PaxosLease: Diskless Paxos for Leases

Marton Trencseni, mtrencseni@scalien.com
Attila Gazso, agazso@scalien.com
Holger Reinhardt, hreinhardt@gmail.com

Abstract

This paper describes PaxosLease, a distributed algorithm for lease negotiation. PaxosLease is based on Paxos, but does not require disk writes and does not make clock synchrony and skew assumptions. PaxosLease is used for master lease negotation in the open-source Keyspace replicated key-value store.

1 Introduction

In concurrent programming, *locks* are a basic primitive which processes use to synchronize access to a shared resource. In a system where locks are granted without an automatic expiry time (and without an overseer process), failure of the lock owner before it releases the lock may cause other processes to block.

In a highly-available distributed system, one wishes to avoid single node failure to cause the entire system to block. Also, ”restarting” the system in case of failure may be harder restarting a multi-threaded program. Thus, in distributed systems, *leases* take the place of locks to avoid starvation. *A lease is a lock with an automatic expiry time.* If the owner of the lock fails or becomes disconnected from the rest, its lease automatically expires and other nodes can acquire it.

We assume the basic set up is as follows: the system consists of a set of *poposers* who follow the proposer's algorithm and a set of *acceptors* who follow the acceptor's algorithm. It is assumed the system is non-byzantine, ie. the nodes do not cheat (are not hacked) by not following their algorithm. The number of acceptors is fixed and does not change.

A naive, majority vote type algorithm can be given which correctly solves the distributed lease problem, *correct* meaning that the lease is held by no more than one node at any time. However, this simple algorithm will frequently *block* in the presence of many proposers, hence the need for a more sophisticated approach.

The naive majority algorithm is as follows: proposers start local timeouts for T seconds and send requests to the acceptors for the lease with timespan T. The acceptors, upon receiving the request, start a timer for T seconds and sends an accept message to the proposer. After the timeout, the acceptors clear their state. If an acceptor receives a request but its state is not empty it either does not answer or sends a reject message. To make sure only one proposer has the lease at any time, the proposer must receive accept messages from a majority of acceptors; then it has the lease until its local timer expires.

As discussed previously, it is possible (and likely), that with many proposers, noone will be able to get a majority and proposers will continually block each other. For example, with three proposers 1, 2 and 3 and three acceptors A, B and C, if the distributed state is such that A accepts 1's request, B 2's and C 3's, then no proposer has a majority. The system must wait until the timeouts expire and the acceptors discard their state at which point the proposers will try again. However, it is likely that the system will block again.

The solution described in this paper is to follow the scheme of Paxos [1], and introduce *prepare* and *propose* phases, which avoids this kind of blocking altogether[1]. Paxos solves the problem of replicated state-machines, where each node has a local copy of a state-machine, and they wish to reach consensus on the next state transition. Paxos is a majority based algorithm, meaning a majority of the nodes have to be up and communicating for consensus to be possible. Paxos deals with reaching consensus on a single state transition, so in practice several Paxos rounds are run in sequence to negotiate the sequence of state transitions [3]. In Paxos, acceptors record their state to disk before sending responses, which guarantees that once a value (state transition) is chosen, it is chosen at all later times; in other words, all state machines go through the same sequence of state transitions, whatever error conditions occur.

Unlike earlier Paxos-based distributed lease algorithms such as Fatlease [5], PaxosLease does not make any time synchrony assumptions about local clocks of nodes (no global synchronization with a bounded skew is required). Also, Fatlease runs Paxos rounds in succession for lease commands, whereas PaxosLease makes use of the temporal nature of leases and avoids this complication altogether for a much simpler and elegant algorithm.

PaxosLease is a natural specialization of Paxos. As in Paxos, it is assumed that the number of nodes is fixed (and node identities are globally known). PaxosLease deals with a special replicated state-machine of the form:

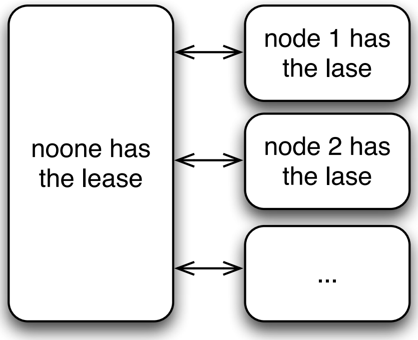

Figure 1: PaxosLease distributed state machine.

To acquire the lease, PaxosLease proposer nodes propose the value: "node i has the lease", and the return to the "noone has the lease" state is automatic after the expiry time. Proposers may also extend their leases by proposing the "node i has the lease" value again before their previous lease expires, and may optionally release the lease before it expires.

Similar to Paxos, PaxosLease intrinsically handles all relevant failure conditions:

1. nodes stop and restart
2. network splits

[1] Another solution may be to let the system block, but introduce an "undo" mechanism which lets some proposers undo their proposals to let one proposer acquire the lease.

3. message loss and reordering
4. in-transit message delays

2 Definitions

A PaxosLease cell consists of proposers and acceptors. We assume there are n acceptors and any number of proposers. In practice, nodes often act as proposers and acceptors, but this is an implementation issue and does not affect the discussion.

Proposers send *prepare request* and *propose request* messages to acceptors, who reply with *prepare response* and *propose response* messages. The messages have the following structure:

1. prepare request = ballot number
2. prepare response = ballot number, answer, accepted proposal
3. propose request = ballot number, lease
4. prepare response = ballot number, answer

A ballot number and a lease together make up a *proposal.* A *lease* is made up of a *proposer id* (who wishes to become the lease owner) and a timespan T.

Acceptors store the following state information:

1. highest ballot number promised: acceptors ignore messages whose ballot number is less than this
2. accepted proposal: the last proposal (ballot number and lease) accepted

There is a globally known maximal lease time M. Proposers always acquire leases for timespans $T < M$.

Ballot numbers are globally unique and monotonously increasing per proposer. In practice, this can is achieved by ballot numbers being made up of a *proposer id* field, a *restart counter* field and a *run counter* field (at the most significant end) specific to this run of the proposer. The restart counter is incremented each time the proposer starts up and is written to stable storage.

PaxosLease preserves the *lease-invariant*: at any given time, there is no more than one proposer which holds the lease.

3 Basic algorithm

This section describes the basic flow of the algorithm in terms of the proposer's and acceptor's algorithm. The proposers send prepare and propose requests, acceptors reply with prepare and propose responses. If all goes well, it takes two round-trip times for the proposer to acquire the lease.

1. A proposer wishes to acquire the lease for $T < M$ seconds. It generates a ballot number `[request.ballotNumber]`, chooses a time T and sends prepare request messages to a majority of acceptors.

```
Proposer::Propose()
{
   state.ballotNumber = NextBallotNumber();
   state.timeout = T
   reqest.type = PrepareRequest
   reqest.ballotNumber = state.timeout
   Broadcast(request)
}
```

2. Acceptors, upon receiving a prepare request check whether the ballot number `[request.ballotNumber]` is higher than their local highest ballot number promised `[state.highestPromised]`. If it is lower, they either discard the message or send a prepare response with a *reject* answer. If it equal or higher, the acceptor constructs a prepare response with an *accept* answer containing the currently accepted proposal `[state.acceptedProposal]`, which may be *empty*. The acceptor sets its highest ballot number promised `[state.highestPromised]` equal to the ballot number of the message `[request.ballotNumber]` and sends the prepare response back to the proposer.

   ```
   Acceptor::OnPrepareRequest()
   {
      if (request.ballotNumber < state.highestPromised)
         return
      state.highestPromised = reqeust.ballotNumber
      response.type = PrepareRespose
      resonse.ballotNumber = request.ballotNumber
      reponse.acceptedProposal = state.acceptedProposal  // may be 'empty'
      Send(response)
   }
   ```

3. The proposer examines the prepare responses from the acceptors. If a majority of acceptors responded with empty proposals, meaning they are open to accept new proposals, the proposer is free to propose itself as the lease owner for time T. It starts a timer which will expire in T seconds and sends propose requests containing the ballot number and the lease (its *proposer id* and T).

   ```
   Proposer::OnPrepareResponse()
   {
      if (response.ballotNumber != state.ballotNumber)
         return // some other proposal
      if (response.acceptedProposal == 'empty')
         numOpen++
      if (numOpen < majority)
         return
      SetTimeout(state.timeout)
      request.type = ProposeRequest
      request.ballotNumber = state.ballotNumber
      request.proposal.proposerID = me
      request.proposal.timeout = state.timeout
      Broadcast(request)
   }

   Proposer::OnTimeout()
   {
      state.leaseOwner = false // set in Proposer::OnProposeResponse()
   }
   ```

4. Acceptors, upon receiving a propose request check whether the ballot number `[request.ballotNumber]` is higher than their highest ballot number promised `[state.highestPromised]`. If it is lower, they can either discard the message or send a propose response with a *reject* answer. If it equal or higher, the acceptor accepts the proposal: if starts a timeout which will expire in T seconds and sets its accepted proposal to the proposal received (if it has stored a previous proposal, it discards it). It then constructs a propose response with an *accept* answer containing the ballot number

`[request.ballotNumber]`. After the timeout expires the acceptor resets its accepted proposal to *empty*. Acceptors never reset their highest ballot number promised, except when restarting.

```
Acceptor::OnProposeRequest()
{
   if (request.ballotNumber < state.highestPromised)
      return
   state.acceptedProposal = request.proposal
   SetTimeout(state.acceptedProposal.timeout)
   response.type = ProposeResponse
   response.ballotNumber = request.ballotNumber
   Send(response)
}

Acceptor::OnTimeout()
{
   state.acceptedProposal = empty
}
```

5. The proposer examines the propose responses messages. If a majority of acceptors responded accepting the proposal, the proposer has the lease until its local timeout, started in step 3. expires. The time at which the proposer receives the last message necessary for a majority is the time at which it acquired the lease and may switch its internal state to ”I have the lease”.

```
Proposer::OnProposeResponse()
{
   if (response.ballotNumber != state.ballotNumber)
      return // some other proposal
   numAccepted++
   if (numAccepted < majority)
      return
   state.leaseOwner = true // I am the lease owner
}
```

Note that acceptors do not store their state to stable storage. Upon restart, they start out in a blank state. To make sure restarting nodes do not violate the lease invariant, nodes wait M seconds before rejoining the network. M is a globally known maximal lease time known to all nodes, and proposers always acquire leases for $T < M$ seconds.

It is important to note that since all that is transmitted is timespans (relative times), only the proposer which has the lease knows it has the lease. It cannot tell other nodes it has the lease (similar to learn messages is classical Paxos or Fatlease), since other nodes cannot know how much time these learn messages spent in transit. Thus, only the proposer which has the lease knows it has the lease. All other nodes know is that they do not have the lease. In other words, each proposer has two states regarding the lease: ”I do not have the lease and I don’t know who has the lease” and ”I have the lease”. Of course, nodes may send out learn messages as *hints* which may be used in higher-level applications or heuristics, but this is beyond the scope of this paper.

It is possible that a proposer cannot get a majority of acceptors to send favorable responses in steps 3. and 5. above. In this case the proposer may sleep for a while and then restart its algorithm at step 1. with a higher ballot number.

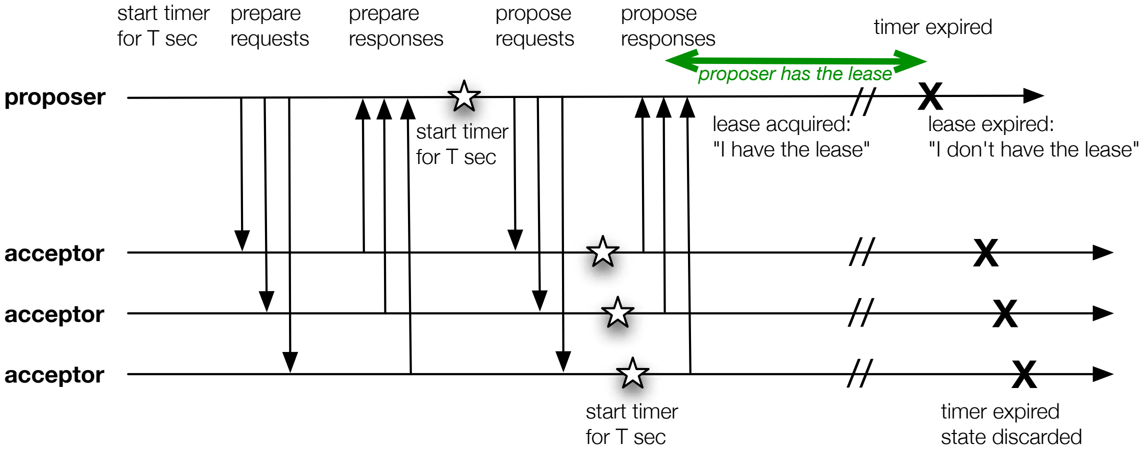

Figure 2: Time flow of a proposer acquiring the lease.

4 Proof of lease invariance

First we give the reader an intuitive notion of why PaxosLease works. Figure 2. (above) is a pictorial explanation: the proposer starts its timer before sending out propose requests, acceptors only start their timer *later*, before sending out propose responses, thus, if a majority of acceptors stored the state and started timers, no other proposer will be able to get the lease before the proposer's timer expires. No two proposers will simultaneously believe themselves to be lease holders.

More formally, PaxosLease guarantees that if proposer i receives propose accept messages from a majority of acceptors for its proposal with ballot number b and timespan T at time t_{now}, then if it started its timer at time t_{start}, no other proposer will receive a similar majority until $t_{end} = t_{start} + T$.

Proof: Suppose proposer p with ballot number b acquired the lease. It received empty prepare responses of type *accept* from a majority of acceptors and started its timer at t_{start}, received propose responses of type *accept* from a majority of acceptors at $t_{acquire}$, and thus has the lease until $t_{end} = t_{start} + T$. Let A_1 be the majority of acceptors who replied with empty prepare responses to p's prepare requests and A_2 be a majority of acceptors who accepted p's proposal and sent prepare responses of type *accept*.

Part 1. No other proposer q can hold the lease between $t_{acquire}$ and t_{end} with ballot number $b' < b$. In order to hold the lease, the proposer q must get a majority A'_2 of acceptors to accept its lease. Let a be an acceptor who is in both A'_2 and A_1: since $b' < b$, a must have first accepted q's proposal and then sent a prepare response to p. However, if a sent an empty prepare response to p its state must have been empty, its timer must have expired, thus q's timer also expired, thus q already lost the lease. There is no overlap between p and q's lease.

Part 2. No other proposer q can hold the lease between $t_{acquire}$ and t_{end} with ballot number $b < b'$. In order to hold the lease, the proposer q must get a majority A'_1 of acceptors to send it empty prepare responses. Let a be an acceptor who is in both A'_1 and A_2: since $b < b'$, a must have first accepted p's proposal and then sent a prepare response to q. However, since a accepted p's proposal, if it sent an empty prepare response to q its state must have been empty, its timer must have expired, thus p's timer also expired, thus p already lost the lease. There is no overlap between p and q's lease.

5 Liveness

Paxos-type algorithms such as PaxosLease contain the possibility of dynamical deadlock: two proposers can continually generate higher and higher ballot numbers, send prepare requests to acceptors, who continually increase their highest ballot number promised, and neither of the proposers can get acceptors to accept their proposals. In practice, this can be worked around by proposers waiting a small but random time before re-running their algorithm.

The main advantage of Paxos-type algorithms is that static deadlocks, as described in the naive majority vote algorithm are not possible because proposers can overwrite acceptor states and the algorithm guarantees that no majority is overwritten.

6 Extending leases

In some situations, once a proposer has the lease, it is important that it can hold on to it beyond its original lease time. A typical case is whent the lease identifies the master node in a distributed system and it is desirable that a node can hold on to its mastership for a long time.

To accomodate this only the proposer's algorithm needs to be modified. In step 3., if a majority responded either with empty proposals *or its existing proposal* whose lease has not yet expired, it can propose itself again as the lease holder. This will allow the proposer to extend its lease by $O(T)$ time. The acceptor's algorithm does not change.

7 Releasing leases

In the algorithm described so far, the proposerss lease automatically expire after some time. In some situations, it is important to release the lease as soon as possible so that other nodes may acquire it. A typical case is a distributed process, where workers acquire a lease for a resource, do work on it, and then wish to release the lease as asson as possible so that other workers can acquire it.

To accomodate this, any proposer may at any point in time send special release messages to acceptors which contain the ballot number of the lease they wish to release. Before sending out the release messages, the proposer switches its internal state from ”I have the lease” to ”I do not have the lease”. If an acceptor receives a release message, it checks whether its accepted ballot number matches. If if does, it discards its state; otherwise it does nothing. Proposers can also send out release messages to other proposers as hints, telling them that they may be able to acquire the lease now.

8 Leases for many resources

The algorithm defines lease actions concerning a single resource R. In practice nodes may deal with many resources, such as leases in a distributed process. PaxosLease can be used by running independent instances of the algorithm for each resource by tagging each message, proposer and acceptor state with a *resource identifier*. A node acting as proposer and acceptor requires no more than ∼ 100 bytes of memory per PaxosLease instance, which means it can handle ∼ 10 million resource leases per gigabyte of main memory. This and the fact that PaxosLease does not require disk syncs or time synchronization means the algorithm may be used in a wide variety of scenarios for fine-grained locking.

9 Implementation

PaxosLease is used for master lease negotiation is Scalien Software's Keyspace replicated key-value store. Since Keyspace is licensed under the Affero GPL, this reference implementation, which contains many practical optimizations is freely available to all interested readers. The source code and binaries may be downloaded at `http://scalien.com`.

10 Genealogy

Leslie Lamport invented Paxos in 1990, but it was only published in 1998. This paper, *'The Part-Time Parliament'* was "greek to many readers", which led to a second paper, *'Paxos Made Simple'*. Paxos solves the distributed consensus problem by introducing prepare and propose phases and having acceptors write their state to stable storage before replying to messages. Multiple rounds of Paxos may be run sequentially to negotiate state transitions of a replicated state machine.

Paxos was popularized by its use in Google's in-house distributed stack, described in *'Paxos Made Live - An Engineering Perspective'* and *'The Chubby Lock Service for Loosely-Coupled Distributed Systems'*. In Google's Chubby, multiple, sequential rounds of Paxos are used to reach consensus on the next write opereration on a replicated database, another way of looking at a replicated state machine.

Fatlease, described in *'FaTLease: Scalable Fault-Tolerant Lease Negotiation with Paxos'* solves the same problem as PaxosLease, but it is much more complicated as it mimicks the multiple Paxos rounds approach instead of simply expiring the state of acceptors, as in PaxosLease. Also, Fatlease requires the nodes to synchronize their clocks, making it unattractive for real-world use. PaxosLease was inspired by Fatlease and addresses the aforementioned weaknesses.

References

[1] L. Lamport, *The Part-Time Parliament*, ACM Transactions on Computer Systems 16, 2 (May 1998), 133-169.

[2] L. Lamport, *Paxos Made Simple*, ACM SIGACT News 32, 4 (Dec. 2001), 1825.

[3] T. Chandra, R. Griesemer, J. Redstone, *Paxos Made Live - An Engineering Perspective*, PODC '07: 26th ACM Symposium on Principles of Distributed Computing

[4] M. Burrows, *The Chubby Lock Service for Loosely-Coupled Distributed Systems*, OSDI'06: Seventh Symposium on Operating System Design and Implementation.

[5] F. Hupfeld et al., *FaTLease: Scalable Fault-Tolerant Lease Negotiation with Paxos*, HPDC08, June 2327, 2008, Boston, Massachusetts, USA.

[6] `http://www.gnu.org/licenses/agpl-3.0.html`