Active Disk Paxos
with infinitely many processes *

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

May 14, 2003

Key-words: **Shared memory**, **Consensus**, **Paxos**, **infinitely many processes**, **non-responsive object faults**

1 Introduction

In this paper we present a solution for universal service replication using a data-centric approach. In this approach, a highly available service is implemented by a replicated set of servers, a threshold of which may be faulty. Each server is responsible solely for implementing certain objects, e.g., a single shared register, that is accessible by any number of clients. Our paradigm provides for coordination and information sharing among transient clients, possibly numerous, 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 [46, 12].

The data-centric paradigm also faithfully reflects recent advances in hardware technology that 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* [8] is an example of a commercially available SAN system that solves many of the coordination, sharing and security issues involved with SANs (see Section 2 for more examples). In this paper, we tackle the issue of scaling the number of clients that are served by a SAN.

From here on, we refer to shared storage units in our data-centric system simply as *objects*. 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 [48, 49]. This leads to the usage of the Paxos protocol [34, 35, 18, 37] and its variants, as is done, e.g., in Petal [38] and Frangipani [51]. 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 in the shared-memory model in the Disk Paxos protocol [23]. In this work we provide several important contributions that enhance this line of research:

- We provide an adaptation of Paxos that supports infinitely many clients;

*A preliminary version of this work appears in Proceedings of the 21st ACM Symposium on Principles of Distributed Computing (PODC '02), August 2002.

- The memory complexity of our solution is constant;

- Our construction makes use of a modular building block, called a *ranked register*, that promotes understanding and analysis of Paxos and of general coordination in distributed systems.

- Both our agreement protocol and our atomic object emulation are built directly over the ranked register abstraction, providing for each an efficient one-tier implementation. In contrast, most atomic object emulation algorithms found in the literature utilize the Consensus object as a building block.

Of the above contributions, the most tangible one is the extension to support infinitely many clients. 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 memory 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 [23]. Also, the complexity of memory (disk) operations is linear in the number of clients.

In contrast, our solution is ignorant of the number of participating clients and their identities. It builds on a strengthening of the shared memory model. Our use of strong memory objects is justified by the impossibility result of Section 7, 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.

Note that the strengthened memory model is justified in practice. First, servers may support arbitrarily complex object semantics, and as for disks, 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., [25]). 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 [26].

The existence of strong shared memory objects does not obviate the need for an agreement protocol. Admittedly, if we had even one reliable read-modify-write object, we could leverage coordination off it to solve agreement, as shown by Herlihy in [30]. However, objects stored by servers or disks could become unavailable. Unfortunately, it is impossible to use a collection of fail-prone read-modify-write objects to emulate a reliable one [31]. Hence, our construction is necessarily more involved. It should be noted that using a farm of shared objects also has the benefit beyond high availability. Even in the case that disks are considered reliable, distributing client accesses among multiple objects prevents unnecessary contention. Hence, our solution provides for both high availability, and for load sharing among storage servers.

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]. (We compare our ranked register abstraction with the round-based register of [6] in Section 4). Briefly, a ranked register supports rr-*read* and rr-*write* operations that are both parameterized by a *rank* whose values are taken from a totally ordered set fixed in advance (e.g., the Paxos *ballots* are integers). 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 and universal object emulation using the abstraction of one reliable shared ranked register that supports infinitely many clients. Briefly, in these implementations 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. (In case of universal object emulation, the agreement value is the operation prefix, possibly extended by the client's input operation). 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.

[1]Section 7 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[30], this a fortiori implies impossibility of constructing agreement.

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 7). Furthermore, no single fail-prone 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 read-modify-write registers, of which a threshold may become non-responsive. 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 objects required for the emulation is determined only by the level of desired fault tolerance, regardless of the number of participating clients.

Our approach is readily implementable in a 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 [42] were designed. We advocate the data-centric approach in more detail in two recent position papers [14, 41].

2 Related work

Our work deals with solving the Consensus problem [36], one of the most fundamental problems in distributed computing. Consensus is the building block for replication paradigms such as state machine replication [33, 50], group membership (see [46, 12] for survey), virtual synchrony [5], atomic broadcast [11], total ordering of messages [32, 22], etc. Consensus is known to be unsolvable in most realistic models such as asynchronous message passing systems [21] and asynchronous shared memory with read/write registers [40, 30, 19] 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., [19, 20, 17]), augmenting the system with unreliable failure detectors [11], and employing randomization (see a survey in [16]). Specifically, our solution uses one of the most widely deployed implementations of the state machine replication [33, 50], the Paxos algorithm [34, 35, 18, 37]. 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 [10] which was shown in [10] to be the weakest failure detector that can be used to solve Consensus.

As shown below in Section 7 though, when an infinite number of processes is present, even if they are all non-faulty, 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 [43, 24]. 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 [23]. 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, 31]. 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 [31], 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 weaker guarantees and supports a slightly different interface. A detailed comparison is provided in Section 4. A different deconstruction of the Paxos protocol is provided in [7]. This deconstruction employs a more abstract shared-object definition, called $\Diamond Register$. The $\Diamond Register$ avoids referring to ranks (or rounds), and thus is a higher level abstraction that does not directly include implementation details. On the other hand, as discussed in [7], the $\Diamond Register$ admits inefficient implementations. This is prevented in the specification of our ranked register.

2.1 SAN technology

Our work was motivated in part 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) [25] 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 [26], security [27] and network protocols [25].

Active Disks [47, 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 [8] in the context of the 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 [45], the Netstation project [28] and the Swarm Scalable Storage System [29]. Petal [38] is a project to research highly scalable block-level storage systems. Frangipani [51] 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 [38] virtual disk storage system and the Frangipani [51] file system is achieved using replicated lock servers which utilize Paxos for consistency. Consequently, Disk Paxos is a natural candidate for enabling lock management

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

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 $1, 2, \ldots$. Clients 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. The shared memory objects themselves may be crash faulty. The model above is called in [31] the *non-responsive crash (NR-Crash)* failure model, and we shall use this name here-on.

According to [31], wait-free consensus is impossible in such a setting. This result holds regardless of the number, size and type of the shared objects used by the implementation. Therefore, similar to the Paxos approach, we overcome this impossibility by augmenting the system with a leader oracle. The oracle guarantees the eventual emergence of a unique non-faulty leader, though when this happens is unknown to the clients themselves.

The boolean failure detector oracle we employ, denoted $\mathcal{L}$, is as follows. Let $\mathcal{L}_i$ denote the local instance of $\mathcal{L}$ at i, with a boolean isLeader() operation returning the current value output by $\mathcal{L}_i$. Then, $\mathcal{L}$ is required to satisfy the following property eventually:

Property 1 (Unique Leader). *There exists a correct process i such that every invocation of $\mathcal{L}_i$.isLeader() returns true, and for each process $j \neq i$, every invocation of $\mathcal{L}_j$.isLeader() returns $false$.*

4 The Ranked Register

Our Consensus and atomic object constructions (see Section 5) employ a special type of shared memory register, called a *ranked register*, which for now we assume is failure-free. In Section 6, we show how to implement a fault tolerant ranked register.

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]. We discuss the differences below. 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.

We now give a formal specification of the ranked register. Let *Ranks* be a totally ordered set of ranks with a distinguished initial rank r_0 such that for each $r \in Ranks$, $r > r_0$; and *Vals* be a set of values with a distinguished initial value v_0. 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: rr-*read*$(r)_i$ by process i, $r \in Ranks$, whose corresponding response is $value(V)_i$, where $V \in RVals$. And rr-*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 rr-*read* and rr-*write* operations on a ranked register take a rank as an additional argument; and its rr-*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 rr-*read* operation R *returns* a value V meaning that the register responds with $value(V)$ in response to R. We also say that a rr-*write* operation W *commits* (*aborts*) if the register responds with *commit* (*abort*) in response to W.

For simplicity, we assume that each run starts with $W_0 =$ rr-*write*$(\langle r_0, \perp\rangle)$ which commits. Furthermore, we will restrict our attention to runs in which invocations of rr-*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* $\text{rr-}write(\langle r, v\rangle)_i$ *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 rr-*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* rr-*read operation* $R = \text{rr-}read(r_2)_i$ sees *a* rr-*write operation* $W = \text{rr-}write(\langle r_1, v\rangle)_j$ *if R returns $\langle r', v'\rangle$ where $r' \geq r_1$.*

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

Property 2 (Safety). *Every* rr-*read operation returns a value and rank that was written in some* rr-*write invocation. Additionally, let* $W = \text{rr-}write(\langle r_1, v\rangle)_i$ *be a* rr-*write operation that commits, and let* $R = \text{rr-}read(r_2)_j$*, such that $r_2 > r_1$. Then R sees W.*

Property 3 (Non-Triviality). *If a* rr-*write operation W invoked with the rank r_1 aborts, then there exists a* rr-*read (*rr-*write) operation with rank $r_2 > r_1$ which is invoked before W returns.*

Property 4 (Liveness). *If an operation (*rr-*read or* rr-*write) is invoked by a non-faulty process, then it eventually returns.*

Allowing rr-*write* to abort sometimes is crucial for its implementability. Suppose that a rr-*read* operation with rank r returns a value written by a rr-*write* operation with a rank $r' < r$. Later, when a subsequent rr-*write* with a rank $r' < r'' < r$ is invoked, it must abort due to this rr-*read*.

Also note that our ranked register specification is very weak: In particular, it allows in some situations for rr-*write* operation to commit even though there exists another previously committed rr-*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 5.1. In particular, each process in our Consensus implementation invokes rr-*write* only after it invokes rr-*read* with the same rank and this rr-*read* returns. Thus, the ranked register Safety property ensures that in every finite execution prefix, each value written by a committed rr-*write* must be returned by one of the rr-*read* operations with a higher rank if such exist. Consequently, in each run of the Consensus implementation, any rr-*write* operation, which is invoked after rr-*read* with a higher rank has returned, would necessarily abort.

Our specification of ranked register is weaker compared with the round-based register of [6]. The round-based register uses notions of partial operation ordering in the definition of the *write-commit* property ("if write(k, v) commits, and no **subsequent** write(k', v') with $k' \geq k$ and $v' \neq v$ is invoked, then any read(k'') that commits, commits with v if $k'' > k$", stressed text added here for clarity). To see that this definition is too strong, consider the following scenario. Suppose that a write $w_1 = write(k_1, v_1)$ is invoked and is still in progress when another write is invoked, $w_2 = write(k_2, v_2)$, with $k_1 > k_2$, $v_1 \neq v_2$. In this case, w_2 may commit. However, a subsequent read may "see" w_1, and the value of w_1 may be returned, contradictory to the requirement. In fact, the distributed implementation of round-based register in [6] does not prevent this. Moreover, it does not seem possible to prevent this in our setting. Finally, we should note that just by dropping 'subsequent' from the specification results in different problems. It is our view that there is no easy way to form the ranked-register specification using operation ordering, and hence, the specification above is qualitatively different from that of the round-based register.

5 Consensus and atomic object emulation

In this section we present the implementations of Consensus and of an arbitrary typed atomic object (a universal construction) based on the ranked register abstraction defined in the previous section. The algorithms in this section use the ranked register as a black box.

In addition to a shared ranked register, our algorithms also employ atomic shared registers. It should be noted that these objects can be implemented in our models in a similar manner to the ranked-register implementation, and hence we omit their explicit constructions.

5.1 Consensus using a ranked register

We now outline an agreement protocol which employs a shared ranked register. The pseudocode of the Consensus implementation is depicted in Figure 1. Each process i iterates through the following steps until the decision is reached: First, i checks whether some process has decided and written the agreement value into the *decision* register. If yes, this value is returned. Otherwise, i calls a local procedure, chooseRank which is assumed to output monotonically increasing values $r \in Ranks$, and then waits until the output of $\mathcal{L}_i$ becomes *true*. Once this happens, the local DECIDE routine is invoked. It takes as arguments i's initial value and the chosen rank. 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 (i.e., Property 1 holds).

```
Shared: Ranked registers rr initialized by rr-write(⟨r0, ⊥⟩) which commits;
Regular register decision, with values in RVals, initialized by write(⟨r0, ⊥⟩)
Local: V ∈ RVals ∪ {abort}, r ∈ Ranks;

Process i:

propose(v), Vals → Vals
    r ← r0;
    while(true) do
        V ← decision.read();
        if (V.value ≠ ⊥)
            return V.value;
        if (L_i.isLeader()) then
            r ← chooseRank(r);
            V ← DECIDE(⟨r, v⟩);
            if (V ≠ abort)
                return V.value;
        fi
    od

Function DECIDE(⟨r, v⟩), RVals → RVals ∪ {abort}:
    V ← rr.rr-read(r)_i;
    if (V.value = ⊥) then
        V.value ← v;
    V.rank ← r;
    if (rr.rr-write(V)_i = commit) then
        decision.write(V);
        return V;
    fi
    return abort;
```

Figure 1: Consensus using a ranked register

We now outline the correctness argument of the agreement algorithm. Recall that W_0 is an initialization rr-*write* operation, assumed to commit at the start of any execution. Ignoring this initialization, the next lemma shows that once a consensus value commits, it remains fixed as the decision value throughout the execution.

Lemma 1. *For any finite execution α, let $W_1 = rr.\mathsf{rr}\text{-}write(\langle r_1, v_1\rangle)$, $W_1 \neq W_0$ be the lowest ranked* rr-*write invocation which commits in α. Then, in any extension of α in which $W = rr.\mathsf{rr}\text{-}write(\langle r, v\rangle)$, $r > r_1$, is invoked, $v = v_1$.*

Proof. Our proof strategy is to build a chain of rr-*write*'s from W_1 to W, such that each W writes the value that it reads from the preceding rr-*write* in the chain. We then show that the same value is written in all of these rr-*write*'s by induction on the length of such chains.

Indeed, let $R = rr.\text{rr-}read(r)$ be the rr-*read* corresponding to W that is executed before W is invoked. By safety, R returns the pair $\langle r_1, w_1 \rangle$ or a higher ranking pair $\langle r_k, w_k \rangle$ that was written in some $W_k = \text{rr-}write(r_k, w_k)$. Since $r_k > r_1$, again the corresponding $rr.\text{rr-}read(r_k)$ returns $\langle r_1, w_1 \rangle$ or a higher ranked written value. And so on. Eventually, we obtain a unique chain $W_1, W_2, ..., W_k, W$, such that for each of $W_2, .., W_k, W$, the corresponding rr-*read* returns the value/rank pair written by the preceding rr-*write* in the chain.

We now show by induction on the length k of the chain that W writes v_1. If $k = 1$, then R returns v_1 and by the agreement protocol W writes v_1.

Otherwise, suppose for all chains of length $< k$ it holds that the last rr-*write* writes v_1, and consider the chain above of length k. For W_k, the (unique) chain from W_1 is $W_1, W_2, ..., W_k$. By the induction hypothesis, W_k writes v_1. Hence, again R reads v_1 and according to the protocol, W writes v_1. □

The following theorem immediately follows from Lemma 1 (and the protocol):

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

Next, we show liveness.

Theorem 2. *If some correct process invokes* propose*, then eventually all correct processes decide.*

Proof. First note that the regular register semantics imply that once some process decides and completes its write operation to the *decision* register, all other process will eventually read this value and decide.

Otherwise, by definition of $\mathcal{L}$, there exists time T such that Property 1 holds at all times $t > T$. Assume that *decision* is not written before T. Since by the theorem precondition, at least one correct process is taking steps after T, Property 1 implies that there exists a correct process i such that at all times $t > T$, $\mathcal{L}_i$.isLeader() returns true, and for all $j \neq i$, $\mathcal{L}_j$.isLeader() returns false. By non-triviality of the ranked register, rr-*write* is guaranteed to commit once it is called with a rank which is the highest among all the ranks ever chosen by any process in the system. Since chooseRank returns monotonically increasing ranks, such a rank is eventually returned by chooseRank at i. Once, rr-*write* commits, i writes the committed value to the *decision* register. Once this happens, all the correct processes eventually decide. □

5.2 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.

An object type $\mathcal{T}$ is a tuple $\langle \Sigma, OP, Res, G \rangle$, where Σ is a set of the type states, OP is a set of operations, Res is a set of responses, and G is a sequential specification of $\mathcal{T}$ which maps the pairs in $OP \times \Sigma$ to the pairs in $Res \times \Sigma$. Let $\sigma, \sigma' \in \Sigma$, $op \in OP$ and $res \in Res$. We say that $\langle \sigma', res \rangle$ is the result of *applying op* to σ iff $\langle \sigma', res \rangle = G(\sigma, op)$.

The atomic object emulation pseudocode appears in Figure 2. The operation submit takes as a parameter an operation to execute, and returns the operation response. For simplicity, we assume that each $op \in OP$ can be submitted at most once throughout the run. In practice, this requirement can be easily enforced by assigning each newly submitted operation a unique id which e.g., can be the pair consisting of the process id and a counter. We assume that the sequential specification of $\mathcal{T}$, G as well as the initial state σ_0 of the type $\mathcal{T}$ instance being emulated is known to all the participating processes. As before, we assume that the chooseRank routine returns unique and monotonically increasing ranks. The ranked register is used to build a common invocation sequence which, when applied to the object states starting from σ_0, ensures that the returned responses are consistent with G. The set of values which are written to and read from the ranked register are taken from the set of all sequences over the set $OP \times Res$. We denote by $\text{APPLY}(seq, op)$, where seq is a sequence over $OP \times Res$ and $op \in OP$, the sequence obtained by appending $\langle op, res \rangle$ to seq, where res is the result of applying op to the final state reached when applying the operations in seq from the initial state σ_0.

```
Types: OpSeq is a sequence over OP × Res;
       RVals = Ranks × OpSeq with selectors rank and prefix;
Shared: A ranked register rr with values in RVals initialized by rr.rr-write(⟨r0, ⟨⟩⟩) which commits.
Local: r ∈ Ranks, initially r = r0; V ∈ RVals;

Process i:

submit(op)_i: OP → Res:
    r ← r0;
    while(true) do
        if (L_i.isLeader()) then
            r ← chooseRank(r);
            V ← ORDER(r, op);
            if (V ≠ abort)
                return res : ∃i⟨op, res⟩ = V.prefix[i];
        fi
    od

Function ORDER(r, op), Ranks × OP → RVals ∪ {abort}
    V ← rr.rr-read(r);
    if (∀i∀res : V.prefix[i] ≠ ⟨op, res⟩)
        V.prefix ← APPLY(V.prefix, op);
    V.rank ← r;
    if (rr.rr-write(V) = commit) then
        return V;
    else
        return abort;
    fi
```

Figure 2: Emulating an arbitrary atomic object using a ranked register

We now set off to show the emulation algorithm correct. The next lemma is similar to Lemma 1 of the Consensus correctness argument:

Lemma 2. *For any execution α, let $W_0 = \text{rr-}write(\langle \pi_0, r_0 \rangle)$ be a* rr-*write operation which commits. Then, in any extension of α in which $W = \text{rr-}write(\langle \pi, r \rangle)$, $r > r_0$ is invoked, π_0 is a prefix of π.*

Proof. By the same argument as in the proof of Lemma 1, there exists a unique chain $W_0, W_1, \ldots, W_k, W$, such that for each $W_1, \ldots, W_k, W$, the corresponding rr-*read* returns the prefix/rank pair written by the preceding rr-*write* in the chain.

We now show by induction on the length k of the chain that π is an extension of π_0. If $k = 0$, then the rr-*read* preceding W reads π_0, and by the ORDER routine, W either writes π_0, or appends an operation/response pair to π_0, and then writes the resulting sequence.

Otherwise, suppose for all chains of length $l < k$, it holds the last rr-*write* writes a sequence π_l which extends π_0, and consider the chain above the length k. For W_k, the (unique) chain from W_0 is $W_0, W_1, \ldots W_k$. By the induction hypothesis, W_k writes a sequence π_k which extends π_0. Hence, again the rr-*read* operation preceding W reads π_k, and according to the ORDER code, W writes a sequence π which is either equal or extends the sequence π_k. □

Theorem 3 (Atomicity). *The algorithm in Figure 2 emulates an atomic object of type $\mathcal{T}$.*

Proof. By Lemma 2, all the operation prefixes written by the committed rr-*write* invocations form a sequence $\bar{\pi} = \pi_0, \pi_1, \ldots, \pi_k$ such that (1) $\pi_0 = \langle\rangle$; and for each $i > 0$: (a) π_i either equal to or extends π_{i-1}; and (b) the rank of rr-*write* which wrote π_i is higher than the rank of rr-*write* which wrote π_{i-1}. Since according to the algorithm, the operation result is obtained by applying all the operations in a committed prefix to the initial object state, the returned results are consistent with $\bar{\pi}$. Moreover, since no lower ranked rr-*write* can commit if it is invoked after a higher ranked rr-*write* has committed, $\bar{\pi}$ preserves the temporal order of non-concurrent rr-*write* operations. □

Finally, the next theorem asserts the liveness:

Theorem 4 (Liveness). *If a correct process i invokes submit$(op)_i$, then there exists a process j such that submit$(op)_j$ eventually returns.*

Proof. The proof is based on the same argument as the proof of Theorem 2. □

The liveness property above is notably weak. In particular, it does not guarantee that every submit operation terminates, but rather, provides a global guarantee of progress; singular operations could in principle be starved. The usual approach to transform this guarantee into a proper liveness provision is to establish a set-object into which pending operations are thrown. Each leader should then help set an order on any operation in the set, and thus, eventually all operations are ordered. We omit the details of this mechanism for brevity.

6 Implementing a ranked register

```
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:
    lock x:
        V ← rmw-read(r)
    unlock x
    return V

rr-write(⟨r, v⟩)_i:
    lock x:
        status ← rmw-write(r, v)
    unlock x
    if (status = ack)
        return commit
    return abort

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

rmw-write(r, v):
    if (x.rR ≤ r ∧ x.wR < r)
        x.wR ← r
        x.val ← v
        return ack
    return nack
```

Figure 3: An implementation of a single ranked register

In this section, we deal with the problem of implementing a wait-free shared ranked register. First, in Section 6.1, we specify how a single ranked register is implemented from a read-modify-write object. Second, in Section 6.2, we present a wait-free self-construction of the ranked register for the NR-Crash failure model.

6.1 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 [30]). 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 $X = (Ranks \times Ranks \times Vals) \cup \{\langle r_0, r_0, \bot\rangle\}$ 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 rr-*read* operation has been invoked; $x.wR$ holds the maximum rank with which a rr-*write* operation has been invoked; and $x.val$ holds the current register value. The implementation pseudocode is depicted in Figure 3. It is quite straight-forward: read returns the currrent value of the register, and records its own rank. Write checks wether a higher ranking read was invoked, aborts if yes, and if not, modifies the value of the register and records its own rank. For clarity, invocations of read-modify-write operations rmw-*read* and rmw-*write* are enclosed within "lock" and "unlock" statements, to indicate that they execute indivisibly.

Lemma 3. *The pseudocode in Figure 3 satisfies Safety.*

Proof. That a rr-*read* operation can only return a valid value that was actually used in a rr-*write* operation or $\langle r_0, \bot\rangle$ is obvious from the code. Now consider a rr-*write* operation $W_1 = \text{rr-}write(\langle r_1, v_1\rangle)_i$ that commits and let $R_2 = \text{rr-}read(r_2)_j$, $r_2 > r_1$ 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 4. *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 5. *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 5. *The pseudocode in Figure 3 is an implementation of a ranked register.*

6.2 A fault-tolerant construction of a ranked register for NR-Crash

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. This construction is also straight-forward: Reading and writing are both done at a majority of the ranked registers. As for rr-*write*, if any of the ranked registers which are accessed returns *abort*, the operation aborts.

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

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

Process i:

rr-read(r)_i:
    S_1 ← ∅
    Invoke rr-read(r)_i in parallel on the ranked registers in {rr_j}_{1...n};
    Accumulate responses in S_1; 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 rr-write(⟨r, v⟩)_i in parallel on the ranked registers in {rr_j}_{1...n};
    Accumulate responses in S_2; wait until |S_2| ≥ ⌈(n+1)/2⌉;
    if (abort ∈ S_2)
        return abort
    return commit
```

Figure 4: A fault-tolerant ranked register construction for NR-Crash

Lemma 6. *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 7. *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. ☐

Lemma 8. *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 6. *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.*

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

Our construction of a fault tolerant ranked-register requires strong (read-modify-write) base objects. In this section we briefly address the natural question of whether this strong memory model is necessary. 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, even if clients are failure-free. The main result of this section is expressed in Theorem 7 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 [9] to prove lower bounds on the number of atomic registers needed to solve mutual exclusion. Though the proof technique below is standard, it should be noted that there is no known direct reduction from mutual exclusion to a ranked register, and hence, the results of [9] do not apply directly to the impossibility of constructing a ranked register. In fact, we conjecture that the ranked register is strictly weaker than the mutual exclusion problem, and hence, that no such reduction is possible.

We start with some definitions. 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 9. *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 10 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 9, 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 10. *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 $1 \ldots k$ 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 9, 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 9, 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 7. *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 10 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 9, 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. □

Acknowledgments

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

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

[8] 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.

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

[10] 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.

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

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

[13] G. V. Chockler, I. Keidar and D. Malkhi. Computing with Byzantine storage. *In preparation.*

[14] G. Chockler, D. Malkhi and D. Dolev. State-Machine Replication with Infinitely Many Processes: A position Paper. In proceedings of the International Workshop on Future Directions in Distributed Computing (FuDiCo), Bertinoro, Italy, 2002.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

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

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

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

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

[32] 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.

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

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

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

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

[37] 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.

[38] 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.

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

[40] 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.

[41] D. Malkhi. From Byzantine Agreement to Practical Survivability. In the International Workshop on Self-Repairing and Self-Configurable Distributed Systems (RCDS 2002) October 2002, Osaka, Japan.

[42] 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.

[43] 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.

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

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

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

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

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

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

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

[51] 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.