**用于松耦合分布式系统的锁服务Chubby**

**摘要**

We describe our experiences with the Chubby lock service, which is intended to provide coarse-grained locking as well as reliable (though low-volume) storage for a loosely-coupled distributed system. Chubby provides an interface much like a distributed file system with advisory locks, but the design emphasis is on availability and reliability, as opposed to high performance. Many instances of the service have been used for over a year, with several of them each handling a few tens of thousands of clients concurrently. The paper describes the initial design and expected use, compares it with actual use, and explains how the design had to be modified to accommodate the differences.

我们描述了我们在Chubby锁服务方面的经历。Chubby的目标是为松散耦合的分布式系统，提供粗粒度的锁定、可靠的存储（尽管容量不大）。Chubby提供了一个很像带有意向锁的分布式文件系统的接口（*interface*)，不过它的**设计重点是可用性和可靠性，而不是高性能**。许多Chubby服务实例已经使用了超过一年，其中的几个实例每个都同时处理着数万台客户端。本文描述了最初的设计和预期使用，将之与实际的使用相比较，并解释了设计是怎样修改以适应这些差别的。

**1. 简介**

This paper describes a *lock service* called Chubby. It is intended for use within a loosely-coupled distributed system consisting of moderately large numbers of small machines connected by a high-speed network. For example, a Chubby instance (also known as a Chubby *cell*) might serve ten thousand 4-processor machines connected by 1Gbit/s Ethernet. Most Chubby cells are confined to a single data centre or machine room, though we do run at least one Chubby cell whose replicas are separated by thousands of kilometres.

本文介绍了一种锁服务：Chubby。它被计划用在由适度规模的大量小型计算机通过高速网络互联构成的的松散耦合的分布式系统中。例如，一个Chubby实例（也叫Chubby单元(*Cell*))可能服务于通过1Gbit/s网络互联的一万台4核计算机。大部分Chubby单元限于一个数据中心或者机房，不过我们在相隔数千公里之外运行着至少一个Chubby单元副本(replica)。

The purpose of the lock service is to allow its clients to synchronize their activities and to agree on basic information about their environment. The primary goals included reliability, availability to a moderately large set of clients, and easy-to-understand semantics; throughput and storage capacity were considered secondary. Chubby’s client interface is similar to that of a simple file system that performs ***whole-file***reads and writes, augmented with advisory locks and with **notification of various events** such as file modification.

锁服务的目的是允许它的客户应用们(clients)同步它们的活动，并对它们所处环境的基本信息取得一致。主要的目标包括在适度大规模的客户应用群时的可靠性、可用性，以及易于理解的语义；吞吐量和存储容量被认为是次要的。Chubby的客户端接口很像这样一个简单的文件系统：能执行整文件的读和写，另外还有意向锁和和多种诸如文件改动等事件的通知。

We expected Chubby to help developers deal with coarse-grained synchronization within their systems, and in particular to deal with the problem of electing a leader from among a set of otherwise equivalent servers. For example, the Google File System [7] uses a Chubby lock to appoint a GFS master server, and Bigtable [3] uses Chubby in several ways: to elect a master, to allow the master to discover the servers it controls, and to permit clients to find the master. In addition, both GFS and Bigtable use Chubby as a well-known and available location to store a small amount of meta-data; in effect they use Chubby as the root of their distributed data structures. Some services use locks to partition work (at a coarse grain) between several servers.

我们预期Chubby帮助开发者处理他们的系统中的粗粒度同步，特别是处理从一组各方面相当的服务器中选举领导者。例如Google文件系统（Google File System）[***[7]***](javascript:void())使用一个Chubby锁来选择GFS Master 服务器，Bigtable[***[3]***](javascript:void())以数种方式使用Chbbuy：选择领导者；使得Master可以发现它控制的服务器；使得客户应用(client)可以找到Master。此外，GFS和Bigtable都用Chubby作为众所周知的、可访问的存储小部分元数据(*meta-data*)的位置；实际上它们用Chubby作为它们的分布式数据结构的根。一些服务使用锁在多个服务器间（在粗粒度级别上）分发工作。

Before Chubby was deployed, most distributed systems at Google used ***ad hoc***methods for primary election (when work could be **duplicated** without harm), or required operator intervention (when correctness was essential). In the former case, Chubby allowed a small saving in computing effort. In the latter case, it achieved a significant improvement in availability in systems that no longer required human intervention on failure.

在Chubby发布之前，Google的大部分分布式系统使用*ad hoc*方法做主从选举(*primary election*)（在工作可能被重复但无害时），或者需要人工干预（在正确性至关重要时）。对于前一种情况，Chubby可以减少计算量。对于后一种情况，它使得系统在失败时不再需要人工干预，显著改进了可用性。

Readers familiar with distributed computing will recognize the election of a primary among peers as an instance of the *distributed consensus* problem, and realize we require a solution using ***asynchronous*** *(*异步的*)*communication; this term describes the behaviour of the vast majority of real networks, such as Ethernet or the Internet, which allow packets to be lost, delayed, and reordered. (Practitioners should normally beware of protocols based on models that make stronger assumptions on the environment.) Asynchronous consensus is solved by the ***Paxos***protocol [12, 13]. The same protocol was used by Oki and Liskov (see their paper on *viewstamped replication* [19, §4]), an equivalence noted by others [14, §6]. Indeed, all working protocols for asynchronous consensus we have so far encountered have Paxos at their core. Paxos maintains safety without timing assumptions, but clocks must be introduced to ensure liveness; this overcomes the impossibility result of Fischer *et al.* [5, §1].

熟悉分布式计算的读者会意识到在多个对等体件选举**primay**是分布式一致性问题的一个特例，同时意识到我们需要一种异步通信的解决方案。异步这个术语描述了绝大多数真实网络如以太网或因特网的行为：它们容许数据包丢失、延时和重排序。（开发者一般应该了解协议集建立在对环境做了很强假设的模型之上。） 异步一致性由Paxos协议***[[12, 13]](javascript:void()" \t "_blank)***解决了。同样的协议被Oki和Liskov使用，其他人使用了等价的协议***[[14, $6]](javascript:void()" \t "_blank)***。实际上，迄今为止我们遇到的所有可用的异步协同协议的核心都有Paxos。Paxos不需要计时假设来维持安全性，但必须引入时钟来确保活跃度。这克服了Fisher等人的不可能性结果。

Building Chubby was an engineering effort required to fill the needs mentioned above; it was not research. We claim no new algorithms or techniques. The purpose of this paper is to describe what we did and why, rather than to advocate it. In the sections that follow, we describe Chubby’s design and implementation, and how it has changed in the light of experience. We describe unexpected ways in which Chubby has been used, and features that proved to be mistakes. We omit details that are covered elsewhere in the literature, such as the details of a consensus protocol or an RPC system.

构建Chubby是一种满足上文提到的各种需求的工程上的工作，不是学术研究。我们声明没有提出新的算法和技术。本文的目的在于描述我们做了什么以及为什么这么做，而不是提出这些。在接下来的章节中，我们描述Chubby的设计和实现，以及在实际过程中它是怎么改变的。我们描述了预料之外的Chubby的使用方式，以及被证明是错误的特性。我们忽略了在其他文献中已经包括的细节，例如一致性协议或者RPC系统。

**2. 设计**

**2.1 基础依据(*Rationale*)**

One might argue that we should have built a library embodying Paxos, rather than a library that accesses a centralized lock service, even a highly reliable one. A client Paxos library would depend on *no* other servers (besides the name service), and would provide a standard framework for programmers, assuming their services can be implemented as state machines. Indeed, we provide such a client library that is independent of Chubby.

有人可能会争论说我们应该构建一个包含Paxos的库，而不是一个访问中心化锁服务的库，即使是一个高可靠的锁服务。客户端Paxos库将不依赖于其他服务器（除了名字服务以外），并且假定他们的服务可以实现为状态机,将为开发者提供标准化的框架。事实上，我们提供了一个这样的与Chubby无关的客户端库。

Nevertheless, a lock service has some advantages over a client library. First, our developers sometimes do not plan for high availability in the way one would wish. Often their systems start as prototypes with little load and loose availability guarantees; invariably the code has not been specially structured for use with a consensus protocol. As the service matures and gains clients, availability becomes more important; replication and primary election are then added to an existing design. While this could be done with a library that provides distributed consensus, a lock server makes it easier to maintain existing program structure and communication patterns. For example, to elect a master which then writes to an existing file server requires adding just two statements and one RPC parameter to an existing system: One would acquire a lock to become master, pass an additional integer (the lock acquisition count) with the write RPC, and add an if-statement to the file server to reject the write if the acquisition count is lower than the current value (to guard against delayed packets). We have found this technique easier than making existing servers participate in a consensus protocol, and especially so if compatibility must be maintained during a transition period.

然而，一个锁服务具有一些客户端库不具有的优点。第一，有时开发者并没有如人们希望的那样，为高可用性做好规划。通常他们的系统从只有很小负载和宽松的可用性保证的原型开始，代码总是没有特别地构造为使用一致性协议。当服务成熟，得到了用户，可用性变得更重要了，复制和选举随后被加入到已有的设计中。 尽管这能通过提供分布式协同的库来搞定，但锁服务更易于保持已经存在的程序结构和通信模式。例如，选择一个master，然后向一个已经存在的文件服务器写入数据，只需要加两条语句和一个RPC参数到已经存在的系统中即可：一条语句请求一个锁以成为master，另外传递一个整数（锁请求计数）给写RPC，再加入一条if语句，如果请求计数小于当前的值（用于防止延时的包），则文件服务器拒绝写入。我们发现这种技术比将已有的服务器加入一致性协议更容易，尤其是在迁移期间必须维持兼容性时。

Second, many of our services that elect a primary or that partition data between their components need a mechanism for advertising the results. This suggests that we should allow clients to store and fetch small quantities of data—that is, to read and write small files. This could be done with a name service, but our experience has been that the lock service itself is well-suited for this task, both because this reduces the number of servers on which a client depends, and because the consistency features of the protocol are shared. Chubby’s success as a name server owes much to its use of **consistent client caching,** rather than time-based caching. In particular, we found that developers greatly appreciated not having to choose a cache timeout such as the DNS time-to-live value, which if chosen poorly can lead to high DNS load, or long client fail-over times.

第二，我们的许多服务在选举parimary，或在它们的各个组件间划分数据时，需要一种公布结果的机制。这意味着我们应该允许客户端存储和取得小量的数据--也就是读写小文件。这能通过名字服务来完成，但是我们的经验是锁服务自身相当适合做这件事，既因为这样减少了客户端要依赖的服务器数，也因为协议的一致性特性是相同的。Chubby的作为一个名字服务器的成功，应很大程度上应归功于**一致的客户端缓存**，而不是基于时间的缓存。特别地，我们发现开发者相当地喜欢不需要选择一个像DNS生存时间值一样的缓存超时值，这个值如果选择不当会导致很高的DNS负载或者较长的客户端故障修复时间。

Third, a lock-based interface is more familiar to our programmers. Both the replicated state machine of Paxos and the critical sections associated with exclusive locks can provide the programmer with the illusion of sequential programming. However, many programmers have come across locks before, and think they know to use them. Ironically, such programmers are usually wrong, especially when they use locks in a distributed system; few consider the effects of independent machine failures on locks in a system with asynchronous communications. Nevertheless, the apparent familiarity of locks overcomes a hurdle in persuading programmers to use a reliable mechanism for distributed decision making.

第三，基于锁的接口更为程序员所熟悉。复制的Paxos状态机和与排他锁关联的临界区都能为程序员提供顺序编程的幻觉。可是，许多程序员已经用过锁了，并且认为他们知道怎么使用它们。颇为讽刺的是，这样的程序员经常是错的，尤其是当他们在分布式系统里使用锁时。很少人考虑单个独立机器的失败对一个异步通信的系统中的锁的影响。不管怎样，对锁的表面上的熟悉性，战胜了试图说服程序员们为分布式决策使用（其他）可靠机制的努力。

Last, distributed-consensus algorithms use quorums to make decisions, so they use several replicas to achieve high availability. For example, Chubby itself usually has five replicas in each cell, of which three must be running for the cell to be up. In contrast, if a client system uses a lock service, even a single client can obtain a lock and make progress safely. Thus, a lock service reduces the number of servers needed for a reliable client system to make progress. In a loose sense, one can view the lock service as a way of providing a generic electorate that allows a client system to make decisions correctly when less than a majority of its own members are up. One might imagine solving this last problem in a different way: by providing a “consensus service”, using a number of servers to provide the “acceptors” in the Paxos protocol. Like a lock service, a consensus service would allow clients to make progress safely even with only one active client process; a similar technique has been used to reduce the number of state machines needed for Byzantine fault tolerance [24]. However, assuming a consensus service is not used exclusively to provide locks (which reduces it to a lock service), this approach solves none of the other problems described above.

最后，分布式一致性算法使用仲裁来做出决策，因此他们使用多个副本来实现高可用性。例如，Chubby本身通常在每个实例中有五个副本，其中三个必须运行才能使实例运行起来。相反，如果客户端系统使用锁定服务，即使是单个客户端也可以获得锁并安全地进行。因此，锁服务能减少可靠的客户端系统运行所需要服务器数目。在更宽泛的意义上，人们能够将锁服务视作提供通用选举的一种方式，允许一个客户系统在少于其多数的成员存活时正确地决策。可以想象以不同的方式解决这个最后的问题：通过提供“一致性服务”，使用多个服务器来提供Paxos协议中的“接受者”。 与锁定服务一样，即使只有一个活动的客户端进程，一致服务也可以让客户安全地运行下去; 类似的技术已用于减少拜占庭容错所需的机器数量[24]。 但是，假设一致性服务不是**唯一**使用的锁服务（将其减少为锁服务），则该方法不会解决上述任何其他问题。

These arguments suggest two key design decisions:

* We chose a lock service, as opposed to a library or service for consensus, and
* we chose to serve small-files to permit elected primaries to advertise themselves and their parameters, rather than build and maintain a second service.

上面这些讨论建议了我们的两个关键的设计决策：

* 我们选择一个锁服务，而不是一个库或者一致性服务，以及
* 我们选择提供小文件，以使得被选出来的primaries可以公布它们自身以及它们的参数，而不是创建和维护另外一个服务。

Some decisions follow from our expected use and from our environment:

* A service advertising its primary via a Chubby file may have thousands of clients. Therefore, we must allow thousands of clients to observe this file, preferably without needing many servers.
* Clients and replicas of a replicated service may wish to know when the service’s primary changes. This suggests that an event notification mechanism would be useful to avoid polling.
* Even if clients need not poll files periodically, many will; this is a consequence of supporting many developers. Thus, caching of files is desirable.
* Our developers are confused by non-intuitive caching semantics, so we prefer consistent caching.
* To avoid both financial loss and jail time, we provide security mechanisms, including access control.

某些设计决策则来自于我们预期的用途和我们的环境：

* 一个通过Chubby文件来公布其Primary的服务，可能拥有数千的客户端。因此，我们必须允许数千的客户端监视这个文件，并且最好不需要太多服务器。
* 客户端和有多个副本的服务的各个副本要知道什么时候服务的primary发生了变化。这意味着一种事件通知机制将很有用，以避免轮询。
* 即使客户端不需要间歇性地轮询文件，很多客户端还是会这样做；这是支持许多开发者的结果。因此，缓存这些文件是很可取的。
* 我们的开发者对不直观的缓存语义感到困扰，所以我们倾向于一致的缓存。
* 为了避免金钱上的损失与牢狱之灾(*jail time*)，我们提供了包括访问控制在内的安全机制。

A choice that may surprise some readers is that we do not expect lock use to be *fine-grained*, in which they might be held only for a short duration (seconds or less); instead, we expect *coarse-grained* use. For example, an application might use a lock to elect a primary, which would then handle all access to that data for a considerable time, perhaps hours or days. These two styles of use suggest different requirements from a lock server.

一个可能让一些读者吃惊的选择是，我们不希望锁被细粒度地使用，这种情况下这些锁可能只被持有一段很短的时间（数秒或更短）；实际上，我们希望粗粒度地使用。例如，一个应用程序可能使用锁来选举一个primary，然后这个primary在一段相当长的时间可能是数小时或数天里，处理所有对数据的访问。这两种使用方式意味着对锁服务器的不同的需求。

Coarse-grained locks impose far less load on the lock server. In particular, the lock-acquisition rate is usually only weakly related to the transaction rate of the client applications. Coarse-grained locks are acquired only rarely, so temporary lock server unavailability delays clients less. On the other hand, the transfer of a lock from client to client may require costly recovery procedures, so one would not wish a fail-over of a lock server to cause locks to be lost. Thus, it is good for coarse-grained locks to survive lock server failures, there is little concern about the overhead of doing so, and such locks allow many clients to be adequately served by a modest number of lock servers with somewhat lower availability.

粗粒度的锁在锁服务器引入的负载要小得多。特别是锁的获取频率通常只会与客户端应用系统的事务频率有很微弱的关联。粗粒度的锁不常被请求，所以临时性的锁服务器不可用给客户端的造成的延时会更少。在另一方面，一个锁从一个客户端到另一个客户端可能需要很大开销的恢复处理，所有人们不希望锁服务器的故障恢复造成锁的丢失。因此，粗粒度的锁能够经历锁服务器的失败而继续有效是很有用的，这里不太在意这样做的开销，并且这样的锁使得许多客户端可由少数可用性稍低的锁服务器服务得足够好。

Fine-grained locks lead to different conclusions. Even brief unavailability of the lock server may cause many clients to stall. Performance and the ability to add new servers at will are of great concern because the transaction rate at the lock service grows with the combined transaction rate of clients. It can be advantageous to reduce the overhead of locking by not maintaining locks across lock server failure, and the time penalty for dropping locks every so often is not severe because locks are held for short periods. (Clients must be prepared to lose locks during network partitions, so the loss of locks on lock server fail-over introduces no new recovery paths.)

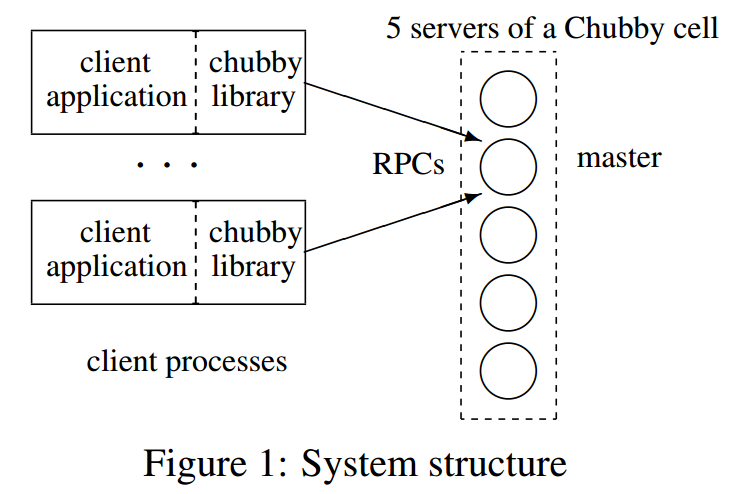
细粒度的锁会有不同的结论。即使锁服务的短暂的不可用也可能导致许多客户端挂起。因为锁服务上的事务频率将随着所有客户端的事务频率之和一起增长，性能和随意增加新服务器的能力十分重要。不需要在锁服务失败期间维持锁可以减少锁定的开销，这是优势；频繁地丢弃锁也不是一个很严重的问题，因为锁只被持有一段很短的时间。（客户端必须准备好在网络分离期间丢失锁，因此锁服务器故障恢复造成的锁的丢失不会引入新的恢复路径。）

Chubby is intended to provide only coarse-grained locking. Fortunately, it is straightforward for clients to implement their own fine-grained locks tailored to their application. An application might partition its locks into groups and use Chubby’s coarse-grained locks to allocate these lock groups to application-specific lock servers. Little state is needed to maintain these fine-grain locks; the servers need only keep a non-volatile, monotonically increasing acquisition counter that is rarely updated. Clients can learn of lost locks at unlock time, and if a simple fixed-length lease is used, the protocol can be simple and efficient. The most important benefits of this scheme are that our client developers become responsible for the provisioning of the servers needed to support their load, yet are relieved of the complexity of implementing consensus themselves.

Chubby被计划为只提供粗粒度的锁定。幸运的是，对于客户端而言，实现根据其自身的应用系统定制的细粒度锁是很简单的。一个应用程序可能将它的锁划分成组，并使用Chubby的粗粒度锁将这些锁分组分配给应用程序特定的锁服务器。维护这些细粒度的锁只需要很少的状态，这些服务器只需要持有一个不常变的、单调递增的请求计数器，这个请求计数器不会经常被更新。客户端能够在解锁时发现丢失了的锁，并且如果使用了一个简单定长的租期(lease)，其协议将会简单高效。该方案最重要的好处是我们的客户端开发人员负责配置支持其负载所需的服务器，也将他们从自己实现一致的复杂性中解放出来。

**2.2 系统结构**

Chubby has two main components that communicate via RPC: a server, and a library that client applications link against; see Figure 1. All communication between Chubby clients and the servers is mediated by the client library. An optional third component, a proxy server, is discussed in Section 3.1.



Chubby有两个主要的通过RPC通信的组件：一个服务器和一个客户端应用链接的库，如图一所示。所有Chubby客户端和服务器之间的通信由客户端库居间达成。还有一个可选的第三个组件，代理服务器，将在第3.1节讨论。

A Chubby cell consists of a small set of servers (typically five) known as *replicas*, placed so as to reduce the likelihood of correlated failure (for example, in different racks). The replicas use a distributed consensus protocol to elect a master; the master must obtain votes from a majority of the replicas, plus promises that those replicas will not elect a different master for an interval of a few seconds known as the *master lease*. The master lease is periodically renewed by the replicas provided the master continues to win a majority of the vote.

一个Chubby单元由一组称作副本集的服务器（典型的是五个）组成，按降低关联失败的可能性来放置（例如分别放在不同的机架）。这些副本使用分布式一致的协议来选举一个Master，Master必须从副本集得到多数投票，并得到副本集在一个持续数秒的被称为master租期(lease)的时间段内不再选举另一个不同的Master的承诺。只要Master继续赢得大多数投票，这个master租期就会周期性地被副本集刷新。

The replicas maintain copies of a simple database, but only the master initiates reads and writes of this database. All other replicas simply copy updates from the master, sent using the consensus protocol.

副本集维护着一个简单数据库的备份集，但是只有master发起对数据库的读写。其他的所有副本简单地从master复制使用一致性协议传送的更新。

Clients find the master by sending master location requests to the replicas listed in the DNS. Non-master replicas respond to such requests by returning the identity of the master. Once a client has located the master, the client directs all requests to it either until it ceases to respond, or until it indicates that it is no longer the master. Write requests are propagated via the consensus protocol to all replicas; such requests are acknowledged when the write has reached a majority of the replicas in the cell. Read requests are satisfied by the master alone; this is safe provided the master lease has not expired, as no other master can possibly exist. If a master fails, the other replicas run the election protocol when their master leases expire; a new master will typically be elected in a few seconds. For example, two recent elections took 6s and 4s, but we see values as high as 30s (§4.1).

客户端们通过发送master位置请求给列在DNS中的副本集来查找master。非master的副本返回master的标识来回应这些请求。一旦一个客户端定位到master，它将所有请求指引到该master，直到该master停止回应或者指明自己不再是master。写请求被通过一致性协议传播给所有副本，这些请求在写入达到Chubby单元中的多数副本时被确认。读请求仅有master响应，这样是安全的只要master租期没有过期，因为没有别的master可能存在。如果一个master失败了，其他的副本在他们的master租期到期时运行选举协议，典型地，一个新的master将在几秒之内选举出来。例如，最近两次的选举花费了6秒和4秒，但是我们也见过高达30秒的。

If a replica fails and does not recover for a few hours, a simple replacement system selects a fresh machine from a free pool and starts the lock server binary on it. It then updates the DNS tables, replacing the IP address of the failed replica with that of the new one. The current master polls the DNS periodically and eventually notices the change. It then updates the list of the cell’s members in the cell’s database; this list is kept consistent across all the members via the normal replication protocol. In the meantime, the new replica obtains a recent copy of the database from a combination of backups stored on file servers and updates from active replicas. Once the new replica has processed a request that the current master is waiting to commit, the replica is permitted to vote in the elections for new master.

如果一个副本失败了并且在几个小时内没有恢复，一个简单的替换系统从一个空闲池选择一台新的机器，并在其上启动锁服务器的二进制文件(*binary*)。然后它将更形DNS表，将失败了的副本的IP替换为新启动的启动的IP。当前的master周期性地轮询DNS并最终注意到这个变化，然后它在该Chubby单元的数据库中更新单元成员列表，这个列表在所有成员之间通过常规的复制协议保持一致。与此同时，新的副本取得存储在文件服务器上的数据库备份和活跃副本的更新的组合。一旦新副本处理了一个当前master在等待提交的请求，新的副本就被许可在新master的选举中投票。

**2.3 文件，目录和句柄**

Chubby exports a file system interface similar to, but simpler than that of UNIX [22]. It consists of a strict tree of files and directories in the usual way, with name components separated by slashes. A typical name is:

/ls/foo/wombat/pouch

Chubby开放出一个类似于UNIX的文件系统[***[22]***](javascript:void())的接口，但比Unix的文件系统更简单。它由一个严格的文件和目录树组成，名字的各部分使用反斜杠划分。一个典型的名字如下：

/ls/foo/wombat/pouch

The ls prefix is common to all Chubby names, and stands for *lock service*. The second component (foo) is the name of a Chubby cell; it is resolved to one or more Chubby servers via DNS lookup. A special cell name local indicates that the client’s local Chubby cell should be used; this is usually one in the same building and thus the one most likely to be accessible. The remainder of the name, /wombat/pouch, is interpreted within the named Chubby cell. Again following UNIX, each directory contains a list of child files and directories, while each file contains a sequence of uninterpreted bytes.

ls (*lock service*)前缀对所有Chubby名称都是通用的，代表锁定服务。第二个部分(foo)是Chubby单元的名字，通过DNS查询被解析到一个或多个Chubby服务器。一个特殊的单元名字local，表明应该使用客户端的本地Chubby单元，这个Chubby单元通常在同一栋楼里，因此这个单元最有可能能访问到。名字的剩余部分，/wombat/pouch，由Chubby单元内部解析。同样跟UNIX一样，每个目录包含一个所有子文件和子目录的列表，而每个文件包含一串不解析的字节。

Because Chubby’s naming structure resembles a file system, we were able to make it available to applications both with its own specialized API, and via interfaces used by our other file systems, such as the Google File System. This significantly reduced the effort needed to write basic browsing and name space manipulation tools, and reduced the need to educate casual Chubby users.

因为Chubby的命名结构组成了一个文件系统，我们既可以通过它的专门API将它开放给应用系统，也可以通过我们的其他文件系统例如GFS使用的接口。这样显著地减少了编写基本浏览和名字空间操作的工具的工作，也减少了培训那些偶然用Chubby的用户的需求。

The design differs from UNIX in a ways that ease distribution. To allow the files in different directories to be served from different Chubby masters, we do not expose operations that can move files from one directory to another, we do not maintain directory modified times, and we avoid path-dependent permission semantics (that is, access to a file is controlled by the permissions on the file itself rather than on directories on the path leading to the file). To make it easier to cache file meta-data, the system does not reveal last-access times.

这种设计使得chubby接口不同于UNIX文件系统，它使得分布更容易。为允许不同目录下的文件服务于不同的Chubby Master，我们没有放出那些将文件从一个目录移动到另一个目录的操作，我们不维护目录修改时间，也避开路径相关的权限语义（也就是文件的访问由其本身的权限控制，而不由它上层路径上的目录控制)。 为使缓存文件元数据更容易，系统不公开最后访问时间。

The name space contains only files and directories, collectively called *nodes*. Every such node has only one name within its cell; there are no symbolic or hard links.

其名字空间包含文件和目录，统一叫做**节点**(nodes)。每个这样的节点在其所在的Chubby单元内只有一个名字，没有符号链接和硬链接。

Nodes may be either permanent or ephemeral. Any node may be deleted explicitly, but ephemeral nodes are also deleted if no client has them open (and, for directories, they are empty). Ephemeral files are used as temporary files, and as indicators to others that a client is alive. **Any node can act as an advisory reader/writer lock;** these locks are described in more detail in Section 2.4.

节点可能是永久的或者瞬时(ephemeral)的。任意节点都可以被显示地(explicitly)删除，但是瞬时节点也会在没有客户端打开它时被删除（另外，对目录而言，在它们为空时被删除）。瞬时文件被用作临时文件，也被用作一个客户端是否存活的指示器（给其他客户端）。任意节点都能作为一个意向性(advisory)的读写锁；这些锁将在2.4节更详细地描述。

Each node has various meta-data, including three names of access control lists (ACLs) used to control reading, writing and changing the ACL names for the node. Unless overridden, a node inherits the ACL names of its parent directory on creation. ACLs are themselves files located in an ACL directory, which is a well-known part of the cell’s local name space. These ACL files consist of simple lists of names of principals; readers may be reminded of Plan 9’s *groups* [21]. Thus, if file F’s write ACL name is foo, and the ACL directory contains a file foo that contains an entry bar, then user bar is permitted to write F. Users are authenticated by a mechanism built into the RPC system. Because Chubby’s ACLs are simply files, they are automatically available to other services that wish to use similar access control mechanisms. The per-node meta-data includes four monotonically increasing 64-bit numbers that allow clients to detect changes easily:

每个节点有多种元数据，包括访问控制列表(ACLs)的三个名字，分别用于控制读、写和修改其ACL。除非被覆盖，节点在创建时继承父目录的ACL名字。ACLs本身是位于一个ACL目录中的文件, 这个ACL目录是Chubby单元的一个为人熟知的本地名字空间。这些ACL文件的内容由简单的访问名字(principals)列表组成；这里读者可能会想起Plan 9的 *groups*[21]。这样，如果文件F的写ACL名字是foo，并且ACL目录中包含了一个名foo的文件，foo文件中包含bar这个条目，那么用户bar就可以写文件F。用户由内嵌在RPC系统里的机制鉴权。因为Chubby的ACLs是平常的文件，它们自动地就可以由其他想使用类似的访问控制机制的服务访问。每个节点的元数据包括四个单调递增的64位编号。这些编号允许客户端很容易地检测变化：

* an instance number; greater than the instance number of any previous node with the same name.
* a content generation number (files only); this increases when the file’s contents are written.
* a lock generation number; this increases when the node’s lock transitions from *free* to *held*.
* an ACL generation number; this increases when the node’s ACL names are written. Chubby实例编号：大于任意先前的同名节点的实例编号。
* 内容的世代编号（只针对文件）：这个编号在文件内容被写入时增加。
* 锁的世代编号：这个编号在节点的锁由自由(free)转换到持有(held)时增加。
* ACL的世代编号：这个编号在节点的ACL名字被写入时增加。
* also exposes a 64-bit file-content checksum so clients may tell whether files differ.

Clients open nodes to obtain *handles* that are analogous to UNIX file descriptors. Handles include:

* check digits that prevent clients from creating or guessing handles, so full access control checks need be performed only when handles are created (compare with UNIX, which checks its permissions bits at open time, but not at each read/write because file descriptors cannot be forged).
* a sequence number that allows a master to tell whether a handle was generated by it or by a previous master.
* mode information provided at open time to allow the master to recreate its state if an old handle is presented to a newly restarted master.

Chubby也放出了一个64位的文件内容的校验码，所以客户端可以分辨文件是否有变化。

客户端通过打开节点得到类似于UNIX文件描述符的句柄(handles)。句柄包括：

* 校验位：阻止客户端自行创建或猜测句柄，所以完整的访问控制检查只需要在句柄创建时执行（对比UNIX，UNIX在打开时检查权限位但在每次读写时不检查，因为文件描述符不能伪造）。
* 一个序列号：这个序列号允许master分辨一个句柄是由它或前面的master生成。
* 模式信息：在句柄打开时设定的是否允许新master在遇见一个由前面的master创建的旧句柄时重建该句柄的状态。

**2.4 锁和序号**

Each Chubby file and directory can act as a reader-writer lock: either one client handle may hold the lock in exclusive (writer) mode, or any number of client handles may hold the lock in shared (reader) mode. Like the mutexes known to most programmers, locks are ***advisory***. That is, they conflict only with other attempts to acquire the same lock: holding a lock called F neither is necessary to access the file F, nor prevents other clients from doing so. We rejected *mandatory* locks, which make locked objects inaccessible to clients not holding their locks:

每个Chubby文件和目录都能作为一个读写锁：要么一个客户端句柄以排他（写者）模式持有这个锁，要么任意数目的客户端句柄以共享（读者）模式持有这个锁。像大部分程序员熟知的互斥器(mutexes)，锁是意向性的(advisory)。就是只与对同一个锁的加锁请求冲突：持有锁F既不是访问文件F的必要条件，也不会阻止其他客户端访问文件F。我们舍弃强制锁―强制锁使得其他没有持有锁的客户端不能访问被锁定的对象:

* Chubby locks often protect resources implemented by other services, rather than just the file associated with the lock. To enforce mandatory locking in a meaningful way would have required us to make more extensive modification of these services.
* We did not wish to force users to shut down applications when they needed to access locked files for debugging or administrative purposes. In a complex system, it is harder to use the approach employed on most personal computers, where administrative software can break mandatory locks simply by instructing the user to shut down his applications or to reboot.
* Our developers perform error checking in the conventional way, by writing assertions such as “lock X is held”, so they benefit little from mandatory checks. Buggy or malicious processes have many opportunities to corrupt data when locks are not held, so we find the extra guards provided by mandatory locking to be of no significant value.
* Chubby锁经常保护由其他服务实现的资源，而不仅仅是与锁关联的文件。以一种有意义的方式执行强制锁定可能需要我们对这些服务做更广泛的修改。
* 我们不想强迫用户在他们为了调试和管理而访问这些锁定的文件时关闭应用程序。在一个复杂的系统中，这种在个人电脑上常用的方法是很难用的。个人电脑上的管理员可以通过指示用户关闭应用或者重启来中止强制锁定。
* 我们的开发者用常规的方式来执行错误检测，例如像“lock X is held”这样的断言，所以他们从强制检查中受益很少。有Bug或者恶意的进程有很多机会在没有持有锁时破坏数据，所以我们发现强制锁定提供的额外保护没有实际价值。

In Chubby, acquiring a lock in either mode requires **write permission** so that an unprivileged reader cannot prevent a writer from making progress.

在Chubby中，请求任意模式的锁都需要**写权限**，因而一个无权限的读者不能阻止一个写者的操作。

Locking is complex in distributed systems because communication is typically uncertain, and processes may fail independently. Thus, a process holding a lock *L* may issue a request *R*, but then fail. Another process may acquire *L* and perform some action before *R* arrives at its destination. If *R* later arrives, it may be acted on without the protection of *L*, and potentially on inconsistent data. The problem of receiving messages out of order has been well studied; solutions include ***virtual time***[11], and ***virtual synchrony***[1], which avoids the problem by ensuring that messages are processed in an order consistent with the observations of every participant.

在分布式系统中，锁定是很复杂的，因为通信经不可靠，并且每个进程可能失败。像这样，一个持有锁 L 的进程可能发起请求 R ，但接着就失败了。另一个进程可能请求锁 L 并在 R 到达目的地之前执行了某些操作。如果 R 来晚了，它可能在没有锁 L 的保护下操作，也有可能在不一致的数据上操作。接收消息顺序紊乱的问题已经被研究得很彻底：解决方案包括*虚拟时间*[***[11]***](javascript:void())和*虚拟同步*[***[1]***](javascript:void())，它通过确保与每个参与者的观察一致的顺序处理消息来避免这个问题。

It is costly to introduce sequence numbers into all the interactions in an existing complex system. Instead, Chubby provides a means by which sequence numbers can be introduced into only those interactions that make use of locks. At any time, a lock holder may request a *sequencer*, an opaque(不透明的) byte-string that describes the state of the lock immediately after acquisition. It contains the name of the lock, the mode in which it was acquired (exclusive or shared), and the lock generation number. The client passes the sequencer to servers (such as file servers) if it expects the operation to be protected by the lock. The recipient server is expected to test whether the sequencer is still valid and has the appropriate mode; if not, it should reject the request. The validity of a sequencer can be checked against the server’s Chubby cache or, if the server does not wish to maintain a session with Chubby, against the most recent sequencer that the server has observed. The sequencer mechanism requires only the addition of a string to affected messages, and is easily explained to our developers.

在一个已有的复杂系统中给所有的交互引入顺序编号的成本很高。作为替代，Chubby提供了一种方法，通过该方法可以将序列号引入仅使用锁的那些交互中。任何时候，锁持有者可能请求一个*序号*，一个不透明的描述了锁刚获得时的状态的字节串(byte-string)。它包括锁的名字，被请求的锁定模式（独占还是共享），以及锁的世代编号。客户端在其希望操作将被锁保护时传递这个序号给服务器（例如文件服务器）。接受服务器将测试这个序号是否仍然有效并且具有适当的锁定模式，如果没有，它将拒绝该请求。序号的有效性可与服务器的Chubby缓存核对，或者如果服务器不想维护一个到Chubby的Session的话，可与服务器最近观察到的序号比对。这种序号机制只需要给影响的消息增加一条字符串，并且很容易地解释给我们的开发者。

Although we find sequencers simple to use, important protocols evolve slowly. Chubby therefore provides an imperfect but easier mechanism to reduce the risk of delayed or re-ordered requests to servers that do not support sequencers. If a client releases a lock in the normal way, it is immediately available for other clients to claim, as one would expect. However, if a lock becomes free because the holder has failed or become inaccessible, the lock server will prevent other clients from claiming the lock for a period called the ***lock-delay***. Clients may specify any lock-delay up to some bound, currently one minute; this limit prevents a faulty client from making a lock (and thus some resource) unavailable for an arbitrarily long time. While imperfect, the lock-delay protects unmodified servers and clients from everyday problems caused by message delays and restarts.

虽然我们发现序号使用简单，但重要的协议也在缓慢演化。因此Chubby提供了一种不完美但是更容易的机制来降低延迟的或者不支持序号的服务器消息乱序的风险。如果一个客户端以通常的方式释放了一个锁，像人们期待的那样，这个锁立即可供其他的客户端索取。然而，如果一个锁是的持有者停止工作或者不可访问，锁服务器将阻止其他的客户端在一个被称作锁延时(lock-delay)的小于某个上界的时间段内索取该锁。客户端可能指定任意的低于某个上界（当前是一分钟）的锁延时，这个限制防止有缺陷的客户端造成一个锁(和像这样的某些资源）在一个随意长的时间里不可用。 尽管不完美，锁延时保护未修改过的服务器和客户端免受消息延时和重启导致的日常问题的影响。

**2.5 事件**

Chubby clients may subscribe to a range of events when they create a handle. These events are delivered to the client asynchronously via an up-call from the Chubby library. Events include:

* file contents modified—often used to monitor the location of a service advertised via the file.
* child node added, removed, or modified—used to implement mirroring (§2.12). (In addition to allowing new files to be discovered, returning events for child nodes makes it possible to monitor ephemeral files without affecting their reference counts.)
* Chubby master failed over—warns clients that other events may have been lost, so data must be rescanned.
* a handle (and its lock) has become invalid—this typically suggests a communications problem.
* lock acquired—can be used to determine when a primary has been elected.
* conflicting lock request from another client—allows the caching of locks.

在创建句柄时，Chubby客户端可能会订阅一系列事件。这些事件通过Chubby库的向上调用(up-call)异步地传递给客户端。这些事件包括：

* 文件内容被修改--常用于监视通过文件公布的某个服务的位置信息。
* 子节点的增加、删除和修改 ― 用于实现镜像(第2.12节)。(除了允许新文件可被发现外，返回子节点上的事件使得可以监控临时文件(ephemeral files)而不影响这些临时文件的引用计数(reference counts))
* Chubby master故障恢复 ― 警告客户端其他的事件可能已经丢失了，所以必须重新检查数据。
* 一个句柄（和它的锁）失效了 ― 这经常意味着某个通讯问题。
* 锁被请求了(lock required) ― 可用于判断什么时候primary选出来了。
* 来自另一个客户端的相冲突的锁请求 ― 允许锁的缓存。

Events are delivered after the corresponding action has taken place. Thus, if a client is informed that file contents have changed, it is guaranteed to see the new data (or data that is yet more recent) if it subsequently reads the file.

事件在相应的动作发生之后被递送。这样，如果一个客户端被告知文件内容发生了变化，一定能保证接下来它读这个文件会看到新的数据（或者比该事件还要更新的数据）。

The last two events mentioned are rarely used, and with hindsight could have been omitted. After primary election for example, clients typically need to communicate with the new primary, rather than simply know that a primary exists; thus, they wait for a file modification event indicating that the new primary has written its address in a file. The conflicting lock event in theory permits clients to cache data held on other servers, using Chubby locks to maintain cache consistency. A notification of a conflicting lock request would tell a client to finish using data associated with the lock: it would finish pending operations, flush modifications to a home location, discard cached data, and release. So far, no one has adopted this style of use.

上面提到的最后两种事件极少使用，事后可能会被省略。例如在primary选举后，客户端通常需要与新的primary联系，而不是简单地知道一个新的primary存在了；因而，它们会等待primary将地址写入文件的文件修改事件。锁冲突事件在理论上允许客户端缓存其他服务器持有的数据，使用Chubby锁来维护缓存的一致性。一个冲突的锁请求将会告诉客户端结束使用与锁相关的数据；它将结束等待进行的操作，将修改刷新到原来的位置，丢弃缓存数据，然后释放锁。到目前为止，没有人采纳这种用法。

**2.6 API**

Clients see a Chubby handle as a pointer to an opaque structure that supports various operations. Handles are created only by Open(), and destroyed with Close().

客户端将Chubby句柄看作一个指向一个支持多种操作的不透明结构的指针。句柄之通过open()创建，以及通过close()销毁。

Open() opens a named file or directory to produce a handle, analogous to a UNIX file descriptor. Only this call takes a node name; all others operate on handles.

Open() 打开一个带名字的文件或目录产生一个句柄，类似于一个UNIX文件描述符。只有这个调用需要一个节点名，所有其他的调用都在句柄上操作。

The name is evaluated relative to an existing directory handle; the library provides a handle on ”/” that is always valid. Directory handles avoid the difficulties of using a program-wide *current directory* in a multi-threaded program that contains many layers of abstraction [18]. The client indicates various options:

这个节点名是相对于某个已知的目录句柄的，客户端库提供了一个一直有效的“/”目录句柄。目录句柄避免了在包含多层抽象的多线程程序中使用程序范围的当前目录的困难[18]。 在调用Open()时，客户端指定了多个选项：

how the handle will be used (reading; writing and locking; changing the ACL); the handle is created only if the client has the appropriate permissions.

句柄将被如何使用（读；写和锁定；改变ACL）；句柄只在客户端拥有合适的权限时创建。

* events that should be delivered (see §2.5).
* the lock-delay (§2.4).
* whether a new file or directory should (or must) be created. If a file is created, the caller may supply initial contents and initial ACL names. The return value indicates whether the file was in fact created.
* 需要传递的事件（查看第2.5节）。
* 锁延时（第2.4节）。
* 是否应该（或者必须）创建一个新的文件或目录。如果要创建一个文件，调用者可能要提供初始内容和初始的ACL名字。其返回值表明这个文件实际上是否已经创建。

Close() closes an open handle. Further use of the handle is not permitted. This call never fails. A related call Poison() causes outstanding and subsequent operations on the handle to fail without closing it; this allows a client to cancel Chubby calls made by other threads without fear of deallocating the memory being accessed by them. The main calls that act on a handle are:

Close() 关闭一个打开的句柄。以后不再允许使用这个句柄。这个调用永远不会失败。一个相近的调用Poison()引起该句柄尚未完成的和接下来的操作失败，但不关闭它，这就允许一个客户端取消由其他线程创建的Chubby调用而不需要担心被它们访问的内存的释放。在句柄上进行的主要调用包括：

GetContentsAndStat() returns both the contents and meta-data of a file. The contents of a file are read **atomically** and in their **entirety**. We avoided partial reads and writes to discourage large files. A related call GetStat() returns just the meta-data, while ReadDir() returns the names and meta-data for the children of a directory.

GetContentsAndStat() 返回文件的内容和元数据。一个文件的内容被原子地、完整地读取。我们避开部分读取和写入以阻碍大文件。一个相关的调用GetStat()只返回元数据，而ReadDir()返回一个目录下的名字和元数据(the names and meta-data for the children of a directory)。

SetContents() writes the contents of a file. Optionally, the client may provide a content generation number to allow the client to simulate compare-and-swap on a file; the contents are changed only if the generation number is current. The contents of a file are always written atomically and in their entirety. A related call SetACL() performs a similar operation on the ACL names associated with the node.

SetContents()将内容写到文件。可选择地，客户端可能提供一个内容世代编号以允许客户端模拟在文件上的比较和交换：只有在世代编号是当前值时内容才被改变。文件的内容总是完整地、原子地写入。一个相关的调用SetACL()在节点关联的ACL名字上执行一个类似的操作。

Delete() deletes the node if it has no children.

Delete() 删除一节节点，如果它没有孩子的话。

Acquire(), TryAcquire(), Release() acquire and release locks.

Accquire(), TryAccquire(), Release() 获得和释放锁。

GetSequencer() returns a sequencer (§2.4) that describes any lock held by this handle.

GetSequencer() 返回一个描述这个句柄持有的锁的序号(sequencer)(见第2.4节)。

SetSequencer() associates a sequencer with a handle. Subsequent operations on the handle fail if the sequencer is no longer valid.

SetSequencer() 将一个序号与句柄关联。在这个句柄上的随后的操作将失败如果这个序号不再有效的话。

CheckSequencer() checks whether a sequencer is valid (see §2.4).

Calls fail if the node has been deleted since the handle was created, even if the file has been subsequently recreated. That is, a handle is associated with an instance of a file, rather than with a file name. Chubby may apply access control checks on any call, but always checks

Open() calls (see §2.3).

CheckSequencer() 检查一个序号是否有效。（见第2.4节）  
如果节点在该句柄创建之后被删除，其上的调用将会失败，即使这个文件随后又重新创建了也是如此。也就是一个句柄关联到一个文件的实例，而不是文件名。Chubby可能在任意的调用上使用访问控制，但是总是检查Open() 调用。

All the calls above take an *operation* parameter in addition to any others needed by the call itself. The operation parameter holds data and control information that may be associated with any call. In particular, via the operation parameter the client may:

* supply a callback to make the call asynchronous,
* wait for the completion of such a call, and/or
* obtain extended error and diagnostic information.

除了调用自身需要的参数之外, 上面的所有调用都带有一个操作参数。这个操作参数持有可能与任何调用相关的数据和控制信息。特别地，通过操作的参数客户端可能：

* 提供一个回调(callback)以使调用以异步方式执行。
* 等待调用的结束，和/或
* 取得展开的错误和诊断信息。

Clients can use this API to perform primary election as follows: All potential primaries open the lock file and attempt to acquire the lock. One succeeds and becomes the primary, while the others act as replicas. The primary writes its identity into the lock file with SetContents() so that it can be found by clients and replicas, which read the file with GetContentsAndStat(), perhaps in response to a file-modification event (§2.5). Ideally, the primary obtains a sequencer with GetSequencer(), which it then passes to servers it communicates with; they should confirm with CheckSequencer() that it is still the primary. A lock-delay may be used with services that cannot check sequencers (§2.4).

客户端可以利用这个API执行primary选举：所有潜在的primaries打开锁文件，并尝试获得锁。其中一个成功获得锁，并成为primary，而其他的则作为副本。这个Primary将它的标识用SetContents()写入到锁文件，在对文件修改事件的响应中，它被客户端和副本们发现，它们用GetContentsAndStat()读取锁文件。理想情况下，Primary通过GetSequencer()取得一个序号，然后将序号传递给与它通信的其他服务器：这些服务器应该用CheckSequencer()确认它仍然是primary。一个锁延时可能与不能检查序号（第2.4节）的服务一起使用。

**2.7 缓存**

为了减少读传输量，Chubby客户端将文件数据和节点元数据（包括文件缺失信息(file absence)) 缓存在内存中的一个一致的(consistent)、write-through的缓存中。这个缓存由一个如后文描述的租期机制(lease mechanism)维护，并通过由master发送的过期信号(invalidations)来维护一致性。Master保留着每个客户端可能正在缓存的数据的列表。这个协议保证客户端要么看到一致的Chubby状态，要么看到错误。

当文件数据或者元数据将被修改时，修改操作被阻塞，同时master发送过期信号给所有可能缓存了这些数据的客户端。这种机制建立在下一节要详细讨论的KeepAlive RPC之上。在接收到过期信号后，客户端清除过期的状态信息，并通过发起它的下一个KeepAlive调用应答服务器。修改操作只在服务器知道每个客户端都将这些缓存失效以后再继续，要么因为客户端答谢了过期信号，要么因为客户端让它的缓存租期(cache lease)过期。

只有一次过期来回(round)是必需的，因为master在缓存过期信号没有答谢期间将这个节点视为不可缓存的(*uncachable*)。这种方式让读操作总是被无延时地得到处理；这是很有用的，因为读操作的数量要大大超过了写操作。另一种选择可能是在过期操作期间阻塞对该节点的访问；这将使得过度急切的客户端在过期操作期间连续不断地以无缓存的访问轰炸master的可能性大大降低，其代价是偶然的延迟。 如果这成为问题，人们可能会想采用一种混合方案，在检测到过载时切换处理策略(tactics)。

缓存协议很简单：它在修改时将缓存的数据失效，而永不去更新这些数据。有可能去更新缓存而不是让缓存失效也会一样简单，但是只更新的协议可能会无理由地低效；某个访问文件的客户端可能无限期地收到更新，从而导致次数无限制的、完全不必要的更新。

尽管提供严格一致性的开销不小，我们拒绝了更弱的模型，因为我们觉得程序员们将发现它们很难用。类似地，像虚拟同步(virtual synchrony)这种要求客户端在所有的消息中交换序号的机制，在一个有多种已经存在的通信协议的环境中也被认为是不合适的。

除了缓存数据和元数据，Chubby客户端还缓存打开的句柄。因而，如果一个客户端打开一个前面已经打开的文件，只有第一次Open()调用时会引起一个给Master的RPC。这种缓存被限制在较低层次(in a minor ways)，所以它不会影响客户端观察到的语义：临时文件上的句柄在被应用程序关闭后，不能再保留在打开状态；而容许锁定的句柄则可被重用，但是不能由多个应用程序句柄并发使用。最后的这个限制是因为客户端可能利用Close()或者Poison()的边际效应：取消正在进行的向Master请求的Accquire()调用。

Chubby的协议允许客户端缓存锁 ― 也就是，怀着这个锁会被同一个客户端重新使用的希望，持有锁比实际需要更长的时间。如果另一个客户端请求了一个冲突的锁，可以用一个事件告知锁持有者，这允许锁持有者只在别的地方需要这个锁时才释放锁（参考§2.5）。

**2.8 会话 和 KeepAlives**

一个Chubby会话是Chubby单元和Chubby客户端之间的一种关系，它存在于么某个时间间隔内，并由称做KeepAlive的间歇性地握手来维持。除非一个Chubby客户端通知master,否则，只要它的会话依然有效，该客户端的句柄、锁和缓存的数据都仍然有效。（然而，会话维持的协议可能要求客户端答谢一个缓存过期信号以维持它的会话，请看下文）

一个客户端在第一次联系一个Chubby单元的master时请求一个新的会话。它要么在它结束时明确地结束会话，或者在会话陷入空转时(在一段时间内没有任何打开的句柄和调用)结束会话。

每个会话都有关联了一个租期(lease) ― 一个延伸向未来的时间间隔，在这个时间间隔内master保证不单方面中止会话。间隔的结束被称作租期到期时间(lease timeout)。Master可以自由地向未来延长租期到期时间，但是不能将它往回移动。

在三种情形下，Master延长租期到期时间：会话创建时、master故障恢复（见下文）发生时和它应答来自客户端的KeepAlive RPC时。在接收KeepAlive时，master通常阻塞这个RPC请求（不允许其返回），直到客户端的上前一个租期接近过期。然后，Master允许RPC返回给客户端，并告知客户端新的租期过期时间。Master可能任意长度地延伸超时时间。默认的延伸是12秒，但一个过载的master可能使用更高的值，以降低它必须处理的KeepAlive调用数目。客户端在收到上一个回复以后立即发起一个新的KeepAlive，这样客户端保证几乎总是有一个KeepAlive调用阻塞在master上。

除了延伸客户端的租期，KeepAlive的回复还被用于将事件和缓存过期传回给客户端。Master允许一个KeepAlive在有事件或者缓存过期需要递送时提前返回。在KeepAlive的回复上搭载事件，保证了客户端不应答缓存过期则不能维持会话，并使得所有的RPC都是从客户端流向master。这样既简化了客户端，也使得协议可以通过只允许单向发起连接的防火墙。

客户端维持着一个本地租期超时，这个本地超时值是Master的租期的较小的近似值。它跟master的租期过期不一样，是因为客户端必须在两方面做保守的假设。一是KeepAlive花在传输上的时间，一是master的时钟超前的度。为了维护一致性，我们要求服务器的时钟频率相对于客户端的时钟频率，不会快于某个常数量(constant factor)。

如果一个客户端的本地租期过期了，它就变得不能确定master是否终结了它的会话。于是这个客户端清空并禁用自己的缓存，我们称它的会话处于危险(jeopardy)中。客户端在等待一个之后的被称作宽限期的间隔，默认是45秒。如果这个客户端和master在宽限期结束之前设法完成了一个成功的KeepAlive，客户端就再开启它的缓存。否则，客户端假定会话已经过期。这样做了，所以Chubby的API调用不会在Chubby单元变成不可访问时无限期阻塞，如果在通讯重新建立之前宽限期结束了，调用返回一个错误。

Chubby库能够通过一个”危险”(jeopardy)事件在宽限期开始时通知应用程序。当知道会话在通讯问题后幸存下来，一个”安全”(safe)事件告知应用程序继续工作；而如果会话时间过去了，一个”过期”(expire)事件被发送给应用程序。这些通知允许应用程序在对它的会话状态不确定时停下来，并且在问题被证明是瞬时的时不需要重启进行恢复。在启动开销很高的服务中，这在避免服务不可用方面可能是很重要的。

如果一个客户端持有某个节点上的句柄 H，并且任意一个H上的操作都因为关联的会话过期了而失败，那么所有接下来的H上的操作（除了Close()和Poison()) 将以同样的方式失败。客户端可以用这去保证网络和服务器不可用时会导致随后的一串操作丢失了，而不是一个随机的操作子序列丢失，这样就允许复杂的修改可以以其最后一次写标记为已提交。

**2.9 故障切换**

chubby_figure2  
当一个master失效了或者失去了master身份时，它丢弃内存中有关会话、句柄和锁的信息。有权力的(authoritative)会话租期计时器开始在master上运行。这样，直到一个新的master被选举出来，租期计时器才停止；这是合法的因为它等价于延长客户端的租期。如果一个master选举很快发生，客户端可以在它们本地（近似的）租期计时器过期之前与新的master联系。如果选举用了很长时间，客户端刷空它们的缓存并等待一个宽限期(grace period)，同时尝试寻找新的master。以这种方式，宽限期(grace period)使得会话在跨越超出正常租期的故障恢复期间被维持。

图2展示了一个漫长的故障恢复事件中的事件序列，在这个恢复事件中客户端必须使用它的宽限期来保留它的会话。时间从左到右延伸，但是时间是不可以往回缩的。客户端会话租期显示为粗箭头，它既被新的、老的Master看到(M1-3，上方)，也被客户端都能看到(C1-3，下方)。倾斜向上的箭头标示KeepAlive请求，倾斜向上的箭头标示这些KeepAlive请求的应答。原来的Master有一个给客户端的租期M1，而客户端有一个(对租期的)保守的估计C1。在通过KeepAlive应答2通知客户端之前，原Master允诺了一个新的租期M2；客户端能够延长它的视线为租期C2。原Master在应答下一个KeepAlive之前死掉了，过了一段时间后另一个master被选出。最终客户端的租期估计值(C2)过期了。然后客户端清空它的缓存，并为宽限期启动一个计时器。

在这段时间里，客户端不能确定它的租期在master上是否已经过期了。它没有销毁它的会话，但是它阻塞所有应用程序对它的API的调用，以防止应用程序观测到不一致的数据。在宽限期开始时，Chubby的库发送一个危险事件给应用程序，允许它停下来直到会话状态能够确定。最终一个新的master选举完成了。Master最初使用对它的前任可能持有的给客户端的租期的保守估算值M3。新Master的来自客户端的第一个KeepAlive请求(4)被拒绝了，因为它的master代数不正确（下文有详细描述）。重试请求(6)成功了，但是通常不去延长master租期，因为M3是保守的。然后它的回复(7)允许客户端延再一次长它的租期(C3)，还可以选择通知应用程序其会话不再处于危险状态。因为宽限期足够长，可以覆盖从租期C2的结束到租期C3的开始之间的间隔，客户端除了观测到延迟不会看到别的。但假如宽限期比这个间隔短，客户端将丢弃会话，并将失败报告给应用程序。

一旦客户端与新的master联系上，客户端库和master协作提供给应用程序一种没有故障发生过的假象。为达到此目的，新的master必须重新构造一个前面的master拥有的内存状态的保守估算（conservative approximation）。它通过读取存储在硬盘上的数据（通过常规的数据库复制协议复制）来完成一部分，通过从客户端获取状态完成一部分，通过保守的假设（conservative assumptions）完成一部分。数据库记录了每个会话、被持有的锁和临时文件。

一个新选出来的master继续处理：  
1. 它先选择一个新的代编号（epoch number），这个编号在客户端每次请求时都要求出示。Master拒绝来自使用较早的代编号的客户端的请求，并提供新的代编号给它们。这保证了新的master将不会响应一个很早的发给前一个master的包，即使此前的这个Master与现在的master在同一台机器上。  
2. 新的Master可能会响应对master的位置的请求（master-location requests），但是不会立即开始处理传入的会话相关的操作。  
3. 它为记录在数据库中的会话和锁构建内存数据结构。会话租期被延长至上一个master可能当时正使用的最大期限。  
4. Master现在允许客户端进行KeepAlive，但不能执行其他会话相关的操作。  
5. It emits a fail-over event to each session; this causes clients to flush their caches (because they may have missed invalidations), and to warn applications that other events may have been lost.  
5. 它发出一个故障切换事件给每个会话；这引起客户端刷新它们的缓存（因为这些缓存可能错过了过期信号（invalidations）），并警告应用程序别的事件可能已经丢失。  
6. Master一直等待，直到每个会话应答了故障切换事件或者使其会话过期。  
7. Master允许所有的操作继续处理。  
8. 如果一个客户端使用一个在故障切换之前创建的句柄（可依据句柄中的序号来判断），master重新创建了这个句柄的内存印象（in-memory representation），并执行调用。如果一个这样的重建后的句柄被关闭后，master在内存中记录它，这样它就不能在这个master代（epoch）中再次创建；这保证了一个延迟的或者重复的网络包不能偶然地重建一个已经关闭的句柄。一个有问题的客户端能在未来的master代中重建一个已关闭的句柄，但倘若该客户端已经有问题的话，则这样不会有什么危害。  
9. 在经过某个间隔（如，一分钟）后，master删除没有已打开的文件句柄的临时文件。在故障切换后，客户端应该在这个间隔期间刷新它们在临时文件上的句柄。这种机制有一个不幸的效果是，如果该文件的最后一个客户端在故障切换期间丢失了会话的话，临时文件可能不会很快消失。

读者将不意外地得知，远远不像系统的其他部分那样被频繁使用的故障切换代码，是一些有趣的bug的丰富来源。

**2.10 数据库实现**

第一版的Chubby使用带复制的Berkeley DB版本[20]作为它的数据库。Berkeley DB提供了映射字节串的键到任意的字节串值上B-树。我们设置了一个按照路径名称中的节数排序的键比较函数，这样就允许节点用它们的路径名作为键，同时保证兄弟节点在排序顺序中相邻。由于Chubby不使用基于路径的权限，数据库中的一次查找就可以满足每次文件访问。

Berkeley DB使用一种分布式一致协议将它的数据库日志复制到一组服务器上。只要加上master租期，就吻合Chubby的设计，这使得实现很简单。相比于在Berkeley DB的B-树代码被广泛使用并且很成熟，其复制相关的代码是最近增加的，并且用户较少。维护者必须优先维护和改进他们的最受欢迎的产品特性。

在Berkeley DB的维护者解决我们遇到的问题期间，我们觉得使用这些复制相关的代码将我们暴露在比我们愿负担的更多的风险中。结果，我们写了一个简单的，类似于Birrell et al.[2]的设计的，利用了日志预写和快照的数据库。数据库日志还是想以前一样利用一致性协议在多个副本之间分布。Chubby至少了Berkeley DB的很少的特性，这样重新可使整个系统大幅简化；例如，我们需要原子操作，但是我们确实不需要通用的事务。

**2.11 备份**

每隔几个小时，Chubby单元的mster将它的数据库快照写到另一栋楼里的GFS文件服务器上 [7]。使用另一栋楼（的文件服务器）既确保了备份会在楼宇损毁中保存下来，也确保备份不会在系统中引入循环依赖，同一栋楼中的GFS单元潜在地可能依赖于Chubby单元来选举它的master。 备份既提供了灾难恢复，也提供了一种初始化一个新创建的副本的数据库的途径，而不会将初始化的压力放在其他正在提供服务的副本上。

**2.12 镜像 (mirroring)**

Chubby允许一组文件(a collection of files)从一个单元镜像到另外一个单元。镜像是很快的，因为文件很小，并且事件机制（参见第2.5节）会在文件增加、删除或修改时立即通知镜像处理相关的代码。在没有网络问题的前提下，变化会一秒之内便在世界范围内的很多个镜像中反映出来。如果一个镜像不可到达，它保持不变直到连接恢复。然后通过比较校验码识别出更新了的文件。

镜像被最常见地用于将配置文件复制到各种各样的遍布全世界的计算集群。一个叫global的特殊的Chubby单元，含有一个子树 /ls/global/master，这个子树被镜像到每个其他的Chubby单元的子树 /ls/cell/slave。这个Global单元很特别，因为它的五个副本分散在全球相隔很远的多个地区，所以几乎总是能够从大部分国家/地区访问到它。

在这些从global单元镜像的文件中，有Chubby自己的访问控制列表，多种含有Chubby单元和其他系统向我们的监控服务广告它们的存在的文件，允许客户端定位巨大的数据集如BigTable单元的指针信息（的文件），以及许多其他系统的配置文件。

**3 扩展机制**

因为Chubby的客户端是独立的进程，所以Chubby必须处理比人们预想的更多的客户端；我们曾经看见过90,000个客户端直接与一个Chubby服务器通讯 ― 这远大于涉及到的机器总数。因为在一个单元之中有一个master，而且它的机器跟那些客户端是一样的，于是客户端们能以巨大的优势(margin)将master打败。因此，最有效的扩展技术减少与master的通信通过一个重大的因数(Thus, the most effective scaling techniques reduce communication with the master by a significant factor)。假设master没有严重的性能缺陷，master上的请求处理中的较小改进只有很小的(have little)效果。我们使用了下面几种方法：

* 我们可以创建任意数量的Chubby单元；客户端几乎总是使用一个邻近的单元（通过DNS找出）以避免对远程机器的依赖。我们的有代表性的发布让一个有几千台机器的数据中心使用一个Chubby单元。
* Master可能会在负载很重时将租期(lease times)从默认的12秒到延长到最大60秒左右，这样它就只需要处理更少的KeepAlive RPC调用。(KeepAlive是到目前为止的占统治地位的请求类型（参考第4.1节），并且未能及时处理它们是一个超负荷的服务器的代表性的失败模式；客户端对其他调用中的延迟变化相当地不敏感。）
* Chubby客户端缓存文件数据，元数据，文件缺失(the absence of files)和打开的句柄，以减少它们在服务器上做的调用。
* 我们使用转换Chubby协议为不那么复杂协议如DNS等的协议转换服务器。我们后文描述其中的一些。

这里我们描绘两种熟悉的机制，代理(proxies)和分区(partitioning)，这两种机制将允许Chubby扩展更多。我们现在还没有在产品环境中使用它们，但是它们已经被设计出来了，并且可能很快被使用。我们还没有显示出考虑扩展到五倍以外的需要：第一，在一个数据中心想要放的或者依赖于单个实例服务的机器的数目是有限制的；第二，因为我们为Chubby的客户端和服务器使用类似配置的机器，硬件上的提升增加了每台机器上的客户端的数量时，同时也增加了每台服务器的容量。

**3.1 代理**

Chubby的协议可以由可信的进程代理（在两边都使用相同的协议），这个进程将来自其他客户端的请求传递给Chubby单元。一个代理可以通过处理KeepAlive和读请求减少服务器负载，它不能减少从代理的缓存中穿过的写入流量。但即使有积极的客户端缓存，写入流量仍远小于Chubby正常负载的百分之一（参见第4.1节），这样代理允许大幅提升客户端的数量。如果一个代理处理*Nproxy*个客户端，KeepAlive流量将减少*Nproxy*倍，而*Nproxy*可能是一万甚至更大。A proxy cache can reduce read traffic by at most the mean amount of read-sharing�a factor of around 10 （参见第4.1节）。但因为读只占当前的Chubby负载的10%以下，KeepAlive流量上的节省仍然是到目前为止更重要的成效。

代理增加了一个额外的RPC用于写和第一次读。人们可能会预期代理会使得Chubby单元的临时不可用次数至少是以前的两倍，因为每个代理后的客户端依赖于两台可能失效的机器：它的代理和Chubby的Master。

机敏的读者会注意到2.9节描述的故障切换，对于代理机制不太理想。我们将的第4.4节讨论这个问题。

**3.2 分区**

像第2.3节提到的，Chubby的接口被选择为Chubby单元的名字空间能在服务器之间划分。虽然我们还不需要它，但是现在的代码(code)能够通过目录划分名字空间。 如果开启划分，一个Chubby单元将由*N 个分区*组成，每个分区都有一组副本(replicas)和一个master。每个在目录 *D* 中的节点 *P(D/C)* 将被保存在分区 *P(D/C)*= hash(*D*) mod *N* 上。注意 *D* 上的元数据可能存储在另一个不同的分区 P(D) = hash(D’) mod N 上， D’ 是 D 的父目录。

分区被计划为以很小的分区之间的通信来启用很大的Chubby单元集(Chubby cells)。尽管Chubby没有硬链接(hard links)，目录修改时间，和跨目录的重命名操作，一小部分操作仍然需要跨分区通信：

* ACL 本身是文件，所以一个分区可能会使用另一个分区做权限校验。 然后， ACL 文件可以很快捷地缓存；只有Open() 和 Delete() 操作需要做 ACL 核查；并且大部分客户端读公开可访问的、不需要ACL的文件。
* 当一个目录被删除时，一个跨分区的调用可能被需要以确保这个目录是空的。

因为每个分区独立地处理大部分调用，我们预期这种通信只会对性能和可用性造成不太大的影响。

除非分区数目 N 很大，我们预期每个客户端将与大部分分区联系。 因此，分区将任意一个分区上的读写通讯量(traffic)降低为原来的N分之一，但是不会降低 KeepAlive 的通信量。如果Chubby需要处理更多客户端的话，我们的应对方案将会联合使用代理和分区。

**4 实际应用，意外和设计错误**

**4.1 实际应用和表现**

下面的表给出了某个Chubby单元的快照的统计信息；其中的RPC频率是从一个10分钟的时间段里计算出来的。这些数字在Google的Chubby单元中是很常见的。  
chubby_table1  
从表中可以看到下面几点：

* 许多文件被用于命名(naming)，参见第$4.3节。
* 参数配置，访问控制和元数据文件(类似于文件系统的super-blocks)很普遍。
* Negative Caching很突出。
* 平均来看，每个缓存文件由230k/24k≈10个客户端使用。
* 较少的客户端持有锁，共享锁很少；这与锁定被用于primary选举和在多个副本之间划分数据的预期是相符合的。
* RPC流量主要是会话 KeepAlive，有少量的读（缓存未命中引起），只有很少的写或锁请求。

现在我们简要描述写我们的Chubby单元不可用的原因。如果我们假定（乐观地）如果一个单元有一个master愿意服务则是”在线的(up)”，在我们的Chubby的采样中，在数周内我们记录下合计61次不可用，合计共有700单元-天(??)。我们排除了维护引起的关闭数据中心时的不可用。所有其他的不可用的原因包括：网络拥塞，维护，超负荷和运营人员引起的错误，软件，硬件。大部分不可用在15s以内或者更短，另外52次在30秒以内；我们的大部分应用程序不会被Chubby的30秒内的不可用显著地影响到。剩下的就此不可用由这些原因引起：网络维护(4)，可疑的网络连接问题(2)，软件错误(2)，超负荷(1)。

在好几打Chubby单元年(cell-years)的运行中，我们有六次丢失了数据，由数据库软件错误(4)和运营人员错误(2)引起；与硬件错误没有关系。具有讽刺意味的是，操作错误与为避免软件错误的升级有关。我们有两次纠正了由非master的副本的软件引起的损坏。

Chubby的数据都在内存中，所以大部分操作的代价都很低。我们生产服务器的中位请求延时始终保持在一毫秒以内(a small fraction of a millisecond)，不管Chubby单元的负载如何，直到Chubby单元超出负荷，此时延迟大幅增加，会话掉线。超负荷通常发生在许多(> 90,000)的会话激活时，但是也能由异常条件引起：当客户端们同时地发起几百万读请求时（在4.3节有描述），或者当客户端库的一个错误禁用了某些读的缓存，导致每秒成千上万的请求。因为大部分RPC是KeepAlive,服务器可以通过延长会话租期(参考第三章)跟许多客户端维持一个急哦较低的中位请求延时。Group commit reduces the effective work done per request when bursts of writes arrive，但是这很少见。

客户端观测到的RPC读延迟受限于RPC系统和网络；对一个本地Chubby单元而言在1毫秒以下，但跨洲则需要250毫秒。RPC写（包括锁操作）被数据库的日志更新进一步地延迟了5-10毫秒，但是如果一个最近刚失败过的客户端要缓存该文件的话，最高可以被延迟几十秒。即使这种写延迟的变化程度对服务器上的中位请求延迟也只有很小的影响，因为写相当地罕见。

如果会话没有丢弃的话，Chubby是相当不受延迟变化的影响的。在一个时间点上(At one point)，我们在Open()中增加了人为的延时以抑制攻击性的客户端(参考第4.5节）；开发人员只会在延时超过十秒并且反复地被延时时会注意到。我们发现扩展Chubby的关键不是服务器的性能；降低到服务器端的通讯可以有远远大得多的作用。没有重要的努力用在调优读/写相关的服务器代码路径上；我们检查了没有极坏的缺陷(bug)存在，然后集中在更有效的扩展机制上。在另一方面，如果一个性能缺陷影响到客户端会每秒读几千次的本地Chubby缓存，开发者肯定会注意到。

**4.2 Java 客户端**

Google的基础架构设施大部分是用C++写的，但是一些正在不断增加中的系统正在用Java编写 [8]。这种趋势呈现出Chubby的一个预料之外的问题，而Chubby有一个复杂的客户端协议和不那么简单的(non-trival)客户端库。

Java通过让它有些不厌其烦地与其他语言连接，以渐进采用的损耗，来鼓励整个程序的移植性。通常的用于访问非原生(non-native)的库的机制是JNI [15]，但JNI被认为很慢并且很笨重。我们的Java程序员们是如此不喜欢JNI以至于为了避免使用JNI而倾向于将很大的库转换为Java，并维护它们。

Chubby的客户端库有7000行代码（跟服务器差不多），并且客户端协议很精细。去维持这样一个用Java写的库需小心和代价，同时一个没有缓存的实现将埋葬Chubby服务器。因此，我们的Java用户跑着多份协议转换服务器，这些服务器暴露一个很类似于Chubby客户端API的简单的RPC协议。即使事后回想，怎么样去避免编写、运行并维护这些额外的服务器仍然是不那么明显的。

**4.3 作为名字服务使用**

尽管Chubby被设计为一种锁服务，我们发现它的最流行的应用是做为名字服务器。

在常规的因特网命名系统 ― DNS中，缓存是基于时间的。DNS条目有一个生存时间(TTL)，DNS数据在这个时间内没有得到刷新的话，就将被丢弃。通常很容易选择一个合适的TTL值，但是当希望对失败的服务做快速替换时，这个TTL会很小，以至于使DNS服务器超载。

例如，很常见地，我们的开发者运行有数千个进程的任务，且其中的每个进程之间都要通信，这引起了二次方级的DNS查询。我们可能希望使用一个60秒的TTL，这将允许行为不正常的客户端没有过长的延迟就被换掉，同时也在我们的环境中不被认为是一个过长的替换时间。在这种情况下，为维持某一个小到3000个客户端的任务的DNS缓存，可能需要每秒15万次查找。（作为比照，一个2-CPU 2.6GHz Xeon DNS服务器么秒可能能处理5万请求）。更大的任务产生更恶劣的问题，并且多个任务可能同时执行。在引入Chubby以前，DNS负载的波动对Google来时曾经是的一个严重问题。

相反，Chubby的缓存使用显示地失效(invalidations)，因此在没有修改时，一个恒定频率的会话KeepAlive请求能无限期地维护一个客户端上的任意数目的缓存条目。 一个2-CPU 2.6GHz Xeon的Chubby Master 曾被见到过处理9万个直接与它通讯（没有Proxy）的客户端；这些客户端包括了具有上文描述过的那种通讯模式的庞大任务。不需要轮询每个名字的提供快速名字更新的能力是如此吸引人，以至于现在Google的大部分系统都由Chubby提供名字服务。

尽管Chubby的缓存允许一个单独的单元能承受大量的客户端，负载峰值仍可能是一个问题。当我们第一次发布基于Chubby的名字服务时，启动了一个3千进程的任务（其结果是产生了9百万个请求）可能将Chubby的Master压垮。为了解决这个问题，我们选择将名字条目组成批次，因而一次查询将返回同一个任务内的大量的（通常是100个）相关进程的名字映射，并缓存起来。

Chubby提供的缓存语义要比名字服务需要的缓存语义更加精确；名字解析只要求定期的通知，而非完全的一致性。因而，这里有一个时机，可以通过引入特别地为名字查找设计的简单协议转换服务器，来降低Chubby的负载。假如我们预见到将Chubby用作名字服务的用法，我们可能选择实现完整的代理，而不是提供这种简单但没有必要的额外的服务器。

有一种更进一步的协议转换服务器存在：Chubby DNS服务器。这中服务器将存储在Chubby里的命名数据提供给DNS客户端。这种服务器很重要，它既能减少了DNS名字到Chubby名字之间的转换，也能适应已经存在的不能轻易转换的应用程序，例如浏览器。

**4.4 故障切换的问题**

原来的master故障切换(§2.9)的设计要求master在新的会话创建时将新会话写入数据库。在Berkeley DB版本的锁服务器中，在许多进程同时启动时创建会话的开销成为问题。为了避免超载，服务器被修改为不在会话创建时保存会话信息到数据库，而是在会话的第一次修改、锁请求或者打开临时文件时写入数据库中。此外，活动会话在每次KeepAlive时以某种概率被记录到数据库中。这样，只读的会话信息的写入在时间上分散开来。

尽管为了避免超载，这种优化是必要的，但它有一种负面效应：年轻的只读的会话可能没有被记录在数据库中，因而在故障切换发生时可能被丢弃。虽然这种会话不持有锁，这也是不安全的；如果所有已记录的会话在被丢弃的会话的租期到期之前签入到新的master中，被丢弃的会话可能在一段时间内读到脏数据。这在实际中很少见，但是在大型系统中，几乎可以肯定某些会话将签入失败，因而迫使新的master必须等待最大租期时间。尽管这样，我们修改了故障切换设计，既避免这个效应，也避开当前这种模式带给代理的复杂性。

在新的设计下，我们完全避免在数据库中记录会话，而是以目前master重建句柄的方式重建它们(参见§2.9,8)。一个新的master现在必须等待一个完整的最坏情况下的租期过期，才能允许操作继续处理。因为它不能得知是否所有的会话都签入(check in)了(参见§2.9,6)。再者，这在实际中只有很小的影响，因为很可能不是所有的会话都会签入。

一旦会话能在没有在磁盘上的状态信息重建，代理服务器就能管理master不知情的会话。一个额外的只提供给代理的操作允许它们修改与锁有关联的会话。这允许在在代理失败时，一个代理可从另一个代理取得一个客户端。Master上增加的唯一必要的修改是保证不放弃与代理会话关联的锁或者临时文件句柄，直到一个新的代理有机会获得它们。

**4.5 攻击性的客户端**

Google的项目团队可自由地设置它们自己的Chubby单元，但这样做加强了他们的维护负担，而且消耗了额外的硬件资源。许多服务因此使用共享的Chubby单元，这使得隔离行为不正常的客户端变得很重要。Chubby是本是计划用于在单个公司内部运行的，因此针对它的恶意的服务拒绝袭击是很罕见的。然而，错误、误解和开发者不一样的预期都导致了类似于袭击的效应。

我们的某些解决方式是很粗暴的。例如，我们检查项目团队计划使用Chubby的方式，并在检查使人满意之前拒绝其访问共享的Chubby名字空间。这种方式的一个问题是开发者经常不能预计他们的服务将来会被如何使用，以及使用量会怎么样增长。

我们的检查的最重要的方面是判断任何的Chubby的资源（RPC频率，硬盘空间，文件数目）是否会随着该项目的用户数量或处理的数据量线性（或者更坏）地增长。任何线性增长必须被由一个修正参数来调低，这个修正参数可以调整为降低Chubby上负载到一个合理边界的值。尽管如此，我们的早期检查仍然不彻底足够。

一个相关的问题是大部分软件文档中缺少性能建议。一个由某个团队编写的模块，可能在一年后被另一个团队复用，并带来极糟糕的后果。有时很难向接口设计者解释他们必须修改他们的接口，不是因为它们不好，而是因为其他开发者可能较少知道RPC的代价。

下面我们列出了我们遇到过的一些问题。

**缺少可应付冲击的缓存(aggressive caching)** 最初，我们没有意识到缓存不存在的文件和重用打开的文件句柄的关键性的需要。 尽管在培训时进行了尝试，我们的开发人员通常在一个文件不存在时，仍会编写无限重试的循环，或者通过重复地关闭/打开一个文件来轮询文件（实际上只需打开一次就可以）。

最初，我们对付这些重试循环的方法是，在某个应用程序短时间内做了许多Open()同一个文件的尝试时，引入指数级递增的延时。在某些情形下，这暴露了开发者接受了的漏洞，但是通常这需要我们花更多的时间做培训。最后，让重复性的Open()调用廉价会更容易一些。

**缺少限额** Chubby从没有被计划用做存放大量数据的存储系统，因此没有存储限额。事后反思，这是很弱智的。Google的某个项目写了一个跟踪数据上传的模块，存储了某些元数据到Chubby中。这种上传很少发生，并且只限于一小群人，因此使用的空间是有上限的。然而，两个其他服务开始使用该模块作为追踪来自于多得多的用户群的上传的手段。不可避免地，他们的服务一直增长，直到对Chubby的使用到达极限：一个1.5M字节的文件在每次用户动作时被整个重写，被这些服务使用的空间超出了其他所有Chubby客户端加起来的空间需求。

我们在文件大小上引入了一个限制（256KB），并推动这些服务迁移到更合适的存储系统上去。但要在这些由很繁忙的人们维护着的生产系统上要取得重大变化是很困难的，―-花了大约一年将这些数据迁移到其他地方。

**发布/订阅** 曾有数次将Chubby的事件机制作为Zephyr[6]式的发布/订阅系统的尝试。Chubby的超重量级的保证和它在维护缓存一致性时使用过期（invalidation）而不是更新，使得它除了无意义的订阅/发布示例之外，都很慢、效率很低。幸运地是，所有这样的应用都在重新设计应用程序的成本变得太大以前被叫停(caught)。

**4.6 Lessons learned**

这里我们列出教训和多种如果有机会的话我们可能会做出的设计变更：

开发者极少考虑可用性。 我们发现开发者极少考虑失败的可能性，并倾向于认为像Chubby这样的服务好像会用于可用。例如，开发者曾经构建了一个用了几百台机器的系统，这个系统在Chubby选举出一个新的Master时就发起一个持续几十分钟的恢复处理过程。这将一个单一故障的后果放大到受影响的机器数与时间的乘积的倍数。我们偏好开发者为短时间的Chubby不可用做好计划，以便这样的事件对他们的应用只有很少影响，甚至没有影响。这是粗粒度锁定的一个有争议的地方，在第2.1节讨论了这个问题。

开发者同样未能成功地意识到服务上线和服务对他们的应用可用的之间的区别。例如，全局Chubby单元(参见 $2.12)基本上总是在线的，因为很少会有两个物理距离遥远的数据中心同时下线。然后，对于某个客户端而言，它被观测到的可用性通常低于这个客户端的观测到的本地Chubby单元的可用性。首先，本地单元较少可能与客户端之间发生网络断开，另外，尽管本地Chubby单元可能会因为维护操作而下线，但同样的维护操作也会直接影响到客户端，因此Chubby的不可用就不会被客户端看到。

我们的API选择同样能影响到开发者处理Chubby不可用的方式。例如，Chubby提供了一个时间允许客户端检测什么时候发生了master的故障切换。本来这是用于客户端检查可能的变化的，因为别的时间可能丢失了。不幸的是，许多开发选择在收到这个事件是关闭他们的程序，因此大幅降低了他们系统的可用性。我们本来有可能通过给“文件改变(file-change)”事件以做得更好，甚或可以保证在故障切换期间不丢失事件。

目前我们用了三种机制防止开发者对Chubby的可用性过分乐观，特别是对全局单元(Global cell) 的可用性过分乐观。首先，如前文提到的那样，我们复查各个项目团队打算如如何使用Chubby，并建议它们不要使用可能将它们的可用性与Chubby的可用性绑定的太紧密。第二，现在我们提供了执行某些高层次任务的库，因此开发者被自动地与Chubby中断隔离。第三，我们利用每次Chubby中断的事后分析作为一种手段，不仅清除Chubby和运维过程中的缺陷(bugs)，还降低应用程序对Chubby的可用性的敏感性 ― 两方面都带来系统的整体上更好的可用性。

差劲的API选择导致了不可预料的后果 对于大部分而言，我们的API演化得很好，但是有一个错误很突出。我们的取消长期运行（long-running）的调用是Close()和Poison() RPC，它们也会丢弃服务器的分配给对应句柄的状态 。这阻止了能请求锁的句柄被 共享，例如，被多个线程共享。我们可能会添加一个 Cancel() RPC以允许更多打开的句柄的共享。

RPC的使用影响了传输协议 KeepAlive被用于刷新客户端的会话租期，也被用于从master端传递事件和缓存过期到客户端。这个设计有自动的符合预期的效应：一个客户端不能在没有应答缓存过期的情况下刷新会话状态。

这似乎很理想，除了这样会在传输协议的选择中引入紧张情形以外。TCP的拥塞时回退(back off)策略不关心更高层的超时，例如Chubby的租期，因此基于TCP的KeepAlive在发生高网络拥塞时导致许多丢失的会话。我们被迫通过UDP而不是TCP发送KeepAlive RPC调用；UDP没有拥塞回避机制，因此我们只有当高层的时间界限必须满足时才使用UDP协议。

我们可能增加一个基于TCP的GetEvent() RPC来增强传输协议，这个RPC将用于在正常情形下传递事件和过期，它与KeepAlives的方式相同。KeepAlive回复仍将包含一个未应答事件列表，因而事件最终都将被应答。

**5 与相关工作的比较 (Comparison with related work)**

Chubby是基于长期稳定的思想（well-established ideas）之上的。Chubby的缓存设计源自于分布式文件系统相关的成果[10]。它的会话和缓存标记(tokens)在行为上与Echo的相应部分类似[17]；会话降低租期压力与V 系统(V System)类似。放出一个一般性的（general-purpose）锁服务的理念最早出现在VMS[23]中，尽管该系统最初使用了一个特殊目的的许可低延迟交互的高速联网系统。 像它的缓存模型，Chubby的API是建立在文件系统模型上的，其中包括类似于文件系统的名字空间要比单纯的文件要方便很多的思想。 [18, 21, 22] （原文：Like its caching model, Chubby’s API is based on a file-system model, including the idea that a filesystem-like name space is convenient for more than just files [18, 21, 22].）

Chubby区别于一个像Echo或者AFS[10]这样的文件系统，主要表现在在它的性能和存储意图上：客户端不读、写或存储大量数据，并且他们不期待很高的吞吐量，甚至如果数据不缓存的话也不预期低延时。它们却期待一致性，可用性和可靠性，但是这些属性在性能不那么重要是比较容易达成。因为Chubby的数据库很小，我们能在线存储它的多个拷贝（通常是五个副本和少数备份）。我们每天做很多次完整备份，并通过数据库状态的校验码（checksums），我们每隔几个小时互相比较副本。对常规文件系统的性能和存储需求的弱化（weakening）允许我们用一个Chubby master服务数千个客户端。我们通过提供一个中心点，许多客户端在这个中心点共享信息和协同活动，解决了一类我们的系统开发人员面对的问题。

各种文献（literature）描述了大量的文件系统和锁服务器，所以不可能做一个彻底的比较。于是我们选择了其中一个做详细比较：我们选择与Boxwood的锁服务***[16](javascript:void()" \t "_blank)***比较，因它是最近设计的，并且设计为运行在松耦合环境下，另外它的设计在多个方面与Chubby不同，某些很有趣，某些是本质上的。（原文：some interesting and some incidental）

Chubby 实现了锁，一个可靠的小文件存储系统，以及在单个服务中的会话/租期机制。相反，Boxwood将这些划分为三部分：锁服务，Paxos服务（一个可靠的状态存储库），以及对应的失败检测服务。 Boxwood系统它本身使用了这三个组件，但是另一个系统可能独立地使用这些构建块。我们怀疑设计上的这个区别源自不同的目标客户群体(target audience)。Chubby被计划用于多种多样的目标客户（audience） 和应用的混合体， 它的用户有创建新分布式系统的专家，也有编写管理脚本的新手。对于我们的环境，一个提供熟知的API的大规模（large-scale）的共享服务似乎更有吸引力。相反，Boxwood提供了一个工具包（至少我们看到的是这样），这个工具包适合于一小部分经验丰富、老练的开发者用在那些可能共享代码但不需要一起使用的项目中。

In many cases, Chubby provides a higher-level interface than Boxwood. For example, Chubby combines the lock and file names spaces, while Boxwood’s lock names are simple byte sequences. Chubby clients cache file state by default; a client of Boxwood’s Paxos service could implement caching via the lock service, but would probably use the caching provided by Boxwood itself.

The two systems have markedly different default parameters, chosen for different expectations: Each Boxwood failure detector is contacted by each client every 200ms with a timeout of 1s; Chubby’s default lease time is 12s and KeepAlives are exchanged every 7s. Boxwood’s subcomponents use two or three replicas to achieve availability, while we typically use five replicas per cell. However, these choices alone do not suggest a deep design difference, but rather an indication of how parameters in such systems must be adjusted to accommodate more client machines, or the uncertainties of racks shared with other projects.

A more interesting difference is the introduction of Chubby’s grace period, which Boxwood lacks. (Recall  
that the grace period allows clients to ride out long Chubby master outages without losing sessions or locks.  
Boxwood’s “grace period” is the equivalent of Chubby’s “session lease”, a different concept.) Again, this difference is the result of differing expectations about scale and failure probability in the two systems. Although master fail-overs are rare, a lost Chubby lock is expensive for clients.

Finally, locks in the two systems are intended for different purposes. Chubby locks are heavier-weight, and  
need sequencers to allow externals resources to be protected safely, while Boxwood locks are lighter-weight,  
and intended primarily for use within Boxwood.

**6. 总结**

Chubby is a distributed lock service intended for coarsegrained synchronization of activities within Google’s  
distributed systems; it has found wider use as a name service and repository for configuration information.

Its design is based on well-known ideas that have meshed well: distributed consensus among a few replicas  
for fault tolerance, consistent client-side caching to reduce server load while retaining simple semantics, timely notification of updates, and a familiar file system interface. We use caching, protocol-conversion servers, and simple load adaptation to allow it scale to tens of thousands of client processes per Chubby instance. We expect to scale it further via proxies and partitioning.

Chubby has become Google’s primary internal name service; it is a common rendezvous mechanism for systems  
such as MapReduce [4]; the storage systems GFS and Bigtable use Chubby to elect a primary from redundant  
replicas; and it is a standard repository for files that require high availability, such as access control lists.

**7 致谢**

Many contributed to the Chubby system: Sharon Perl wrote the replication layer on Berkeley DB; Tushar  
Chandra and Robert Griesemer wrote the replicated database that replaced Berkeley DB; Ramsey Haddad  
connected the API to Google’s file system interface; Dave Presotto, Sean Owen, Doug Zongker and Praveen  
Tamara wrote the Chubby DNS, Java, and naming protocol-converters, and the full Chubby proxy respectively; Vadim Furman added the caching of open handles and file-absence; Rob Pike, Sean Quinlan and Sanjay Ghemawat gave valuable design advice; and many Google developers uncovered early weaknesses.

**参考文献**

[1] BIRMAN, K. P., AND JOSEPH, T. A. Exploiting virtual synchrony in distributed systems. In 11th SOSP (1987), pp. 123�138.  
[2] BIRRELL, A., JONES, M. B., AND WOBBER, E. A simple and efficient implementation for small databases. In 11th SOSP (1987), pp. 149�154.  
[3] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. Bigtable: A distributed structured data storage system. In 7th OSDI (2006).  
[4] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In 6th OSDI 2004), pp. 137�150.  
[5] FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. Impossibility of distributed consensus with one faulty process. J. ACM 32, 2 (April 1985), 374�382.  
[6] FRENCH, R. S., AND KOHL, J. T. The Zephyr Programmer’s Manual. MIT Project Athena, Apr. 1989.  
[7] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. In 19th SOSP (Dec. 2003), pp. 29�43.  
[8] GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. Java Language Spec. (2nd Ed.). Addison-Wesley, 2000.  
[9] GRAY, C. G., AND CHERITON, D. R. Leases: An efficient fault-tolerant mechanism for distributed file cache  
consistency. In 12th SOSP (1989), pp. 202�210.  
[10] HOWARD, J., KAZAR, M., MENEES, S., NICHOLS, D., SATYANARAYANAN, M., SIDEBOTHAM, R., AND WEST, M. Scale and performance in a distributed file system. ACM TOCS 6, 1 (Feb. 1988), 51�81.  
[11] JEFFERSON, D. Virtual time. ACM TOPLAS, 3 (1985), 404�425.  
[12] LAMPORT, L. The part-time parliament. ACM TOCS 16, 2 (1998), 133�169.  
[13] LAMPORT, L. Paxos made simple. ACM SIGACT News 32, 4 (2001), 18�25.  
[14] LAMPSON, B. W. How to build a highly available system using consensus. In Distributed Algorithms, vol. 1151 of LNCS. Springer�Verlag, 1996, pp. 1�17.  
[15] LIANG, S. Java Native Interface: Programmer’s Guide and Reference. Addison-Wesley, 1999.  
[16] MACCORMICK, J., MURPHY, N., NAJORK, M., THEKKATH, C. A., AND ZHOU, L. Boxwood: Abstractions as the foundation for storage infrastructure. In 6th OSDI (2004), pp. 105�120.  
[17] MANN, T., BIRRELL, A., HISGEN, A., JERIAN, C., AND SWART, G. A coherent distributed file cache with directory write-behind. TOCS 12, 2 (1994), 123�164.  
[18] MCJONES, P., AND SWART, G. Evolving the UNIX system interface to support multithreaded programs. Tech. Rep. 21, DEC SRC, 1987.  
[19] OKI, B., AND LISKOV, B. Viewstamped replication: A general primary copy method to support highly-available distributed systems. In ACM PODC (1988).  
[20] OLSON, M. A., BOSTIC, K., AND SELTZER, M. Berkeley DB. In USENIX (June 1999), pp. 183�192.  
[21] PIKE, R., PRESOTTO, D. L., DORWARD, S., FLANDRENA, B., THOMPSON, K., TRICKEY, H., AND WINTERBOTTOM, P. Plan 9 from Bell Labs. Computing Systems 8, 2 (1995), 221�254.  
[22] RITCHIE, D. M., AND THOMPSON, K. The UNIX timesharing system. CACM 17, 7 (1974), 365�375.  
[23] SNAMAN, JR., W. E., AND THIEL, D. W. The VAX/VMS distributed lock manager. Digital Technical Journal 1, 5 (Sept. 1987), 29�44.  
[24] YIN, J., MARTIN, J.-P., VENKATARAMANI, A., ALVISI, L., AND DAHLIN, M. Separating agreement from execution for byzantine fault tolerant services. In 19th SOSP (2003), pp. 253�267.

======================

**术语**

cell Chubby 单元Chubby cell Chubby 单元session 会话bug 缺陷antipodes 洲际section $4.1 [P11]:but 250msbetweenantipodes.lock 锁locking 锁定