The Chubby lock service for loosely-coupled distributed systems

Mike Burrows, *Google Inc*.

Abstract

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.

1 Introduction

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.

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.

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.

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.

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* 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].

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.

2 Design

2.1 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.

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.

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.

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.

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.

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.

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.

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.

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.

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.)

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.

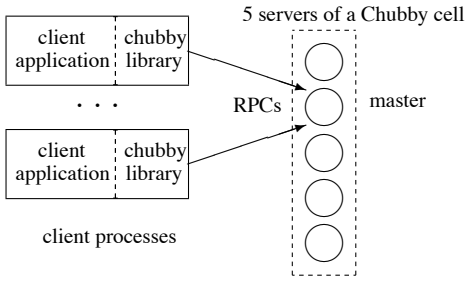

Figure 1: System structure

2.2 System structure

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.

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.

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.

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).

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.

2.3 Files, directories, and handles

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
```

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.

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.

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.

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.

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.

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:

- 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 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.

2.4 Locks and sequencers

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:

- 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.

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

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.

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 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.

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.

2.5 Events

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.

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.

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 modifi-

cation 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.

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()`.

`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.

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:

- 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.
- 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.

`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:

`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.

`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.

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

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

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

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

`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).

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.

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).

2.7 Caching

To reduce read traffic, Chubby clients cache file data and node meta-data (including file absence) in a consistent, write-through cache held in memory. The cache is maintained by a lease mechanism described below, and kept consistent by invalidations sent by the master, which keeps a list of what each client may be caching. The protocol ensures that clients see either a consistent view of Chubby state, or an error.

When file data or meta-data is to be changed, the modification is blocked while the master sends invalidations for the data to every client that may have cached it; this mechanism sits on top of KeepAlive RPCs, discussed more fully in the next section. On receipt of an invalidation, a client flushes the invalidated state and acknowl-

edges by making its next KeepAlive call. The modification proceeds only after the server knows that each client has invalidated its cache, either because the client acknowledged the invalidation, or because the client allowed its cache lease to expire.

Only one round of invalidations is needed because the master treats the node as *uncachable* while cache invalidations remain unacknowledged. This approach allows reads always to be processed without delay; this is useful because reads greatly outnumber writes. An alternative would be to block calls that access the node during invalidation; this would make it less likely that over-eager clients will bombard the master with uncached accesses during invalidation, at the cost of occasional delays. If this were a problem, one could imagine adopting a hybrid scheme that switched tactics if overload were detected.

The caching protocol is simple: it invalidates cached data on a change, and never updates it. It would be just as simple to update rather than to invalidate, but update-only protocols can be arbitrarily inefficient; a client that accessed a file might receive updates indefinitely, causing an unbounded number of unnecessary updates.

Despite the overheads of providing strict consistency, we rejected weaker models because we felt that programmers would find them harder to use. Similarly, mechanisms such as virtual synchrony that require clients to exchange sequence numbers in all messages were considered inappropriate in an environment with diverse pre-existing communication protocols.

In addition to caching data and meta-data, Chubby clients cache open handles. Thus, if a client opens a file it has opened previously, only the first `Open()` call necessarily leads to an RPC to the master. This caching is restricted in minor ways so that it never affects the semantics observed by the client: handles on ephemeral files cannot be held open if the application has closed them; and handles that permit locking can be reused, but cannot be used concurrently by multiple application handles. This last restriction exists because the client may use `Close()` or `Poison()` for their side-effect of cancelling outstanding `Acquire()` calls to the master.

Chubby's protocol permits clients to cache locks—that is, to hold locks longer than strictly necessary in the hope that they can be used again by the same client. An event informs a lock holder if another client has requested a conflicting lock, allowing the holder to release the lock just when it is needed elsewhere (see §2.5).

2.8 Sessions and KeepAlives

A Chubby session is a relationship between a Chubby cell and a Chubby client; it exists for some interval of time, and is maintained by periodic handshakes called KeepAlives. Unless a Chubby client informs the master otherwise, the client's handles, locks, and cached data all remain valid provided its session remains valid. (However, the protocol for session maintenance may require the client to acknowledge a cache invalidation in order to maintain its session; see below.)

A client requests a new session on first contacting the master of a Chubby cell. It ends the session explicitly either when it terminates, or if the session has been idle (with no open handles and no calls for a minute).

Each session has an associated lease—an interval of time extending into the future during which the master guarantees not to terminate the session unilaterally. The end of this interval is called the session lease timeout. The master is free to advance this timeout further into the future, but may not move it backwards in time.

The master advances the lease timeout in three circumstances: on creation of the session, when a master fail-over occurs (see below), and when it responds to a KeepAlive RPC from the client. On receiving a KeepAlive, the master typically blocks the RPC (does not allow it to return) until the client's previous lease interval is close to expiring. The master later allows the RPC to return to the client, and thus informs the client of the new lease timeout. The master may extend the timeout by any amount. The default extension is 12s, but an overloaded master may use higher values to reduce the number of KeepAlive calls it must process. The client initiates a new KeepAlive immediately after receiving the previous reply. Thus, the client ensures that there is almost always a KeepAlive call blocked at the master.

As well as extending the client's lease, the KeepAlive reply is used to transmit events and cache invalidations back to the client. The master allows a KeepAlive to return early when an event or invalidation is to be delivered. Piggybacking events on KeepAlive replies ensures that clients cannot maintain a session without acknowledging cache invalidations, and causes all Chubby RPCs to flow from client to master. This simplifies the client, and allows the protocol to operate through firewalls that allow initiation of connections in only one direction.

The client maintains a local lease timeout that is a conservative approximation of the master's lease timeout. It differs from the master's lease timeout because the client must make conservative assumptions both of the time its KeepAlive reply spent in flight, and the rate at which the master's clock is advancing; to maintain consistency, we require that the server's clock advance no faster than a known constant factor faster than the client's.

If a client's local lease timeout expires, it becomes unsure whether the master has terminated its session. The client empties and disables its cache, and we say that its session is in *jeopardy*. The client waits a further interval called the grace period, 45s by default. If the client and master manage to exchange a successful KeepAlive be-

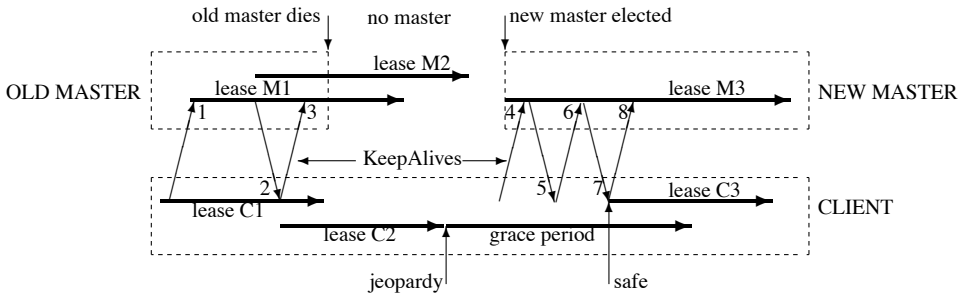

Figure 2: The role of the grace period in master fail-over

fore the end of the client's grace period, the client enables its cache once more. Otherwise, the client assumes that the session has expired. This is done so that Chubby API calls do not block indefinitely when a Chubby cell becomes inaccessible; calls return with an error if the grace period ends before communication is re-established.

The Chubby library can inform the application when the grace period begins via a *jeopardy* event. When the session is known to have survived the communications problem, a *safe* event tells the client to proceed; if the session times out instead, an *expired* event is sent. 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. This can be important in avoiding outages in services with large startup overhead.

If a client holds a handle H on a node and any operation on H fails because the associated session has expired, all subsequent operations on H (except `Close()` and `Poison()`) will fail in the same way. Clients can use this to guarantee that network and server outages cause only a suffix of a sequence of operations to be lost, rather than an arbitrary subsequence, thus allowing complex changes to be marked as committed with a final write.

2.9 Fail-overs

When a master fails or otherwise loses mastership, it discards its in-memory state about sessions, handles, and locks. The authoritative timer for session leases runs at the master, so until a new master is elected the session lease timer is stopped; this is legal because it is equivalent to extending the client's lease. If a master election occurs quickly, clients can contact the new master before their local (approximate) lease timers expire. If the election takes a long time, clients flush their caches and wait for the grace period while trying to find the new master. Thus the grace period allows sessions to be maintained across fail-overs that exceed the normal lease timeout.

Figure 2 shows the sequence of events in a lengthy master fail-over event in which the client must use its grace period to preserve its session. Time increases from left to right, but times are not to scale. Client session leases are shown as thick arrows both as viewed by both the old and new masters (M1-3, above) and the client (C1-3, below). Upward angled arrows indicate KeepAlive requests, and downward angled arrows their replies. The original master has session lease M1 for the client, while the client has a conservative approximation C1. The master commits to lease M2 before informing the client via KeepAlive reply 2; the client is able to extend its view of the lease C2. The master dies before replying to the next KeepAlive, and some time elapses before another master is elected. Eventually the client's approximation of its lease (C2) expires. The client then flushes its cache and starts a timer for the grace period.

During this period, the client cannot be sure whether its lease has expired at the master. It does not tear down its session, but it blocks all application calls on its API to prevent the application from observing inconsistent data. At the start of the grace period, the Chubby library sends a *jeopardy* event to the application to allow it to quiesce itself until it can be sure of the status of its session.

Eventually a new master election succeeds. The master initially uses a conservative approximation M3 of the session lease that its predecessor may have had for the client. The first KeepAlive request (4) from the client to the new master is rejected because it has the wrong master epoch number (described in detail below). The retried request (6) succeeds but typically does not extend the master lease further because M3 was conservative. However the reply (7) allows the client to extend its lease (C3) once more, and optionally inform the application that its session is no longer in jeopardy. Because the grace period was long enough to cover the interval between the end of lease C2 and the beginning of lease C3, the client saw nothing but a delay. Had the grace period been less than that interval, the client would have abandoned the session and reported the failure to the application.

Once a client has contacted the new master, the client library and master co-operate to provide the illusion to the application that no failure has occurred. To achieve this, the new master must reconstruct a conservative approximation of the in-memory state that the previous master had. It does this partly by reading data stored stably on disc (replicated via the normal database repli-

cation protocol), partly by obtaining state from clients, and partly by conservative assumptions. The database records each session, held lock, and ephemeral file.

A newly elected master proceeds:

1. It first picks a new client *epoch number*, which clients are required to present on every call. The master rejects calls from clients using older epoch numbers, and provides the new epoch number. This ensures that the new master will not respond to a very old packet that was sent to a previous master, even one running on the same machine.
2. The new master may respond to master-location requests, but does not at first process incoming session-related operations.
3. It builds in-memory data structures for sessions and locks that are recorded in the database. Session leases are extended to the maximum that the previous master may have been using.
4. The master now lets clients perform KeepAlives, but no other session-related operations.
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.
6. The master waits until each session acknowledges the fail-over event or lets its session expire.
7. The master allows all operations to proceed.
8. If a client uses a handle created prior to the fail-over (determined from the value of a sequence number in the handle), the master recreates the in-memory representation of the handle and honours the call. If such a recreated handle is closed, the master records it in memory so that it cannot be recreated in this master epoch; this ensures that a delayed or duplicated network packet cannot accidentally recreate a closed handle. A faulty client can recreate a closed handle in a future epoch, but this is harmless given that the client is already faulty.
9. After some interval (a minute, say), the master deletes ephemeral files that have no open file handles. Clients should refresh handles on ephemeral files during this interval after a fail-over. This mechanism has the unfortunate effect that ephemeral files may not disappear promptly if the last client on such a file loses its session during a fail-over.

Readers will be unsurprised to learn that the fail-over code, which is exercised far less often than other parts of the system, has been a rich source of interesting bugs.

2.10 Database implementation

The first version of Chubby used the replicated version of Berkeley DB [20] as its database. Berkeley DB provides B-trees that map byte-string keys to arbitrary byte-string values. We installed a key comparison function that sorts first by the number of components in a path name; this allows nodes to by keyed by their path name, while keeping sibling nodes adjacent in the sort order. Because Chubby does not use path-based permissions, a single lookup in the database suffices for each file access.

Berkeley DB's uses a distributed consensus protocol to replicate its database logs over a set of servers. Once master leases were added, this matched the design of Chubby, which made implementation straightforward.

While Berkeley DB's B-tree code is widely-used and mature, the replication code was added recently, and has fewer users. Software maintainers must give priority to maintaining and improving their most popular product features. While Berkeley DB's maintainers solved the problems we had, we felt that use of the replication code exposed us to more risk than we wished to take. As a result, we have written a simple database using write ahead logging and snapshotting similar to the design of Birrell *et al*. [2]. As before, the database log is distributed among the replicas using a distributed consensus protocol. Chubby used few of the features of Berkeley DB, and so this rewrite allowed significant simplification of the system as a whole; for example, while we needed atomic operations, we did not need general transactions.

2.11 Backup

Every few hours, the master of each Chubby cell writes a snapshot of its database to a GFS file server [7] in a different building. The use of a separate building ensures both that the backup will survive building damage, and that the backups introduce no cyclic dependencies in the system; a GFS cell in the same building potentially might rely on the Chubby cell for electing its master.

Backups provide both disaster recovery and a means for initializing the database of a newly replaced replica without placing load on replicas that are in service.

2.12 Mirroring

Chubby allows a collection of files to be mirrored from one cell to another. Mirroring is fast because the files are small and the event mechanism (§2.5) informs the mirroring code immediately if a file is added, deleted, or modified. Provided there are no network problems, changes are reflected in dozens of mirrors world-wide in well under a second. If a mirror is unreachable, it remains unchanged until connectivity is restored. Updated files are then identified by comparing their checksums.

Mirroring is used most commonly to copy configuration files to various computing clusters distributed around the world. A special cell, named `global`, contains a subtree `/ls/global/master` that is mirrored to the

subtree `/ls/`*cell*`/slave` in every other Chubby cell. The global cell is special because its five replicas are located in widely-separated parts of the world, so it is almost always accessible from most of the organization.

Among the files mirrored from the global cell are Chubby's own access control lists, various files in which Chubby cells and other systems advertise their presence to our monitoring services, pointers to allow clients to locate large data sets such as Bigtable cells, and many configuration files for other systems.

3 Mechanisms for scaling

Chubby's clients are individual processes, so Chubby must handle more clients than one might expect; we have seen 90,000 clients communicating directly with a Chubby master—far more than the number of machines involved. Because there is just one master per cell, and its machine is identical to those of the clients, the clients can overwhelm the master by a huge margin. Thus, the most effective scaling techniques reduce communication with the master by a significant factor. Assuming the master has no serious performance bug, minor improvements in request processing at the master have little effect. We use several approaches:

- We can create an arbitrary number of Chubby cells; clients almost always use a nearby cell (found with DNS) to avoid reliance on remote machines. Our typical deployment uses one Chubby cell for a data centre of several thousand machines.
- The master may increase lease times from the default 12s up to around 60s when it is under heavy load, so it need process fewer KeepAlive RPCs. (KeepAlives are *by far* the dominant type of request (see 4.1), and failure to process them in time is the typical failure mode of an overloaded server; clients are largely insensitive to latency variation in other calls.)
- Chubby clients cache file data, meta-data, the absence of files, and open handles to reduce the number of calls they make on the server.
- We use protocol-conversion servers that translate the Chubby protocol into less-complex protocols such as DNS and others. We discuss some of these below.

Here we describe two familiar mechanisms, proxies and partitioning, that we expect will allow Chubby to scale further. We do not yet use them in production, but they are designed, and may be used soon. We have no present need to consider scaling beyond a factor of five: First, there are limits on the number of machines one would wish to put in a data centre or make reliant on a single instance of a service. Second, because we use similar machines for Chubby clients and servers, hardware improvements that increase the number of clients per machine also increase the capacity of each server.

3.1 Proxies

Chubby's protocol can be proxied (using the same protocol on both sides) by trusted processes that pass requests from other clients to a Chubby cell. A proxy can reduce server load by handling both KeepAlive and read requests; it cannot reduce write traffic, which passes through the proxy's cache. But even with aggressive client caching, write traffic constitutes much less than one percent of Chubby's normal workload (see §4.1), so proxies allow a significant increase in the number of clients. If a proxy handles N_{proxy} clients, KeepAlive traffic is reduced by a factor of N_{proxy}, which might be 10 thousand or more. A proxy cache can reduce read traffic by at most the mean amount of read-sharing—a factor of around 10 (§4.1). But because reads constitute under 10% of Chubby's load at present, the saving in KeepAlive traffic is by far the more important effect.

Proxies add an additional RPC to writes and first-time reads. One might expect proxies to make the cell temporarily unavailable at least twice as often as before, because each proxied client depends on two machines that may fail: its proxy and the Chubby master.

Alert readers will notice that the fail-over strategy described in Section 2.9, is not ideal for proxies. We discuss this problem in Section 4.4.

3.2 Partitioning

As mentioned in Section 2.3, Chubby's interface was chosen so that the name space of a cell could be partitioned between servers. Although we have not yet needed it, the code can partition the name space by directory. If enabled, a Chubby cell would be composed of N *partitions*, each of which has a set of replicas and a master. Every node D/C in directory D would be stored on the partition $P(D/C) = \texttt{hash}(D) \bmod N$. Note that the meta-data for D may be stored on a different partition $P(D) = \texttt{hash}(D') \bmod N$, where D' is the parent of D.

Partitioning is intended to enable large Chubby cells with little communication between the partitions. Although Chubby lacks hard links, directory modified-times, and cross-directory rename operations, a few operations still require cross-partition communication:

- ACLs are themselves files, so one partition may use another for permissions checks. However, ACL files are readily cached; only `Open()` and `Delete()` calls require ACL checks; and most clients read publicly accessible files that require no ACL.
- When a directory is deleted, a cross-partition call may be needed to ensure that the directory is empty.

Because each partition handles most calls independently of the others, we expect this communication to have only a modest impact on performance or availability.

Unless the number of partitions N is large, one would expect that each client would contact the majority of the partitions. Thus, partitioning reduces read and write traffic on any given partition by a factor of N but does not necessarily reduce KeepAlive traffic. Should it be necessary for Chubby to handle more clients, our strategy involves a combination of proxies and partitioning.

4 Use, surprises and design errors

4.1 Use and behaviour

The following table gives statistics taken as a snapshot of a Chubby cell; the RPC rate was a seen over a ten-minute period. The numbers are typical of cells in Google.

time since last fail-over	18 days
fail-over duration	14s
active clients (direct)	22k
additional proxied clients	32k
files open	12k
naming-related	60%
client-is-caching-file entries	230k
distinct files cached	24k
names negatively cached	32k
exclusive locks	1k
shared locks	0
stored directories	8k
ephemeral	0.1%
stored files	22k
0-1k bytes	90%
1k-10k bytes	10%
$>$ 10k bytes	0.2%
naming-related	46%
mirrored ACLs & config info	27%
GFS and Bigtable meta-data	11%
ephemeral	3%
RPC rate	1-2k/s
KeepAlive	93%
GetStat	2%
Open	1%
CreateSession	1%
GetContentsAndStat	0.4%
SetContents	680ppm
Acquire	31ppm

Several things can be seen:

- Many files are used for naming; see §4.3.
- Configuration, access control, and meta-data files (analogous to file system super-blocks) are common.
- Negative caching is significant.
- 230k/24k≈10 clients use each cached file, on average.
- Few clients hold locks, and shared locks are rare; this is consistent with locking being used for primary election and partitioning data among replicas.
- RPC traffic is dominated by session KeepAlives; there are a few reads (which are cache misses); there are very few writes or lock acquisitions.

Now we briefly describe the typical causes of outages in our cells. If we assume (optimistically) that a cell is "up" if it has a master that is willing to serve, on a sample of our cells we recorded 61 outages over a period of a few weeks, amounting to 700 cell-days of data in total. We excluded outages due to maintenance that shut down the data centre. All other causes are included: network congestion, maintenance, overload, and errors due to operators, software, and hardware. Most outages were 15s or less, and 52 were under 30s; most of our applications are not affected significantly by Chubby outages under 30s. The remaining nine outages were caused by network maintenance (4), suspected network connectivity problems (2), software errors (2), and overload (1).

In a few dozen cell-years of operation, we have lost data on six occasions, due to database software errors (4) and operator error (2); none involved hardware error. Ironically, the operational errors involved upgrades to avoid the software errors. We have twice corrected corruptions caused by software in non-master replicas.

Chubby's data fits in RAM, so most operations are cheap. Mean request latency at our production servers is consistently a small fraction of a millisecond regardless of cell load until the cell approaches overload, when latency increases dramatically and sessions are dropped. Overload typically occurs when many sessions ($> 90,000$) are active, but can result from exceptional conditions: when clients made millions of read requests simultaneously (described in Section 4.3), and when a mistake in the client library disabled caching for some reads, resulting in tens of thousands of requests per second. Because most RPCs are KeepAlives, the server can maintain a low mean request latency with many active clients by increasing the session lease period (see §3). Group commit reduces the effective work done per request when bursts of writes arrive, but this is rare.

RPC read latencies measured at the client are limited by the RPC system and network; they are under 1ms for a local cell, but 250ms between antipodes. Writes (which include lock operations) are delayed a further 5-10ms by the database log update, but by up to tens of seconds if a recently-failed client cached the file. Even this variability in write latency has little effect on the mean request latency at the server because writes are so infrequent.

Clients are fairly insensitive to latency variation provided sessions are not dropped. At one point, we added artificial delays in `Open()` to curb abusive clients (see §4.5); developers noticed only when delays exceeded ten seconds *and* were applied repeatedly. We have found that the key to scaling Chubby is not server performance; reducing communication to the server can have far greater

impact. No significant effort has been applied to tuning read/write server code paths; we checked that no egregious bugs were present, then focused on the scaling mechanisms that could be more effective. On the other hand, developers do notice if a performance bug affects the local Chubby cache, which a client may read thousands of times per second.

4.2 Java clients

Google's infrastructure is mostly in C++, but a growing number of systems are being written in Java [8]. This trend presented an unanticipated problem for Chubby, which has a complex client protocol and a non-trivial client-side library.

Java encourages portability of entire applications at the expense of incremental adoption by making it somewhat irksome to link against other languages. The usual Java mechanism for accessing non-native libraries is JNI [15], but it is regarded as slow and cumbersome. Our Java programmers so dislike JNI that to avoid its use they prefer to translate large libraries into Java, and commit to supporting them.

Chubby's C++ client library is 7000 lines (comparable with the server), and the client protocol is delicate. To maintain the library in Java would require care and expense, while an implementation without caching would burden the Chubby servers. Thus our Java users run copies of a protocol-conversion server that exports a simple RPC protocol that corresponds closely to Chubby's client API. Even with hindsight, it is not obvious how we might have avoided the cost of writing, running and maintaining this additional server.

4.3 Use as a name service

Even though Chubby was designed as a lock service, we found that its most popular use was as a name server.

Caching within the normal Internet naming system, the DNS, is based on time. DNS entries have a *time-to-live* (TTL), and DNS data are discarded when they have not been refreshed within that period. Usually it is straightforward to pick a suitable TTL value, but if prompt replacement of failed services is desired, the TTL can become small enough to overload the DNS servers.

For example, it is common for our developers to run jobs involving thousands of processes, and for each process to communicate with every other, leading to a quadratic number of DNS lookups. We might wish to use a TTL of 60s; this would allow misbehaving clients to be replaced without excessive delay and is not considered an unreasonably short replacement time in our environment. In that case, to maintain the DNS caches of a single job as small as 3 thousand clients would require 150 thousand lookups per second. (For comparison, a 2-CPU 2.6GHz Xeon DNS server might handle 50 thousand requests per second.) Larger jobs create worse problems, and several jobs many be running at once. The variability in our DNS load had been a serious problem for Google before Chubby was introduced.

In contrast, Chubby's caching uses explicit invalidations so a constant rate of session KeepAlive requests can maintain an arbitrary number of cache entries indefinitely at a client, in the absence of changes. A 2-CPU 2.6GHz Xeon Chubby master has been seen to handle 90 thousand clients communicating directly with it (no proxies); the clients included large jobs with communication patterns of the kind described above. The ability to provide swift name updates without polling each name individually is so appealing that Chubby now provides name service for most of the company's systems.

Although Chubby's caching allows a single cell to sustain a large number of clients, load spikes can still be a problem. When we first deployed the Chubby-based name service, starting a 3 thousand process job (thus generating 9 million requests) could bring the Chubby master to its knees. To resolve this problem, we chose to group name entries into batches so that a single lookup would return and cache the name mappings for a large number (typically 100) of related processes within a job.

The caching semantics provided by Chubby are more precise than those needed by a name service; name resolution requires only timely notification rather than full consistency. As a result, there was an opportunity for reducing the load on Chubby by introducing a simple protocol-conversion server designed specifically for name lookups. Had we foreseen the use of Chubby as a name service, we might have chosen to implement full proxies sooner than we did in order to avoid the need for this simple, but nevertheless additional server.

One further protocol-conversion server exists: the Chubby DNS server. This makes the naming data stored within Chubby available to DNS clients. This server is important both for easing the transition from DNS names to Chubby names, and to accommodate existing applications that cannot be converted easily, such as browsers.

4.4 Problems with fail-over

The original design for master fail-over (§2.9) requires the master to write new sessions to the database as they are created. In the Berkeley DB version of the lock server, the overhead of creating sessions became a problem when many processes were started at once. To avoid overload, the server was modified to store a session in the database not when it was first created, but instead when it attempted its first modification, lock acquisition, or open

of an ephemeral file. In addition, active sessions were recorded in the database with some probability on each KeepAlive. Thus, the writes for read-only sessions were spread out in time.

Though it was necessary to avoid overload, this optimization has the undesirable effect that young read-only sessions may not be recorded in the database, and so may be discarded if a fail-over occurs. Although such sessions hold no locks, this is unsafe; if all the recorded sessions were to check in with the new master before the leases of discarded sessions expired, the discarded sessions could then read stale data for a while. This is rare in practice; in a large system it is almost certain that some session will fail to check in, and thus force the new master to await the maximum lease time anyway. Nevertheless, we have modified the fail-over design both to avoid this effect, and to avoid a complication that the current scheme introduces to proxies.

Under the new design, we avoid recording sessions in the database at all, and instead recreate them in the same way that the master currently recreates handles (§2.9,¶8). A new master must now wait a full worst-case lease timeout before allowing operations to proceed, since it cannot know whether all sessions have checked in (§2.9,¶6). Again, this has little effect in practice because it is likely that not all sessions will check in.

Once sessions can be recreated without on-disc state, proxy servers can manage sessions that the master is not aware of. An extra operation available only to proxies allows them to change the session that locks are associated with. This permits one proxy to take over a client from another when a proxy fails. The only further change needed at the master is a guarantee not to relinquish locks or ephemeral file handles associated with proxy sessions until a new proxy has had a chance to claim them.

4.5 Abusive clients

Google's project teams are free to set up their own Chubby cells, but doing so adds to their maintenance burden, and consumes additional hardware resources. Many services therefore use shared Chubby cells, which makes it important to isolate clients from the misbehaviour of others. Chubby is intended to operate within a single company, and so malicious denial-of-service attacks against it are rare. However, mistakes, misunderstandings, and the differing expectations of our developers lead to effects that are similar to attacks.

Some of our remedies are heavy-handed. For example, we review the ways project teams plan to use Chubby, and deny access to the shared Chubby name space until review is satisfactory. A problem with this approach is that developers are often unable to predict how their services will be used in the future, and how use will grow. Readers will note the irony of our own failure to predict how Chubby itself would be used.

The most important aspect of our review is to determine whether use of any of Chubby's resources (RPC rate, disc space, number of files) grows linearly (or worse) with number of users or amount of data processed by the project. Any linear growth must be mitigated by a compensating parameter that can be adjusted to reduce the load on Chubby to reasonable bounds. Nevertheless our early reviews were not thorough enough.

A related problem is the lack of performance advice in most software documentation. A module written by one team may be reused a year later by another team with disastrous results. It is sometimes hard to explain to interface designers that they must change their interfaces not because they are bad, but because other developers may be less aware of the cost of an RPC.

Below we list some problem cases we encountered.

Lack of aggressive caching Originally, we did not appreciate the critical need to cache the absence of files, nor to reuse open file handles. Despite attempts at education, our developers regularly write loops that retry indefinitely when a file is not present, or poll a file by opening it and closing it repeatedly when one might expect they would open the file just once.

At first we countered these retry-loops by introducing exponentially-increasing delays when an application made many attempts to `Open()` the same file over a short period. In some cases this exposed bugs that developers acknowledged, but often it required us to spend yet more time on education. In the end it was easier to make repeated `Open()` calls cheap.

Lack of quotas Chubby was never intended to be used as a storage system for large amounts of data, and so it has no storage quotas. In hindsight, this was naïve.

One of Google's projects wrote a module to keep track of data uploads, storing some meta-data in Chubby. Such uploads occurred rarely and were limited to a small set of people, so the space was bounded. However, two other services started using the same module as a means for tracking uploads from a wider population of users. Inevitably, these services grew until the use of Chubby was extreme: a single 1.5MByte file was being rewritten in its entirety on each user action, and the overall space used by the service exceeded the space needs of all other Chubby clients combined.

We introduced a limit on file size (256kBytes), and encouraged the services to migrate to more appropriate storage systems. But it is difficult to make significant changes to production systems maintained by busy people—it took approximately a year for the data to be migrated elsewhere.

Publish/subscribe There have been several attempts to use Chubby's event mechanism as a publish/subscribe

system in the style of Zephyr [6]. Chubby's heavyweight guarantees and its use of invalidation rather than update in maintaining cache consistency make it a slow and inefficient for all but the most trivial publish/subscribe examples. Fortunately, all such uses have been caught before the cost of redesigning the application was too large.

4.6 Lessons learned

Here we list lessons, and miscellaneous design changes we might make if we have the opportunity:

Developers rarely consider availability We find that our developers rarely think about failure probabilities, and are inclined to treat a service like Chubby as though it were always available. For example, our developers once built a system employing hundred of machines that initiated recovery procedures taking tens of minutes when Chubby elected a new master. This magnified the consequences of a single failure by a factor of a hundred both in time *and* the number of machines affected. We would prefer developers to plan for short Chubby outages, so that such an event has little or no affect on their applications. This is one of the arguments for coarse-grained locking, discussed in Section 2.1.

Developers also fail to appreciate the difference between a service being up, and that service being available to their applications. For example, the global Chubby cell (see §2.12), is almost always up because it is rare for more than two geographically distant data centres to be down simultaneously. However, its *observed availability for a given client* is usually lower than the observed availability of the client's local Chubby cell. First, the local cell is less likely to be partitioned from the client, and second, although the local cell may be down often due to maintenance, the same maintenance affects the client directly, so Chubby's unavailability is not observed by the client.

Our API choices can also affect the way developers chose to handle Chubby outages. For example, Chubby provides an event that allows clients to detect when a master fail-over has taken place. The intent was for clients to check for possible changes, as other events may have been lost. Unfortunately, many developers chose to crash their applications on receiving this event, thus decreasing the availability of their systems substantially. We might have done better to send redundant "file change" events instead, or even to ensure that no events were lost during a fail-over.

At present we use three mechanisms to prevent developers from being over-optimistic about Chubby availability, especially that of the global cell. First, as previously mentioned (§4.5), we review how project teams plan to use Chubby, and advise them against techniques that would tie their availability too closely to Chubby's. Second, we now supply libraries that perform some high-level tasks so that developers are automatically isolated from Chubby outages. Third, we use the post-mortem of each Chubby outage as a means not only of eliminating bugs in Chubby and our operational procedures, but of reducing the sensitivity of applications to Chubby's availability—both can lead to better availability of our systems overall.

Fine-grained locking could be ignored At the end of Section 2.1 we sketched a design for a server that clients could run to provide fine-grained locking. It is perhaps a surprise that so far we have not needed to write such a server; our developers typically find that to optimize their applications, they must remove unnecessary communication, and that often means finding a way to use coarse-grained locking.

Poor API choices have unexpected affects For the most part, our API has evolved well, but one mistake stands out. Our means for cancelling long-running calls are the `Close()` and `Poison()` RPCs, which also discard the server state for the handle. This prevents handles that can acquire locks from being shared, for example, by multiple threads. We may add a `Cancel()` RPC to allow more sharing of open handles.

RPC use affects transport protocols KeepAlives are used both for refreshing the client's session lease, and for passing events and cache invalidations from the master to the client. This design has the automatic and desirable consequence that a client cannot refresh its session lease without acknowledging cache invalidations.

This would seem ideal, except that it introduced a tension in our choice of protocol. TCP's back off policies pay no attention to higher-level timeouts such as Chubby leases, so TCP-based KeepAlives led to many lost sessions at times of high network congestion. We were forced to send KeepAlive RPCs via UDP rather than TCP; UDP has no congestion avoidance mechanisms, so we would prefer to use UDP only when high-level time-bounds must be met.

We may augment the protocol with an additional TCP-based `GetEvent()` RPC which would be used to communicate events and invalidations in the normal case, used in the same way KeepAlives. The KeepAlive reply would still contain a list of unacknowledged events so that events must eventually be acknowledged.

5 Comparison with related work

Chubby is based on well-established ideas. Chubby's cache design is derived from work on distributed file systems [10]. Its sessions and cache tokens are similar in behaviour to those in Echo [17]; sessions reduce the overhead of leases [9] in the V system. The idea of exposing a general-purpose lock service is found in VMS [23],

though that system initially used a special-purpose high-speed interconnect that permitted low-latency interactions. Like its caching model, Chubby's API is based on a file-system model, including the idea that a file-system-like name space is convenient for more than just files [18, 21, 22].

Chubby differs from a distributed file system such as Echo or AFS [10] in its performance and storage aspirations: Clients do not read, write, or store large amounts of data, and they do not expect high throughput or even low-latency unless the data is cached. They do expect consistency, availability, and reliability, but these attributes are easier to achieve when performance is less important. Because Chubby's database is small, we are able to store many copies of it on-line (typically five replicas and a few backups). We take full backups multiple times per day, and via checksums of the database state, we compare replicas with one another every few hours. The weakening of the normal file system performance and storage requirements allows us to serve tens of thousands of clients from a single Chubby master. By providing a central point where many clients can share information and co-ordinate activities, we have solved a class of problems faced by our system developers.

The large number of file systems and lock servers described in the literature prevents an exhaustive comparison, so we provide details on one: we chose to compare with Boxwood's lock server [16] because it was designed recently, it too is designed to run in a loosely-coupled environment, and yet its design differs in various ways from Chubby, some interesting and some incidental.

Chubby implements locks, a reliable small-file storage system, and a session/lease mechanism in a single service. In contrast, Boxwood separates these into three: a *lock service*, a *Paxos service* (a reliable repository for state), and a *failure detection service* respectively. The Boxwood system itself uses these three components together, but another system could use these building blocks independently. We suspect that this difference in design stems from a difference in target audience. Chubby was intended for a diverse audience and application mix; its users range from experts who create new distributed systems, to novices who write administration scripts. For our environment, a large-scale shared service with a familiar API seemed attractive. In contrast, Boxwood provides a toolkit that (to our eyes, at least) is appropriate for a smaller number of more sophisticated developers working on projects that may share code but need not be used together.

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 Summary

Chubby is a distributed lock service intended for coarse-grained 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 Acknowledgments

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.

References

[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 time-sharing 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.