PaxosLease: Diskless Paxos for Leases

Marton Trencseni

March 4, 2009

1 Introduction

1.1 Failure conditions

PaxosLease handles the following failure conditions:

1. Nodes stop and restart
2. Network splits
3. Message loss
4. Message reordering
5. In-transit message delays

PaxosLease does not handle Byzantine failures and messages corruption.

2 Preliminaries

2.1 Definitions

2.1.1 Actors

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

2.1.2 Messages

Proposers send *prepare request* and *propose request* messages to acceptors, who reply with *prepare response* and *propose response* messages. Proposers send *learn* messages to learners.

The messages have the following structure:

1. *<prepare request> = <ballot number>*
2. *<prepare response> = <ballot number, accepted proposal>*
3. *<propose request> = <proposal>*

4. <*propose response*> = <*ballot number*>

5. <*learn*> = <*value*>

A *proposal* consists of a *ballot number* and a *value*.

2.1.3 State

Acceptors store the following state information:

1. *highest ballot promised*: the highest ballot number sent in a prepare response

2. *accepted proposal*: the last proposal accepted

2.1.4 Ballot numbers

Ballot numbers are positive integers generated by proposers. Different proposers generate different ballot numbers. In implementations, this is achieved by choosing ballots numbers such that *ballot number mod n = proposer id*.

2.1.5 Chosen value

A proposal p containing value v is chosen if *accepted proposal* $= p$ for a majority of acceptors.

2.2 Algorithm

2.2.1 Basic flow

A proposer wishing to propose a value first generates a *globally unique ballot number* and sends prepare request messages to a majority of acceptors.

The acceptor, upon receiving a prepare request with *ballot number*, checks whether *ballot number* > *highest ballot promised*. If not, the acceptor does not respond to the message. If yes, the acceptor constructs a prepare response containing the *ballot number* and the currently accepted proposal. If it has not accepted a proposal, it sends special *empty proposal*. The acceptor sets *highest ballot promised* := *ballot number* and then sends the prepare response to the proposer.

The proposer, upon receiving prepare responses from a majority of acceptors, examines the responses. If a majority of nodes responded with empty values, the proposer is free to propose its own value. If not, it must propose the value belonging to the highest *accepted ballot number* among the prepare responses. The proposer sends propose requests containing the generated ballot number and the value to the acceptors who responded.

The acceptor, upon receiving a propose request with proposal p containing *ballot number*, checks its *highest ballot promised*. If *ballot number* < *highest ballot promised* it ignores the message, else it accepts the message by setting *accepted proposal* := p. Finally it sends a propose response with the ballot number.

2.3 Paxos

Augmenting the above rules we get Paxos: acceptors store their state to stable storage before sending responses.

2.3.1 Consistency

Paxos guarantees that only one value is chosen.

PROOF. Suppose two proposals p and p' with ballot numbers b and b' and values v and v' are chosen. Since ballot numbers are unique, if $b = b'$ then $p = p'$ and $v = v'$. Suppose $b < b'$. For value v' to be chosen, p' must be proposed.

2.3.2 Lemma

If a proposal p_0 with ballot number b_0 and value v_0 is chosen, then all higher numbered proposals must contain v_0.

PROOF OF LEMMA. Let $p_1 = (b_1, v_1)$ be the proposal with the lowest ballot number such that $b_0 < b_1$. Let A_0 be a majority of acceptors who accepted p_0 and let A_1 be a majority of acceptors who sent prepare responses to p_1. There must be an acceptor a who is in both sets, who must have *first* accepted p_0 and *later* replied to the prepare request with b_1. In the prepare reply a indicated that it accepted p_0, and since p_1 is the lowest numbered proposal after p_0 the proposer could not have recieved a reply containing a higher numbered proposal, it must have used the value v_0 from proposal p_0, thus $v_0 = v_1$. Repeating this argument for the next proposal p_2, if a majority of nodes sent prepare replys, at least one must have contained p_0 or p_1, thus p_2 must contain the value v_0. Continuing *ad infinitum* proves the lemma.

3 PaxosLease with perfect clocks

3.1 Acquiring leases

Augmenting the basic flow with the following rules we get $PaxosLease_{acquire}$ (in the perfectly synchronized clock case):

1. There is a global maximal lease time T known to all nodes.
2. Upon restart, nodes do not rejoin the network for T time.
3. Proposers use increasing ballot numbers for their proposals, until they restart: for proposals issued within time T, proposals issued at a later time carry higher ballot numbers.
4. Since proposers may reuse ballot numbers across restarts, in order to distuingish messages (eg. prepare responses) from previous proposals, all messages are tagged with a t_{tag} time so that this timestamp and the ballot number together are unique. Using the expiry time t_e of leases as a tag time is a safe choice, since ballot numbers are not reused if the proposer does not restart, and if it does, the expiry time t_e will always be different than in a previous proposal. Acceptors, when replying to messages, simply copy the timetag of the request into the response. Proposers, when collecting responses, discard ones where this timetag doesn't match the current proposal's.
5. Proposers can only propose themselves as lease owners.

6. Proposers abandon proposals after T seconds: if a proposer sends propose requests at time t' with value "$acquire, k, t_e$", it sent out all the appropriate prepare requests in the time interval (t, t') where $t' - T < t$ and $t_e < t + T$.

7. Messages concerning expired leases are discarded: if an acceptor, proposer or learner receives a message with value "$acquire, k, t_e$" and $t_e < t$ then the message is automatically discarded.

8. Acceptors use a timer and discard their state: before sending a propose response for a value "$acquire, k,_e$" the node resets its timer to expire at t_e.

9. If an acceptor's timer expires at t_e, it discards any accepted proposals by setting *accepted proposal* := *empty proposal*.

Note that nodes do not have to store their state on stable storage, hence the title "diskless Paxos"!

3.1.1 Consistency

$PaxosLease_{acquire}$ guarantees that if value "$acquire, k, t_e$" is chosen at time t, then no other value is chosen until time t_e.

PROOF. Suppose two proposals p and p' with ballot numbers b and b' and values $v =$ "$acquire, k, t_e$" and $v' =$ "$acquire, k', t'_e$" are chosen at times t and t' such that $t < t' < t_e < t + T$ and $t < t' < t'_e < t' + T$.

Let A be a majority of acceptors who accepted p at time t and let A' be a majority of acceptors who accepted p' at time t'. There must be an acceptor a who is in both sets and since $t < t' < t + T$ node a did not restart in the meantime. Acceptors, within time T accept proposals with increasing ballot numbers, thus $b \leq b'$.

3.1.2 Lemma

If a proposal p_0 with ballot number b_0 and value $v_0 =$ "$acquire, k, t_e$" is chosen at time t, then all higher numbered proposals chosen at time t', such that $t < t' < t_e$, must contain v_0.

PROOF OF LEMMA. Proposals accepted at time t' must have been issued within the time interval $(t' - T, t')$. Let $p_1 = (b_1, v_1)$ be the proposal issued within time interval $(t' - T, t')$ with the lowest ballot number such that $b_0 < b_1$. Let A_0 be a majority of acceptors who accepted p_0 and let A_1 be a majority of acceptors who sent prepare responses to p_1. There must be an acceptor a who is in both sets and could not have restarted, thus a must have *first* accepted p_0 and *later* replied to the prepare request with b_1. In the prepare reply a indicated that it accepted p_0, and since p_1 is the lowest numbered proposal after p_0 and issued in the time interval $(t' - T, t')$ the proposer could not have recieved a reply containing a higher numbered proposal that would be accepted at time t' by acceptors, it must have used the value v_0 from proposal p_0, thus $v_0 = v_1$. Repeating this argument for the next proposal, as in the previous lemma, and continuing *ad infinitum* proves the lemma.

3.2 Extending leases

$PaxosLease_{acquire,extend}$: The above algorithm can be extended to allow proposers to *extend their leases*. Only the proposer's rule is changed: if a proposer k receives prepare responses from a majority of acceptors and the highest numbered reply contains the value "$acquire, k, t_e$" such that $t < t_e$ (its lease has not yet expired) then it is free to propose a different value "$acquire, k, t'_e$" such that $t < t_e < t'_e < t + T$.

3.2.1 Consistency

$PaxosLease_{acquire,extend}$ guarantees that if value "$acquire, k, t_e$" is chosen at time t and value "$acquire, k', t'_e$" is chosen at time t' such that $t < t' < t_e$ then $k = k'$.

PROOF. Suppose two proposals p and p' with ballot numbers b and b' and values $v =$ "$acquire, k, t_e$" and $v' =$ "$acquire, k', t'_e$" are chosen at times t and t' such that $t < t' < t_e < t + T$ and $t < t' < t'_e < t' + T$.

The first part of the proof showing the $b \leq b'$ is the same as in the previous lemma.

3.2.2 Lemma

If a proposal p_0 with ballot number b_0 and value $v_0 =$ "$acquire, k, t_e$" is chosen at time t, then all higher numbered proposals chosen at time t', such that $t < t' < t_e$, must contain values "$acquire, k', t'_e$" such that $k = k'$.

PROOF OF LEMMA. Proposals accepted at time t' must have been issued within the time interval $(t' - T, t')$. Let $p_1 = (b_1, v_1)$ be the proposal issued within time interval $(t' - T, t')$ with the lowest ballot number such that $b_0 < b_1$. Let A_0 be a majority of acceptors who accepted p_0 and let A_1 be a majority of acceptors who sent prepare responses to p_1. There must be an acceptor a who is in both sets and could not have restarted, thus a must have *first* accepted p_0 and *later* replied to the prepare request with b_1. In the prepare reply a indicated that it accepted p_0, and since p_1 is the lowest numbered proposal after p_0 and issued in the time interval $(t' - T, t')$ the proposer could not have recieved a reply containing a higher numbered proposal that would be accepted at time t' by acceptors, it must have used the value v_0 from proposal p_0, *or if the proposer is the kth, issued a different proposal containing itself.* Repeating this argument for the next proposal, as in the previous lemma, and continuing *ad infinitum* proves the lemma.

3.3 Releasing leases

In the formulation of PaxosLease (and Paxos) so far, a value was chosen if a majority of acceptors accepted a proposal. In practice, nodes must be able to *learn* that a value was chosen. In classical Paxos, a proposer learns that a value was chosen when it receives *propose responses* for a proposal p from a majority of acceptors. Then it can choose to inform learners by sending them *learn* messages.

In the full formulation of PaxosLease, $PaxosLease_{acquire,extend,release}$, we change the definition of chosen value: a proposal p containing value v is chosen if the proposer receives propose response messages from a majority of acceptors. Note that this definition is stronger then the previous, so all proofs presented so far remain valid.

Now we can complete PaxosLease by introducing special "$release, k, t_r$" values. Both the proposer's and acceptor's rules are amended.

1. An acceptor, receiving a proposal p with ballot number b (such that *highest ballot promised* <= b) and value "$release, k, t_r$", if currently holding accepted value "$acquire, k, t_e$" where $t_r < t_e < t$ must change t_e to t_r in its accepted value and reset its timer to t_r.

2. Proposer k, at time t learning that its value "$acquire, k, t_e$" was chosen, where $t < t_e < t+T$, may either extend its lease as described above or release its lease at time t' by issuing a proposal with value "$release, k, t_r$" where $t < t' < t_r < t_e < t + T$.

3.3.1 Consistency

$PaxosLease_{acquire,extend,release}$ guarantees that if value "$acquire, k, t_e$" is chosen at time t and value "$acquire, k', t'_e$" is chosen at time t' such that $t < t' < t_e$ and proposer k did not issue proposals containing *release* messages in the meantime, then $k = k'$.

PROOF. The proof is the same as above.

3.4 Learning

The Paxos notion of learners cannot be supported in PaxosLease *if proposers can release leases*. To see why, suppose proposer k acquires the lease until time t_e, informs a learner, and later releases the lease at time $t_r < t_e$. If the learner is cut off from the rest of the nodes, it will not get notified of the release, and incorrectly assume k is holding it until t_e, even though another proposer may have acquired it after t_r.

If the learner is a client who is *issuing commands* to the proposer to acquire and release leases, it will always have safe information regarding the lease. However, in this case the client can be regarded as a process on the proposer, so this distinction is irrelevant for PaxosLease.

4 PaxosLease with skewed clocks

So far, we have assumed that all nodes know some global time t. In practice this is not true, so we assume the kth node has a local clock $c_k(t)$ with a maximal skew S such that

1. $|c_k(t) - c_j(t)| < S$ for all times t and nodes k, j

2. the global maximal lease time T is chosen such that $S < T$.

Changes are required to handle clock skew:

1. When restarting, proposers and acceptors must wait for $T + S$ time

2. If a proposer k wants to acquire a lease until local time t_e. It must take into account that acceptors' clocks may be ahead of its own by S time. It does this by proposing the value "$acquire, k, t'_e$", where $c_k(t) + S < t'_e = t_e + S < c_k(t) + T$. This ensures that an acceptor, even if its clock is ahead of k's by S time, does not delete the accepted value before k's local time reaches t_e.

3. After a proposer has acquired a lease until time t_e, it may notify learners, but it must take into account that learners clocks' may be behind the acceptor's clock by S time. If value "$acquire, k, t'_e$" ($t'_e = t_e + S$) is accepted by a majority of acceptors,

proposer k sends learn messages containing time t_e. This ensures that a learner, even if its clock is behind the acceptors' by S time, does not falsely believe that k owns the lease after acceptor's local time reaches t'_e and they discard their state.

4. If proposer k wishes to release a lease at local time t_r that it holds until local time t_e (the previously accepted value is "$acquire, k, t_e + S$"), it must issue proposal "$release, k, t'_r$" where $c_k(t) + S < t'_r = t_r + S < t_e$. This ensures that an acceptor, even if its clock is ahead of k's by S time, does not delete the accepted value before k's local time reaches t_e.

5. Proposals must send "$release...$" values to learners before proposing the value. If it wishes to release the resource at local time t_r, it sends the value "$release, k, t_r$" to the proposers. Since it will propose the value "$release, k, t'_r = t_r + S$" to the acceptors, this ensures that learner's do not falsely believe k holds the lease after acceptor's local time reashes t'_r and they discard their state.

5 Leases for many resources

The algorithm defines lease actions concerning a single resource R. In practice, nodes wish to deal with several resources, such as write locks for files in a distributed filesystem. The solution is to run several independent instances of PaxosLease in parallel, each with its own proposer, acceptor and learner states. To identify leases in messages, prefix each messages with a *resources identifier*.

A node acting as proposer, acceptor and learner (a common setup in distributed systems) requires 50-100 bytes of memory for each PaxosLease instance, or 10-20 million resource leases per gigabyte of main memory.