Chubby: 为松散耦合的分布式系统提供锁服务

摘要

我们将介绍我们关于Chubby锁服务的一些经验，这是一个为松散耦合式的分布式系统提供的粗粒度锁服务，就像可靠的存储服务一样（尽管容量小）。Chubby提供了一个与分布式文件系统的协同锁很相似的接口，但是设计重点在于可用性和可靠性，而不是高性能。许多个服务实例已经使用了超过一年的时间，这些服务中的大部分都有着几万个客户同时在线的并发量。这篇论文将介绍最初的设计以及预期的用途，并对比实际用途，解释为了适应这些差异，设计做出了哪些修改。

1 介绍

这篇论文介绍一个叫做Chubby的锁服务，旨在用于由高速网络连接中等规模的小型机器组成的松散耦合分布式系统。例如，一个Chubby实例（也被称为Chubby单元）可能用于由1Gbit/s的以太网连接起来的一万台四核机器上。大多数Chubby单元会被使用在单数据中心或者机房里，尽管我们至少会部署一个在数千公里外有副本的Chubby单元。

锁服务的目的在于允许多个客户端同步它们的行为，并且在环境的基本信息方面达成一致。对于一个中等规模的客户端集合来说，其主要目标包括可靠性、可用性，以及具有容易理解的语意。吞吐量和存储空间则是次要考虑的目标。Chubby的客户端接口与一个简单的文件系统很相似，这种文件系统会执行整个文件的读写，并且使用协同锁做为加强，同时会对各种类型的事件提供通知机制，例如文件修改。

我们希望Chubby帮助开发者处理系统内的粗粒度同步，特别是能够解决在一组服务器中选出领导者的问题。例如，GFS(谷歌文件系统)使用Chubby锁来指派一个主服务，而Bigtable则在几个方面使用Chubby：1.选举一个主结点，2.允许主结点发现其控制的子结点，允许客户端寻找他们的主结点。另外，GFS和Bigtable都使用Chubby作为一个众所周知并且可用的位置来存储少量的元数据。实际上它们都是将Chubby做为它们分布式数据结构的根。有些服务则使用锁在几个服务之间区分工作（粗粒度）。

在Chubby上线之前，谷歌的许多分布式系统使用 `ad hoc` 做为主要的选举方式（当工作重复做不会导致不良影响时），或者需要操作人员的介入（需要保证正确性时）。对于前者，Chubby允许了其能保存少量的计算结果。对于后者，Chubby在系统的可用性上实现了重大的提升，不再需要人类在系统出错时手动介入。

如果读者熟悉分布式计算的话，会意识到在几个主要结点间做选举是针对分布式一致性问题的一个例子，并且知道我们需要一个使用异步通信的方案；这个术语描述了绝大多数真实网络的行为，例如以太网和因特网，它们都允许数据包被丢失、延迟、重新排序（从业人员通常需要提防将协议建立在那些对环境做了高度假设的基础之上）。异步一致性问题使用Paxos[12,13]协议作为解决方案。Paxos协议也被用在Oki和Liskov项目上（可以阅读他们关于viewstamped replication的论文[19, §4]）。实际上，到目前为止，我们所有可行的异步一致性协议都使用Paxos作为它们的核心。Paxos并不依赖时间假设来保证安全性，但是需要引入时钟来确保活性；这做到Fischer的研究结果中认为不可能的事[5, §1]。

构建Chubby需要工程上的努力才能完成上面的目标；这其实并不是一项研究工作，我们并不会介绍新的算法或者技术。这篇论文的主要目的是介绍我们做了什么、为什么要这样做，而不是去提倡它。在接下来的章节，我们介绍Chubby的设计和实现，以及它是如何根据经验进行调整的。我们介绍一些使用Chubby的意想不到的方式，以及被证明是错误的特性。我们在文献的其他地方忽略了一些细节，比如一致性算法或者RPC系统。

2 设计

2.1 合理性

有人可能会认为我们应当将Paxos算法封装成一个库，而不是通过调用某个库去访问一个中心化的锁服务，即便它非常可靠。一个客户端的Paxos库不会依赖于其他的服务（除了名称服务），假设程序员的服务能够以状态机的形式实现，那么也能够为程序员提供一个标准的框架。实际上，我们就提供了这样的一个独立于Chubby的客户端库。

对比客户端库，锁服务有几个优势。第一，我们的开发者在最开始的时候并不会考虑高可用。多数情况下他们的系统都是以从较低负载以及较低的可用性保证开始开发的；代码中并不会有专门的数据结构用来实现一致性协议。随着服务的成熟以及用户的增长，可用性变得越来越重要；副本和初级的选举机制会被添加到已有的设计当中。这种情况下可以使用提供分布式一致性协议的库，加上使用锁服务就可以让维护现有的程序结构以及交互模式变得更加简单。例如，为了选举出一个用来写入已有文件的文件服务的主结点，只需要添加两个语句以及一个RPC参数到现有的系统中即可。获得锁的将成为主结点，通过在写RPC时添加一个额外的整数（对锁获得情况的计数），以及添加一个if语句以拒绝那些计数值低于当前实际值的写请求。比起让现有的服务实现一致性协议，特别是必须得保证兼容性的情况下，我们发现使用这种技术实现起来会更加容易。

第二，我们的许多服务会选择一个主要的服务，或者在他们的组件之间对数据做区分，这些都需要一个能够传播结果的机制。这提醒我们需要允许客户端能够存储数据或者取回少量的数据，即能够读写小文件。这可以通过名称服务器来完成，但我们的经验证明使用锁服务可能会更加适合，一方面是因为能够减少客户端所以来的服务数量，另一方面是因为是共享协议提供的一致性。Chubby在名称服务器上的成功很大一方面是其使用了一致性客户缓存，而不是基于时间的缓存。特别是我们发现开发者非常感激不用去选择缓存的超时时间，例如DNS缓存的生存周期，如果选择不佳的话可能会导致高的DNS负载或者长的客户端故障时间。

第三，对于我们的程序员来说，他们更熟悉基于锁的接口。Paxos的复制状态机以及与排他锁关联的临界区能够为程序员提供循序编程的感觉。许多程序员都遇到过锁，并且认为他们知道如何使用它。讽刺的是，这些程序员通常都是错的，特别是当他们在一个分布式系统中使用锁的时候。很少有人会考虑独立的某台机器发生故障时对异步通信系统中的锁的影响。尽管如此，对于锁的较高熟悉程度，还是能够说服程序员去克服在使用可靠的分布式锁服务中遇到的障碍。

最后，分布式一致性算法使用多数表决的方法来做决策，所以它们使用副本集来达到高可用。例如，Chubby最少需要3个才能保证正常工作，通常使用5个副本。作为对比，如果一个客户端系统使用锁服务，即便是单独的客户端也能获得锁来保证程序的安全性。所以，一个锁服务能够减少客户端所依赖的服务数。还有一点没有提到,可以将锁服务做为通用的选举服务提供给客户端系统，让其正常工作的成员数量少于大多数的时候能够做出正确的决策。在最后一个问题上，有个想象中可能存在的解决方案：使用若干服务作为Paxos协议中的"acceptors"，提供一致性服务。与锁服务相似，一个一致性服务能够让客户端的进程变得更加安全，即便只有一个；一种类似的技术也已经用于减少Byzantine容灾所需要的状态机数量。然而，假设一个一致性服务没有专门用于提供锁，那么这种解决方法将不能用于解决上面提到的问题。

这些论点提出了两个在决策设计中的建议：

- 我们提供了一个锁服务，而不是一个库或者是一致性服务。
- 我们选择服务小文件，允许被选出来的主去传播它们和它们的参数，而不是构造和维护另外一个服务。

下面的决定来自于我们预期的使用方法以及我们的环境：

- 一个通过Chubby传播其主的服务可能拥有几千个客户端。因此，我们必须允许几千个客户端观察该文件，并且最好不要使用太多服务器。
- 客户端和复制服务的副本们可能希望知道服务的主发生了变化。这说明提供一个事件通知机制比起长轮询会更加有用。
- 即便有些客户端不需要长时间轮询文件，还是有其他客户端需要；所以，文件缓存是必要的，这也是为了能够支持更多的开发者。
- 我们的开发者对不直观的缓存语义感到困惑，所以我们更倾向于提供一致性缓存。
- 为了避免经济的损失，我们提供了安全机制，包括访问控制。

有个选择可能会让一些读者感到惊讶，我们并不期望锁被用在只有几秒或者更少的细粒度上面。取而代之的是，我们希望锁是粗粒度的。例如，一个应用可能使用一个锁去选择一个主，这个主在接下来很长一段时间，有可能是几小时甚至是几天，会处理所有的访问请求。这两种使用方式会对锁服务器提出不同的要求。

粗粒度的锁对服务器施加的负载会小很多。特别是，请求锁的频率通常和客户端应用的交互频率只有微弱的关系。粗粒度的锁只能很少的人获取，所以锁服务器的暂时不可用对客户端造成的延迟影响也会减少。从另一个方面来说，在客户端之间传输锁的过程是昂贵的，所以我们不希望任何服务器的故障会导致锁的丢失。因此，对于粗粒度的锁来说在服务器故障中恢复是更加有优势的，这样做的开销也很小，并且在只牺牲部分可用性的情况下，这样的锁只需要中等规模的服务器就能服务数量众多的客户端。

细粒度的锁会得出不同的结论，甚至服务的短时间不可用就可能导致许多客户端产生很高的延迟。性能和服务器的横向扩展能力是非常值得注意的，因为锁服务的事务速率会随着客户端事务速率的增长而增长。这能够带来的好处是能够在锁服务故障时不维护锁，以减少锁的负载，并且频繁得丢失锁对时间的影响也不会太严重，因为锁的有效期很短。（客户端必须做好在网络分区时会丢失锁的准备，所以在锁服务发生故障时也不会导致产生出新的恢复路径。）

Chubby只提供粗粒度的锁。幸运的是，客户端也可以直接根据他们的应用实现细粒度锁。一个应用可以将它们的锁区分成组，并且使用Chubby的粗粒度锁将这些组分配到指定应用的服务上。维护这些细粒度的锁需要一点状态信息；服务器只需要保持一个非易失，单调递增并且很少更新的计数器。客户端可以在解锁时了解丢失的锁，如果使用了定长的租约信息，协议可以变得简单高效。使用这种方案的好处就是我们的开发者只需要提供能够支持它们的负载的服务器配置即可，而不用关心如何实现复杂的一致性。

2.2 系统结构

Chubby有两个通过RPC通信的重要组件，一个是服务端，以及一个能够让客户端程序互相连接的库；参见图1。所有的Chubby客户端和服务器的通讯都在客户端库处理。第三个代理服务器组件是可选的，一个代理服务，这个将在3.1节讨论。

一个Chubby单元包含一小组服务器（通常是5个）来做副本集，以便于减少失败的可能性（例如，在不同的机架上的服务器）。副本集采用分布式一致性协议来选择主节点；这个主节点必须是多数副本集选举出来的，并且保证那些副本集在主节点的任期内不能选出另外的主节点。这个主节点的租约定期通过副本集更新，以便让主继续赢得大多数选票。

副本会维护一个简单数据库的副本，但只有主会初始化该数据库的读写操作。所有其他副本都是简单从主那里复制更新，并使用一致性协议进行发送。

客户端通过发送主的定位请求到在DNS里的副本列表来寻找主。非主的副本会响应该请求主的位置。一旦一个客户端定位到主了，该客户端会将所有请求直接发送到主上，直到其无法响应客户端的请求或者告诉客户端它不是主了。写请求会通过一致性算法传播到所有副本上；这样的请求会在写入大多数副本后会认为已经被接受。读请求会通过主满足安全性；如果主的租约没有过期，那么就是安全的，因为没有其他主存在的可能。如果一个主宕机了，当其租约过期时，其他的副本会运行选举协议选出新的主。一个新的主一般会在几秒内被选出来。举个例子，两个最近的选举分别用了6s和4s，但是我们会看到数值其实高达30s(§4.1)。

如果副本故障并且没有在几小时之内恢复，一个简单的替换系统会从一个空闲的池中选择一个新的机器，然后在上面启动锁服务的二进制。接下来再更新DNS表，使用新机器的IP替换掉故障机器的IP。当前的主会定时轮询服务器，最终关注到变化后，会更新数据库中中Chubby单元的成员列表。这个列表会使用常规的复制协议在所有成员之间保持一致。与此同时，新的副本会从存储在文件系统的数据库备份中取得最近几次的拷贝，并且在活跃的的那些副本那里取得更新。一旦新的那个副本已经处理了一个请求，并且当前的主正在等待提交，那么这个副本就允许在选举中投票以选出新的主。

2.3 文件，目录，和句柄

Chubby对外导入一个与Unix很像但比Unix简单的文件系统接口。在通常情况下，它包含了一个严格的文件和目录树，并用斜杠分隔名字组件。一个典型的名字是：

> /ls/foo/wombat/pouch

`ls` 是所有Chubby的文件名字通用的前缀，并且代表了锁服务。第二个组件（foo）是Chubby单元的名字；它通过DNS查找解析到一个或多个Chubby服务器。一个特殊的单元名 `local` 表明客户端本地的Chubby单元应该被使用。该单元通常是在同一个构建环境中，因此是最有可能被访问到的一个。名字的剩余部分， `/wombat/pounch` ,在被命名的Chubby单元中进行解析。同样跟随Unix，每一个目录都包含一个文件列表和目录列表，当中的每个文件都包含了一串未被解释的字节。

由于Chubby的命名结构酷似一个文件系统，我们可以通过自己的专用API和其他文件系统使用的接口使应用程序可用，例如GFS。这能够显著减少写基础的浏览或维护命名空间工具的需要，同时也降低了Chubby用户的学习成本。

而设计中与Unix不同之处在于简化的分布式操作。为了能够让不同目录中的文件能够让不同的Chubby主服务到，我们并不对外提供把一个文件从一个目录移到另一个目录的操作，我们也不目录的修改时间，并且避免路径相关的权限语义（即文件的访问控制由文件自身决定，与其上层目录无关）。为了能够让缓冲元数据变得更加容易，系统也不会显示上一次的访问时间。

命名空间只包含文件和目录，统称为节点。每个这样的节点在Chubby单元中只有唯一一个名字；它们都不会有符号链接或者硬链接。

节点既可能是永久性的，也可能是临时的。任何节点都有可能被显式删除，但如果没有客户端打开它们，临时节点也会被删除（对于目录来说意味着其为空）。临时文件除了除了本身的作用外，还用来指示客户端是否还活着。任何节点都能做为读写的协同锁；这些锁将在2.4节做更多的介绍。

每个节点都有各种元数据，包括三个用于控制读取，写入和更改ACL名称的访问控制列表（ACL）的名称节点。除非被覆盖，否则节点会继承创建它的父目录的访问控制名。访问控制列表即那些存放在ACL目录下的文件，这些文件也是Chubby单元本地命名空间众所周知的一部分。这些ACL文件由简单的委托人的名单组成。读者可能会想起Plan 9的小组。因此，如果文件F写了其ACL名称为 `foo` ,ACL目录包含了文件 `foo` ，该文件中有一个叫做 `bar` 的条目，那么接下来用户 `bar` 将有权限对文件F做写入。用户的验证通过内建于RPC系统的机制完成。因为Chubby的ACLs只是简单的文件，所以对于其他想使用类似的访问控制机制的服务，也是可以直接使用的。

每一个节点的元数据包含4个单调递增的64位数字，用于让客户端更容易得检测到变化：

- 一个实例数字；该数字会大于先用拥有同样名字的节点
- 一个由内容生成的数字（只针对文件）；当文件被写的时候该数字会增加
- 一个由锁生成的数字；当节点的锁事务从可用到被保持时该数字会增加
- 一个由ACL生成的数字；当节点的ACL名字被写入时该数字会增加

Chubby同时会导出一个64位文件内容检验码，所以客户端它判断文件是否有变化

客户端会打开节点用以获取与Unix文件描述符类似的句柄。该句柄包括：

- 检查码，用于防止客户端创建或者探测句柄，所以当只有句柄被创建时，完全的访问控制才是需要的（和Unix在文件打开时检查权限位相比，Chubby不是在每一次读写操作时进行，因此文件描述符不可能被伪造。）
- 一个数字串，允许通过一个主表示一个句柄是否是由它生成的，或者是之前的主生成的。
- 在开放时间提供的模型信息允许一个主在旧的句柄无法在重启后的主上使用时，能重新创建它的状态。

2.4 锁和序列

每一个Chubby文件和目录都可以做为读写锁；即任意一个客户端句柄都能在独占（写）模式下持有锁，或者有多个数量的客户端句柄在将锁保持在共享（读）模式下。就跟大多数程序认识的互斥一样，这里的锁也属于协同锁。所以只有在其他人尝试获得同一个锁的情况下会导致冲突；持有一个称为F的锁即不是访问文件F所必须的，也不是为了防止其他的客户端访问。我们拒绝强制锁，这会让被锁住的对象无法被那些没有持有锁的客户端访问：

- Chubby锁通常是保护正在被其他服务使用中的资源，而不仅仅将锁和文件关联起来。如果想以有意义的方式强制使用强制锁，将会要求我们对那些服务做出更多的修改。
- 我们不希望强制用户为了调试程序或是管理而需要在访问文件时关闭应用。大多数人的计算机在一个复杂的系统内要使用这种方法访问是很困难的。管理软件只能简单得通过引导用户关闭或重启它的应用来关闭强制锁。
- 我们的开发者通过一种常规的方式做错误检查，即写类似 `lock X is held` 的断言，所以他们很难从强制锁中获益。Bug或者恶意进程有很多种方式在锁没有被持有的情况下破坏数据，所以我们发现额外的强制锁并无法提供重要的价值。

在Chubby中， 不管在哪个模式下进行写都需要获得一个锁，所以一个没有任何特权的读者是无法干扰写者的活动的。

分布式系统中的锁是复杂的，因为通信无法确定，并且进程可能会独立失败。所以，一个持有锁L的进程可能会发起一个请求R，但接下来进程退出。其他进程有可能在R请求到达其目的地之前获得了锁L并且执行了一些操作。如果R接下来到了，其有可能在没有锁L的保护下执行，并且会有潜在的数据不一样问题。消息被乱序接受这个问题已经能够很好得解决了；解决方案包括虚拟时间[11]、虚拟同步，通过确保按照与每个参与者的观察一致的顺序处理消息来避免该问题。

要将序列号引入现有复杂系统的所有交互中，其代价是高昂的。相反，Chubby提供一个方法，将序列号引入到那些只与锁有关的交互中。在任何时候，一个锁的持有者可能会向序列发生器请求一个不透明字符串，该字符串描述锁在被获得后的即时状态。其包含了锁的名称，被获得时的工作模式（排他或共享），还有一个锁生成的数字。客户端将序列发生器传送给服务端（如文件服务器），如果其希望它的操作能够被锁保护起来。接受的服务器被用于测试序列发生器是否可用并且是否处在合适的模式下；如果不是，其会拒绝请求。序号发生器的可用性能够通过服务端的Chubby缓存检查，如果服务端不愿意维护Chubby的会话，检查则可以通过服务器最近观察到的序列发生器进行。序列发生器的机制要求只有字符串的添加能够影响到消息，这很容易向我们的开发者解释。

尽管我们发现序列发生器很容易使用，但是重要的协议却发展缓慢。因此Chubby提供了一个不完美但是简单的机制来减少延迟的风险或者重新对不支持序列发生器的请求进行排序。如果一个客户端在正常方式下释放了锁，其会立即对其他客户端可用，就像人们所期望的那样。然而，如果一个锁是因为持有者发生错误或者无法访问被释放掉，锁服务会在一定时间内防止其他客户端占用该锁，这个时间称为锁的延迟。客户端可以在一定上限内指定任意一个锁的延迟，目前该上限是一分钟。这个限制可以防止有故障的客户端在一个任意长的时间内锁定一个资源。尽管不完美，但锁延迟每天都在保护因为信息延迟或者重启导致的问题中没能修改的服务和客户端，

2.5 事件

Chubby客户端在创建句柄时可能会订阅一系列事件。这些事件通过向上调用的方式异步得从Chubby库传送到客户端。事件包括：

- 文件内容修改 —— 通常用于通过文件来监控服务通知的位置
- 子结点的添加、移除、或修改 —— 通常用于实现镜像（$2.12）。（另外还用于允许新文件被发现，为子结点返回节点让监控那些短暂存在的文件而不影响他们的引用计数变得可能）。
- Chubby主节点出故障 —— 警告客户端其他事件可能已经被丢失，所以数据必须被重新扫描。
- 一个句柄（包括它的锁）已经变得不可用 —— 这通常表明这是一个通信问题。
- 获得锁 —— 能被用来检测一个主要节点什么时候能被选举出来
- 来自其他客户端的锁冲突请求 —— 允许做锁的缓存

事件是在相应的行为发生后传送的。因此，如果一个客户端被通知文件已经发生变化，其保证接下来的读操作能看到文件的新数据（或者是最近的数据）

最后提到的两个事件很少被使用，事后也可以被忽略掉。例如选举结果出来后，客户端通常需要和选出来的主进行通信，而不是简单得知道主已经存在。因此，他们等待一个文件的的变化事件以知道新的主已经将其地址写在了文件上。冲突锁理论上允许客户端缓存保存在其他服务局上的数据，并使用Chubby锁来维护缓存的一致性。冲突锁的请求通知会告诉客户端使用数据将锁关联起来；其将会完成排队中的操作、刷新修改到一个主页的位置、丢弃缓存数据、然后释放等行为。到目前为止，没有人适应这种风格的用法。

2.6 API

Chubby的句柄在客户端看来就像是一个不透明的结构体,该结构体支持多种操作。句柄只能使用 `Open()` 创建，并且使用 `Close()` 销毁。

`Open()` 打开一个已经命名的文件或者目录并返回一个与Unix文件描述符类似的句柄。只有这种调用需要用到节点名称；其他所有的操作都作用通过句柄。

节点的名称是通过已有的目录句柄评估的；库提供了一个基于 / 的句柄，该句柄永远可用。在包含了多层抽象的多线程程序中，目录句柄避开了在程序范围内使用当前目录的困难。

客户端指示多种操作：

- 句柄会怎样被使用（读；写或者锁住；更改ACL）;句柄只有在客户端拥有合适的权限时才能会创建。
- 应该被传递的事件（见$2.5）
- 锁的延迟
- 新的文件或者目录是否应该（或者是必须）被创建。如果一个文件被创建了，调用者可能会提供初始化的内容和初始化的ACL名称。返回值会指示文件是否真的被创建成功

`Close()` 关闭一个打开的句柄。接下来对该句柄是使用都是不被允许的。这个调用永远不会失败。如果有关闭一个句柄，调用 `Poison()` 会导致退出和接下来对该句柄操作的失败。这允许了一个客户端取消其他Chubby客户端发起的调用而不用担心回收的内存被他们重新访问。

对句柄的主要操作包含：

`GetContentsAndStat()` 返回所有的内容以及文件的元数据，文件的内容会以原子和完整条目的方式读取。我们避免了部分读取和写入来阻止大文件。一个相关的调用 `GetStat()` 则只返回元数据，同时 `ReadDir()` 返回一个目录下面的子文件的名称和元数据。

`SetContents()` 向一个文件写入内容。客户端可以选择提供一个基于内容生成的数字，以允许客户端对该文件模拟CAS操作。只有当生成的数字是当前的，内容才会变化。文件的内容总是以原子和完整条目的形式写入。一个相关的调用 `SetACL()` 在与ACL名称相关联的节点上执行类似的操作。

`Delete()` 删除那些没有孩子的节点

`Acquire()`，`TryAcquire()`，`Release()` 获取和释放锁

`GetSequencer()` 返回一个描述该句柄持有的任意锁的序列生成器（$2.4）

`SetSequencer()` 将一个序列生成器与句柄关联起来，如果该序列生成器不可用则接下来的操作将会失败

`CheckSequencer()` 检查一个序列生成器是否可用（见$2.4）

如果句柄在创建后节点被删除了，那么调用将会失败，尽管文件在接下来已经被重新创建。一个句柄是与一个文件的实例相关联的，而不是文件的名称。Chubby可能在任何调用中使用访问控制检查，但总是会检查 `Open()` 调用（见$2.3）

除了调用本身所需的其他调用之外，上面的所有呼叫都需要一个操作参数。操作参数持有与其他调用相关联的数据和控制信息。特别是，通过操作参数客户端可以：

- 支持回调以实现调用的异步
- 等待该调用的完成
- 获取额外的错误和诊断信息

客户端能够使用这些API去执行如下主的选举：所有潜在的主打开一个锁文件并尝试获得锁。其中一个成功并且成为了主，同时其他做副本。主将会使用 `SetContents ()` 把它的标识写入锁文件中，所以通过 `GetContentsAndStat` 读文件，或者是文件改动事件的响应，其能够被客户端和副本发现。在理想情况下，主会通过 `GetSequencer` 获取一个序列生成器，接下来将其传递到与主通信的服务器上。它们都需与 `CheckSequencer()` 确认其是否依旧为主。锁的延迟可以用于无法检查序列生成器的服务（$2.4）。

2.7 缓存

为了减少读的流量，Chubby的客户端会缓存文件数据以及节点的元数据（包括文件的缺失）在一块具有一致性、使用直写模式（write-through）的内存中。该缓存由下面描述的租约机制维护，并由记录有每个客户端可能缓存的列表的主发送出来的无效信号保持一致性。协议确保客户端能够互相看到一个一致的Chubby状态视图，或者是一个错误。

当文件的数据或者元数据被改变时，修改会被阻塞，直到主发送数据的无效信号到每一个可能缓存该数据的客户端上。这种机制建立在KeepAlive的RPCs机制上，下一节会更详细讨论改机制。在收到无效信号后，客户端会刷新无效的状态并且通过下一次的KeepAlive调用做确认。修改只会有在服务器知道每个客户端都将缓存置为无效后才会被处理，要么是因为客户端确认了缓存的失效，要么是客户端确认其缓存的数据租约到期。

当缓存无效保存，保持在未确认的状态时，主会将其所在的节点标记为不可达，因此只需要一轮无效即可。这种处理方式能够让读总是能够被无延时得处理。这是很有效果的因为读的数量会远大于写。另外一个替代方案将会是一种延时的方案，其会在节点不可用期间访问节点。这种方案会降低缓存穿透的可能性，尽管会导致偶然的延时。如果选择这种方案存在问题，可以设想采用混合方案，在检测到负载过大的时候切换方案。

缓存协议很简单，它在数据变动时将缓存数据置为非可用，并且从不更新它。更新而不置为非可用也是一种简单的方案，但是这样在某种程度上会影响到效率；一个访问文件的客户端可能要即时收到文件的更新，但也会导致收到很多不必要的更新通知。

2.8 会话与心跳

Chubby的会话是指一个Chubby客户端与一个Chubby单元之间的联系；它会存在一段时间，并且通过周期性的握手即KeepAliveb来维护。除非Chubby客户端通知主，否则只要会话是有效的，那么其相关的客户端句柄，锁和缓存数据就全都依旧有效（然而维护会话的协议可能需要客户端知道缓一个存的失效，具体看下面论述）

客户端请求一个新的会话时会先与Chubby单位的主节点通讯.其在终止或者会话已经被闲置时（即没有打开的句柄并且在一分钟内没有调用）也会明确得结束会话。

每一个会话关联一个持续到以后的一段时间的租约，在此期间主节点保证不会单方面结束会话。这段时间结束后即session租约到期。主节点可以让到期时间延后，但绝不能提前。

主节点可以因为以下三个因素让租期时间延后：会话创建时，当主节点发生故障(见下文),从客户端响应了一个心跳.接受到了心跳，主节点一般会阻止RPC(不让它返回)直到客户端先前的租期即将到期。这是主节点才会返回RPC给客户端,并且通知客户端租期到期.主节点可以延长任意长度的超时时间.默认延长12秒，但是对与超时加载主节点会设置长的时间以便于减少处理心跳的代码的调用数量.在接受之前的返回后，客户端会马上开始发送新的心跳.因此，客户端确定总有一个心跳呼叫在主节点被阻塞.

除此之外，KeepAlive回复被用作发送事件和缓存无效给客户端。当一个事件或者无效(invalidation)被发送过去，主节点允许一个KeepAlive提早返回.在KeepAlive回复捎带事件是确保客户端在没有清楚缓存无效时无法维护session并且导致Chubby RPCs从客户端流向主节点.这让客户端的工作降低复杂度，并且在启动连接时允许协议通过防火墙进行单方面操作，

客户端维护的本地租约到期时间是根据主节点租约到期时间的保守估计.它跟主节点的租约到期时间不同因为客户端必须保守估计他的KeepAlive应答过程中耗费的时间和主节点的时钟 is advancing的速度；为了保证一致，我们需要服务器的cpu时钟速度不会比客户端的速度快一个已知的常数

（主节点发生故障，不回复心跳，让客户端租约到期）如果客户端的本地租约到期时间到了，它不管主节点是否终止了会话，客户端会清空并且禁用缓存，这时候我们就称会话is in jeopardy。客户端会等待默认为45s的宽限期。如果在宽限期结束前重新恢复了心跳，客户端就重新启用缓存。否则客户端假设这个会话过期。这样就完成了以便于当Chubby单元不可访问时，不会让Chubby的API无限阻塞；并且会回调错误函数当宽限期结束还没有建立连接时

在宽限期开始计数时可以使用Chubby的库发送jeopardy事件来通知客户端。当在宽限期结束前重新建立了连接时可以用Chubby库发送safe事件来告知客户端继续。如果会话过期，那就发送通知过期的事件。当不清楚会话状态时（主节点挂掉，不知是否继续使用session），这些事件可以让客户端应用自己停止会话，并且如果它知道问题可以很快被解决可以自行恢复（This information allows the application to quiesce itself when it is unsure of the status of its session, and to recover without restarting if the problem proves to be transient.） 这些重要的举措能避免高负荷运转的服务挂掉.

如果客户端拥有节点的句柄H并且在这个H上的操作（除了Close()和Poison()）都失败了，那是因为会话过期了。与其丢失任意序列（ arbitrary subsequence），不如让客户端用这个来判断（guarantee ）网络和服务器停止运转引起的只有 a suffix of a sequence of operations 丢失，从而在最后写入提交时能够标记到复杂的变化。