Konrad-Zuse-Zentrum
für Informationstechnik Berlin

Takustraße 7
D-14195 Berlin-Dahlem
Germany

BJÖRN KOLBECK, MIKAEL HÖGQVIST, JAN STENDER,
FELIX HUPFELD

Fault-tolerant and Decentralized Lease Coordination in Distributed Systems

ZIB-Report 10-02 (February 2010)

Fault-tolerant and decentralized lease coordination for distributed systems

Björn Kolbeck, Mikael Högqvist, Jan Stender, Felix Hupfeld

Abstract

Applications which need exclusive access to a shared resource in distributed systems require a fault-tolerant and scalable mechanism to coordinate this exclusive access. Examples of such applications include distributed file systems and master/slave data replication.

We present Flease, an algorithm for decentralized and fault-tolerant lease coordination in distributed systems. Our algorithm allows the processes competing for a resource to coordinate exclusive access through leases among themselves without a central component. The resulting system easily scales with an increasing number of nodes and resources. We prove that Flease ensures exclusive access, i.e. guarantees that there is at most one valid lease at any time.

1 Introduction

A broad range of applications require exclusive access to a shared resource in a distributed system. A resource could be a file, a hard disk block or an exclusive master role: Replicated file systems [1, 20, 21] have to ensure consistency of a file's replicas; shared-disk file systems [18, 19] must coordinate concurrent writes to the same disk block and master/slave replication need a single master which orders updates. The last example is particularly interesting as it is often used to simplify the design of distributed systems [16]. For these applications we need a mechanism to arbitrate which node gets access, i.e. who becomes the temporary *owner* of a resource. Such an mechanism must be:

fault tolerant. If the owner of a resource crashes or is disconnected another node should be able to become the new owner. Likewise, the mechanism itself must not introduce a single point of failure to the system.

scalable. The mechanism must be able to automatically scale with number of nodes and resources. This explicitly excludes any manual partitioning of the system into smaller cells, we expect the mechanism to grow with the number of nodes. Regarding the number of resources, this means that we don't want to artificially group resources into larger units. The resources should be as fine-grained as necessary to avoid contention and to distribute the workload across many machines.

Leases [10] are a mechanism to coordinate exclusive access which meets our requirements. Leases are fault-tolerant locks that grant exclusive access to a

resource for a limited pre-defined time span. If the process holding a lease crashes, the resource becomes automatically available again as soon as the lease expires. There is no need for complex failure detectors, the lease time-out serves as an intrinsic failure detector based on real-time clocks available in all modern computers.

We present Flease, an algorithm for decentralized, fault-tolerant lease coordination. Flease is built on top of a quorum-based distributed register [2] derived from Paxos [12] and can tolerate message loss and the failure of a minority of the participants. In contrast to the regular Paxos algorithm, our algorithm allows processes to recover without using persistent state. As a fully decentralized algorithm, Flease can coordinate a large number of fine-granular leases. Since the nodes competing for the resource coordinate the leases among themselves, Flease easily scales with the size of the system.

We continue the paper with an overview of related work on algorithms for distributed lease coordination for various system models and centralized lock services used in practical systems. We start the description of Flease with a review of its underlying round-based register. We then construct a basic version of Flease that does not include lease renewal and assumes perfectly synchronized clocks. We further refine the algorithm to allows lease renewals. Finally, we presents the full Flease algorithm for processes with loosely synchronized clocks and illustrate how processes can recover without persistent state.

2 Related Work

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

Chockler and Malkhi [6] presented a fault-tolerant algorithm for timed asynchronous systems with shared memory. They specifically designed their algorithm for SAN-based file systems in which a shared memory is present. This model is not applicable to shared-nothing architectures as used e.g. in distributed file systems exploiting commodity hardware.

A leader election algorithm for the timed asynchronous model is also presented by Fetzer et al. in [7]. This algorithm implements a leader election with expiration time which is basically a lease. The algorithm is not fault-tolerant and does not consider processes which recover after a crash.

An algorithm for truly asynchronous system was presented by Boichat et al. [3]. However, due to the lack of time in asynchronous systems, their leases approach works with logical time. This means that their variation of leases do 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 [15] argued that consensus with Paxos can be made more efficient by using a single master elected with a lease.

Flease is a simplified version of FaTLease [11] which uses regular consensus with Paxos to agree on a lease owner. A scheme with instances similar to Multipaxos [17] 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.

Another option for lease coordination is a central lock service. This central service, however, doesn't meet our requirements as it does not scale with the number of resources in the system. The number of leases the lock service can handle limits the size of the system and the granularity of the resources. In practice, this approach is widely used at the cost of having very coarse-grained resources [16].

The most prominent example is Google's Chubby lock service [4, 5] which is implemented using Paxos to replicate the lease database. Chubby is used by other services at Google, e.g. for master election in the Google File System (GFS) [9]. For file replication, the WheelFS [20] authors suggest to use a central lock service for master-leases. The Frangipani [21] file system design included a Paxos-replicated configuration service which issues locks to partition-masters, similar to Chubby. Farsite [1], a distributed peer-to-peer file system, uses leases for data and metdata access. To avoid contention on metdata entries, each field of a metadata record has its own lease [8].

The Paxos algorithm [12, 13, 14] 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 it is widely used. The Paxos algorithm relies on a quorum approach and is consequently able to tolerate failure of a minority (up to $\lceil \frac{n+1}{2} \rceil$ out of n processes) 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 that can be significant. For the Flease algorithm we use the abstraction of a round-based register which was derived from Paxos in a modularized deconstruction by Boichat et. al [2].

3 The Flease Algorithm

The main building block of Flease is a round-based register derived from Paxos [2]. It has the same properties as Paxos regarding process failures and message loss but assumes processes to be crash-stop as it lacks persistent storage. The register implements a shared read-modify-write variable in a distributed system which arbitrates concurrent accesses. The semantics of the register resembles that of a microprocessor's test-and-set operation.

Flease stores the currently valid lease in the register. If a process wants to acquire the lease or wants to find out which process holds the lease, it starts by reading the register's value. If the register is empty or the lease stored in the register has expired, the process creates a new lease and stores it in the register. The currently valid lease, or the newly created lease, is returned as the result.

We start the presentation with this basic algorithm. The basic version assumes perfectly synchronized clocks and does not allow leases to be renewed. In a refined version we include lease renewal. For the final version we take care of loosely synchronized clocks. Finally, we describe how the crash-stop register can be turned into a crash-recovery register without persistent storage exploiting the loosely synchronized clocks.

3.1 System Model and Definitions

We assume a system model similar to the timed asynchronous model defined in [7] with a finite and fixed set of processes $\Pi = p_1, p_2..p_n$ with each processes making progress at its own speed. We also assume that each process p_i has access to a local (hardware) clock c_i. These clocks exhibit a strictly monotonically increase of the time they return, i.e. $c_i(t) < c_i(t')$ if $t < t'$. We require that the processes maintain a loosely synchronized time, i.e. there is a known upper bound ϵ on the drift between any two clocks. This means 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 or equal 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. To simplify the presentation, we start with an initial version of Flease that assumes perfectly synchronized clocks, i.e. $\epsilon = 0$. In the final version, we extend the algorithm to also consider loosely synchronized clocks.

We assume the communication channels to be unreliable in the sense that messages can be lost and delayed but are not altered or duplicated. To simplify the presentation we start with a crash-stop model in which processes stop after failing. For the final version of the algorithm, we extend this to a crash-recovery model where processes recover and re-join the system after failing. We do not assume that our processes have access to stable storage.

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 $c_i(t_{now}) < t$. A lease has expired if $c_i(t_{now}) > t$ with t_{now} as the current time. We define the maximum time span of a lease to be valid as t_{max}. For the system to make progress, we require that $t_{max} > \epsilon$.

3.2 The Distributed Round-Based Register

The algorithm for the register is shown in figure 1. The register has two operations: READ(k) and WRITE(k, v). k is a unique identifier generated by the process initiating the operation. In Paxos k is the ballot number of the proposal. We assume that there is total order on the values for k. v is the value to be written to the register. Both operations either commit or abort. If read commits, it returns the current value v of the register or $\perp$ if the register is empty. The full algorithm and proof of the following lemmas can be found in [2].

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') can commit with $k' \leq k$ or WRITE($k'', *$) can commit with $k'' \leq k$.

Algorithm 1 round based register from [2]

$read_i \leftarrow 0$
$write_i \leftarrow 0$
$v_i \leftarrow \perp$

procedure READ(k)
  send (READ,k) to all processes in Π
  wait until **received** (ackREAD,k,$*$,$*$) or (nackREAD,k)
    from $\lceil \frac{n+1}{2} \rceil$ processes
  if received at least one (nackREAD,k) **then**
    return (abort,$\perp$)
  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 $\lceil \frac{n+1}{2} \rceil$ 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 $write_i \geq k$ **or** $read_i \geq k$ **then**
    send (nackREAD,k) to p_j
  else
    $read_i \leftarrow k$
    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
    $write_i \leftarrow k$
    $v_i \leftarrow v$
    send (ackWRITE,k) to p_j
  end if
end upon

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, commits with v if $k'' > k$.

3.3 The basic Flease algorithm

Algorithm 2 The basic algorithm

```
procedure GetLease(k)
    if read(k) = (commit, λ) then
        if λ =⊥ or λ.t < t_now then
            λ ← (p_i, t_now + t_max)
        end if

        if write(k,λ) = commit then
            return (commit,λ)
        end if
    end if
    return (abort,⊥)
end procedure
```

The basic version ofFlease is shown in algorithm 2. The GetLease procedure returns either the currently valid lease or a new lease with the local process as the lease owner.

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 hosts 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 which 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 take one of the 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 to the assumption that $t > t_{now}$ and $t' > t_{now}$.

3.4 Including Lease Renewals

With the basic version (algorithm 2), the lease owner will lose its lease when the old lease has timed out and the new lease is being coordinated. During this time, which takes at least two message round trips, there is no lease and consequently the resource cannot be accessed. To avoid these interruptions, the owner of a lease should be allowed to extend the lifetime of the lease as long as the original lease is still valid. Algorithm 3 shows an extended version of the basic algorithm which includes lease renewals.

Algorithm 3 The extended algorithm with lease renewal

```
procedure GetLease(k)
    if read(k) = (commit, λ) then
        if λ =⊥ or λ.t < t_now then
            λ ← (p_i, t_now + t_max)
        else if λ.p = p_i then
            λ ← (p_i, t_now + t_max)
        end if

        if write(k,λ) = commit then
            return (commit,λ)
        end if
    end if
    return (abort,⊥)
end procedure
```

To allow lease renewal we need to relax the lease invariant to require the 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 as 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 which are valid at the same time and have different lease owners. Depending on the value of $\lambda.t$, process p_j will take 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 $t' > t_{now}$.

3.5 Allowing Processes to Recover

The round-based register assumes a crash-stop model as it does not use persistent storage to recover the register content 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) and k as well as v must be stored for WRITE(k, v). Thus, the state of the register on each node i comprises of the three values $read_i, write_i$ and v_i.

With leases 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. Therefore, we can turn the register into a crash-recovery model for our leases. We require a recovering process to wait until t_{max} has passed, before it can rejoin the system. During this waiting period, the process is not allowed to participate in lease coordination and must not send messages. By waiting until t_{max} has passed, we can guarantee that any lease which 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 which recovered from a crash has lost its complete state which also includes the $read_i$ and $write_i$. To guarantee that any k' used after such a crash is larger than the maximum k seen before the crash, we take again 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.

3.6 Final Algorithm with Loosely Synchronized Clocks

An algorithm based on perfectly synchronized clocks is of little practical use. To make Flease suitable for real-world use, we extend it to work with loosely synchronized clocks as in FaTLease. 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 (Fig. 3) which also takes the clock skew 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 holder still considers the lease to be valid or expired due to the clock skew. However, after ϵ time, any host can safely assume that the lease has expired on all hosts and can execute the regular algorithm 3.

It is easy to see that the algorithm is correct regarding the lease invariant as this part is identical to algorithm 3.

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

Algorithm 4 The full Flease algorithm for loosely synchronized clocks

```
procedure GetLease(k)
    if read(k) = (commit, λ) then
        if λ.t < t_now and λ.t + ε > t_now then
            wait for ε
            return GetLease(k')                          ▷ with k' > k
        end if

        if λ =⊥ or λ.t < t_now then
            λ ← (p_i, t_now + t_max)
        else if λ.p = p_i then
            λ ← (p_i, t_now + t_max)
        end if

        if write(k,λ) = commit then
            return (commit,λ)
        end if
    end if
    return (abort,⊥)
end procedure
```

clocks. However, the process with the fastest clock (i.e. maximum $c(t)$) will always acquire the lease. To circumvent this problem, we consider time-ranges for comparison rather than just the timestamps. As long as $|t - t'| \leq \epsilon$ we consider the messages to be equal and use the process id and a random value to distinguish the messages.

4 Discussion

We have presented Flease, a decentralized and fault-tolerant algorithm for lease coordination in distributed systems. Our proofs demonstrate that Flease guarantees the exclusiveness of leases.

Compared to coordinating leases with Paxos, Flease offers the same fault-tolerance and cost in terms of messages and message round-trips. However, it is more efficient as it does not require persistent storage consuming less resources on the hosts it is executed on. This is particularly important in distributed file systems and databases which require the maximum I/O performance. For these systems, Flease offers a scalable alternative to centralized lock services.

5 Acknowledgments

We thank Minor Gordon for his detailed comments. This work has been funded partially by the European Commission's IST programme as part of XtreemOS project under contract #FP6-033576.

References

[1] 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*, pages 1–14, New York, NY, USA, 2002. ACM Press.

[2] R. Boichat, P. Dutta, S. Frolund, and R. Guerraoui. Deconstructing paxos. *SIGACT News*, 34(1):47–67, 2003.

[3] 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)*, page 180, Washington, DC, USA, 2002. IEEE Computer Society.

[4] M. Burrows. Chubby distributed lock service. In *Proceedings of the 7^{th} Symposium on Operating System Design and Implementation, OSDI'06*, Seattle, WA, November 2006.

[5] T. D. 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*, pages 398–407, New York, NY, USA, 2007. ACM Press.

[6] G. Chockler and D. Malkhi. Light-weight leases for storage-centric coordination. *Int. J. Parallel Program.*, 34(2):143–170, 2006.

[7] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Trans. Parallel Distrib. Syst.*, 10(6):642–657, 1999.

[8] 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*, pages 321–334, Berkeley, CA, USA, 2006. USENIX Association.

[9] 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*, pages 29–43, New York, NY, USA, 2003. ACM Press.

[10] 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*, pages 202–210, New York, NY, USA, 1989. ACM Press.

[11] 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*, pages 1–10, New York, NY, USA, 2008. ACM.

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

[13] L. Lamport. Paxos made simple. *SIGACT News*, 32(4):18–25, 2001.

[14] L. Lamport. Lower bounds on asynchronous consensus. In A. Schiper, A. A. Shvartsman, H. Weatherspoon, and B. Y. Zhao, editors, *Future Directions in Distributed Computing*, volume 2584 of *Lecture Notes in Computer Science*, pages 22–23. Springer, 2003.

[15] B. W. Lampson. How to build a highly available system using consensus. In *WDAG '96: Proceedings of the 10th International Workshop on Distributed Algorithms*, pages 1–17, London, UK, 1996. Springer-Verlag.

[16] M. K. McKusick and S. Quinlan. Gfs: Evolution on fast-forward. *Queue*, 7(7):10–20, 2009.

[17] R. D. Prisco, B. Lampson, and N. Lynch. Revisiting the Paxos algorithm. *Theor. Comput. Sci.*, 243(1-2):35–91, 2000.

[18] 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*, page 19, Berkeley, CA, USA, 2002. USENIX Association.

[19] S. R. Soltis, T. M. Ruwart, and M. T. O'Keefe. The global file system. In *5th NASA Goddard Conference on Mass Storage Systems and Technologies*, 1996.

[20] 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*, pages 43–58, Berkeley, CA, USA, 2009. USENIX Association.

[21] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: a scalable distributed file system. *SIGOPS Oper. Syst. Rev.*, 31(5):224–237, 1997.