Light-Weight Leases for Storage-Centric Coordination

Gregory Chockler*, Dahlia Malkhi†

April 22, 2004

Abstract

We propose light-weight *lease* primitives to leverage fault-tolerant coordination among clients accessing a shared storage infrastructure (such as network attached disks or storage servers). In our approach, leases are implemented from the very shared data that they protect. That is, there is no global lease manager, there is a lease per data item (e.g., a file, a directory, a disk partition, etc.) or a collection thereof. Our lease primitives are useful for facilitating exclusive access to data in systems satisfying certain timeliness constraints. In addition, they can be utilized as a building block for implementing dependable services resilient to timing failures. In particular, we show a simple lease based solution for fault-tolerant Consensus which is a benchmark distributed coordination problem.

Keywords: leases, file systems, mutual exclusion, consensus

[1]CS and AI Laboratory, MIT, Email: `grishac@csail.mit.edu`. Research supported by NSF grant #CCR-0121277, NSF-Texas Engineering Experiment Station grant #64961-CS, Air Force Aerospace Research-OSR contract #F49620-00-1-0097, and MURI AFOSR contract #SA2796PO 1-0000243658.

[2]School of Computer Science and Engineering, The Hebrew University of Jerusalem. Email: `dalia@cs.huji.ac.il`.

1 Introduction

Motivation. In recent years, advances in hardware technology have made possible a new approach for storage sharing, in which clients access disks directly over a *storage area network* (SAN). 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.

The traditional approach to implementing locks in SAN-based file systems designates a lock manager to administer shared access [11, 32], thus creating a performance and an availability bottleneck. 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.

Fault tolerance. 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 [35, 9]), or object store controllers (see [36, 22]). 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 [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., [7, 8, 13]). In the remainder of this paper, we largely ignore replication and 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.

Uniform solutions. 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 [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 per data item 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. There still remain tasks that are best handled by a coordinated group of SAN managers. For example, SAN managers need to administer volume assignments and configuration information. 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 [19, 14]. 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 [28].

Contribution. In the remainder of this exposition, we provide a formal treatment of the problem at hand, in which disks are simply considered to be persistent shared memory containers accessed by multiple fail-prone clients. 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

In this paper we apply the real-time mutual exclusion theory to support locking in practical SAN-based systems. In the following, we survey the current state-of-the-art in these two areas.

2.1 Locking Support in SAN-based file systems

Traditionally, SAN-based file systems rely on separate servers to maintain their meta-data and coordinate access to the user data on storage devices. 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) [37] and IBM StorageTank [32]. More examples can be found in [22].

The vision of a storage-centric locking was first realized in the Global File System (GFS) project [34, 38, 39] 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 [35, 9] 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 [36, 22] is a research file system implemented over object store devices [33] 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.

2.2 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 [3, 4, 5] and the *unknown delay model* of [2].

The known delay model was first formally defined in [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 [26]. In [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 [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 [31] 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 [31] 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 [31, 2]. The fault-tolerant (wait-free) timing based algorithms of [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 [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 [31]. This results in a model analogous to the partial synchrony model of [18]. However, as is, the partial synchrony model of [31] 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 [31] to allow non-atomic memory access as well as evaluating applicability of other timed models (e.g., [1], or the timed I/O automata model of [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 [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 [31] 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 [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.

2.3 Other work on locks and leases

Gray and Cheriton were the first to employ leases in [23] for constructing fault-tolerant distributed systems. Lampson advocates in [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 [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.

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 [13]. Then, in Section 3.3, we augment the basic model with necessary timeliness assumptions by adapting the timed asynchronous model of [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 [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., [13, 8]). 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)* [30] 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

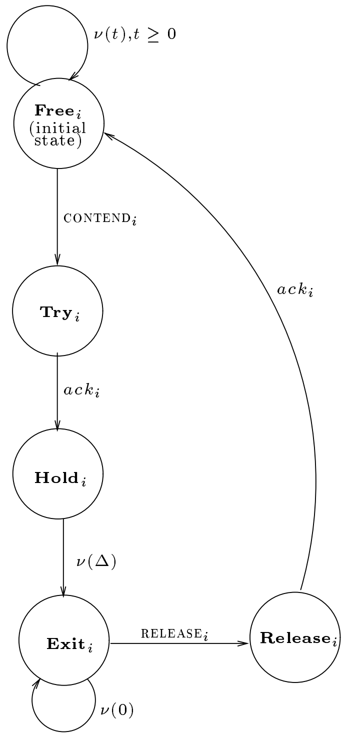

Figure 1: Well-formed interaction of process i and the Δ-Lease object

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 the start 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 consists of the following two operations for each process i: CONTEND$_i$ and RELEASE$_i$. The responses to these operations are 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 Figure 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 to the *Exit* state. At this state, the application invokes RELEASE and returns to the Free state upon the ack response.

In the states Free, Hold and Exit, 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 $t \geq 0$ time passage). However, we assume that the transition from the state Exit to the Release state is instantaneous (indicated by a 0 time passage).

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

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.*
3. *Release Progress: At any point in an execution, if a correct process i is in the Release state, then at some later point process i enters the Free state.*

5 The Lease Implementation

The Δ-*Lease* object implementation appears in Figure 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 4–8, where it waits until the current lease holder either relinquishes the lease, or the lease period Δ expires without RELEASE being called. The latter could happen if the current lease holder crashes before calling RELEASE. 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.

Upon RELEASE, a special $\perp$ value is written to x to indicate the fact that no process is currently holding the lease. This way a newly contending process could avoid the delay statement in line 2.6 and proceed directly to 2.9.

We now prove that the implementation in Figure 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:

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

That is, denote by $L.w$ the last *write* operation invoked during L (i.e., the last time line 10 in Figure 2 is activated). Denote by $L.r$ the *read* operation that follows $L.w$ (on line 12). If there exists a *read* operation invoked from line 7, denote by $L.r''$ 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 12 followed by a delay of $(\Delta + 5\delta)$. Otherwise, let $L.r'$ be the last *read* operation during L from line 1 or line 12 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.

```
Shared:
  x ∈ TS⊥;
Local:
  x1, x2 ∈ TS⊥.

CONTEND:
  x2 ← read(x);
  do
    if (x2 ≠ ⊥) then {
      do
        x1 ← x2;
        delay(Δ + 5δ);
        /* Δ + 6δ for the ◇ND renewals */
        x2 ← read(x);
      until x1 = x2 ∨ x2 = ⊥;
    }
    Generate a unique timestamp ts;
    write(x, ts);
    delay(2δ);
    x2 ← read(x);
  until x2 = ts;
  return ack;

RELEASE:
  write(x, ⊥);
  return ack;
```

Figure 2: The Δ-Lease Implementation.

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 $L_0.w$, if $L.r''$ is invoked, then it is invoked after $s_0 + \delta$.*

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 Figure 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 Figure 3(b). Since $L'.w$ is invoked after

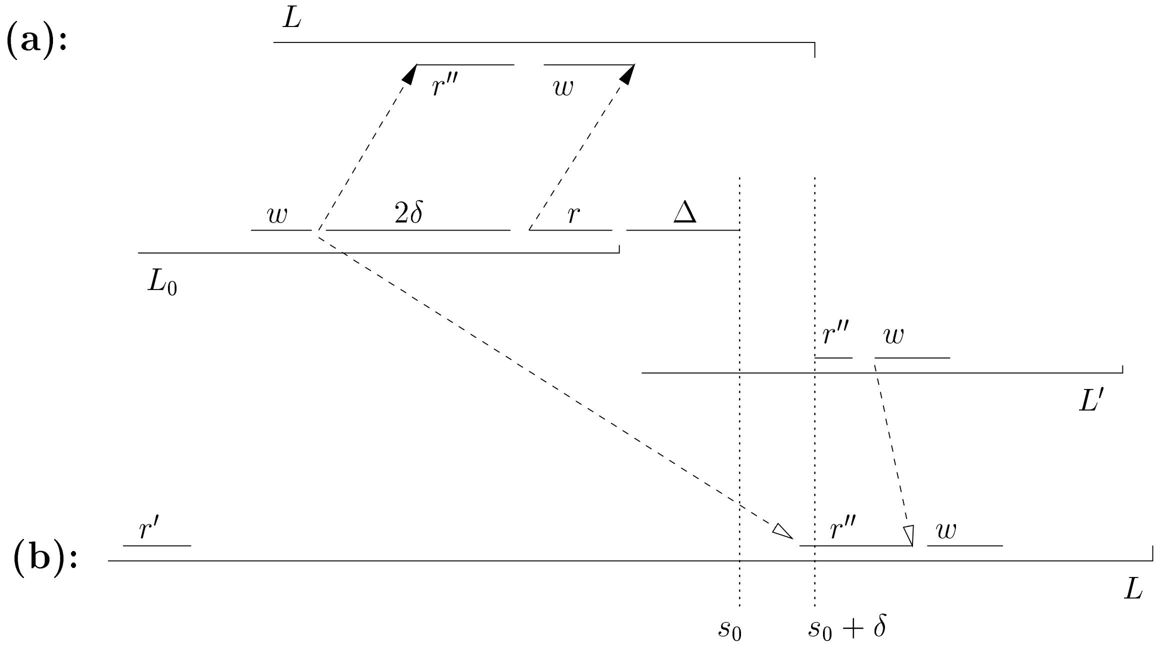

Figure 3: Possible placements of overlapping CONTEND operations L_0 and L.

$s_0 + \delta$, and since by assumption, $L.r'$ finishes before $s_0 + \delta$, we get that $L.r' \to L'.w$. 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 Figure 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. $\square$

We are now ready to prove Safety.

Lemma 2 (Safety). *The implementation in Figure 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, suppose that $L'.r''$ has never been invoked. Then, $L.r'$ must have returned $\perp$. Therefore, $L.r'$ must have been invoked before $L.w$ returns. Therefore, $L'.w$ returns before $L.delay(2\delta)$ terminates. Hence, $L'.w \to L.r$, and by regularity of x, both $L.w$ and $L.w'$ must appear in both $\pi_{L.r}$ and $\pi_{L.r'}$. Since $L.r$ returns the value written by $L.w$, $L.w'$ precedes $L.w$

in π. However, by assumption, $L.r'$ must return the value written by $L.w'$. Therefore, $L.w$ precedes $L.w'$ in π. A contradiction.

Next, suppose that $L'.r''$ was invoked. Then, it must have been 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 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 $s+4\delta$, $w \not\rightarrow w'$. 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) $W \rightarrow r$, and (ii) there does not exist any *write* event ω in π_r that follows W in π such that $W \rightarrow \omega$ and ω is invoked before r returns.

To see that (i) holds, let $w' \in W$. Since $w \not\rightarrow w'$, we have that w' terminates at most δ after w; since r starts 2δ after w's termination, $w' \rightarrow r$. To see (ii), first note that if $W \rightarrow \omega$, 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 Figure 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 2.4–8) 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 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. ☐

Finally, since the RELEASE code is trivially live, we proved the following

Theorem 1. *The implementation in Figure 2 satisfies Property 1.*

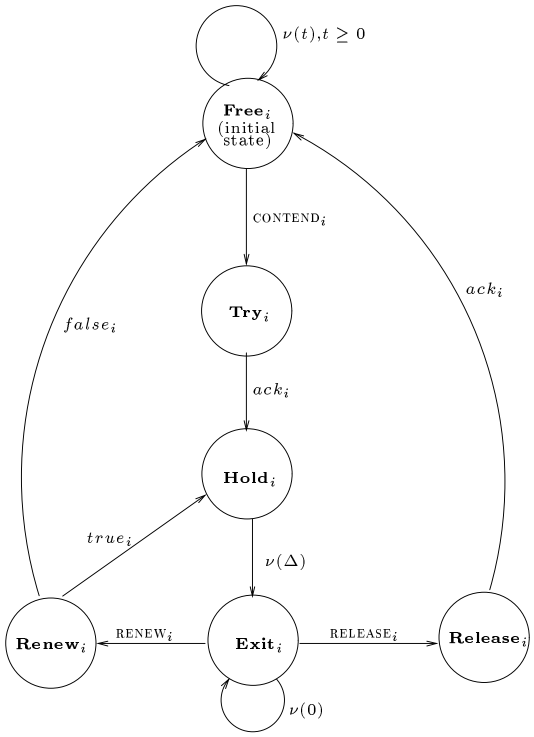

Figure 4: Well-formed interaction of process i and the Δ-Lease object with renewals

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 Figure 4. It allows an application in the Exit state to 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 a process is allowed to renew its lease for several times in a row before relinquishing the lease with the RELEASE operation.

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

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.*

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 Figure 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 $t_k - \delta$, $L.r' \to L.w$. 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. $\square$

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 $t' \leq t + \Delta$, we get that the lease period of L' overlaps with $[t_0, s_k]$. A contradiction. $\square$

The following lemma follows immediately from Lemma 5 and Lemma 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 $\Diamond$ND renewal

The RENEW operation implementation for the $\Diamond$ND model is shown in Figure 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 $\Diamond$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), $L.r$.

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 7 of Figure 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 12 of Figure 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 12 of of Figure 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.

RENEW:

$x_1 \leftarrow read(x)$;
if $(x_1.id \neq ts.id)$ then
    return $false$;
$ts.counter \leftarrow ts.counter + 1$;
$write(x, ts)$;
$delay(2\delta)$;
$x_1 \leftarrow read(x)$;
if $(x_1 = ts)$ then
    return $true$;
else
    return $false$;

Figure 5: $\Diamond$ND Renew Implementation.

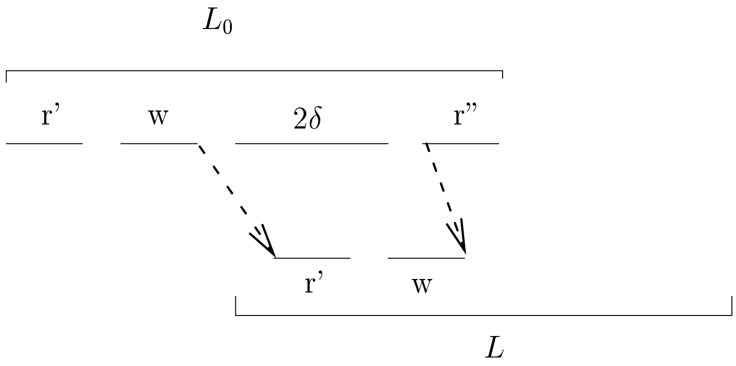

Figure 6: Overlapping renewals.

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 Figure 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 $s_0 + \delta$ 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 $L_0.w \rightarrow L.r''$. We now show that $L.r' \rightarrow L_0.w$. 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, $L.r' \rightarrow L_0.w$. 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, $L.w \to L'.r''$. Since $L.'w$ is the first *write* to follow $L.w$, and since $L'.r'' \to L'.w$, 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 $L.w \to L'.r''$. 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 $L'.r' \to L.w$. 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 (◇ND Renewal Correctness). *The ◇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 [19, 14]. $\mathcal{L}$ is defined as follows: Let $\mathcal{L}_i$ denote the local instance of $\mathcal{L}$ at a process p_i, with a boolean $\mathsf{isLeader}()$ 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 $\mathcal{L}_i.\mathsf{isLeader}()$ returns true, and for each process $p_j \neq p_i$, every invocation of $\mathcal{L}_j.\mathsf{isLeader}()$ returns $false$.*

The lease based implementation of $\mathcal{L}$ appears in Figure 7. A complete Consensus algorithm based on $\mathcal{L}$ appears in [14]. Here, we include it in Appendix A for completeness.

```
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;
```

Figure 7: The Lease-based Leader Oracle implementation

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 $T \geq GST$ 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 lines 6–7 of the code in Figure 7. By the renewal liveness, some of the processes renewing its lease at line 6 at the time $t \geq T$ 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 $L.\text{CONTEND}()$ after GST, $L.\text{CONTEND}()$ will return at time $t \geq T$. 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 Preliminary Performance Assessment

To assess the scalability of the lease implementation, we carried out preliminary simulation studies. The simulation results appear in Figures 8 and 9.

In our experiments, we assumed that read and write operations take times exponentially distributed with mean 1. Subsequently, the lease delays were measured in the units of the mean read/write delay. In all the experiments, δ was set to 2, and Δ was set to 1. The choice

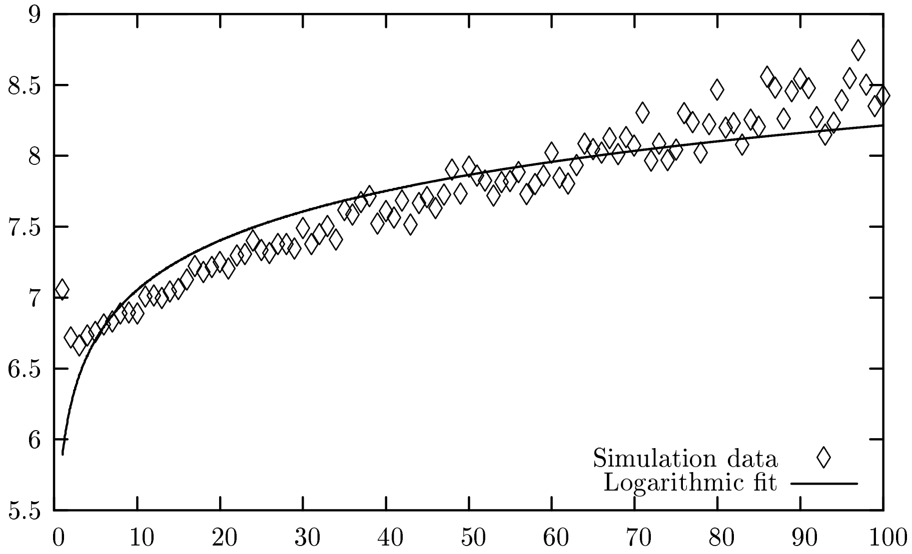

Figure 8: Delay until the first client gets the lease

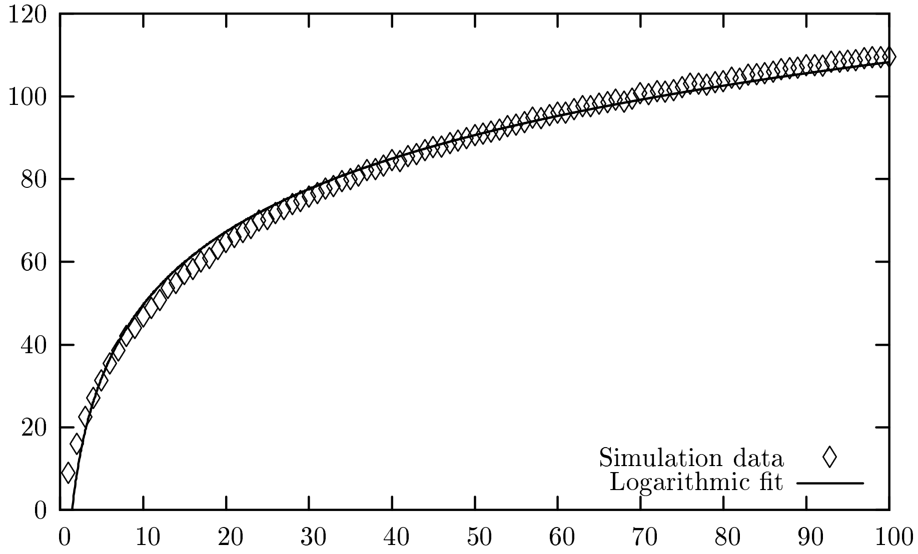

Figure 9: Delay until all the clients get the lease

of $\delta = 2$ is justified by both the exponential distribution properties, and the simulation studies. The experiments vary the number n of contending processes. All contending processes start simultaneously, and contend for the lease once until they obtain it. Subsequently, they release it after $\Delta = 1$ delay. The graph in Figure 8 shows the average delay until the first process obtains the lease as a function of the number of simultaneously contending processes; and the graph in Figure 9 shows the average delay until *all* the contending processes succeed to obtain their leases. The first graph fits into a $O(\ln(n))$ curve and the second one fits into a $O(n + \ln(n))$ curve. These results suggest good scalability features for the real implementation and are also consistent with the exponential distribution analysis of [20].

Both analytical and empirical performance evaluation of the lease algorithms as well as their implementation in the real storage system is the subject of the ongoing work.

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 [6] indicate that in scalable settings, the costs 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.

References

[1] M. Abadi and L. Lamport. An Old-Fashioned Recipe for Real Time. *ACM Transactions on Programming Languages and Systems* 16, 5 (September 1994) 1543-1571.

[2] R. Alur, H. Attiya, and G. Taubenfeld. Time-adaptive algorithms for synchronization. *SIAM J. 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, 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*, pages 243-252, College Park, Maryland, March 27-30, 2000. IEEE Computer Society.

[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*, pages 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. In *Distributed Computing* 16(1):1–20, 2003.

[20] E. Gafni and M. Mitzenmacher. Analysis of Timing-Based Mutual Exclusion with Random Times. Proceedings of the *18th Annual ACM Symposium on Principles of Distributed Computing (PODC'99)*, pages 13–21, May 3-6, 1999, Atlanta, Georgia, USA.

[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, 2003, Florence, Italy.

[23] C. Gray and D. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. Proceedings of the *12th ACM Symposium on Operating Systems Principles*, pages 202–210, 1989.

[24] P. Jayanti, T. Chandra, and S. Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM* 45(3), pages 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)*, Springer-Verlag LNCS 1151:1-17, 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, 2001, Newport, Rhode Island, USA.

[30] N. Lynch. Distributed Algorithms. Morgan Kaufman Publishers, San Mateo, CA, 1996.

[31] N. Lynch and N. Shavit. Timing-based mutual exclusion. In Proceedings of the *13rd Real-Time Systems Symposium*, pages 2–11, Phoenix, Arizona, December 1992. IEEE Computer Society.

[32] J. Menon, D. Pease, R. Rees, L. Duyanovich, and B. Hillsberg. StorageTank, a Heterogeneous Scalable SAN File System. *IBM Systems Journal*, 2(42), 2003.

[33] The Object-Based Storage Devices Technical Work Group. `www.snia.org/tech_activities/workgroups/osd`.

[34] K. Preslan, et al. A 64-bit, Shared Disk File System for Linux. In Proceedings of the *16th IEEE Symposium on Mass Storage Systems*, pages 22-41, San Diego, California, March 15-18, 1999. IEEE Computer Society.

[35] 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*, pages 262-274, San Diego, California, March 15-18, 1999. IEEE Computer Society.

[36] 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, 2003. IEEE Computer Society.

[37] 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.

[38] 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.

[39] 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.

A Uniform Consensus based on $\mathcal{L}$

Our Consensus implementation utilizes the *ranked register* primitive of [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: $\mathsf{rr}\text{-}read(r)_i$ by process i, $r \in Ranks$, whose corresponding response is $value(V)_i$, where $V \in RVals$. And $\mathsf{rr}\text{-}write(V)_i$ by process i, $V \in RVals$, whose reply is either $commit_i$ or $abort_i$.

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

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

Property 4 (Safety). *Every* rr*-read operation returns a value and rank that was written in some* rr*-write invocation. Additionally, let* $W = \mathsf{rr}\text{-}write(\langle r_1, v\rangle)_i$ *be a* rr*-write operation that commits, and let* $R = \mathsf{rr}\text{-}read(r_2)_j$*, such that* $r_2 > r_1$*. Then* R *sees* W*.*

Property 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 Figure 10. Please refer to [14] for the correctness proof.

```
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;
```

Figure 10: Consensus using a ranked register and $\mathcal{L}$