International Journal of Parallel Programming, Vol. 34, No. 2, April 2006 (© 2006)
DOI: 10.1007/s10766-006-0008-z

Light-Weight Leases for Storage-Centric Coordination

Gregory Chockler[1,3] and Dahlia Malkhi[2]

Reaching agreement among processes sharing read/write memory is possible only in the presence of an eventual unique leader. A leader that fails must be recoverable, but on the other hand, a live and well-performing leader should never be decrowned. This paper presents the first leader algorithm in shared memory environments that guarantees an eventual leader following global stabilization time. The construction is built using light-weight *lease* and *renew* primitives. The implementation is simple, yet efficient. It is *uniform*, in the sense that the number of potentially contending processes for leadership is not a priori known.

KEY WORDS: Leases; file systems; mutual exclusion; consensus.

1. INTRODUCTION

An attractive way to model a distributed system in which clients access information on remote servers is a shared memory system, in which processes interact through access to shared persistent objects. A quintessential building block for coordination in such settings is distributed agreement. However, in order to allow wait-free agreement to be solved, it is known that the environment must support an eventually unique leader.(30) Intuitively, this requisite enables a unique leader to be established and enforce a decision. In this paper we focus on the problem of implementing such

[1]IBM Haifa Labs, Haifa University Campus, Mount Carmel, Haifa 31905, Israel. E-mail: Chockler@il.ibm.com
[2]School of Computer Science and Engineering, The Hebrew University of Jerusalem, and Microsoft Research, Silicon Valley. E-mail: dalia@cs.huji.ac.il.
[3]To whom correspondence should be addressed.

eventually safe leader election from the bare shared memory environment, under an eventual synchrony assumption.

Our approach introduces as a building block the abstraction of an *eventual lease*. Informally, a lease is a shared object that supports a CONTEND operation, such that when CONTEND returns 'true' at any process, it does not return 'true' to any other process for a pre-designated period. The lease automatically expires after the designated time period. In addition, our lease supports a RENEW operation which allows a non-faulty leader to remain in leadership (indefinitely).

Leases and Mutual Exclusion. A lease is substantially different from a mutual-exclusion object, and hence is not solved by the vast literature on mutual exclusion. By its very nature, it becomes possible for other processes to recover an acquired lease regardless of the actions of the others, including the case that the process that held the lease has failed. Additionally, a lease may not be safe for an arbitrarily long period. Indeed, in an eventual (or intermittently) synchronous settings, any lease's pre-designated exclusion period entails no safety guarantee during periods of asynchrony. However, when the system stabilizes, all previous (possibly simultaneous) leases expire, and safety is recovered by the very nature of leases. Thus, despite any transient periods of instability, leases guarantee that once a system becomes synchronous for sufficiently long, it will be possible for processes to acquire exclusive leases. Renewals also provide automatic recovery: Only one renewal emerges successfuly after system stabilization, despite any unstable past periods, and despite the possibile existence of multiple simultaneous lease holders before the stability.

The Model. This paper provides the first implementation of leases from regular shared mutli-reader/multi-writer *read/write* registers.[13] The model of synchrony we employ, called the *Eventual Known Delay Timed* ($\diamond$ND) model, is an extension of the timed-asynchronous communication model of [17] to shared memory systems. In addition to arbitrarily long period of asynchrony, the model admits the following types of faults. First, by the very definition of leases, any acquired lease is recoverable, even if the process that holds it becomes slow or crashed. Second, we may employ standard constructions of a reliable regular register from a collection of fail-prone shared objects (see e.g., Ref. 13). In this way, leases can be constructed from a set of shared objects exhibiting non-responsive memory faults of up to a minority.

Uniformity. It is known that supporting mutual exclusion with read/write registers incurs a cost that is linear in the maximum potential number of participating processes, in terms of both the memory consumption and the number of shared memory accesses.[12] Indeed, many similar abstractions such as failure detectors, or the Ω leader oracle of Ref. (16) are defined for a group of known members. To circumvent this limitation,

we adopt a timing-based locking approach that was originally suggested by Fischer.[26] This results in a very simple locking protocol, that uses a *single* read/write register to support exclusion among a priori unknown (but eventually finite) number of client processes.

We enhance Fischer's scheme with a number of important modifications. First, in order to support automatic recovery of the locks held by failed processes, we augment the scheme with an expiration mechanism so that a lock is *leased* to a process for a pre-defined time period. Once the lease period expires, the lock is relinquished and subsequently, can be granted to another process. (In the following, we will use terms locks and leases interchangeably). Another important extension we present is the support for automatic lease renewal. This leads to efficient utilization of the lease by a leader who holds the lease and continues doing useful work.
Reaching Coordination. The common approach for reaching consensus among multiple servers in such tasks is to employ the Paxos paradigm.[27] This paradigm preserves uniqueness of decisions through a three phase commit protocol, and relies on timeliness conditions for progress. Our leases serve as a fundamental enabler of the Paxos paradigm in storage-centric systems, and a necessary building block for the agreement algorithms in Refs 14, 19. Our leases guarantee exclusion to clients once the system stabilizes (and remains stable for long enough), regardless of any past timing violations. This allows our lease to support an eventual leader-election primitive, a necessary building block for implementing dependable services resilient to timing failures.

We show a simple lease based solution for fault-tolerant consensus that guarantees agreement at all times but can fail to make progress when the system is unstable. The latter can be used to realize efficient, always safe fault-tolerant locking using a hierarchical approach described by Lampson in Ref. 28.
Contribution. Our work provides the following formal contribution. It gives a specification of leases, including a renewal operation. It provides an efficient way to implement leases for an unbounded number of unreliable client processes. The solution applies ideas originally developed for mutual exclusion in synchronous shared memory to derive light-weight lease primitives for highly decentralized and unreliable distributed settings. Finally, we show a simple lease based solution for fault-tolerant Consensus which is a benchmark distributed coordination problem.

2. RELATED WORK

2.1. Time Based Mutual Exclusion

Algorithms for mutual exclusion in the presence of failures must be based on timeliness assumptions, as they have to be able to attain progress

in spite of process failures while executing in their critical section. There are two commonly used timing assumptions in this context: The *known delay model* of Refs 3–5 and the *unknown delay model* of Ref. 2.

The known delay model was first formally defined in Ref. 4. The first mutual exclusion algorithm explicitly based on the known delay assumption was the famous Fischer algorithm, which was first mentioned by Lamport in Ref. 26. In Ref. 26, another timing based algorithm is presented. This algorithm assumes a known upper bound on time a process may spend in the critical section.

Alur et al. consider in Ref. 2 the unknown delay model: The time it takes for a process to make a step is bounded but unknown to the processes. The paper presents algorithms for mutual exclusion and Consensus in this model. A remarkable feature of these algorithms is their ability to preserve safety even in completely asynchronous runs. However, they are guaranteed to satisfy progress only if the system behaves synchronously throughout the entire run. The mutual exclusion algorithm of Ref. 32 combines the ideas of Fischer and Lamport's fast mutual exclusion algorithm[26] to derive a timing based algorithm that guarantees progress when the system stabilizes while being safe at all times. However, the algorithm of Ref. 32 is not fault-tolerant.

As far as we know the eventual known delay timed ($\diamond$ND) model introduced in this paper was never considered in the shared memory context. Most of the existing time based algorithms are either not fault-tolerant,[4,5] or resilient only to the timing failures.[2,32] The fault-tolerant (wait-free) timing based algorithms of Ref. 3 are not suitable for the $\diamond$ND model as they might violate safety and/or liveness even during synchronous periods if the delay constraints do not hold right from the beginning of the run.

The $\diamond$ND model considered in this paper is an extension of a standard asynchronous shared memory model to include timeliness assumptions based on the absolute real-time. To this end, the $\diamond$ND model postulates the existence of bounded drift local hardware clocks accessible to each process. In this respect, the $\diamond$ND model closely resembles the timed asynchronous model of Cristian and Fetzer defined in Ref. 17. An alternative approach to model timeliness in shared memory environments is to postulate the existence of a known upper bound on relative process speeds as it is done by Lynch and Shavit in Ref. 32. This results in a model analogous to the partial synchrony model of Ref. 18. However, as is, the partial synchrony model of Ref. 32 is inappropriate for our purposes as it does not distinguish between local process steps and those involving a shared memory access. This distinction is important if non-atomic shared objects (such as regular registers) are assumed. Relaxing the partial synchrony model of Ref. 32 to allow

non-atomic memory access as well as evaluating applicability of other timed models (e.g., Ref. 1, or the timed I/O automata model of Ref. 25) remains a subject of the future work.

Other properties that are of interest to us is the ability of timing based algorithms to support exclusion among arbitrarily many client processes and to work with weaker registers and/or a small number thereof. The latter is particularly important in failure prone environments as in these environments the registers must be first emulated out of possibly faulty components. In this respect the original solution by Fischer is superior to all the other algorithms as it is based on a single multi-writer multi-reader register. In fact, as we show in this paper, the register is only required to support regular semantics (in the sense of Ref. 13), and hence may be emulated efficiently even in a message passing setting. This solutions was therefore chosen as a basis for our lease implementation. The algorithms of Refs 32 and 4 are also oblivious to the number of participants and use two and three shared atomic registers, respectively.

The goodness of timing based mutual exclusion algorithms are frequently assessed in terms of their performance in contention free runs. In particular, a good algorithm is expected to avoid delay statements when there are no contention. The performance of the timing based algorithms under various levels of contention is analyzed in Ref. 20. The paper examines (both analytically and in simulations) the expected throughput of timed based mutual exclusion algorithms under various statistical assumptions on the arrival rate and the service time. The question of further optimizing our leases approach for contention free runs is left for future research.

2.2. Other Work on Locks and Leases

Gray and Cheriton were the first to employ leases in Ref. 23 for constructing fault-tolerant distributed systems. Lampson advocates in Ref. 28, 29 the use of leases to improve the Paxos algorithm. Boichat et al.[10] introduce asynchronous leases as an optimization to the atomic broadcast algorithms based on the rotating coordinator paradigm. Chockler et al.[15] show a randomized backoff based algorithm for implementing leases in a setting similar to the $\diamond$ND model of this paper. However, the algorithm of Ref. 15 guarantees progress only probabilistically, and relies on shared objects that can measure the passage of time. Finally, Cristian and Fetzer[17] show an implementation of leases in timed asynchronous message passing systems.

2.3. Locking Support in SAN-based File Systems

Our work is of independent practical importance in storage area network (SAN)-based file systems. In recent years, advances in hardware technology have made possible a new approach for storage sharing, in which clients access disks directly over a high-speed network. By allowing the data to be transferred directly from network attached disks to clients, SAN has the potential to improve scalability (through eliminating the file server bottleneck) and performance (through shorter data paths). However, without properly restricting concurrent access to shared data by clients, shared data would be rendered inconsistent. Therefore, a scalable and efficient locking support is widely recognized as a key requisite for realizing the SAN technology's full potential.

Traditionally, SAN-based file systems rely on separate servers to maintain their meta-data and coordinate access to the user data on storage devices.[11,33] The meta-data servers can be replicated for better availability and load balancing. The server replicas are kept in a consistent state using a group-communication substrate. However, the cluster of replicated meta-data servers still remains the performance and availability hotspot as all the file-system operations (even those targeted to different objects) must consult the meta-data servers before accessing the storage. Examples of the systems whose design follows this approach include the IBM General Parallel File System (GPFS)[38] and IBM StorageTank.[33] More examples can be found in Ref. 22.

The vision of a storage-centric locking was first realized in the Global File System (GFS) project[35,39,40] developed in the University of Minnesota. In GFS, the cluster nodes physically share storage devices connected via a high-speed network. GFS utilizes fine grain test-and-set locks provided by specialized SCSI devices[9,36] to implement atomic execution of file system operations.

Amiri et al.[6] proposes base storage transactions (BSTs) as a core paradigm for maintaining low-level integrity of striped storage (such as RAID) in the face of concurrent client accesses. In particular, the paper discusses *device-served locking* as an alternative to traditional centralized locking schemes. It demonstrates through an extensive empirical performance study that device-served locking provides better performance under high contention, and is therefore, more scalable.

zFS[22,37] is a research file system implemented over object store devices[34] directly accessible over a SAN. In zFS, each storage device maintains a coarse grain lock which can be used by a lease manager to obtain an exclusive access (a major lease) to the entire device. The lease manager is then responsible for administering fine grain locks to clients requesting access to individual data items stored on the device.

The symmetrical locking mechanisms above all guarantee availability of lock information in face of process failures. However, none of these systems support data and lock replication and therefore, do not guarantee availability in the face of storage device failures. As a partial solution, a reliability hardware (such as RAID) may be employed in these systems to mask the storage failures to some extent. In addition, both GFS and zFS require sophisticated storage hardware which must be able to support read–modify–write instructions and, in the case of zFS, also be capable of measuring real time passages.

An alternative approach, put forth in this paper, is to employ a *storage-centric* locking, i.e., to co-locate locks with the very data items that are protected by these locks. This way, the cost of locking is folded into the cost of accessing the data itself, and the locks availability is the same as that of the data itself. The challenge is in providing an efficient and fault-tolerant implementation.

A naive per-datum lock design would associate a strong object that directly implements locking (such as test-and-set) with each data item. However, this approach has several drawbacks: First, it necessitates a sophisticated support on behalf of the storage hardware such as SCSI controllers enhanced with device locks (see Refs 9, 36), or object store controllers (see Refs 22, 37). These hardware enhancements still remain proprietary and it is unclear whether they will be accepted by the storage manufacturers in the future.

Second, data is frequently replicated on several storage units (e.g., a file may be striped, or mirrored) for availability and fault-tolerance. As a result, it is desirable to have the locks replicated as well so that the same level of availability is preserved. Unfortunately, as it was proved in Ref. 24, it is impossible to use a collection of fail-prone strong objects (such as test-and-set, compare-and-swap, etc.) to implement a reliable one.

We therefore opt for an alternative approach which is to build locks from weaker objects, i.e., read/write registers. Thus, deployment becomes a non-issue, as designating a read/write word per file or per block on a disk is trivially done. In case that multi disk locking is required, a single **reliable** read/write register is implementable using a farm of failure-prone storage units (see, e.g., Refs 7, 8, 13).

3. SYSTEM MODEL

We will start by defining a basic asynchronous shared memory model and the regular register properties (Section 3.1). We will follow the basic formalism of Ref. 13. Then, in Section 3.3, we augment the basic model

with necessary timeliness assumptions by adapting the timed asynchronous model of Ref. 17 to the shared memory environment.

3.1. The Basic Model

Our basic model is an asynchronous shared memory model consisting of finite but *a priori* unknown universe of processes $p_1, p_2, \ldots$ communicating by means of a finite collection of shared objects, $O_1, \ldots, O_n$. Every shared object has a *sequential specification* defining the object behavior when accessed sequentially. A sequence of operations on a shared object is *legal* if it belongs to the sequential specification of the shared object. In this paper, we reduce our attention to read/write shared objects. A sequence of operations on a read/write shared object is *legal* if each read operation returns the value written by the most recent write operation if such exists, or an initial value otherwise.

The operations on objects have non-zero duration, commencing with an *invocation* and ending with a *response*. An *execution* of an object is a sequence of possibly interleaving invocations and responses. For an execution σ and a process p_i, we denote by $\sigma|i$ the subsequence of σ containing invocations and responses performed by p_i. Processes may fail by crashing. A process is called *correct* in an execution σ if it never crashes throughout σ. Otherwise, a process is called *faulty* in σ. A threshold t of the objects may suffer non-responsive crash failures,[24] i.e., may stop responding to incoming invocations.

An execution σ is *admissible* if the following is satisfied: (1) Every invocation by a correct process in σ has a matching response; and (2) For each process p_i, $\sigma|i$ consists of alternating invocations and matching responses beginning with an invocation. In the rest of this paper, only admissible executions will be considered.

Given an execution σ, we denote by $ops(\sigma)$ (resp. $write(\sigma)$) the set of all operations (resp. all *write* operations) in σ; and for a *read* operation r in σ, we denote by $writes_{\leftarrow r}$ the set of all *write* operations w in σ such that w begins before r ends in σ. The operations in $ops(\sigma)$ are partially ordered by a $\rightarrow_\sigma$ relation satisfying $o_1 \rightarrow_\sigma o_2$ iff o_1 ends before o_2 begins in σ. In the following, we will often omit the execution subscript from $\rightarrow$ if it is clear from the context.

Our definition of regularity for a multi-reader/multi-writer read/write shared object is similar to the **MWR2** condition of Ref. 13. It is as follows:

Definition 1 (Regularity). An execution σ satisfies regularity if there exists a permutation π of all the operations in $ops(\sigma)$ such that for any *read* operation r, the projection π_r of π onto $writes_{\leftarrow r} \cup \{r\}$ satisfies:

1. π_r is a legal sequence.
2. π_r is consistent with the $\rightarrow$ relation on $ops(\sigma)$.

A *read/write* shared object is regular if all its executions satisfy regularity.

3.2. Masking Object Failures

Given a collection of $n > 2t$ shared objects up to t of which can suffer from non-responsive crash failures, it is possible to construct a wait-free regular register defined in the previous section (see e.g., Refs 8, 13). The resulting reliable registers can then be used to construct higher level services. Hence, in this paper we will follow a modular approach: i.e., we will assume that reliable registers are available, and develop algorithms in a shared memory model with reliable registers.

3.3. The Augmented Model

In the augmented model, each process is assumed to have access to a hardware clock with some predetermined granularity. We also assume that each process can suspend itself by executing a *delay* statement. Thus, a call to $delay(t)$ will cause the caller to suspend its execution for t consecutive time units. We model the system behavior as a *General Timed Automaton* (GTA)[31] which is a state machine augmented with special *time-passage* events $\nu(t)$, $t \in \mathbb{R}$. The time-passage event $\nu(t)$ denotes the passage of real time by the amount t.

The system is called *stable* over a time interval $[s, t]$, called a *stability period*, if the following holds during $[s, t]$: (1) The processes' clock drift with respect to the real-time is bounded by a known constant ρ. For simplicity we assume that $\rho = 0$ (it is easy to extend our results to clocks with $\rho \neq 0$); and (2) The time it takes for a correct process to complete its access to a shared memory object, i.e., to invoke an operation and receive a reply, is strictly less than a known bound δ.

In the following, we will be interested mainly in properties exhibited by the system during stability periods. To simplify the presentation, we will consider a timed model, which we call an *Eventually Known Delay Timed* model, or $\diamond$ND, with stability periods of infinite duration: i.e., we assume that for each run there exists a *global stabilization time* (GST) such that the system is stable forever after GST (i.e., during $[GST, \infty)$). In the remainder of the presentation, all properties and correctness proofs regard operations as starting after GST.

We will also consider a special case of the $\diamond$ND model, which we call a *Known Delay Timed* model, or ND, that requires each run to be stable right from the outset.

4. THE LEASE SPECIFICATION WITHOUT RENEWALS

We define the Δ-*Lease* object as a shared memory object that can be concurrently accessed by any number of processes, and whose interface for each process i consists of a single operation CONTEND$_i$. The response to the CONTEND operation is ack_i. We assume that the interaction between each process i and the lease object is *well-formed* in the sense that it is consistent with the state diagram depicted in Fig. 1.

A process that is not holding a lease is in the state *Free*. We assume that each process execution always starts from the Free state. A process that attempts to acquire the lease, invokes CONTEND and moves to the state *Try*. Once CONTEND returns, the process moves to the state *Hold* assuming the lease for the next Δ time units. Once the lease expires, the process moves back to the Free state.

In the states Free and Hold, the process executes the code specified by the application program. We do not put any restrictions on the time spent in the Free state (indicated by $t \geqslant 0$ time passage).

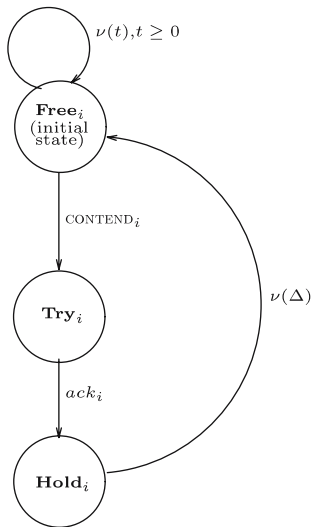

Fig. 1. Well-formed interaction of process i and the Δ-Lease object.

A Δ-*Lease* object is required to satisfy the following property after time $t \geqslant GST$:

Property 1. At any point in an execution, the following holds:

1. *Safety*: At most one process is in the Hold state.
2. *Contend Progress*: If no process is in the Hold state, and some correct process is in the Try state, then at some later point some correct process enters the Hold state.

5. THE LEASE IMPLEMENTATION

The Δ-*Lease* object implementation appears in Fig. 2. It utilizes a single shared multi-reader multi-writer regular register x. A process that tries to acquire the lease writes a unique timestamp to the register x and delays for 2δ time. If upon the delay expiration, the process reads its own value back, then it acquires the lease and enters the Hold state. Otherwise, it backs off to the loop in lines 3–7, where it waits until the lease period Δ expires. Note that each process has to write a unique timestamp (e.g., id and a sequence number) into x. This is necessary in order to prevent a process that acquires the lease for several times in a row from being falsely suspected by other processes.

We now prove that the implementation in Fig. 2 satisfies the Δ-Lease object properties.

Throughout the proof, we make use of the following assumptions and notations. Let L be a CONTEND operation. We denote the sequence of *read*/*write* operations by which L terminates by:

$$L.r', (\text{delay } \Delta + 5\delta), L.r'', L.w, (\text{delay } 2\delta), L.r.$$

That is, denote by $L.w$ the last *write* operation invoked during L (i.e., the last time line 9 in Fig. 2 is activated). Denote by $L.r$ the *read* operation that follows $L.w$ (on line 11), and by $L.r''$ the one immediately preceding $L.w$. Let $L.r'$ be the last *read* operation during L from line 1 or line 11 that precedes $L.w$.

Finally, for the execution considered in all proofs, let π be a serialization of the operations that upholds the regularity of x.

Lemma 1. Let L_0 be a CONTEND operation invoked by process p that returns at time t_0. Denote $s_0 = t_0 + \Delta$ the expiration time of L_0. Then for all CONTEND operations L such that $L.w$ appears in π after $L_0.w$, $L.r''$ is invoked after $s_0 + \delta$.

```
Shared:
  x ∈ TS_⊥;
Local:
  x_1, x_2 ∈ TS_⊥.

CONTEND:
(1)   x_2 ← read(x);
(2)   do
(3)     do
(4)         x_1 ← x_2;
(5)         delay(Δ + 5δ);
            /* Δ + 6δ for the ◊ND renewals */
(6)         x_2 ← read(x);
(7)     until x_1 = x_2;
(8)     Generate a unique timestamp ts;
(9)     write(x, ts);
(10)    delay(2δ);
(11)    x_2 ← read(x);
(12)  until x_2 = ts;
(13)  return ack;
```

Fig. 2. The Δ-Lease Implementation.

Proof. Assume to the contrary, and let L be a CONTEND operation such that $L.w$ is the first *write* in π that breaks the conditions of the lemma.

Clearly, $L.w$ does not precede $L_0.r$ in $\pi_{L_0.r}$, for else $L_0.r$ cannot return the value written by $L_0.w$. Furthermore, since all *write* operations w such that $w \rightarrow L_0.r$ must appear in $\pi_{L_0.r}$ before $L_0.r$, and because by assumption $L_0.w$ precedes $L.w$ in π, $L.w \not\rightarrow L_0.r$. Putting this together with the fact that the response of $L_0.w$ and the start of $L_0.r$ are separated by a 2δ delay, we have $L_0.w \rightarrow L.r''$ (see Fig. 3(a)). Hence, $L_0.w \in \pi_{L.r''}$.

Next, we show that $L_0.w$ is the last *write* preceding $L.r''$ in $\pi_{L.r''}$. Let $L' \neq L$ be a CONTEND operation such that $L'.w$ is between $L_0.w$ and $L.r''$ in $\pi_{L.r''}$. By assumption, $L'.r''$ must be invoked after $s_0 + \delta$. Since, by definition of $\pi_{L.r''}$, $L'.w$ must be invoked before $L.r''$ returns, $L.r''$ returns after $s_0 + \delta$, as depicted in Fig. 3(b). Since $L'.w$ is invoked after $s_0+\delta$, and since by assumption, $L.r'$ finishes before $s_0+\delta$, we get that $L.r' \rightarrow L'.w$. Putting this together with the assumption that $L'.w$ precedes $L.r''$ in $\pi_{L.r''}$, we obtain that $L.r'$ and $L.r''$ will return different values in which case the lease implementation implies that the *write* statement is not reached. Hence, $L.w$ could not have been invoked. Thus, $L_0.w$ is the last

write preceding $L.r''$ in $\pi_{L.r''}$ implying that $L.r''$ returns the value written by $L_0.w$.

By construction, $L.r''$ is preceded by a $5\delta + \Delta$ delay preceded by another *read* operation $L.r'$ such that the timestamp values returned by these two *read*'s are identical. However, it is easy to see that $L_0.w$ is contained in full between these two reads. Indeed, we already know that $L_0.w \to L.r''$. We now show that $L.r' \to L_0.w$. Indeed, the earliest time that $L_0.w$ can be invoked is $s_0 - \Delta - 4\delta$. Since by assumption $L.r''$ is invoked before $s_0+\delta$, $L.r'$ **returns** before $s_0+\delta-(\Delta+5\delta) = s_0-\Delta-4\delta$ (see Fig. 3(b)). Therefore, $L.r' \to L_0.w$. Thus, regularity of x and the timestamp uniqueness imply that $L.r'$ and $L.r''$ return different timestamps in which case the lease implementation implies that the *write* statement is not reached. Hence, $L.w$ could not have been invoked. A contradiction.
We are now ready to prove Safety.

Lemma 2 (Safety). The implementation in Fig. 2 satisfies Property 1.1.

Proof. Let L be a CONTEND operation by process p that returns at time t. Denote $s = t + \Delta$. Suppose to the contrary that another CONTEND operation L' returns at time t' within the interval $[t, s]$.

First, $L'.r''$ must be invoked before $s+\delta$. By Lemma 1, putting $L_0 = L$ we get that $L.w$ does not precede $L'.w$ in π. Second, $L.r''$ must be invoked before t', and a fortiori, before $t' + \Delta + \delta$. Applying Lemma 1

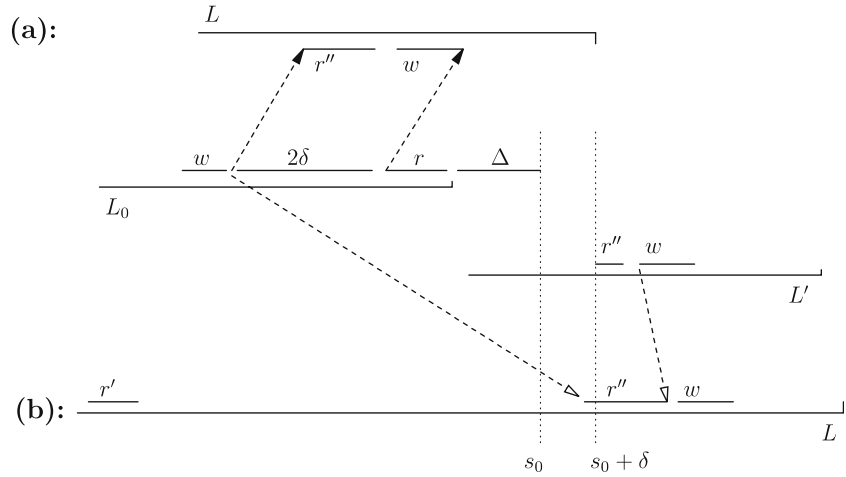

Fig. 3. Possible placements of overlapping CONTEND operations L_0 and L.

again, with $L_0 = L'$, we get that $L.w'$ does not precede $L.w$ in π. A contradiction.
We now turn our attention to proving Progress. We first prove the following technical fact.

Lemma 3. Let q be a process that performs an operation $w_1 = write$ that returns at time t. If no process returns from a CONTEND operation after t, then for each $s > t$, the interval $[s, s + 5\delta]$ contains a complete write invocation (i.e., from its invocation to its response).

Proof. Suppose to the contrary. By assumption, no *write* operation is invoked between s and $s + 4\delta$. Let W be the last *write* invoked before s, or possibly the set of concurrent, latest *write*s invoked before s. Formally, W is the set of all w such that (1) w is invoked before s; and (2) for any write w' invoked by $s + 4\delta$, $w \not\to w'$. W is not empty because w_1 starts before s, and no write is invoked in the interval $[s, s + 4\delta]$.

Let $w \in W$, and let $r = read$ be the corresponding read operation, invoked by the same process 2δ after w. We claim that (i) $W \to r$, and (ii) there does not exist any *write* event ω in π_r that follows W in π such that $W \to \omega$ and ω is invoked before r returns.

To see that (i) holds, let $w' \in W$. Since $w \not\to w'$, we have that w' terminates at most δ after w; since r starts 2δ after w's termination, $w' \to r$. To see (ii), first note that if $W \to \omega$, then by definition ω cannot be invoked before s. Second, by assumption, no *write* is invoked between s and $s+4\delta$, but r terminates by $s+4\delta$ at the latest. So ω cannot be invoked before r returns, and hence is not in π_r.

Hence, by the regularity of x, all *Read's* corresponding to *write*'s in W must return the value of the last *write* in π from W. The *read* corresponding to this *write* then sees x unchanged, and its initiator is allowed to obtain the lease. A contradiction.

Lemma 4 (Progress). The implementation in Fig. 2 satisfies Property 1.2.

Proof. Suppose that no process is holding the lease at time t. Let p be a correct process that is still contending at t. Suppose for contradiction that no CONTEND operation returns after t.

First, eventually some process, say q_1, invokes an operation $w_1 = write$. This is due to the fact that the wait-loop at the start of the CONTEND algorithm (lines 2.3–7) terminates at some process when no *write*'s are performed.

By Lemma 3, if there is no successful CONTEND after w_1 returns, then every instance of the loop by q_1 observes at least one new written value. Thus, the test in line 7 2 remains false. Hence, q_1 does not perform any further *write*'s. Let an operation $w_2 = write$ by q_2 be observed by q_1. Again, so long as there is no successful CONTEND, by Lemma 3, q_2 performs no further *write*'s. And so on.

Since the number of processes is finite, eventually all processes are in their wait loop and no process writes. This is a contradiction.
Hence, we proved the following:

Theorem 1. The implementation in Fig. 2 satisfies Property 1.

6. LEASE RENEWALS

In many situations, it is important to enable the current lease holder to renew its lease without contention. For example, this is the case when a lease holder requires more time to complete an operation than the alloted period. Another example is the use of leases to obtain a leader, in which case we wish the leader to perpetuate so long as it is alive.

In this and the following section, we consider lease renewals. We start by extending the lease specification in Section 4 to include lease renewals.

The Δ-*Lease* object with renewals supports for each process i, an additional RENEW$_i$ operation whose response is either $true_i$ or $false_i$. The extended well-formedness condition is given by the state diagram depicted in Fig. 4. It introduces the *Renew* state where the renew implementation code is executed, and the *Exit* state where an application can attempt lease renewal by calling the RENEW operation. If the call to RENEW returns $true$, the process assumes the lease for another Δ time units. Otherwise, it returns to the state Free. Note that we assume that the transition from state Exit to the Renew state is instantaneous (indicated by a 0 time passage). Note also that a process is allowed to renew its lease for several times in a row.

In addition to Property 1, a Δ-*Lease* object with renewals is required to satisfy the following properties after time $t \geqslant GST$:

Property 2. At any point in an execution, the following holds:

1. *Renewal Safety*: If a correct process i is in the Renew state, then no other process is in the Hold state.
2. *Renewal Progress*: At any point in an execution, if a correct process i is in the Renew state, then at some later point process i enters the Hold state.

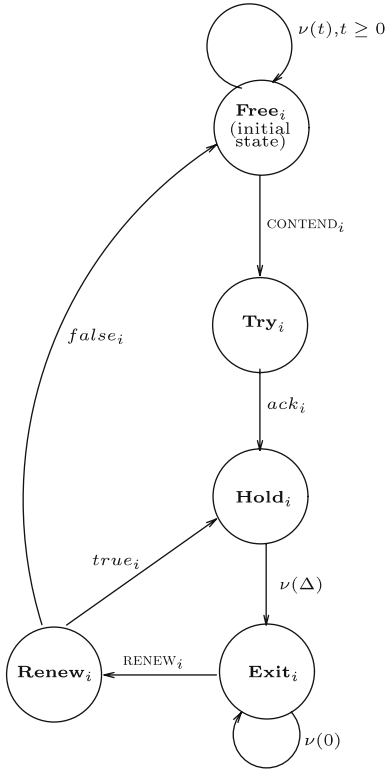

Fig. 4. Well-formed interaction of process i and the Δ-Lease object with renewals.

7. IMPLEMENTING RENEWALS

In this section we address the lease renewals implementation. We consider two implementation options: The first one is suitable for the ND model, and is extremely efficient. The second one works in the $\diamond$ND model, and guarantees stabilization of renewal: Only one renewal emerges successfully after GST, despite any unstable past periods, and despite the possible existence of multiple simultaneous lease holders before GST. The $\diamond$ND renewal protocol is somewhat more costly.

7.1. ND Renewal

The renewal implementation in the ND model is extremely simple: A process whose previously granted lease expires can renew it for another

Δ time units by simply executing lines 8–9 of the Δ-*Lease* implementation in Fig. 2. More precisely, we define the *renew* operation as follows:

RENEW:
Generate a unique timestamp ts;
$write(x, ts)$;
return $true$;

We now prove the correctness of the ND renewal scheme. Since liveness trivially holds, we are only left with proving safety.

Lemma 5. Consider a sequence $\ell = L_0 rn_1 rn_2 \ldots rn_k$ of lease operations by process p. Suppose that L_0 is a successful CONTEND operation that returns at time t_0, and rn_i is a successful RENEW operation that returns at time t_i. Then there exists no CONTEND operation L by process $q \neq p$ such that $L.w$ is invoked within the interval $[t_0, t_k + \Delta + 2\delta]$.

Proof. By induction on length of ℓ. For the base case, let $\ell = L_0 rn_1$. Suppose to the contrary that there exists a CONTEND operation L such that $L.w$ is invoked within $[t_0, t_1 + \Delta + 2\delta]$. First, note that $L_0.w \rightarrow L.w$, and therefore, $L_0.w$ precedes $L.w$ in π. Therefore, by Lemma 1, $L.r''$ must be invoked after $t_0 + \Delta + \delta$. Since $rn_1.w$ is invoked at $t_0 + \Delta$, it must return by $t_0 + \Delta + \delta$, and therefore, $rn_1.w \rightarrow L.r''$. Since $L.r''$ is invoked before $t_1 + \Delta + 2\delta$, $L.r'$ returns before $t_1 + \Delta + 2\delta - (\Delta + 5\delta) = t_1 - 3\delta$. Since $rn_1.w$ must be invoked at $t_1 - \delta$ the earliest, $L.r' \rightarrow rn_1.w$. Therefore, by regularity of x and timestamp uniqueness, $L.r'$ and $L.r''$ will return different values violating the necessary condition for the *write* statement of the CONTEND implementation to be reached. Hence, $L.w$ cannot be invoked. A contradiction.

Assume that the result holds for all sequences ℓ of length $k-1$, and consider a sequence $\ell' = \ell\ rn_k$. Assume to the contrary. By the inductive assumption, $L.w$ must be invoked after $t_{(k-1)} + \Delta + 2\delta$. Therefore, $rn_k.w \rightarrow L.r''$. On the other hand, $L.r''$ must be invoked before $t_k + \Delta + 2\delta$. Therefore, $L.r'$ must return before $t_k - 3\delta$. Since the earliest time $rn_k.w$ can be invoked is $t_k - \delta$, $L.r' \rightarrow L.w$. Therefore, by regularity of x and timestamp uniqueness, $L.r'$ and $L.r''$ will return different values violating the necessary condition for the *write* statement of the CONTEND implementation to be reached. Hence, $L.w$ cannot be invoked. A contradiction.

Lemma 6. Suppose that a process p returns from a RENEW operation rn at time t. Then, there exists no process $q \neq p$ whose RENEW operation rn' returns within the interval $[t, t + \Delta]$.

Proof. Suppose to the contrary that rn' returns at time t' within the interval $[t, t + \Delta]$. By well-formedness, both p and q must have been invoked contend operations L and L' in the past to acquire their initial leases. Suppose that L and L' return at times $c < t$ and $c' < t'$ respectively. Assume, w.l.o.g, that $c < c'$. By Lemma 5, putting $t_0 = c$ and $t_k = t + \Delta$, and because $t' \leqslant t + \Delta$, we get that the lease period of L' overlaps with $[t_0, s_k]$. A contradiction.
The following lemma follows immediately from Lemmas 5 and 6.

Lemma 7 (ND Renewal Safety). The ND renewal implementation satisfies Properties 1.1 and 2.1.

We proved the following:

Theorem 2 (ND Renewal Correctness). The ND renewal implementation satisfies Properties 1 and 2.

7.2. ◇ND Renewal

The RENEW operation implementation for the ◇ND model is shown in Fig. 5. For simplicity, we require that timestamps consist of two fields: the process id and a monotonically increasing counter.

Throughout the proof of correctness of the ◇ND renewal scheme, we make use of the following notation. Let L be a CONTEND or RENEW operation. As in the previous section, we denote the sequence of *read*/*write* operations by which L terminates by:

(in CONTEND only: $L.r'$, delay$\Delta + 6\delta$), $L.r''$, $L.w$, (delay 2δ), $L.r$.

```
RENEW:
(1)   x_1 ← read(x);
(2)   if (x_1.id ≠ ts.id) then
(3)     return false;
(4)   ts.counter ← ts.counter + 1;
(5)   write(x, ts);
(6)   delay(2δ);
(7)   x_1 ← read(x);
(8)   if (x_1 = ts) then
(9)     return true;
(10)  else
(11)    return false;
```

Fig. 5. ◇ND Renew Implementation.

That is, $L.w$ is the last *write* operation invoked within L, and $L.r''$, $L.r$ and the read operations immediately preceding and following $L.w$, respectively. If L is a CONTEND operation, and there exists a *read* operation invoked from line 6 of Fig. 2, then $L.r''$ denotes the one immediately preceding $L.w$. If $L.r''$ exists, it is immediately preceded by a *read* operation $L.r'$ from line 1 or line 11 of Fig. 2 followed by a delay of $(\Delta + 5\delta)$. Otherwise, let $L.r'$ be the last *read* operation during L from line 1 or line 11 of Fig. 2 that precedes $L.w$. then in addition, the read operation preceding $L.r''$ is denoted $L.r'$.

Finally, for the execution considered in all proofs, let π be a serialization of the operations that upholds the regularity of x.

Lemma 8. Let L_0 be a lease operation (contend or renew) invoked by process p that returns successfully at time t_0. Denote $s_0 = t_0 + \Delta$ the expiration time of L_0. Then there exists no *write* operation w in π after $L_0.w$, such that w is invoked before $s_0 + \delta$.

Proof. Assume to the contrary, and let $L.w$ be the first *write* in π that breaks the lemma.

Clearly, $L.w$ does not precede $L_0.r$ in $\pi_{L_0.r}$, for else $L_0.r$ cannot return the value written by $L_0.w$. Furthermore, since all *write* operations w such that $w \rightarrow L_0.r$ must appear in $\pi_{L_0.r}$ before $L_0.r$, and because by assumption $L_0.w$ precedes $L.w$ in π, $L.w \not\rightarrow L_0.r$. Putting this together with the fact that the response of $L_0.w$ and the start of $L_0.r$ are separated by a 2δ delay, we have $L_0.w \rightarrow L.r''$ (see Fig. 6). Hence, $L_0.w \in \pi_{L.r''}$.

Furthermore, by assumption $L.w$ is the first *write* such that (1) $L.w$ follows $L_0.w$ in π; and (2) $L.w$ is invoked before $s_0+\delta$. Since $L_0.w \in \pi_{L.r''}$ any write $w \neq L.w$ that follows $L_0.w \in \pi_{L.r''}$ must be invoked after $s_0+\delta$. Since, by definition of $\pi_{L.r''}$, w must be invoked before $L.r''$ terminates, $L.r''$ terminates after $s_0 + \delta$. Consequently, $L.w$ would be invoked after

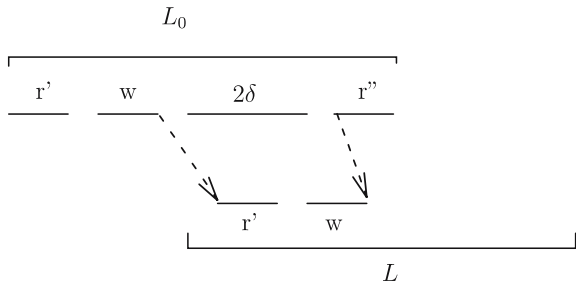

Fig. 6. Overlapping renewals.

$s_0+\delta$ contradicting the assumption. Since $L.w \notin \pi_{L.r''}$, the only remaining possibility is that $L_0.w$ is the last *write* in $\pi_{L.r''}$, and so $L.r''$ returns the value of $L_0.w$.

Next, we consider the case that L is a CONTEND operation separately from the case that it is a RENEW operation. First, consider that L is a RENEW operation. Then the analysis above shows that $L.r''$ returns the timestamp written in $L_0.w$, hence L is unsuccessful.

Second, assume that L is a CONTEND operation. Here, $L.r''$ is preceded by a $6\delta + \Delta$ delay preceded by another *read* operation $L.r'$: and the timestamp values returned by these two *read*'s are identical. However, it is easy to see that $L_0.w$ is contained in full between these two reads. We already know that $L_0.w \to L.r''$. We now show that $L.r' \to L_0.w$. Indeed, the earliest time that $L_0.w$ can be invoked is $s_0-\Delta-4\delta$. Since by assumption $L.w$ is invoked before $s_0+\delta$, $L.r'$ is invoked before $s_0+\delta-(\Delta+6\delta) = s_0 - \Delta - 5\delta$. Therefore, $L.r' \to L_0.w$. Thus, regularity of x and the timestamp uniqueness imply that $L.r'$ and $L.r''$ return different timestamps in which case the lease implementation implies that the *write* statement is not reached. Hence, $L.w$ could not have been invoked. A contradiction.
We are now ready to prove Safety:

Lemma 9. Assume that a lease operation L (CONTEND or RENEW) by process p returns successfully at time t. Let $s = t + \Delta$. Then there exists no successful CONTEND or RENEW operation L' by a process $q \neq p$ that returns during the interval $[t, s]$.

Proof. Suppose to the contrary that L' returns successfully at time t' within the interval $[t, s]$. First, $L'.w$ must be invoked before $s + \delta$. By Lemma 8, putting $L_0 = L$ we get that $L.w$ does not precede $L'.w$ in π. Second, $L.w$ must be invoked before t', and a fortiori, before $t' + \Delta + \delta$. Applying Lemma 8 again, with $L_0 = L'$, we get that $L.w'$ does not precede $L.w$ in π. A contradiction.

Lemma 10. Assume that a RENEW operation L by a process p is invoked at time t_1 and returns successfully at time t_2. Then there exists no successful CONTEND or RENEW operation L' by a process $q \neq p$ that returns during the interval $[t_1, t_2]$.

Proof. Suppose to the contrary that L' returns at a time t' within the interval $[t_1, t_2]$. First, $L'.w$ must be invoked before $s + \delta$. By Lemma 8, putting $L_0 = L$ we get that $L'.w$ must precede $L.w$ in π. Furthermore, applying Lemma 8 again with $L_0 = L'$, we get that $L.w$ must be invoked

after $t' + \Delta + \delta$. Therefore, $L'.w \rightarrow L.r''$ so that $L'.w \in \pi_{L.r''}$, and $L'.w$ precedes $L.r''$ in $\pi_{L.r''}$.

First, suppose that $L.w$ is the first *write* operation by p in π after $L'.w$. Hence, there is no *write* operation by p in $\pi_{L.r''}$ following $L'.w$. Then by regularity of x, and because L is a RENEW operation, $L.r''$ returns a timestamp written by a process $q \neq p$, contradicting to the fact that L is successful.

Next, suppose that there exists a *write* operation $L''.w$ by p in $\pi_{L.r''}$ that follows $L'.w$. Since L is a RENEW operation, L'' must be the successful lease (RENEW or CONTEND) operation immediately preceding L. Applying Lemma 8 with $L_0 = L'$, we get that $L''.w$ must be invoked after $t' + \Delta + \delta$ implying that L starts after $t' + \Delta + \delta$ (i.e., $t_1 > t' + \Delta + \delta$).
We proved the following

Theorem 3 (Renewal Safety). $\diamond$ND renew implementation satisfies Properties 1.1 and 2.1.

Finally, we prove Liveness:

Lemma 11. Assume that a correct process p obtains the lease in a CONTEND or RENEW operation L at time t. Then, a RENEW operation rn invoked by p at $s = t + \Delta$, returns successfully.

Proof. For rn to be successful, first $rn.r''$ must return the timestamp written by $L.w$. This holds by the fact that $L.r$ returns the value of $L.w$, and by Lemma 8, since no other *write* operation that follows $L.w$ in π is invoked before $s + \Delta + \delta$.

Second, $rn.r$ needs to return the value written by $rn.w$. Suppose to the contrary that some lease operation L' overwrites $rn.w$. Let $L'.w$ be the first *write* in π by process $q \neq p$ that follows $L.w$ and precedes $rn.r$ in $\pi_{rn.r}$.

By Lemma 8, $L'.w$ is invoked after $s + \delta$. Hence, $L.w \rightarrow L'.r''$. Since $L.'w$ is the first *write* to follow $L.w$, and since $L'.r'' \rightarrow L'.w$, we have that $L'.r''$ returns the timestamp written by p in $L.w$. By construction, this occurs only if L' is a CONTEND (not RENEW) operation. Still, for $L'.w$ to be invoked, $L'.r'$ and $L'.r''$ must return the same timestamp. We now show this is impossible.

We already know that $L.w \rightarrow L'.r''$. By construction, $L'.r''$ follows a delay of $\Delta + 6\delta$ after the termination of $L'.r'$. If $L'.r''$ is invoked no later than $s + 2\delta$, then $L'.r'$ terminates by $s - \Delta - 4\delta$. Since the earliest that $L.w$ is invoked is $t - 4\delta$, we have $L'.r' \rightarrow L.w$. We get that $L.w$ is a *write*

that occurs completely between $L'.r'$ and $L'.r''$, and so they must return different timestamps.

We are left with the possibility that $L'.r''$ is invoked after $s + 2\delta$. Because $L'.w$ precedes $rn.r$ in $\pi_{rn.r}$, the latest that $L'.r''$ may be invoked is $s+5\delta$. Hence, $L'.r'$ terminates by $s-\delta$. We now get that $rn.w$ is a *write* that occurs completely between $L'.r'$ and $L'.r''$, and so they return different timestamps.

Hence, $L.r'$ and $L'.r''$ must see different values, in contradiction to the assumption that $L'.w$ is invoked after $L'.r''$. Hence, $rn.r$ returns the same value as $rn.w$, and the renewal succeeds.
We proved the following

Theorem 4 ($\diamond$ND Renewal Correctness). The $\diamond$ND renewal implementation satisfies Properties 1 and 2.

8. LEADER ELECTION

In this section we show the lease based implementation of the Boolean failure detector oracle, denoted $\mathcal{L}$, that is required by the Consensus algorithms of Refs. 14, 19. $\mathcal{L}$ is defined as follows: Let $\mathcal{L}_i$ denote the local instance of $\mathcal{L}$ at a process p_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 3 (Unique Leader). There exists a correct process p_i such that every invocation of $\mathcal{L}_i$.isLeader() returns $true$, and for each process $p_j \neq p_i$, every invocation of $\mathcal{L}_j$.isLeader() returns $false$.

The lease based implementation of $\mathcal{L}$ appears in Fig. 7. A complete Consensus algorithm based on $\mathcal{L}$ appears in Ref. 14. Here, we include it in Section 9 for completeness.

The following theorem establishes the correctness of the leader oracle implementation in the $\diamond$ND model.

Theorem 5. The pseudocode in Figure 7 eventually satisfies Property 3 in the $\diamond$ND model.

Proof. Let $T \geqslant GST$ be the time such that all the leases acquired before GST have expired and all the faulty processes have crashed by T. Let $Leaders_T$ be the set of processes that are still leaders after T. If $Leaders_T \neq \emptyset$, then all the processes in $Leaders_T$ must be executing

```
Shared Δ-Lease object L;
Local Boolean leader;
(1)  forever do
(2)       leader ← false;
(3)       L.CONTEND();
(4)       leader ← true;
(5)       delay(Δ);
(6)       while(L.RENEW()) do
(7)            delay(Δ);
(8)  od;

isLeader:
     return leader;
```

Fig. 7. The Lease-based Leader Oracle implementation.

lines 6–7 of the code in Fig. 7. By the renewal liveness, some of the processes renewing its lease at line 6 at the time $t \geqslant T$ will succeed to renew its lease at each renewal attempted after t. By the renewal safety, starting from time t on, this process will remain the exclusive lease holder.

If $Leaders_T = \emptyset$, then by the lease liveness, for some process p invoking L.CONTEND() after GST, L.CONTEND() will return at time $t \geqslant T$. By the renewal liveness, p will succeed to renew its lease at each renewal attempted after t. By the renewal safety, starting from time t on, p will remain the exclusive lease holder.

9. UNIFORM CONSENSUS BASED ON $\mathcal{L}$

Our Consensus implementation utilizes the *ranked register* primitive of Ref. 14 defined as follows: 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 $abort_i$.

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

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

Property 4 (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 5 (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 6 (Liveness) If an operation (rr-*read* or rr-*write*) is invoked by a non-faulty process, then it eventually returns.

The pseudocode of the Consensus implementation is shown in Fig. 8. Please refer to Ref. 14 for the correctness proof.

10. PRACTICAL CONSIDERATIONS

There are a number of considerations worthy of noting in the context of practical distributed storage systems. First, a standard concurrency policy is to allow either multiple simultaneous readers, or one exclusive writer. Our leases easily support this paradigm. More specifically, in our scheme, access is granted to contending processes by writing their names onto a shared read/write register. Therefore, multiple-readers can be supported simply by having readers use a common name (e.g., "reader"), and writers use their own identity.

Another important concern is caching. In a scalable system, a client obtaining a lease on a file may hold the file for some period of time, and work on a local cached copy of the file. However, the lease for the file has to be renewed periodically, which in our approach, implies writing to disk. The obvious concern is that lease-renewal could subvert the benefits of caching.

We expect this **not** to be the case for several reasons. First, comparing our storage-centric lock-renewal with the standard lease-manager approach, it is disputable that writing to a disk over a modern SAN is less efficient than sending a message to the lease manager. First, an advanced storage controller (like IBM's Shark or Total Storage Volume Controller[21]) provides a sophisticated caching which is also fault-tolerant. So writing to a disk can be as fast as writing to a process. Moreover, measurements performed in Ref. 6 indicate that in scalable settings, the costs

```
Shared: Ranked registers rr, initialized by rr-write(⟨r_0, ⊥⟩)
which commits;
Regular register decision, with values in RVals,
initialized by write(⟨r_0, ⊥⟩)
Local: V ∈ RVals ∪ {abort},
r ∈ Ranks;

Process i:

propose(v), Vals → Vals
    r ← r_0;
    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;
```

Fig. 8. Consensus using a ranked register and $\mathcal{L}$.

of accessing a remote disk are significantly outweighed by the overhead of going through a bottleneck lease manager. Further assessing the cost tradeoffs of our approach under different conditions is a topic of further study.

Additionally, the performance gain of caching should be always weighed against the end-user guarantees. Suppose that a client holding a cached data is falsely suspected, and the lease is granted to another client. Then, when the original client eventually attempts to write the cached data

back to disk, its write would be aborted to prevent inconsistency. Subsequently, all the modifications issued by the end-user will be lost. In order to provide a reasonable level of end-user semantics, the cached copy must be synchronized with the disk copy frequently enough. Thus, the lease renewal can be piggybacked on these synchronization messages.

ACKNOWLEDGMENTS

We are thankful to Ohad Rodeh for introducing us to the subject of the locking support in SAN and many fruitful discussions. This research was supported by NSF grant #CCR-0121277, NSF-Texas Engineering Experiment Station grant #64961-CS, Air Force Aerospace Research-OSR contract #F49620-00-1-0097, and MURI AFOSR contract #SA2796PO 1-0000243658.

REFERENCES

1. M. Abadi and L. Lamport, An Old-Fashioned Recipe for Real Time, *ACM Transactions on Programming Languages and Systems* **16**(5):1543–1571 (September 1994).
2. R. Alur, H. Attiya, and G. Taubenfeld, Time-Adaptive Algorithms for Synchronization. *SIAM Journal on Computing* **26**(2):539–556 (1997).
3. R. Alur and G. Taubenfeld, How to share a data structure: A fast timing-based solution, in *Proceedings of the 5th IEEE Symposium on Parallel and Distributed Processing*, pp. 470–477 (1993).
4. R. Alur and G. Taubenfeld, Fast Timing-based Algorithms, *Distributed Computing*, **10**(1):1–10 (1996).
5. R. Alur and G. Taubenfeld, Contention-free Complexity of Shared Memory Algorithms, *Information and Computation*, **126**(1):62–73 (1996).
6. K. Amiri, G. A. Gibson, and R. Golding, Highly Concurrent Shared Storage, in *Proceedings of the International Conference on Distributed Computing Systems* (*ICDCS2000*), (April 2000).
7. H. Attiya, A. Bar-Noy, and D. Dolev, Sharing Memory Robustly in Message-Passing Systems. *Journal of the ACM* **42**(1):124–142 (1995).
8. H. Attiya and A. Bar-Or, Sharing Memory with Semi-Byzantine Clients and Faulty Storage Servers. *The 22nd Symposium on Reliable Distributed Systems* (SRDS), (October, 2003).
9. A. Barry et al. An Overview of Version 0.9.5 Proposed SCSI Device Locks, in *Proceedings of the 17th IEEE Symposium on Mass Storage Systems*, pp. 243–252, College Park, Maryland, March 27–30, IEEE Computer Society, (2000).
10. R. Boichat, P. Dutta, and R. Guerraoui, Asynchronous Leasing. *Invited Paper at the 7th IEEE International Workshop on Object-oriented Real-time Dependable Systems* (*WORDS 2002*), San Diego, California (January 2002).
11. 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).
12. J. Burns and N. Lynch, Bounds on Shared Memory for Mutual Exclusion, *Information and Computation* **107**(2):171–184 (December 1993).

13. Cheng Shao, E. Pierce, J. Welch, Multi-Writer Consistency Conditions for Shared Memory Objects. in *Proceedings of the 17th International Symposium on Distributed Computing* (*DISC'2003*), (to appear).
14. G. Chockler and D. Malkhi, Active Disk Paxos with Infinitely Many Processes. *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing* (*PODC*), (August 2002).
15. G. Chockler, D. Malkhi, and M. K. Reiter, Backoff Protocols for Distributed Mutual Exclusion and Ordering. *Proceedings of the 21st International Conference on Distributed Computing Systems*, pp. 11–20, (April 2001).
16. T. D. Chandra and S. Toueg, Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM* **43**(2):225–267, (March 1996).
17. F. Cristian and C. Fetzer, The Timed Asynchronous Distributed System Model, *IEEE Transactions on Parallel and Distributed Systems* **10**(6):642–657, (1999).
18. C. Dwork, N. Lynch, and L. Stockmeyer, Consensus in the Presence of Partial Synchrony. *Journal of the ACM* **35**(2):288–323, (1988).
19. E. Gafni and L. Lamport, Disk Paxos, *Distributed Computing* **16**(1):1–20, (2003).
20. E. Gafni and M. Mitzenmacher, Analysis of Timing-Based Mutual Exclusion with Random Times, in *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing* (*PODC'99*), pp. 13–21, May 3–6, Atlanta, Georgia, USA (1999).
21. J. S. Glider, C. F. Fuente, and W. J. Scales, Software Architecture of a SAN Storage Control System. *IBM Systems Journal*, **2**(42) (2003).
22. R. Golding and O. Rodeh, Group Communication – Still Complex after All These Years, in *International Workshop on Large-Scale Group Communication* (*in conjunction with SRDS'2003*), October 5, Florence, Italy (2003).
23. C. Gray and D. Cheriton, Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency, in *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pp. 202–210 (1989).
24. P. Jayanti, T. Chandra, and S. Toueg, Fault-Tolerant Wait-free Shared Objects. *Journal of the ACM* **45**(3):451–500 (May 1998).
25. D. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager, Timed I/O Automata. Manuscript in progress (2003).
26. L. Lamport, A Fast Mutual Exclusion Algorithm. *ACM Transactions on Computer Systems*, **5**(1):(February 1987) 1–11. Also appeared as SRC Research Report 7.
27. L. Lamport, Paxos Made Simple. *Distributed Computing Column of SIGACT News* **32**(4):34–58 December (2001).
28. B. W. Lampson, How to Build a Highly Available System using Consensus, in *Proceedings of the 10th International Workshop on Distributed Algorithms* (*WDAG*), Vol. 1151: pp. 1–17, Springer-Verlag LNCS Berlin (1996).
29. B. W. Lampson, The ABCD's of Paxos. Lamport Celebration Lecture 2, *Presented on the 20th Annual ACM Symposium on Principles of Distributed Computing* (*PODC'01*), August 26–29, Newport, Rhode Island, USA, (2001).
30. 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*), pp. 280–295, The Netherlands (1994).
31. N. Lynch, *Distributed Algorithms*. Morgan Kaufman Publishers, San Mateo, CA (1996).
32. N. Lynch and N. Shavit, Timing-based Mutual Exclusion, in *Proceedings of the 13rd Real-Time Systems Symposium*, pp. 2–11, Phoenix, Arizona, IEEE Computer Society, (December 1992).

33. J. Menon, D. Pease, R. Rees, L. Duyanovich, and B. Hillsberg. StorageTank, a Heterogeneous Scalable SAN File System. *IBM Systems Journal* **2**(42) (2003).
34. The Object-Based Storage Devices Technical Work Group. `www.snia.org/tech_activities/workgroups/osd`.
35. K. Preslan, et al. A 64-bit, Shared Disk File System for Linux, in *Proceedings of the 16th IEEE Symposium on Mass Storage Systems*, pp. 22–41, San Diego, California, March 15–18, IEEE Computer Society, (1999).
36. K. Preslan, S. Soltis, C. Sabol, and M. O'Keefe, Device Locks: Mutual Exclusion for Storage Area Networks, in *Proceedings of the 16th IEEE Symposium on Mass Storage Systems*, pp. 262–274, San Diego, California, March 15–18, IEEE Computer Society, (1999).
37. O. Rodeh and A. Teperman. zFS – a scalable distributed file system using object disks, in *Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 207–218, San Diego, California, April 7–10, IEEE Computer Society, (2003).
38. F. Schmuck and R. Haskin, GPFS: A Shared-Disk File System for Large Computing Clusters. in *Proceedings of the First Conference on File and Storage Technologies* (*FAST*) (January 2002).
39. S. Soltis, T. Ruwart, and M. O'Keefe, The Global File System, in *Proceedings of the 5th NASA Goddard Conference on Mass Storage Systems and Technologies*, College Park, Maryland (September, 1996).
40. S. Soltis, G. Erickson, K. Preslan, M. O'Keefe, and T. Ruwart, The Design and Performance of a Shared File System for IRIX, in *Proceedings of the 6th NASA Goddard Conference on Mass Storage Systems and Technologies*, College Park, Maryland, March 23–26 (1998).