resulting state. Since $s'' \overset{j}{\sim} s'$ for all $j \neq i$, we can extend α_2 by running α_1 from s''.

By the Safety property of the ranked register, the rr-$read$ operation R must return the value written by W in this execution. However, it returns a value written by a rr-$write$ operation with rank at most r thus violating safety. A contradiction. □

We now set off to prove the lower bound. We use the following strategy: We first prove using Lemma 9 that with any algorithm implementing the ranked register for $n \geq 1$ processes, it is possible to bring the system to a state where at least $n-1$ shared variables are covered while running only $n-1$ processes. In this state we invoke a rr-$write$ operation whose rank is higher than the the rank of every operation invoked so far. Since this rr-$write$ operation must commit (Non-Triviality), by Lemma 8, it must write to some shared variable which has not been covered yet. This implies that another shared variable is needed in addition to the $n-1$ covered ones.

LEMMA 9. *Suppose that there exists an algorithm that implements a ranked register for $n \geq 1$ processes using only shared atomic read/write registers. Let s be any reachable system state. Then for any k, $1 \leq k \leq n-1$, there exists a state s_k which is reachable from s using steps of processes $1 \dots k$ only, such that at least k distinct variables are covered in s_k.*

PROOF. The proof is by induction on k.

Basis: $k = 1$. Let s be any system state. We first run process 1 until it returns from the last operation invoked on 1, if any. This must happen due to the Liveness property of the ranked register. Let t be the resulting system state.

In t, we let process 1 invoke a rr-$write$ operation W whose rank is higher than the ranks of all operations invoked so far. By Non-Triviality, W must commit. By Lemma 8, W must write some shared variable which is not covered in state s. We then run 1 until it covers this variable. The resulting state s_1 satisfies the lemma requirements.

Inductive step: Suppose the lemma holds for k, where $1 \leq k \leq n-2$. Let us prove it for $k+1$. Using the induction hypothesis, we run k processes from s until the state s_k is reached where at least k distinct shared variables are covered. Starting in s_k, Starting in t, we run process $k+1$ until the last operation invoked on $k+1$ returns. This must happen due to Liveness. Let t be the resulting state.

In t we let process $k+1$ invoke a rr-$write$ operation W whose rank is higher than the ranks of all operations invoked so far. By Non-Triviality, W must commit. Moreover, by Lemma 8, W must write some shared variable which is not covered in s_k. So we run $k+1$ until it covers this shared variable. The resulting state s_{k+1} satisfies the lemma requirements. □

We are now ready to prove the main theorem:

THEOREM 6. *If there exists an algorithm that implements a ranked register for $n \geq 1$ processes, then it must use at least n shared atomic read/write registers.*

PROOF. Assume in contradiction that there exists an algorithm which implements a ranked register for $n \geq 1$ processes using $n-1$ shared read/write registers.

Let s be the initial system state. Note that there are no covered variables in s. We use the result of Lemma 9 and run $n-1$ processes from s until the state s_{n-1} is reached where the processes cover $n-1$ distinct shared variables. We then invoke a rr-$write$ operation W on process n whose rank is higher than the ranks of all operations invoked so far. By Non-Triviality, W must commit. By Lemma 8, W must write some shared variable which is not covered in s_{n-1}. However, all $n-1$ shared variables are covered in s_{n-1}. A contradiction. □

6. CONCLUSION

The paper presents a solution for the Consensus problem with an unbounded number of processes, which is suitable for state-of-the-art SAN environments. Two hurdles must be overcome in solving the agreement problem. One is the uncertainly of failure detection in asynchronous settings; this difficulty has received overwhelming attention in the distributed computing community, and is not further pursued here. The other is the challenge of maintaining a safe decision with finite memory when an unbounded number of processes exist. The solution we give is to use stronger type of memory objects in order to emulate a shared memory abstraction of a reliable ranked register. The required memory objects are readily available in Active Disks and in NASD [22] controllers, and should serve as a reference to the kind of disk functionality that is useful for file system implementors. They can also be naturally supported in the most common client-server settings. The resulting construction is modular and memory efficient.

Acknowledgments

We are thankful to Ittai Abraham and Danny Dolev for helpful discussions of the results in this paper.

7. REFERENCES

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

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

[3] K. Amiri, G. A. Gibson, R. Golding. Highly concurrent shared storage. In Proceedings of the *International Conference on Distributed Computing Systems (ICDCS2000)*, April 2000.

[4] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Roselli, R. Wang. Serverless network file systems. *ACM Transactions on Computer Systems* 14(1):41–79, February 1996.

[5] i K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In Proceedings of the *11th Annual Symposium on Operating Systems Principles*, pages 123–138, November 1987.

[6] R. Boichat, P. Dutta, S. Frolund and R. Guerraoui. Deconstructing Paxos. *Technical Report DSC ID:200106*, Communication Systems Department (DSC), École Polytechnic Fédérale de Lausanne (EPFL), January 2001. Available at `http://dscwww.epfl.ch/EN/publications/documents/tr01_006.pdf`.

[7] R. Burns. Data management in a distributed file system for Storage Area Networks. PhD Thesis. Department of Computer Science, University of California, Santa Cruz, March 2000.

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

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

[10] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM* 43(2):225–267, March 1996.

[11] G. V. Chockler, I. Keidar and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys* 33(4):1–43, December 2001.

[12] 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, April 2001.

[13] B. Chor and C. Dwork. Randomization in Byzantine agreement. *Advances in Computing Research, Randomness in Computation*, volume 5, JAI Press, edited by S. Micali, pp. 443–497, 1989.

[14] F. Cristian and C. Fetzer. The Timed Asynchronous distributed system model. In Proceedings of the *28th*

Annual International Symposium on Fault-Tolerant Computing, June 1998.

[15] R. DePrisco, B. Lampson and N. Lynch. Fundamental study: Revisiting the Paxos algorithm. *Theoretical Computer Science* 243:35–91, 2000.

[16] D. Dolev, C. Dwork and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM* 34(1):77–97, January 1987.

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

[18] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM* 32(2):374–382, April 1985.

[19] A. Fekete, N. Lynch and A. Shvartsman. Specifying and using a partitionable group communication service. *ACM Transactions on Computer Systems* 19(2):171–216, May 2001.

[20] E. Gafni and L. Lamport. Disk Paxos. In Proceedings of *14th International Symposium on Distributed Computing (DISC'2000)*, pages 330–344, October 2000.

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

[22] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg and J. Zelenka. A cost-effective high-bandwidth storage architecture. In Proceedings of the *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 1998.

[23] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, H. Gobioff, E. Riedel, D. Rochberg and J. Zelenka. Filesystems for Network-Attached Secure Disks. *Technical Report CMU-CS-97-118*, July 1997.

[24] H. Gobioff, G. A. Gibson, D. Tygar. Security for Network Attached Storage Devices. *Technical Report CMU-CS-97-185*, October 1997.

[25] S. Hotz, R. Van Meter and G. Finn. Internet protocols for network-attached peripherals. In Proceedings of the *Sixth NASA Goddard Conference on Mass Storage Systems and Technologies in conjunction with 15th IEEE Symposium on Mass Storage Systems*, 1998.

[26] J. H. Hartman, I. Murdock and T. Spalink. The Swarm scalable storage system. In Proceedings of the *19th IEEE International Conference on Distributed Computing Systems (ICDCS '99)*, June 1999.

[27] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems* 11(1):124–149, January 1991.

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

[29] I. Keidar and D. Dolev. Totally ordered broadcast in the face of network partitions: exploiting group communication for replication in partitionable networks. In *Dependable Network Computing, Chapter 3*, D. Avresky Editor, Kluwer Academic Publications. January 2000.

[30] L. Lamport. Time, clocks, and the ordering of events in distributed systems. *Communications of the ACM* 21(7):558–565, July 1978.

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

[32] L. Lamport. Paxos made simple. *Distributed Computing Column of SIGACT News* 32(4):34–58, December 2001.

[33] L. Lamport, R. Shostak and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems* 4(3):382–401, July 1982.

[34] B. W. Lampson. How to build a highly available system using Consensus. In Proceedings of the *10th International Workshop on Distributed Algorithms (WDAG)*, Springer-Verlag LNCS 1151:1-17, Berlin, 1996,

[35] 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, October 1996.

[36] M. C. Loui and H. H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes, In Franco P. Preparata, editor, *Parallel and Distributed Computing*, volume 4 of *Advances in Computing Research*, pages 163–183. JAI Press, Greenwich, Conn., 1987.

[37] D. Malkhi and M. K. Reiter. An architecture for survivable coordination in large-scale systems. *IEEE Transactions on Knowledge and Data Engineering* 12(2):187–202, March/April 2000.

[38] M. Merritt and G. Taubenfeld. Computing with infinitely many processes. In Proceedings of *14th International Symposium on Distributed Computing (DISC'2000)*, pages 164–178, October 2000.

[39] A. Mostfaoui and M. Raynal. Leader-based consensus. *Parallel Processing Letters* 11(1):95–107 2001.

[40] National Storage Industry Consortium. `http://www.nsic.org/nasd`.

[41] D. Powell, editor. Group communication. *Communications of the ACM* 39(4), April 1996.

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

[43] M. D. Skeen. Nonblocking commit protocols. In *SIGMOD International Conference Management of Data*, 1981.

[44] M. D. Skeen. Crash Recovery in a Distributed Database System. PhD thesis, UC Berkeley, May 1982.

[45] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys* 22(4):299–319, December 1990.

[46] 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, October 1997.