Flease - Lease Coordination without a Lock Server

Björn Kolbeck, Mikael Högqvist, Jan Stender
Zuse Institute Berlin
Berlin, Germany
{kolbeck,stender,hoegqvist}@zib.de

Felix Hupfeld
Google Switzerland GmbH
Zurich, Switzerland
hupfeld@google.com

***Abstract*—Large-scale distributed systems often require scalable and fault-tolerant mechanisms to coordinate exclusive access to shared resources such as files, replicas or the primary role. The best known algorithms to implement distributed mutual exclusion with leases, such as Multipaxos, are complex, difficult to implement, and rely on stable storage to persist lease information.**

In this paper we present FLEASE, an algorithm for fault-tolerant lease coordination in distributed systems that is simpler than Multipaxos and does not rely on stable storage. The evaluation shows that FLEASE can be used to implement scalable, decentralized lease coordination that outperforms a central lock service implementation by an order of magnitude.

***Keywords*-leases; distributed mutual exclusion; lock service; Paxos;**

I. INTRODUCTION

The problem of coordinating exclusive access to a shared resource crops up in a wide range of distributed applications, including: SAN-based cluster file systems, which need to ensure exclusive access to disk blocks [1], [2]; replicated file systems, where file updates must be serialized [3], [4], [5], [6]; and primary/backup schemes [7], [8] that ensure only one server has the primary role. The latter application is particularly important as it is often easier to implement a service with a primary server that replicates its state to backup servers than it is to implement a fully decentralized system where all decisions are coordinated [9], [10].

Systems for all of these applications have the same basic structure: *processes* compete for exclusive access to a set of *resources*. Once a process has gained the right to exclusive access, it holds a *lock* on the resource and is called the *owner* of the resource. The problem of guaranteeing exclusive access in such systems can be broken down into two sub-problems:

- **revocation.** If the process owning a resource crashes or is disconnected, ownership of the resource must be revoked and assigned to another process.
- **agreement.** All processes need to agree that a specific single process is the owner of a resource.

The revocation sub-problem can be solved by *leases* [11]. A lease is a token that grants access to a resource for a predefined period of time. Its timeout acts as an implicit revocation mechanism. The resource becomes available again as soon as the lease times out, regardless of whether the owner has crashed, has been disconnected or has simply ceased responding in a timely way. Due to their simplicity, leases are employed in many distributed systems, such as replicated file systems [12], [4], [13], [14], the Hadoop Map-Reduce framework, SAN-based file systems [1] and Google's BigTable [15].

Agreement, the second sub-problem, needs to be solved for leases as well: at any point in time there may exist at most one valid lease for a resource in the system. This agreement can be formulated as a distributed consensus problem, a problem for which there are known solutions, such as Multipaxos [16]. Numerous lock services in large-scale production systems, such as Google's Chubby [17] and Microsoft's Centrifuge [18], rely on Multipaxos. Typically there is a single Multipaxos instance per lease. Within each instance Multipaxos executes Paxos [19] to reach consensus on the owner and timeout of the lease. When a lease times out, a new instance is needed to agree on the next lease. Because of this multiplicity of instances Multipaxos is very complex and error-prone to implement [20]. Paxos also requires two writes to stable storage per lease, which increases response latency and limits the throughput of the system to the bandwidth of the disk.

In this paper we introduce FLEASE, a novel algorithm for lease coordination based on Paxos that is less complex than Multipaxos and does not require stable storage. FLEASE is founded on a round-based register abstraction that was derived from Paxos [21]. By using the register FLEASE also inherits the fault-tolerance of Paxos: it reaches agreement as long as a majority of processes responds and it can deal with host failures and message loss as well as reordering and delays. Like Paxos, FLEASE is suitable for use in real systems.

In contrast to Paxos, FLEASE takes advantage of lease timeouts to avoid persisting state to stable storage. Diskless operation means FLEASE can run concurrently on the same machine with I/O-intensive applications such as file or database servers. In section III, we show how this feature facilitates the design of systems that coordinate leases in a decentralized manner. The decentralized design exhibits greater scalability than a central lock service, which is often is a bottleneck [9], [22].

In the next section (Section II), we present FLEASE along with a proof of its correctness (Section II-C to II-H). In Section III we discuss the decentralized design. We then evaluate FLEASE alongside a central lock service in several experiments, which center on scalability and heavy I/O load (Section IV). Section V surveys related algorithms and lock services as well as the systems that use them.

II. THE FLEASE ALGORITHM

The main building block of FLEASE is a round-based register derived from Paxos by Boichat et al. [21]. The register has the same properties as Paxos regarding process failures and message loss but assumes a crash-stop behavior of processes as it lacks persistent storage. This abstraction of Paxos's core allows us to clearly illustrate the basic ideas of FLEASE as well as the differences between it and regular Paxos. We can also take advantage of the proven correctness of the register and of Paxos itself in order to prove the correctness of FLEASE.

A. System Model and Definitions

We assume a system model similar to the timed asynchronous model defined in [23] with a finite and fixed set of processes $\Pi = p_1, p_2..p_n$ with each process p_i making progress at its own speed. We also assume that each process has access to a local (hardware) clock c_i. These clocks increase strictly monotonically, i.e. $c_i(t) < c_i(t')$ if $t < t'$. We require that the processes maintain a loosely-synchronized time, such that there is a known upper bound ϵ on the drift between any two clocks. This implies that the difference of the time reported by two clocks $c_i(t)$ and $c_j(t)$ at the global time t is always less than or equal to ϵ. At any time t the following condition must hold: $\forall p_i, p_j \in \Pi \ (\epsilon \geq |c_i(t) - c_j(t)|)$. Our assumption of loosely-synchronized clocks is a stricter requirement than the maximum known drift of the clock rates as required by the timed asynchronous model. In order to simplify the presentation we start with an initial version of FLEASE that assumes perfectly-synchronized clocks, i.e. $\epsilon = 0$. We subsequently extend the algorithm to also allow loosely-synchronized clocks.

We assume that communication channels are unreliable in the sense that messages can be lost and delayed but are not altered or duplicated.

We do not assume that processes have access to stable storage. In the basic version of the algorithm, we ignore the problem of processes losing their state when they crash and assume a crash-stop model. For the final version of the algorithm, we extend this to a crash-recovery model, where processes can recover from a crash and re-join the system with an empty state.

A lease is defined as a tuple $\lambda = (p_i, t)$. The lease is held by process p_i and is valid as long as $c_i(t_{now}) \leq t$ with t_{now} as the current time. A lease has expired if $c_i(t_{now}) > t$. We define the maximum time span of lease validity as t_{max}. For the system to make progress, we require that $t_{max} > \epsilon$. Both t_{max} and the clock drift ϵ are system-wide constants known to all processes. In Section II-G we explain how to choose concrete values for these constants.

B. The Distributed Round-Based Register

The distributed round-based register implements a shared read-modify-write variable in a distributed system. The register arbitrates concurrent accesses, with semantics that resemble those of a microprocessor's test-and-set operation. This register is the core of the Paxos algorithm.

The register algorithm is shown in figure 1. The register has two operations: READ(k) and WRITE(k, v). k is a unique identifier (or ballot number, in Paxos terms) generated by the process initiating the operation; in Section II-H we show how to generate this identifier with a total order among processes. v is the value to be written to the register. Both operations either commit or abort. If a read commits, it returns the current value v of the register or $\perp$ if the register is empty. A detailed description of the algorithm and proofs of the following lemmas can be found in [21].

Lemma R1: Read-abort: If READ(k) aborts, then some operation READ(k') or WRITE($k', *$) was invoked with $k' \geq k$.

Lemma R2: Write-abort: If WRITE($k, *$) aborts, then some operation READ(k') or WRITE($k', *$) was invoked with $k' > k$.

Lemma R3: Read-write-commit: If READ(k) or WRITE($k, *$) commits, then no subsequent READ(k') may commit with $k' \leq k$ or WRITE($k'', *$) may commit with $k'' < k$.

Lemma R4: Read-commit: If READ(k) commits with v and $v \neq \perp$, then some operation WRITE(k', v) was invoked with $k' < k$.

Lemma R5: Write-commit: If WRITE(k, v) commits and no subsequent WRITE(k', v') is invoked with $k' \geq k$ and $v \neq v'$, then any READ(k'') that commits will do so with v if $k'' > k$.

C. The basic FLEASE *algorithm*

Similar to Paxos, processes in FLEASE can have two roles. *Proposers* actively try to acquire a lease or attempt to find out which process holds a lease. *Acceptors* are passive, receiving read and write messages of the round-based register and storing k and v according to the algorithm of the round-based register.

The basic version of the FLEASE algorithm is shown in algorithm 2. The GETLEASE procedure is executed by a proposer when it wants to acquire the lease. The proposer starts by reading the register value (line 2). If the register is either empty or the lease has timed out, the proposer will overwrite the register with its own lease (line 3 and 4). If the register contains a valid lease, it does not change the

Algorithm 1 round based register for process p_i, from [21]

```
read_i ← 0
write_i ← 0
v_i ← ⊥

procedure READ(k)
    send (READ,k) to all processes in Π
    wait until received (ackREAD,k,∗,∗)
            or (nackREAD,k) from ⌈(n+1)/2⌉ processes
    if received at least one (nackREAD,k) then
        return (abort,⊥)
    else
        select the [ackREAD,k,k′,v] with the highest k′
        return (commit,v)
    end if
end procedure

procedure WRITE(k,v)
    send (WRITE,k,v) to all processes in Π
    wait until received (ackWRITE,k) or (nackWRITE,k)
            from ⌈(n+1)/2⌉ processes
    if received at least one (nackWRITE,k) then
        return abort
    else
        return commit
    end if
end procedure

upon receive (READ,k) from p_j
    if write_i ≥ k or read_i ≥ k then
        send (nackREAD,k) to p_j
    else
        read_i ← k
        send (ackREAD,k,write_i,v_i) to p_j
    end if
end upon

upon receive (WRITE,k,v) from p_j
    if write_i > k or read_i > k then
        send (nackWRITE,k) to p_j
    else
        write_i ← k
        v_i ← v
        send (ackWRITE,k) to p_j
    end if
end upon
```

Algorithm 2 The basic algorithm

```
 1: procedure GETLEASE(k)
 2:     if READ(k) = (commit, λ) then
 3:         if λ =⊥ or λ.t < t_now then
 4:             λ ← (p_i, t_now + t_max)
 5:         end if

 6:         if WRITE(k,λ) = commit then
 7:             return (commit,λ)
 8:         end if
 9:     end if
10:     return (abort,⊥)
11: end procedure
```

register. Then the proposer writes the lease (either its own or the valid lease it has just read) into the register (line 6) and returns the lease as the result of GETLEASE.

The proposer must always write the lease because writes can be incomplete. An incomplete write occurs when a proposer p crashes while sending write messages to the acceptors or when a majority of the messages are lost. In these cases p will return either nothing as it has crashed or will return (abort,$\perp$) (line 10). However, when another proposer p' runs FLEASE after p has failed, the outcome of the READ operation is undefined. The next read might return the last value p tried to write or the value before the incomplete write. This depends on which acceptors respond to p'. Ergo, p' must ensure that any lease it returns has been successfully written to the register, regardless of whether the lease is owned by p' or the lease has been read from the register. To ensure this, p' must wait for the WRITE to commit before it is allowed to return the lease. This WRITE, together with Lemma R3, ensures that all proposers running after p' will return the lease p' has returned.

To illustrate what would happen without this write step, assume that p_1 writes a lease to the register but fails due to message loss. Now p_2 reads the register and reads the lease from p_1 and returns the lease for p_1. Then p_3 reads the register but gets only responses from acceptors that did not receive p_1's message. p_3 would now write its own lease to the register and the system would have two valid leases at the same time.

D. Proof

In order to prove the correctness of FLEASE we must demonstrate that the algorithm guarantees that there is at most one valid lease at any point in time. We call this the *Lease Invariant*.

Property L1 (Lease Invariant): If a process p decides $\lambda = (p, t)$ then any other process will decide λ until $t_{now} > t$. This is similar to the agreement property of consensus but allows processes to decide a different value after the lease has timed out.

Proof by contradiction: Assume two processes p_i and p_j decide two different values $\lambda = (p, t)$ and $\lambda' = (p', t')$ with $\lambda \neq \lambda'$, $t > t_{now}$ and $t' > t_{now}$, i.e. two different leases are valid at the same time. Without loss of generality, we assume that $k' > k$ and that p_i decides λ after committing GETLEASE(k). Afterwards p_j decides λ' after committing GETLEASE(k'). Following Algorithm 2, p_j must commit READ(k') before calling WRITE(k', λ'). The read-abort property of the register (lemma R1) ensures that the READ will commit because $k' > k$. Due to the write-commit property of the register (lemma R5), the READ will commit with λ as this value was previously written by p_i. Depending on the value of $\lambda.t$, process p_j will make one of two decisions:

Case 1: $\lambda.t \geq t_{now}$ (the lease λ is still valid)
According to the algorithm, p_j will WRITE(k',λ) and decide $\lambda' = \lambda$. However, this is a contradiction to the assumption that $\lambda' \neq \lambda$.

Case 2: $\lambda.t < t_{now}$ (the lease λ has expired)
In this case, p_j would WRITE(k',λ') and decide $\lambda' \neq \lambda$ but is allowed to do so as we require p_j to decide λ only until $t_{now} > \lambda.t$. This is a contradiction of the assumption that $t > t_{now}$ and $t' > t_{now}$.

E. Lease Renewals

With the basic version (algorithm 2), a lease owner will lose its lease in the period after an old lease has timed out and before a new lease has been coordinated. During this time, which takes at least two message round trips, there is no lease and consequently the resource cannot be accessed. The effect of lease renewals is illustrated in Figure 1. To avoid these interruptions, the owner of a lease should be allowed to prolong the lifetime of the lease as long as the original lease is still valid. Algorithm 3 shows an extended version of the basic FLEASE algorithm that includes lease renewals. In line 6, the proposer extends the lifetime of its lease if it is still valid (line 5).

Algorithm 3 The extended algorithm with lease renewal

```
 1: procedure GETLEASE(k)
 2:     if READ(k) = (commit, λ) then
 3:         if λ =⊥ or λ.t < t_now then
 4:             λ ← (p_i, t_now + t_max)
 5:         else if λ.p = p_i then
 6:             λ ← (p_i, t_now + t_max)
 7:         end if

 8:         if WRITE(k,λ) = commit then
 9:             return (commit,λ)
10:         end if
11:     end if
12:     return (abort,⊥)
13: end procedure
```

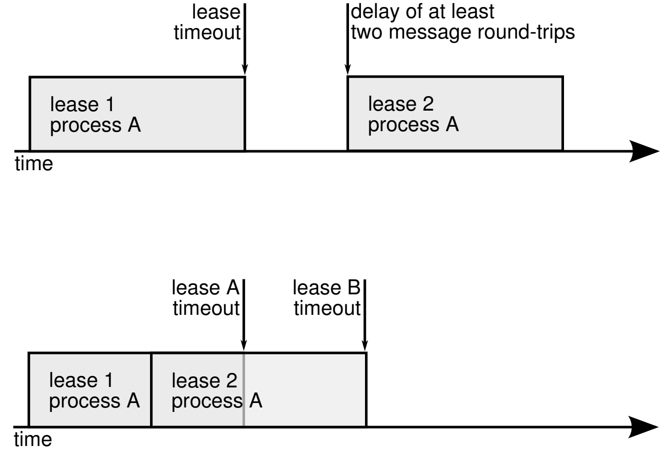

Figure 1. Lease coordination without renewal (top) and with lease renewal (bottom).

F. Proof

To allow lease renewal we must relax the lease invariant to require processes to output the same lease owner, not necessarily the same lease timeout.

Property L2 (Lease Invariant): If a process p decides $\lambda = (p_l, t)$ then any other process will decide $\lambda' = (p'_l, t')$ with $p'_l = p_l$ and $t' \geq t$ until $t_{now} > t$.

The proof by contradiction is similar to the proof for property L1. However, we assume that two processes p_i and p_j decide two different values $\lambda = (p_l, t) \neq \lambda' = (p'_l, t')$ with $t > t_{now}$ and $t' > t_{now}$ and $p_l \neq p'_l$, i.e. two different leases with different lease owners are valid at the same time. Depending on the value of $\lambda.t$, process p_j will make one of the three decisions:

Case 1: $\lambda.t \geq t_{now}$ (lease λ is still valid)

Case 1a: $\lambda.p \neq p_j$ (p_j does not hold the lease)
Same as case 1 in proof of lemma 1.

Case 1b: $\lambda.p = p_j$ (p_j holds the lease)
According to the algorithm, p_j will WRITE(k',λ') and decide λ' with $p'_l = p_l$ and $t' > t$. However, this is a contradiction to the assumption that $p' \neq p$.

Case 2: $\lambda.t < t_{now}$ (the lease has expired)
In this case, p_j would WRITE(k',λ') and decide λ' but is allowed to do so as we require p_j to decide λ only until $t_{now} > \lambda.t$. This is a contradiction to the assumption that $t > t_{now}$ and $t' > t_{now}$.

G. Allowing Processes to Recover

The round-based register assumes a crash-stop model, as it does not use persistent storage to recover the content of the register after a crash. In order to allow the register to recover from a crash, the values for k must be stored on stable storage for each READ(k). k and v must also be stored

for WRITE(k, v). Thus, the state of the register on each node i consists of the three values $read_i, write_i$ and v_i.

Here we can exploit the fact that leases expire: in our algorithm, an empty register or a register with a lease that has expired are equal. Thus we can turn the register into a crash-recovery model for our leases. In order to do this we require a recovering process to wait until t_{max} has passed before it may rejoin the system. During this waiting period the process is not allowed to participate in lease coordination and must not send messages. By forcing processes to wait until t_{max} has passed, we can guarantee that any lease that was in the register when the process crashed has timed out.

A second problem with crash-recovery is that we have to ensure that the register will abort a READ or WRITE with k' if k' is smaller than the k used for the previous READ and/or WRITE operation (lemmas R1 and R2). However, a process that recovered from a crash has lost its complete state, which includes $read_i$ and $write_i$. To guarantee that any k' used after such a crash is larger than the maximum k seen before the crash, we again take advantage of synchronized clocks. We use the current time as the ballot number and therefore guarantee that $k' > k$ always holds. To distinguish messages sent at the same time, we use a ballot number $k = (t, id_p)$ with id_p being a unique process id. The total order on k is then defined as $k < k' \Leftrightarrow (k.t < k'.t) \vee (k.t = k'.t \wedge k.id_p < k'.id_p)$.

H. Final Algorithm with Loosely-Synchronized Clocks

An algorithm based on perfectly-synchronized clocks is of little practical value. To make FLEASE suitable for real-world use, we extend it to work with loosely-synchronized clocks. As mentioned earlier, we expect host clocks to be loosely-synchronized, i.e. the difference between any two clocks does not exceed a certain maximum.

Algorithm 4 is an extended version of the algorithm with lease renewal (algorithm 3) that also takes clock drift into account. We introduce a safety period sp between the time when the lease expires and when a new lease can be issued. During the safety period it is unknown if the current lease owner still considers the lease to be valid or expired due to clock drift. However, after ϵ time, any process can safely assume that the lease has expired on all hosts and can execute the regular algorithm 3.

The algorithm is correct with respect to the lease invariant, since the section that modifies the lease is identical to algorithm 3.

I. Crash-Recovery with Loosely-Synchronized Clocks

Since we assume that $\epsilon < t_{max}$, we can also guarantee $k' > k$ in the case of loosely-synchronized clocks. However, the process with the fastest clock (i.e. maximum $c(t)$) will always acquire the lease. To alleviate this problem, we use intervals when comparing timestamps. We use an interval length of $t_{max} - \epsilon < t_{max}$ to guarantee that this works with

Algorithm 4 The full FLEASE algorithm for loosely-synchronized clocks

```
 1: procedure GetLease(k)
 2:     if READ(k) = (commit, λ) then
 3:         if λ.t < t_now and λ.t + ε > t_now then
 4:             wait for ε
 5:             return GetLease(k')              ▷ with k' > k
 6:         end if

 7:         if λ = ⊥ or λ.t < t_now then
 8:             λ ← (p_i, t_now + t_max)
 9:         else if λ.p = p_i then
10:             λ ← (p_i, t_now + t_max)
11:         end if

12:         if WRITE(k,λ) = commit then
13:             return (commit,λ)
14:         end if
15:     end if
16:     return (abort,⊥)
17: end procedure
```

processes that recover after a crash. We define this interval as

$$I(t) = \left\lfloor \frac{t}{t_{max} - \epsilon} \right\rfloor$$

We now extend $k = (t, r, id_p)$ to include an additional message number r that is used to distinguish the messages sent by a process within the same interval. Since r must only be unique within an interval, there is no need to store this number on stable storage. With this interval, we redefine the total order $<$ on the values for k as

$$\begin{aligned} & k < k' \Leftrightarrow \\ & (I(k.t) < I(k'.t)) \\ \vee \quad & (I(k.t) = I(k'.t) \wedge k.r < k'.r) \\ \vee \quad & (I(k.t) = I(k'.t) \wedge k.r = k'.r \wedge k.id_p < k'.id_p) \end{aligned}$$

This new definition of k ensures that in most cases all processes have equal chances to get a lease. Only in a period of ϵ time before and after an interval, the process with the fastest clock will still get the lease.

J. Practical Considerations

Like any algorithm working with leases, FLEASE requires loosely-synchronized clocks and the parameters ϵ and t_{max} must be carefully chosen for the particular environments FLEASE is used in. To choose a value for ϵ, the developer needs to take into account the drift of the local real time clocks of the nodes and the imprecision of the clock synchronization algorithm used. [23] gives more details on real time clocks and the drift rates.

When choosing t_{max} we need to take the maximum message round-trip delay into account. t_{max} must be longer than twice the maximum message round-trip in the system. This ensures that a lease is still valid after FLEASE was executed on the proposer.

t_{max} should be set as high as possible to reduce the frequency of lease renewals and to give lease owners enough time to execute operations on owned resources. On the other hand, t_{max} should be as short enough to ensure that interruptions due to crashed lease owners are kept to a minimum. The right balance between these two goals depends on the application, e.g. an application such as a online store must provide short timeouts to ensure low latency in the presence of crashes. In contrast, a batch application might be able to tolerate longer timeouts, in the range of several minutes.

Like Paxos and Multipaxos, FLEASE assumes that the group of nodes participating in FLEASE executions is fixed. For applications that require this group to be mutable, FLEASE has to be complemented with a set membership service [24].

III. DECENTRALIZED LEASE COORDINATION

Current research and production systems that require exclusive access are often built around replicated central lock services that coordinate lease agreement [14], [4], [12], [6], [15]. The central lock service decides which process becomes the owner of the resource and ensures that only one process has a valid lease per resource (Figure 2a). In this setup, each process can request a lease for any resource.

Often, the assumption that all process accesses all resource is too general. In many systems, a resource is assigned to a small group of processes and only the processes in that group access the resource. For example, changes to file replicas only need to be coordinated among the servers hosting the replicas (a pattern that can be found in Google's GFS [6], CEPH [13] or WheelFS [14]). In these systems the *natural partitioning* of processes according to which resources they access is a core design feature that enables the systems to scale to a large number of resources.

Introducing a central lock service into such systems sacrifices the scalability achieved through the natural partitioning and creates a bottleneck [9], [22]: The number of resources the system can handle is limited by the throughput of the central lock service. Similarly, the availability of the entire system depends on this central component.

Decentralized lease coordination (Figure 2c) automatically takes advantage of natural partitions by relegating access coordination for a given resource to the processes that a resource is assigned to, rather than relying on a central component. The decentralized design removes the central lock service and thereby enhances scalability and availability.

However, the algorithms at the core of existing central lock services make decentralized lease coordination infeasible. Replicated lock services rely on either Multipaxos [17], [14], [4], [12] or a primary/backup replication scheme with two-phase-commit [25], both of which require two writes to stable storage per lease acquisition. The extra disk traffic can adversely affect the IOOPs and throughput of I/O-intensive applications such as distributed file systems, as we show in the next section.

Implementing such a decentralized design with FLEASE has numerous advantages over the centralized approach:

1) Scalability. The system scales with the number of resources and machines in the system as the leases are coordinated among the processes. This removes the bottleneck of the central lock service and also reduces the need for makeshift solutions like volume leases [26] or manual partitioning.
2) Availability. The availability of a resource depends only on the availability of the processes which the resource is assigned to. In contrast, with a central lock service, the overall system availability depends on the availability of the machines of the lock service. This is particularly important when resources are distributed across datacenters but the lock service is bound to a single datacenter.
3) Lower latency. By avoiding stable storage, FLEASE reduces the latency per request. In addition, the decentralized setup also removes the round-trip between clients and the central lock service.

IV. EVALUATION

In order to evaluate the claims above we conducted three experiments with our full implementation of FLEASE. In the first experiment, we compare the throughput of FLEASE and a central lock service with an increasing number of nodes and resources in the system. We also look at the maximum throughput of the central lock service. In the second experiment, we compare the throughput of both systems under heavy I/O load. In a third experiment, we investigate how FLEASE scales with increasing process group sizes.

The experiments used Apache Zookeeper 3.2.2 [25] as the central lock service and the FLEASE implementation that is part of the XtreemFS distributed file system [27]. Zookeeper is the central lock and configuration service of the Apache Hadoop project, an open source implementation of the MapReduce framework. Zookeeper's functionality is comparable to the proprietary Chubby implementation at Google [17], the main difference being that the Zookeeper service is replicated with a primary/backup scheme and uses a two-phase commit for write operations, whereas Chubby uses Multipaxos for service replication. The implementations of Zookeeper and FLEASE are comparable: both are written in Java with efficient network IO (Java NIO). In terms of communications cost and latency, both FLEASE and Zookeeper require two network round trips for a lease. However, due to its centralized nature Zookeeper also incurs the cost of

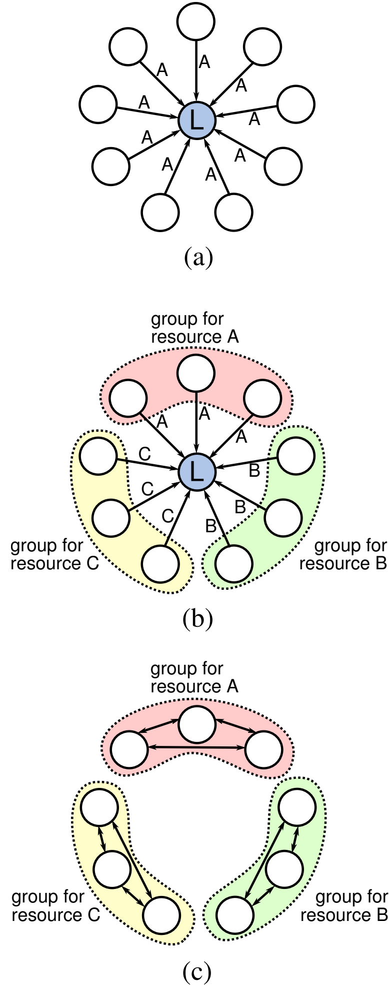

Figure 2. Lease coordination design. (a) truly centralized design, all processes compete for one resource at the central lock service. (b) independent groups with central lock service (c) independent groups with decentralized lease coordination. Circles represent processes, arrows indicate communication between processes.

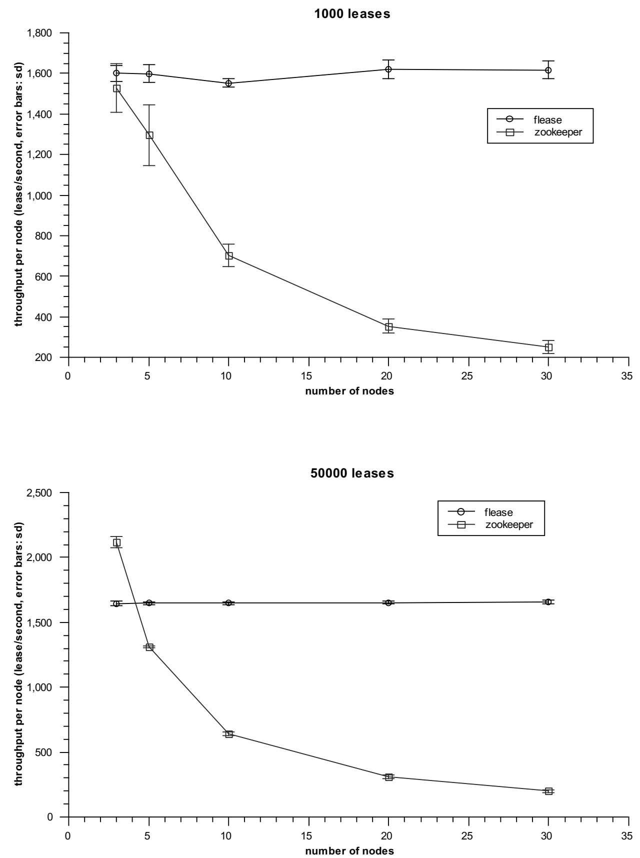

Figure 3. Throughput for batches of 1,000 (top) and 50,000 (bottom) leases with 3, 5, 10, 20 and 30 concurrent clients. Error bars show the standard deviation.

communication between service clients and servers. The latter cost is included in the measurements presented here.

We ran all experiments on a 33-machine cluster connected with regular 1GBit Ethernet. Each machine has two quad-core Opterons running at 2.3 GHz, 8GB RAM and one SATA hard drive. All machines run Linux 2.6.31 and a 64bit Java 1.6.0_18 virtual machine. In addition, we used a Sun Fire X4540 (Quad-core Opteron 2.3GHz, 32GB RAM, 48 SATA HDDs) machine to record file system traces with a FUSE module that records and sends operations to the local file system.

A. Throughput

To asses the maximum throughput of both systems, we measured service throughput in leases per second on each machine. All machines simultaneously submitted a batch of leases, and we measured the locally-observed throughput for batches of 1,000 and 50,000 leases with 3, 5, 10, 20 and 30 machines. For FLEASE we use a group size of 3 machines, i.e. each machine uses a set of two other machines for coordinating its leases. Similarly, we replicated the Zookeeper service across three servers, with the other 30 machines acting as clients requesting leases at the master server. We disabled synchronous disk writes for Zookeeper to ensure that we measured the throughput of the software, not the performance of our hardware.

Figure 3 depicts the average throughput per machine in leases per second. As expected FLEASE produces a constant throughput that is independent of the number of machines (and resources) we use. The results for three machines demonstrate that our FLEASE implementation still has potential for optimization, as it can handle fewer leases per second than the Zookeeper implementation. The throughput for Zookeeper is limited by the maximum throughput of the replicated server, which means that the throughput per machine drops as expected when adding machines.

The experiment shows that a central lock service limits the

benchmark	average	min	max
dbench 3.04, 10 clients	423	1	1576
linux kernel build	1308	1	2169

Table I
FILE OPEN RATE IN OPEN CALLS PER SECOND FOR VARIOUS BENCHMARKS.

overall scalability of the system. Zookeeper has a maximum throughput of 7,336 leases per second. In contrast, FLEASE can handle up to 51,029 leases per second (30 machines, 10,000 leases per batch).

To put these numbers into perspective, we used an approach similar to the way Schmuck et al. [1] evaluated the performance of the GPFS lock manager. We recorded traces from a dbench [28] run with 10 concurrent clients and from a Linux kernel build on a local system. Dbench is a file system benchmark that simulates load with traces recorded from Windows workstations. Then we extracted the file open rate from the trace. The results are shown in table I. In distributed or replicated file systems where a lock per file is used, the maximum throughput of 7,336 leases would be sufficient to handle the average load of 165 dbench clients or six clients compiling the Linux kernel. In contrast, with FLEASE we can easily remove this bottleneck and build a scalable system that takes advantage of the decentralized lease coordination.

B. Throughput under I/O Load

The algorithms used in central lock services such as Zookeeper and Chubby require stable storage to operate correctly. In order to assess the impact of I/O cross-traffic on these services, we ran IOZone [29] on the same machine as the Zookeeper servers and FLEASE while the latter were attempting to coordinate leases. We ran an IOZone throughput test with one thread writing a 10 GB file in 512k chunks, with an additional delay of 7ms between operations to ensure the machine didn't get overloaded by IOZone alone. Then we executed a throughput test similar to the first experiment with three machines and batch sizes of 100 to 50,000 leases. For this experiment, we activated the forceSync option of Zookeeper to ensure that data has been safely written to disk before answering requests. By default, Zookeeper batches requests into larger blocks before syncing to disk.

The results in figure 4 clearly show that Zookeeper's performance depends on the I/O load generated by other processes on the machine. In contrast, FLEASE is only CPU bound and is not affected by concurrent I/O load. This experiment demonstrates that algorithms that require stable storage are not suitable for decentralized setups when I/O bandwidth is required for the main task of the application.

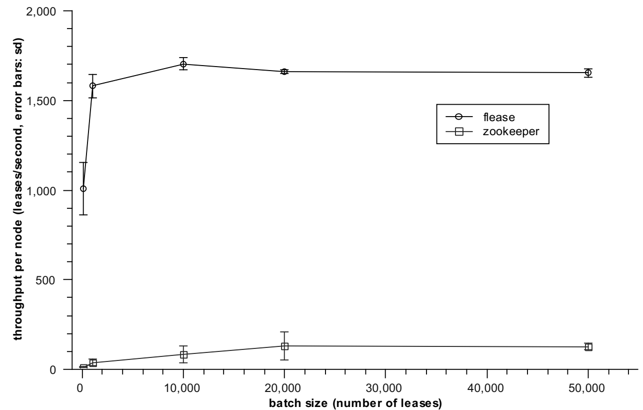

Figure 4. Throughput for batches of 100 to 50,000 leases on three nodes with concurrent IOZone throughput test. Error bars show the standard deviation.

C. Group Size

Finally, we investigated the throughput of FLEASE with varying group sizes. Since FLEASE is CPU-bound, we expect its throughput to decrease with larger groups due to an increase in the number of messages FLEASE processes must exchange. We measured the throughput for a 10,000 lease batch submitted by all 30 machines. However, this time we used groups of 3, 4, 5, 6, 7, 8, 9, 10 and 15 machines, i.e. each machine selected 2..9 or 14 other machines randomly to coordinate its leases with. Figure 5 plots the throughput per node. As expected, the throughput drops with larger group sizes due to the increasing number of messages. This experiment indicates that FLEASE in a decentralized setup is suitable for large numbers of machines as long as the groups of processes trying to access a lease are relatively small.

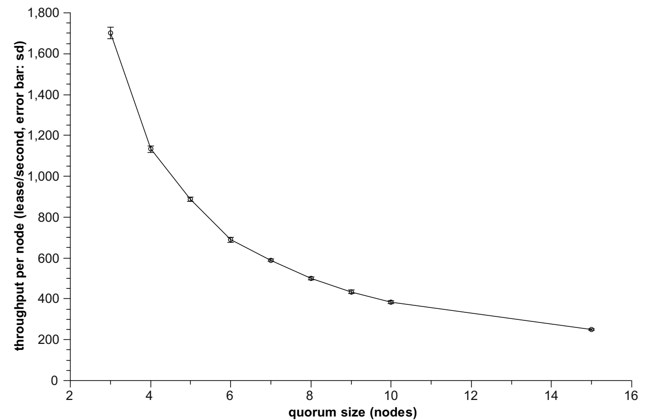

Figure 5. Throughput per node for groups of 3, 4, 5, 6, 7, 8, 9, 10 and 15 nodes with a batch size of 10,000 leases. Error bars show standard deviation.

V. Related Work

Several algorithms for distributed lease coordination have been developed and studied for various system models.

In [30] Chockler and Malkhi introduced a fault-tolerant algorithm for timed asynchronous systems with shared memory. They specifically designed their algorithm for SAN-based file systems with (hardware) shared memory. This model is not applicable to shared-nothing architectures as used in e.g. distributed file systems running on commodity hardware.

Fetzer et al. have developed a leader election algorithm for the timed asynchronous model [23]. This algorithm implements leader election with expiration time, which is essentially a lease. The algorithm is not fault-tolerant and does not consider processes that recover after a crash.

An algorithm for truly asynchronous systems was presented by Boichat et al. [31]. However, due to the lack of time in asynchronous systems, their lease approach works with logical time. This means that their variation of leases does not guarantee exclusive access at a point in time. Rather, the goal of their leases is to speed up the execution of algorithms in asynchronous systems by reducing concurrency through a coordinator role. In a more general context, Lampson [32] argued that consensus with Paxos can be made more efficient by using a single master elected with a lease.

FaTLease [33] utilizes regular consensus with Paxos to agree on a lease owner. A scheme with instances similar to Multipaxos [16] is used for continuous lease coordination. This results in a far more complex algorithm compared to FLEASE. Distinguished renew-instances in FaTLease ensure that a lease can be renewed by the owner even when other processes try to acquire the lease.

Central lock services are widely employed for lease coordination. The most prominent example is Google's Chubby [17], [20], which is implemented using Multipaxos to replicate the lease database. Other services at Google rely heavily on Chubby for e.g. master election in the Google File System (GFS) [6]. Centrifuge [18] is a decentralized configuration service with a centralized lock service that is also implemented using Paxos.

For file replication, the authors of WheelFS [14] recommend using a central lock service for master leases. The Frangipani [4] file system design, in contrast, included a Paxos-replicated configuration service that issues locks to partition masters, which is conceptually similar to Chubby. Farsite [12], a distributed peer-to-peer file system, employs leases for data and metadata access. To avoid contention on metadata entries, each field of a metadata record has its own lease [34]. In Farsite leases are coordinated by a directory group, which is implemented as a replicated service. Zookeeper [25], the lock service of the Hadoop project, uses a token-based algorithm for leader election and two-phase-commit with majorities for data replication. The token-based algorithm makes Zookeeper unsuitable for use in multi-datacenter installations.

Many systems [35], [36], [3], [37] rely on heartbeat messages between servers or between a central configuration service and the servers. When a process fails to receive heartbeat messages from another, resource-owning process the former process infers that the latter has crashed or is disconnected. If the resource owner is unreachable, another machine takes over ownership of the resource. This approach is often used in cluster environments but can lead to inconsistencies: heartbeat messages are not a reliable failure detector during network partitions, when two server processes may not be able to contact each other but can still receive requests from client machines.

Paxos [19], [38], [39] is a well-studied algorithm that implements consensus in the timed asynchronous system model. Due to its simplicity in design and direct applicability to real-world systems Paxos is widely used. The algorithm relies on a quorum approach and is thus able to tolerate the failure of a minority of processes (up to $\lceil \frac{n+1}{2} \rceil$ out of n processes). It is also able to tolerate message loss and delay. The algorithm works in two phases in which a proposer exchanges messages with all other processes in the system. During each phase, all processes have to write their state to stable storage. The requirement of persistent storage adds extra latency to the system, which can be significant. For the FLEASE algorithm we use the abstraction of a round-based register that was derived from Paxos in a modularized deconstruction by Boichat et. al [21].

VI. Conclusion

We have presented FLEASE, a novel algorithm for lease coordination in distributed systems, and have proved its correctness. The system assumptions made and failure cases considered for FLEASE make it suitable for real-world systems. An implementation of FLEASE is freely available under the BSD license as part of the XtreemFS file system.

We illustrated how FLEASE enables decentralized lease coordination that avoids the scalability bottlenecks of central lock services, and demonstrated this fact in our evaluation of FLEASE and Zookeeper. We have shown that FLEASE overcomes the problems of current algorithms used in central lock services.

Acknowledgment

We thank Minor Gordon for his valuable feedback and assistance. The research leading to these results has received funding from the European Union Seventh Framework Programme under grant agreement FP7-ICT-257438.

References

[1] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," in *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2002, p. 19.

[2] J. Menon, D. A. Pease, R. Rees, L. Duyanovich, and B. Hillsberg, "IBM storage tank – a heterogeneous scalable SAN file system," *IBM Syst. J.*, vol. 42, no. 2, pp. 250–267, 2003.

[3] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, "Grid resource management—crush: controlled, scalable, decentralized placement of replicated data," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM Press, 2006, p. 122.

[4] C. A. Thekkath, T. Mann, and E. K. Lee, "Frangipani: a scalable distributed file system," *SIGOPS Oper. Syst. Rev.*, vol. 31, no. 5, pp. 224–237, 1997.

[5] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, and L. Shrira, "Replication in the harp file system," in *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM Press, 1991, pp. 226–238.

[6] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM Press, 2003, pp. 29–43.

[7] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg, "The primary-backup approach," pp. 199–216, 1993.

[8] P. A. Alsberg and J. D. Day, "A principle for resilient sharing of distributed resources," in *ICSE '76: Proceedings of the 2nd international conference on Software engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1976, pp. 562–570.

[9] M. K. McKusick and S. Quinlan, "GFS: Evolution on fast-forward," *Queue*, vol. 7, no. 7, pp. 10–20, 2009.

[10] A. Thomson and D. J. Abadi, "The case for determinism in database systems," in *VLDB*, 2010.

[11] C. Gray and D. Cheriton, "Leases: an efficient fault-tolerant mechanism for distributed file cache consistency," in *SOSP '89: Proceedings of the twelfth ACM symposium on Operating systems principles*. New York, NY, USA: ACM Press, 1989, pp. 202–210.

[12] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, "Farsite: federated, available, and reliable storage for an incompletely trusted environment," in *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*. New York, NY, USA: ACM Press, 2002, pp. 1–14.

[13] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *OSDI*, 2006, pp. 307–320.

[14] J. Stribling, Y. Sovran, I. Zhang, X. Pretzer, J. Li, M. F. Kaashoek, and R. Morris, "Flexible, wide-area storage for distributed systems with wheelfs," in *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*. Berkeley, CA, USA: USENIX Association, 2009, pp. 43–58.

[15] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 1–26, 2008.

[16] R. D. Prisco, B. Lampson, and N. Lynch, "Revisiting the Paxos algorithm," *Theor. Comput. Sci.*, vol. 243, no. 1-2, pp. 35–91, 2000.

[17] M. Burrows, "Chubby distributed lock service," in *Proceedings of the 7th Symposium on Operating System Design and Implementation, OSDI'06*, Seattle, WA, November 2006. [Online]. Available: http://labs.google.com/papers/chubby.html

[18] A. Adya, J. Dunagan, and A. Wolman, "Centrifuge: Integrated lease management and partitioning for cloud services," in *NSDI'10: Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation*. USENIX, 2010.

[19] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998. [Online]. Available: citeseer.ist.psu.edu/lamport89parttime.html

[20] T. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: an engineering perspective," in *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. New York, NY, USA: ACM Press, 2007, pp. 398–407.

[21] R. Boichat, P. Dutta, S. Frolund, and R. Guerraoui, "Deconstructing paxos," *SIGACT News*, vol. 34, no. 1, pp. 47–67, 2003.

[22] T. Chandra, "Sibyl: a system for large scale machine learning," http://ladisworkshop.org/sites/default/files/LADIS 2010 actual.pdf.

[23] F. Cristian and C. Fetzer, "The timed asynchronous distributed system model," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 6, pp. 642–657, 1999.

[24] A. Schiper and S. Toueg, "From Set Membership to Group Membership: A Separation of Concerns," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 2, pp. 2–12, 2006.

[25] The Apache Hadoop Project, "Zookeeper website," http://hadoop.apache.org/zookeeper/.

[26] J. Yin, L. Alvisi, M. Dahlin, and C. Lin, "Volume leases for consistency in large-scale systems," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 11, no. 4, pp. 563 –576, jul. 1999.

[27] F. Hupfeld, T. Cortes, B. Kolbeck, J. Stender, E. Focht, M. Hess, J. Malo, J. Marti, and E. Cesario, "XtreemFS: a case for object-based storage in Grid data management," in *3rd VLDB Workshop on Data Management in Grids, co-located with VLDB 2007*, 2007.

[28] Samba Project, "The dbench website," http://dbench.samba.org/.

[29] IOZone.org, “The iozone file system benchmark,” http://www.iozone.org/.

[30] G. Chockler and D. Malkhi, “Light-weight leases for storage-centric coordination,” *Int. J. Parallel Program.*, vol. 34, no. 2, pp. 143–170, 2006.

[31] R. Boichat, P. Dutta, and R. Guerraoui, “Asynchronous leasing,” in *WORDS ’02: Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*. Washington, DC, USA: IEEE Computer Society, 2002, p. 180.

[32] B. W. Lampson, “How to build a highly available system using consensus,” in *WDAG ’96: Proceedings of the 10th International Workshop on Distributed Algorithms*. London, UK: Springer-Verlag, 1996, pp. 1–17.

[33] F. Hupfeld, B. Kolbeck, J. Stender, M. Högqvist, T. Cortes, J. Marti, and J. Malo, “Fatlease: scalable fault-tolerant lease negotiation with paxos,” in *HPDC ’08: Proceedings of the 17th international symposium on High performance distributed computing*. New York, NY, USA: ACM, 2008, pp. 1–10.

[34] J. R. Douceur and J. Howell, “Distributed directory service in the farsite file system,” in *OSDI ’06: Proceedings of the 7th symposium on Operating systems design and implementation*. Berkeley, CA, USA: USENIX Association, 2006, pp. 321–334.

[35] Lustre, Inc., “Lustre architecture whitepaper,” 2002.

[36] J. Maccormick, C. A. Thekkath, M. Jager, K. Roomp, L. Zhou, and R. Peterson, “Niobe: A practical replication protocol,” *Trans. Storage*, vol. 3, no. 4, pp. 1–43, 2008.

[37] A. Robertson, “Linux-HA heartbeat system design,” in *ALS’00: Proceedings of the 4th annual Linux Showcase & Conference*. Berkeley, CA, USA: USENIX Association, 2000, pp. 20–20.

[38] L. Lamport, “Paxos made simple,” *SIGACT News*, vol. 32, no. 4, pp. 18–25, 2001.

[39] ——, “Lower bounds on asynchronous consensus,” in *Future Directions in Distributed Computing*, ser. Lecture Notes in Computer Science, A. Schiper, A. A. Shvartsman, H. Weatherspoon, and B. Y. Zhao, Eds., vol. 2584. Springer, 2003, pp. 22–23.