【译】PaxosLease：实现租约的无盘Paxos算法

原文名： PaxosLease: Diskless Paxos for Leases

翻译： Jerry Lee oldratlee<at>gmail<dot>com

Marton Trencseni, mtrencseni@scalien.com
Attila Gazso, agazso@scalien.com

这篇论文描述了PaxosLease算法，一种用于租约协商的分布式算法。PaxosLease基于Paxos算法，但无需写盘和时钟同步。PaxosLease在开源的分布式复制KV存储Keyspace中被用来做Master租约协商。

1. 介绍

在并发编程中，*锁* 是进程用来同步共享资源访问的基本原语。在锁以不设置过期时间的方式分配（也没有一个监督进程）的系统中，锁的持有者在释放锁之前如果失效（Failure），就可能导致其它进程阻塞。

在高可用系统中，期望避免单点失效导致整个系统阻塞的情况。另外，“重启”失效的系统会比重启一个多线程程序要更困难。因此，在分布式系统中，*租约* 取代锁以避免饿死的情况。*租约* 就是 *有过期时间的锁* 。如果锁的持有者失效了或是和其它结点断开连接，它的租约会自动过期，其它结点可以得到租约。

我们假设基本的步骤如下：系统由一组请求者和一组接受者组成，请求者和接受者都有各自的算法；系统没有拜占庭问题，即结点之间不会通过不遵守各自算法作弊（也没有被Hack）。接受者的数目是固定不变的。

一个朴素的多数派投票式的算法可以正确地解决分布式租约的问题；这里 *正确* 的意思是，任何时候租约不会被多于一个结点持有。但是，这个简单的算法在有多个请求者时会频繁 *阻塞* ，因此需要一个更成熟的方案。

朴素的多数派算法是这样的： 请求者启动一个开始本地超时计时，超时时间T秒，然后向接受者发送请求时长为T的租约。接受者收到请求后启动一个时长为T秒的定时器，然后发送接受消息给请求者。超时之后，接受者清除自己的状态。如果接受者收到一个请求但他的状态不是空，则接受者不回应或是发一个拒绝消息。为确保任何时间只有一个请求者能获得租约，请求者必须收到多数派的接受者的接受消息；这样它获取租约直到它本地的定时器超时。

正如上面讨论的，有多个请求者时，有可能（而且很有可能）没有请求者能得到多数派，请求者会一直互相阻塞着。举个例子，有3个请求者1、2、3和三个接受者A、B、C，如果分布状态是这样的：A接受1的请求，B接受2的和C接受3的，然后没有一个请求者得到多数派的接受。系统必须等到超时过期，接受者清空自己的状态，这时请求者会再重试。但很很可能会再次阻塞。

在本文描述的解决方法是采取Paxos [1] 方案，引入 *准备* 和 *提议* 阶段，这样可以完全避免这类阻塞问题 [*] 。Paxos解决复制状态机的问题，每个结点有一个本地的状态机拷贝，希望在下一个状态转换时结点间达成一致。Paxos是一个基于多数派的算法，意味着，多数派的结点没有宕并且之间可以通信，是可能的。Paxos达成一致的处理在一个状态转换上，所以在实践中，需要逐次运行多个Paxos实例来协商出一序列的状态转换 [3] 。Paxos中，接受者在发送响应之前要先把自己的状态记录到盘上，以保证一旦一个值（状态转换）被选定，之后一直选定该值；换句话说，不管是否有出错情况出现，所有的状态机经历相同的状态转换序列。

不像之前的那些基于Paxos的分布式租约算法，比如Fatlease [5] ，PaxosLease不对结点的本地时钟做任务时间同步的假设（也不需要全局的同步）。另外，Fatlease为了租约命令连续地运行Paxos实例，而PaxosLease利用租约的临时性完全避免了这样的复杂性，是一个更简单和优雅的算法。

PaxosLease是Paxos的一个自然特化变种。因为在Paxos中假设结点个数是固定的（并且结点标识是全局已知的）。PaxosLease处理一个特殊的复制状态机，形式是：

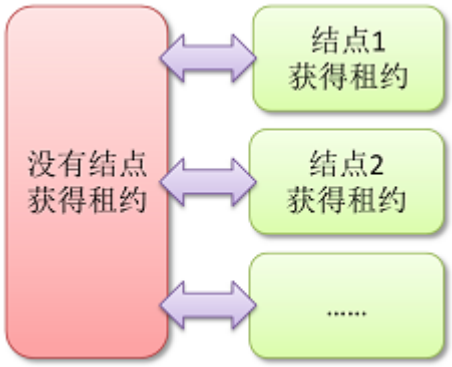

图1：PaxosLease的分布式状态机

为了获得租约，PaxosLease的请求者结点提交的值是“结点i持有租约”，在租约过期后将会自动返回“没有结点持有租约”。请求者也可以延长租约通过在前一次租约过期之前再次提交“结点i持有租约”值，或者在过期之前释放租约（可选操作）。

类似于Paxos，PaxosLease本质上处理了所有有关的失效情况：

1. 结点停止和重启
2. 网络分割不通
3. 消息丢失和乱序
4. 传输中的消息延时

2. 定义

一个PaxosLease单元由请求者和接受者组成。我们假设有 n 个接受者和任意个的请求者。在实践中，结点常常会同时扮演请求者和接受者的角色，但这是个实现上的问题不会影响这里的讨论。

请求者发送 *准备请求* （Prepare Request）和 *提议请求* （Propose Request）消息给接受者；接受者回应的是 *准备响应* （Prepare Response）和 *提议响应* （Propose Response）消息。这些消息有下面的结构：

1. 准备请求 = 投票编号
2. 提议请求 = 投票编号，响应结果，已经接受了的提案
3. 准备响应 = 投票编号，租约
4. 提议响应 = 投票编号，响应结果

投票编号和租约两者组成 *提案* （Proposal）。 *租约* 由 *请求者id* （希望成为租约持有者的结点）和 时间间隔 T 组成。

接受者存储下面的状态信息：

1. 承诺的最高编号： 接受者忽略投票编号小于该值的消息
2. 已经接受的提案： 最后一个接受的提案（投票编号 和 租约）

有一个全局已知的最大租约时间 M 。请求者请求的租约时间间隔 T 总是 $< M$ 。

每个请求者的投票编号是全局唯一的并且单调增加。在实践中，实现的方式可以是，投票编号由 *请求者id* 字段 ，一个 *重启计数器* 和 一个 请求次数的 *计数器* 字段 组成（可以处理最坏的情况）。每次请求者启动时重启计数器递增，并写到可靠的存储中。

PaxosLease保证了 *租约不变式* ：在任何给定的时间点，不会有多余1个请求者持有租约。

3. 基本算法

这一节描述分别从请求者和接受者出发的算法基本流程。请求者发送准备和提议请求，接受者回应准备和提议响应。如果一切正常和话，请求者获得租约花费两轮通信的时间。

1. 一个请求者想要获得租约，时长 $T < M$ 。它生成投票编号 [request.ballotNumber]，然后发送准备请求给多数派的接受者。

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

2. 接受者，当收到准备请求时，检查 [request.ballotNumber] 是否高于自己在 [state.highestPromised] 里承诺的本地投票编号中的最大值。如果提议请求的投票编号更低则可以丢弃这个消息，或者发送一个响应结果是 *拒绝* 的准备响应。如果相等或者更高，接受者用 *接受* 的回答构造一个准备响应，回答中有当前已接受的提案 [state.acceptedProposal] ，提案可以为空。接受者设置已承诺的最高投票编号 [state.highestPromised] 为 请求消息的投票编号 [request.ballotNumber] ，然后把这个准备响应发回给请求者。

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

3. 请求者检查从接受者过来的准备响应。如果有多数派的接受者响应的是空的提案，意味着他们可以接受新的提案，请求者可以提交它自己作为租约的获得者，时长是 T 。请求者启动一个定时器，过期时间是 T 秒，发送提议请求，其中包含了投票编号 和 租约（它自己的 *请求者id* 和 T ）。

```
Proposer::OnPrepareResponse()
{
    if (response.ballotNumber != state.ballotNumber)
        return // some other proposal
    if (response.acceptedProposal == 'empty')
    numOpen++
    if (numOpen < majority)
        return
    state.timeout = T
    SetTimeout(state.timeout)
```

```
    request.type = ProposeRequest
    request.ballotNumber = state.ballotNumber
    request.proposal.proposerID = self.proposerID
    request.proposal.timeout = state.timeout
    Broadcast(request)
}
Proposer::OnTimeout()
{
    state.ballotNumber = empty // set in Proposer::Propose()
    state.leaseOwner = false // set in Proposer::OnProposeResponse()
}
```

4. 接受者，当收到提议请求时，检查投票编号 [request.ballotNumber] 是否高于自己在 [state.highestPromised] 里承诺的本地投票编号中的最大值。如果提议请求的投票编号更低则可以丢弃这个消息，或者发送一个响应结果是 *拒绝* 的提议响应。如果相等或者更高，接受者接受这个提议：启动过期时间T的超时计时，设置它已接受的提案为这个收到的提案（如果还存着前一个提案，丢弃掉）。接受者用 *接受* 的回答构造一个提议响应，回答中有投票编号 [request.ballotNumber] 。在超时过期后，接受者重置它已接受的提案为 *空* 。接受者决不重置它的已承诺的最高投票编号，除非在重启的时候。

```
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. 请求者检查提议响应消息。如果有多数派的接受者响应了接受提案，则这个请求者获得了租约直到本地的定时器超时（在第3步中启动）。它收到多数派消息的最后一条的时间点就是它获得租约的时间点，可以切换它的内部状态到“我持有租约”。

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

可以看到，接受者没有把自己的状态存到存储上。重启时，请求者以空白状态启动。为了保证重启中结点不会破坏租约不变式，结点要在重新加入网络前等待 *M* 秒。 *M* 是一个全局已知最大租约时间，所有的结点都知道，请求者请求的租约时长 *T* 总是 *< M秒* 。

传递都是时间间隔（相对时间），这一点很重要，导致只有获取了租约的请求者才知道自己有租约。该请求者不能告诉其它结点它获取了租约（与经典Paxos的学习消息类似），因为其它结点不能知道学习消息在传输的过程要消耗多少时间。因此，只有获得了租约的请求者知道它自己持有租约。所有其它的结点知道的是自己没有获得租约。换句话说，每个请求者关于租约有两种状态：“我没有租约，我也不知道谁持有租约” 和 “我持有租约”。当然，结果可以发出学习消息作为 *hint* ，这可以用在高级应用中或是用来探索，但这些使用方式超出本论文范围。

有可能一个请求者在第3步和第5步中没有得到多数派接受者赞同响应。这种情况下，请求者可以休眠一会儿再重新从第1步用更高的投票编号执行算法。

4. 租约不变式证明

我们先给出为什么PaxosLease可以工作的直觉感受。图2是以画图方式的解释：请求者在发送提议请求之前开启定时器，接受者只能 *一段时间后* 开启他们的定时器；接受者在发送提议响应之前开启定时器的。因此，如果有多数派的接受者存下了状态并开启定时器，在请求者定时器过期前，将没有其它的请求者可以得到租约。将没有2个请求者同时认为自己是租约的持有者。

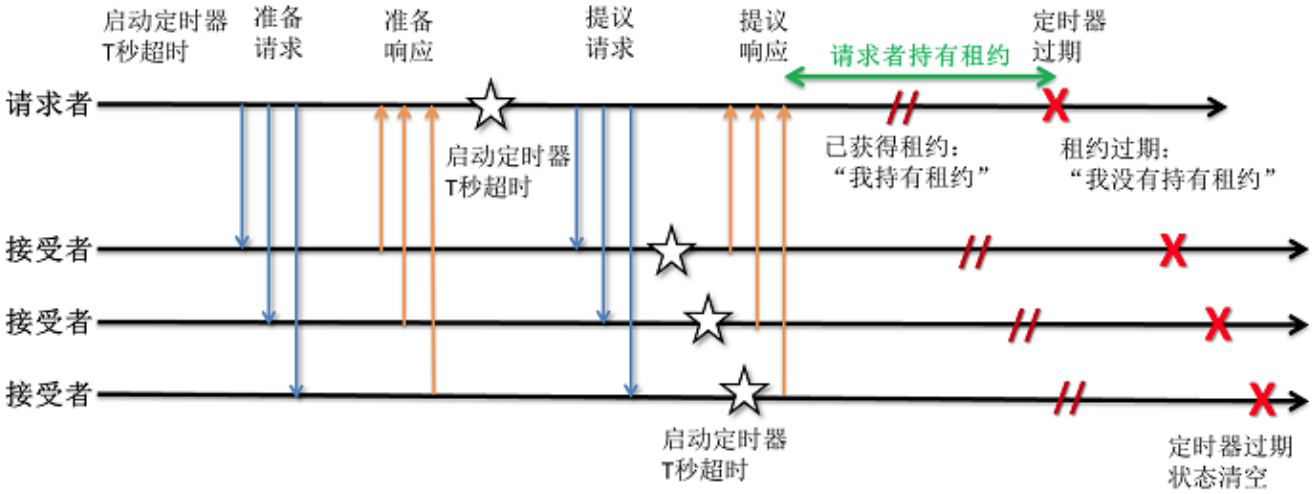

图2：一个请求者获得租约的时间流程图

更正式地说，PaxosLease保证了如果请求者 i 发出的投票编号是 b 和 时长是 T 的提案 从多数派的接受者那里接收到了接受消息，假定请求者在时间点 t_{now} 启动定时器，那么没有其它请求者能再接收到多数派的接受消息，直到 $t_{end} = t_{start} + T$ 。

证明：假定请求者 p 用投票编号 b 获得了租约。它从多数派的接受者那里收到了类型是 *接受* 的空准备响应，在时间点 t_{start} 启动定时器，在时间点 $t_{acquire}$ 从多数派的接受者那里收到了类型是 *接受* 的提议响应，这样请求者持有租约直到 $t_{end} = t_{start} + T$。令 A_1 为用空准备响应回应 p 的准备请求的接受者多数派，令 A_2 为接受 p 提案 并且 发送类型是 *接受* 的准备响应的接受者多数派。

第一部分： 在 $t_{acquire}$ 到 t_{end} 的时间内，没有其它的请求者 q 能以 $b' < b$ 的投票编号的请求来获得租约。为了持有租约，请求者 q 必须得到多数派接受者 A'_2 的接受。 令 a 为同时在 A'_2 和 A_1 的接受者。因为 $b' < b$ ， a 必须是先接受了 q 的提案然后发送准备响应给 p 的。但是如果 a 发送一个空准备响应给 p 它的状态必须为空，它的定时器必须已经过期了，即 q 的定时器过期了，因此 q 已经失去了租约。在 p 和 q 的租约之间没有重叠。

第二部分：在 $t_{acquire}$ 到 t_{end} 的时间内，没有其它的请求者 q 能以 $b < b'$ 的投票编号的请求来获得租约。为了持有租约，请求者 q 必须得到多数派接受者 A'_1 给它发送空个准备响应。 令 a 为同时在 A'_1 和 A_2 的接受者。因为 $b < b'$ ， a 必须是先接受了 p 的提案然后发送准备响应给 q 的。但是既然 a 接受了 p 的提案，如果它发送一个空个准备响应给 q 它的状态必须是空的，它的定时器必须已经过期了，即 p 的定时器过期了，因此 p 已经失去了租约。在 p 和 q 的租约之间没有重叠。

5. 活性（Liveness）

Paxos类型的算法比如PaxosLease，有动态死锁的可能：两个请求者可能连续地生成越来越高的投票编号，发送准备请求给接受者，接受者连续地增加自己承诺的最高投票编号，结果没有请求者可以让接受者接受提案。在实践中，可以通过让请求者在重新执行算法前等待一小段随机的时间来规避。

Paxos类型的算法一个主要的优点是没有静态死锁，在朴素的投票算法中有说到。没有静态死锁是因为请求者可以覆盖接受者的状态，算法又保证了多数派是不会被覆盖的。

6. 延长租约

在某些情况下，一旦一个请求者持有资源后可以持续持有而不是一个原来的租给时间，这一点很重要。一个典型的场景是，在分布式系统中当租约指出Master结点后，期望这个结点可以长时间作为Master。

为了适应这个需求，只要请求者的算法需要修改。要第3步中，如果多数派响应了空的提案或是*已存在提案*（即这个提案中的该请求者的租约还没有过期），它可以再次提议自己为租约的持有者。这样允许请求者延长它的租约 $O(T)$ 的时间。接受者的算法无需修改。

7. 释放租约

到现在的算法描述中，请求者的租约是在一定时间后自动过期的。在一些情况下，尽快释放租约让其它的结点获取是很重要的。一个典型例子是分布式处理，处理进程获得一个资源的租约，执行其上的操作，然后期望尽快释放租约好让其它处理进行获得。

为了适应这个需求，请求者可以发送一个特定释放消息给接受者，消息中包含了它要释放租给的投票编号。在发送释放消息之前，请求者把内部状态从“我持有租约”切换到“我没有持有租约”。当接受者收到释放租约时，查检是否与已接受的投票编号相同。如相同则清空自己的状态；否则不做任何操作。请求者也可以发送一个释放消息给其它请求者作为提示，告诉他们可以去获取租约了。

8. 多个资源的租约

算法定义了关于一个资源 R 的租约动作。在实践中，结点会要处理多个资源，比如一个分布式处理中要用的租约。PaxosLease可以为各个资源运行独立的实例，不同的实例的消息、请求者和接受者状态标志上 *资源标识* 。一个结点作为请求者和接受者，每个PaxosLease实例消耗内存不超过 ~100字节，这样结点上1G内存可以处理 ~1千万个资源租约。再加上PaxosLease不需要硬盘同步和时钟同步，该算法可以用在很多需要细粒度锁的场景上。

9. 实现

在Scalien的分布式复制key-value存储Keyspace中 [†]译注，PaxosLease用于Master的租约协商。Keyspace作为PaxosLease的参考实现，包含了很多实践上的优化。由于基于开源AGPL许可[6]，感兴趣的读者可以自由获取Keyspace实现。源代码和二进制文件可以在http://scalien.com [‡]译注 下载。

10. 宗谱

Leslie Lamport在1990年发明Paxos算法，但在1998才发表的。这篇论文《The Part-Time Parliament》对于很多读者过于极客，这导致第二篇论文《Paxos Made Simple》 [2] 。Paxos通过引入个准备和提议两个阶段和让接受者在响应消息前把自己状态写入稳定存储，解决了发布式一致性问题。多轮的Paxos可以顺序运行以协调复制状态机的状态转换。

在论文《Paxos Made Live - An Engineering Perspective》和《The Chubby Lock Service for Loosely-Coupled Distributed Systems》 [4] 中描述的Google内部的分布式实现栈用了Paxos，这让Paxos流行起来。在Google的Chubby中，多轮顺序执行Paxos以达到，在复制数据库中下次写操作上的一致性，提供了思考复制状态机的另一种方法。

《FaTLease: Scalable Fault-Tolerant Lease Negotiation with Paxos》中描述的Fatlease解决了和PaxosLease一样的问题，但它结构更复杂，因为模仿了在Google论文中提到的多轮Paxos，而不是PaxosLease所用的简单的接受者状态超时。另外，FaTLease需要结点同步他们的时钟，这一点使的它在现实世界使用中没有吸引力。PaxosLease灵感来自于FaTLease，解决了上述的缺点。

参考文献

[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), 18-25.

[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] AGPL License. http://www.fsf.org/licensing/licenses/agpl-3.0.html

注释

[*] 另一个解决方法是，让系统阻塞，但是引入一个“撤销”机制，让请求者撤销他的请求从而让某个其它的请求者可以获得租约。
[†] 译注，scalien的GitHub代码工程在 https://github.com/scalien
[‡] 译注，这个网站已经没有内容了，Keyspace源代码可以在 https://github.com/scalien/keyspace 下载。

评论|建议

发布

3 条评论 38 条新浪微博 2 条腾讯微博 最新 最早 最热

OFSBH

万 .部 A 片高.清 国产.日韩 hTTp://uVU.cc/iqVn

17小时前 回复 顶 转发

wEajh

这个 更C刺j激，A 片：htTP://uVU.Cc/iqVr

2016年10月11日 回复 顶 转发

zITgc

全都到碗里来 ！美臀/丝袜/美熟女乱伦精品大合集 ！！！【 v.ht/xZiU 】

2016年9月25日 回复 顶 转发

dsdoc.net正在使用多说