Paxos Made Live - An Engineering Perspective

Tushar Chandra
Robert Griesemer
Joshua Redstone

June 20, 2007

Abstract

We describe our experience in building a fault-tolerant data-base using the Paxos consensus algorithm. Despite the existing literature in the field, building such a database proved to be non-trivial. We describe selected algorithmic and engineering problems encountered, and the solutions we found for them. Our measurements indicate that we have built a competitive system.

1 Introduction

It is well known that fault-tolerance on commodity hardware can be achieved through replication [17, 18]. A common approach is to use a consensus algorithm [7] to ensure that all replicas are mutually consistent [8, 14, 17]. By repeatedly applying such an algorithm on a sequence of input values, it is possible to build an identical log of values on each replica. If the values are operations on some data structure, application of the same log on all replicas may be used to arrive at mutually consistent data structures on all replicas. For instance, if the log contains a sequence of database operations, and if the same sequence of operations is applied to the (local) database on each replica, eventually all replicas will end up with the same database content (provided that they all started with the same initial database state).

This general approach can be used to implement a wide variety of fault-tolerant primitives, of which a fault-tolerant database is just an example. As a result, the consensus problem has been studied extensively over the past two decades. There are several well-known consensus algorithms that operate within a multitude of settings and which tolerate a variety of failures. The Paxos consensus algorithm [8] has been discussed in the theoretical [16] and applied community [10, 11, 12] for over a decade.

We used the Paxos algorithm ("Paxos") as the base for a framework that implements a fault-tolerant log. We then relied on that framework to build a fault-tolerant database. Despite the existing literature on the subject, building a production system turned out to be a non-trivial task for a variety of reasons:

- While Paxos can be described with a page of pseudo-code, our complete implementation contains several thousand lines of C++ code. The blow-up is not due simply to the fact that we used C++ instead of pseudo notation, nor because our code style may have been verbose. Converting the algorithm into a practical, production-ready system involved implementing many features and optimizations – some published in the literature and some not.

- The fault-tolerant algorithms community is accustomed to proving short algorithms (one page of pseudo code) correct. This approach does not scale to a system with thousands of lines of code. To gain confidence in the "correctness" of a real system, different methods had to be used.

- Fault-tolerant algorithms tolerate a limited set of carefully selected faults. However, the real world exposes software to a wide variety of failure modes, including errors in the algorithm, bugs in its

©ACM 2007. This is a minor revision of the work that will be published in the proceedings of ACM PODC 2007.

implementation, and operator error. We had to engineer the software and design operational procedures to robustly handle this wider set of failure modes.

- A real system is rarely specified precisely. Even worse, the specification may change during the implementation phase. Consequently, an implementation should be malleable. Finally, a system might "fail" due to a misunderstanding that occurred during its specification phase.

This paper discusses a selection of the algorithmic and engineering challenges we encountered in moving Paxos from theory to practice. This exercise took more R&D efforts than a straightforward translation of pseudo-code to C++ might suggest.

The rest of this paper is organized as follows. The next two sections expand on the motivation for this project and describe the general environment into which our system was built. We then provide a quick refresher on Paxos. We divide our experiences into three categories and discuss each in turn: algorithmic gaps in the literature, software engineering challenges, and unexpected failures. We conclude with measurements of our system, and some broader observations on the state of the art in our field.

2 Background

Chubby [1] is a fault-tolerant system at Google that provides a distributed locking mechanism and stores small files. Typically there is one Chubby instance, or "cell", per data center. Several Google systems – such as the Google Filesystem (GFS) [4] and Bigtable [2] – use Chubby for distributed coordination and to store a small amount of metadata.

Chubby achieves fault-tolerance through replication. A typical Chubby cell consists of five replicas, running the same code, each running on a dedicated machine. Every Chubby object (e.g., a Chubby lock, or file) is stored as an entry in a database. It is this database that is replicated. At any one time, one of these replicas is considered to be the "master".

Chubby clients (such as GFS and Bigtable) contact a Chubby cell for service. The master replica serves all Chubby requests. If a Chubby client contacts a replica that is not the master, the replica replies with the master's network address. The Chubby client may then contact the master. If the master fails, a new master is automatically elected, which will then continue to serve traffic based on the contents of its local copy of the replicated database. Thus, the replicated database ensures continuity of Chubby state across master failover.

The first version of Chubby was based on a commercial, third-party, fault-tolerant database; we will refer to this database as "3DB" for the rest of this paper. This database had a history of bugs related to replication. In fact, as far as we know, the replication mechanism was not based on a proven replication algorithm and we do not know if it is correct. Given the history of problems associated with that product and the importance of Chubby, we eventually decided to replace 3DB with our own solution based on the Paxos algorithm.

3 Architecture outline

Figure 1 illustrates the architecture of a single Chubby replica. A fault-tolerant replicated log based on the Paxos algorithm sits at the bottom of the protocol stack. Each replica maintains a local copy of the log. The Paxos algorithm is run repeatedly as required to ensure that all replicas have identical sequences of entries in their local logs. Replicas communicate with each other through a Paxos-specific protocol.

The next layer is a fault-tolerant replicated database which includes a local copy of the database at each replica. The database consists of a local *snapshot* and a *replay*-log of database operations. New database operations are submitted to the replicated log. When a database operation appears at a replica, it is applied on that replica's local database copy.

Finally, Chubby uses the fault-tolerant database to store its state. Chubby clients communicate with a single Chubby replica through a Chubby-specific protocol.

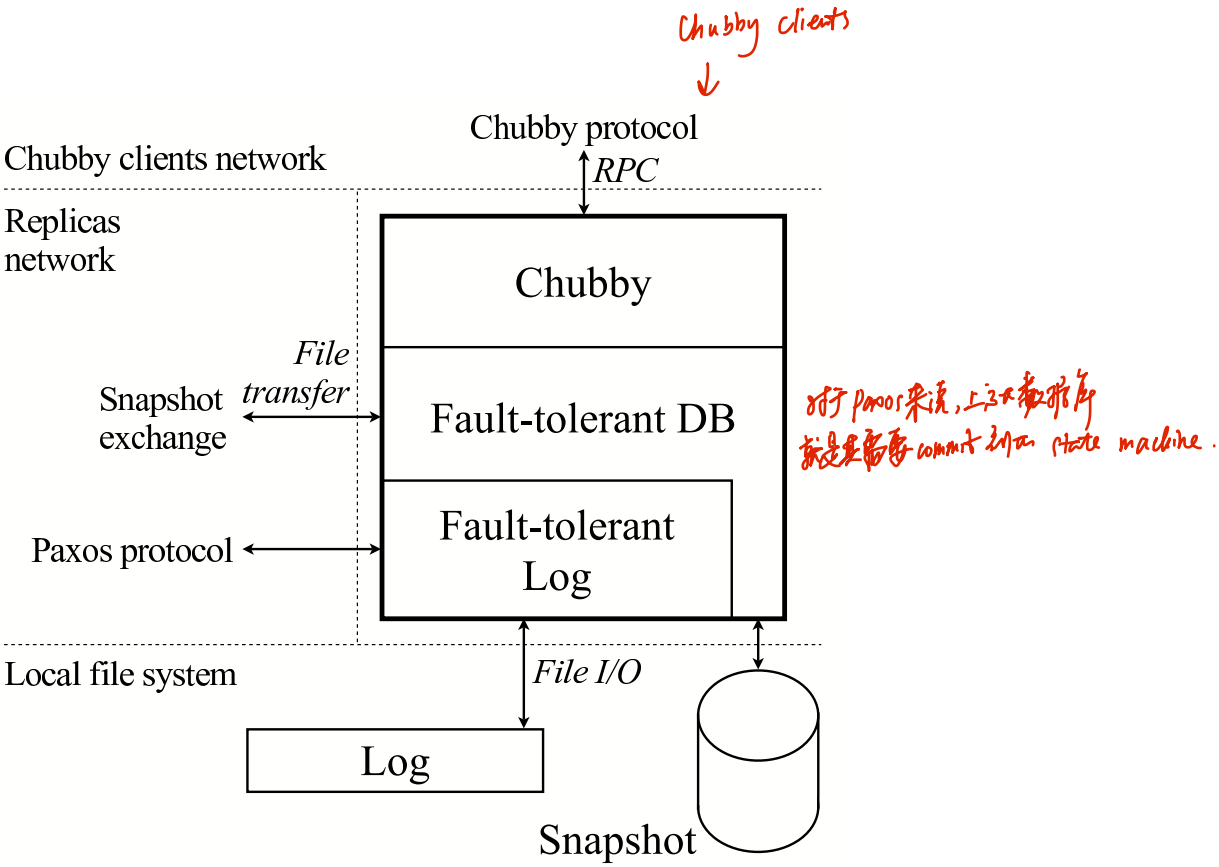

Figure 1: A single Chubby replica.

We devoted effort to designing clean interfaces separating the Paxos framework, the database, and Chubby. We did this partly for clarity while developing this system, but also with the intention of reusing the replicated log layer in other applications. We anticipate future systems at Google that seek fault-tolerance through replication. We believe that a fault-tolerant log is a powerful primitive on which to build such systems.

Our fault-tolerant log's API is depicted in Figure 2. It contains a call to *submit* a new value to the log. Once a submitted value enters the fault-tolerant log, our system invokes a callback to the client application at each replica and passes the submitted value.

Our system is multi-threaded and multiple values can be submitted concurrently on different threads. The replicated log does not create its own threads but can be invoked concurrently by any number of threads. This approach to threading helps in testing the system, as we will show in more detail later in the paper.

4 On Paxos

In this section we give an informal overview of the basic Paxos algorithm and outline how to chain together multiple executions of it (Multi-Paxos). We refer the reader to the literature for more formal descriptions and correctness proofs [8, 9, 16]. Readers who are familiar with Paxos may skip directly to the next section.

4.1 Paxos Basics

Paxos is a consensus algorithm executed by a set of processes, termed *replicas*, to agree on a single value in the presence of failures. Replicas may crash and subseqently recover. The network may drop messages

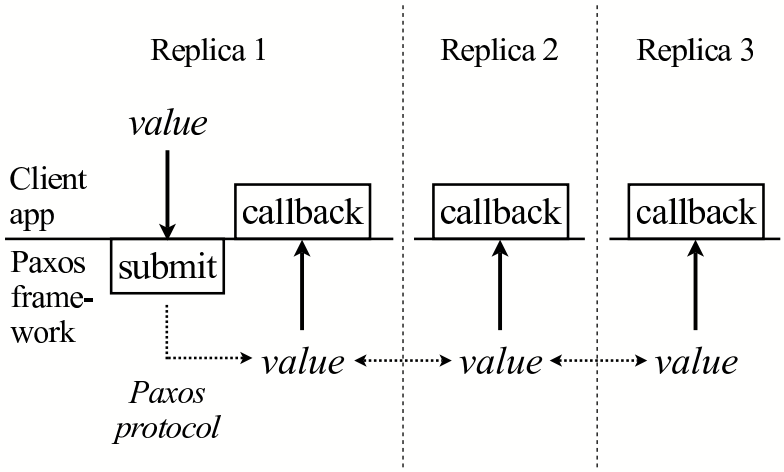

Figure 2: API for fault-tolerant log.

between replicas. Replicas have access to persistent storage that survives crashes. Some replicas may *submit* values for consensus. If eventually a majority of the replicas run for long enough without crashing and there are no failures, all running replicas are guaranteed to *agree* on one of the values that was submitted. In our system, the value to be agreed upon is the next entry in a (replicated) log as described in the introduction.

The algorithm consists of three phases, which may be repeated (because of failures):

1. Elect a replica to be the *coordinator*.
2. The coordinator selects a value and broadcasts it to all replicas in a message called the *accept* message. Other replicas either *acknowledge* this message or *reject* it.
3. Once a majority of the replicas *acknowledge* the coordinator, consensus has been reached, and the coordinator broadcasts a *commit* message to notify replicas.

To provide some intuition about how the algorithm works, consider first the case in which there is only a single coordinator and no failures. Consensus is reached once a majority of replicas receive the accept message from the coordinator and acknowledge it. Subsequently, if any minority of the replicas fail, we are still guaranteed that at least one replica will be alive that received the consensus value.

In reality the coordinator may fail. Paxos does not require that only one replica act as coordinator at a time. Multiple replicas may decide to become coordinators and execute the algorithm at any time. Typically the system is engineered to limit coordinator turnover, as it can delay reaching consensus.

This flexible election policy means there may be multiple replicas who simultaneously believe they are the coordinator. Further, these coordinators may select different values. Paxos ensures consensus can be reached on a single value (it can be from any coordinator) by introducing two extra mechanisms: 1) assigning an ordering to the successive coordinators; and 2) restricting each coordinator's choice in selecting a value.

Ordering the coordinators allows each replica to distinguish between the current coordinator and previous coordinators. In this way, replicas can reject messages from old coordinators and prevent them from disrupting consensus once it is reached. Paxos orders the coordinators by assigning them an increasing *sequence number* as follows. Each replica keeps track of the most recent sequence number it has seen so far. When a replica wants to become coordinator, it generates a unique[1] sequence number higher than any it has seen,

[1]For example, in a system with n replicas, assign each replica r a unique id i_r between 0 and $n-1$. Replica r picks the smallest sequence number s larger than any it has seen such that $s \bmod n = i_r$.

and broadcasts it to all replicas in a *propose* message. If a majority of replicas reply and indicate they have not seen a higher sequence number, then the replica acts as a coordinator. These replies are called *promise* messages since replicas promise henceforth to reject messages from old coordinators. This propose/promise message exchange constitutes step 1 listed above.

Once consensus is reached on a value, Paxos must force future coordinators to select that same value in order to ensure continued agreement. To guarantee this, the promise messages from replicas include the most recent value they have heard, if any, along with the sequence number of the coordinator from whom they heard it. The new coordinator chooses the value from the most recent coordinator. If none of the promise messages contain a value, the coordinator is free to choose a submitted value.

The reasoning why this works is subtle, but proceeds roughly as follows. The new coordinator requires a response to the propose message from a majority of replicas. Therefore, if consensus was achieved by a previous coordinator, the new coordinator is guaranteed to hear about the value decided upon from at least one replica. By induction, that value will have the highest sequence number of all responses received, and so will be selected by the new coordinator.

4.2 Multi-Paxos

Practical systems use Paxos as a building block to achieve consensus on a *sequence of values*, such as in a replicated log. The simple way to implement this is to repeatedly execute the Paxos algorithm. We term each execution an *instance* of Paxos. We refer to *submitting* a value to Paxos (or equivalently, to the log) to mean executing an instance of Paxos while submitting that value.

In Multi-Paxos some slow (*lagging*) replicas might not have participated in recent Paxos instances. We use a *catch-up* mechanism to enable lagging replicas to catch up with *leading* replicas.

Each replica maintains a locally persistent log to record all Paxos actions. When a replica crashes and subsequently recovers, it replays the persistent log to reconstruct its state prior to crashing. Replicas also use this log when helping lagging replicas to catch up. The Paxos algorithm as described thus far requires all message senders to log their state before sending messages – thus the algorithm requires a sequence of five writes (for each of the propose, promise, accept, acknowledgment, and commit messages) to disk on its critical path. Note that all writes have to be flushed to disk immediately before the system can proceed any further. In a system where replicas are in close network proximity, disk flush time can dominate the overall latency of the implementation.

There is a well-known optimization to reduce the number of messages involved by chaining together multiple Paxos instances [9]. Propose messages may be omitted if the coordinator identity does not change between instances. This does not interfere with the properties of Paxos because any replica at any time can still try to become coordinator by broadcasting a propose message with a higher sequence number. In order to avail itself of this optimization, a Multi-Paxos algorithm may be designed to pick a coordinator for long periods of time, trying not to let the coordinator change. We refer to this coordinator as the *master*. With this optimization, the Paxos algorithm only requires a single write to disk per Paxos instance on each replica, executed in parallel with each other. The master writes to disk immediately after sending its accept message and other replicas write to disk prior to sending their acknowledge message.

In order to get additional throughput in a concurrent system, it is possible to batch a collection of values submitted by different application threads into a single Paxos instance.

5 Algorithmic challenges

While the core Paxos algorithm is well-described, implementing a fault-tolerant log based on it is a non-trivial endeavor. Some of the complications are due to imperfections found in the real world (such as hard disk failures, or finite resources), and some are due to additional requirements (for instance, “master leases”). Many of these challenges have algorithmic solutions that are intimately connected with the core Paxos algorithm. In the following we describe a number of mechanisms that we introduced.

5.1 Handling disk corruption

Replicas witness disk corruption from time to time. A disk may be corrupted due to a media failure or due to an operator error (an operator may accidentally erase critical data). When a replica's disk is corrupted and it loses its persistent state, it may renege on promises it has made to other replicas in the past. This violates a key assumption in the Paxos algorithm. We use the following mechanism to address this problem [14].

Disk corruptions manifest themselves in two ways. Either file(s) contents may change or file(s) may become inaccessible. To detect the former, we store the checksum of the contents of each file in the file[2]. The latter may be indistinguishable from a new replica with an empty disk – we detect this case by having a new replica leave a marker in GFS after start-up. If this replica ever starts again with an empty disk, it will discover the GFS marker and indicate that it has a corrupted disk.

A replica with a corrupted disk rebuilds its state as follows. It participates in Paxos as a non-voting member; meaning that it uses the catch-up mechanism to catch up but does not respond with promise or acknowledgment messages. It remains in this state until it observes one complete instance of Paxos that was started after the replica started rebuilding its state. By waiting for the extra instance of Paxos, we ensure that this replica could not have reneged on an earlier promise.

This mechanism enables the following optimization to improve the latency of the system. Since the system can now deal with occasional disk corruption, under some circumstances it may be acceptable not to flush writes to disk immediately[3]. While we have considered schemes to exploit this observation, we have not implemented them yet.

5.2 Master leases

When the basic Paxos algorithm is used to implement a replicated data structure, reads of the data structure require executing an instance of Paxos. This serializes the read with respect to updates and ensures that the *current* state is read. In particular, read operations cannot be served out of the master's copy of the data structure because it is possible that other replicas have elected another master and modified the data structure without notifying the old master. In this case, the read operation at the master runs the risk of returning stale data. Since read operations usually comprise a large fraction of all operations, serializing reads through Paxos is expensive.

The workaround is to implement *master leases* [5] with the following semantics: as long as the master has the lease, it is guaranteed that other replicas cannot successfully submit values to Paxos. Thus a master with the lease has up-to-date information in its local data structure which can be used to serve a read operation purely locally. By making the master attempt to renew its lease before it expires we can ensure that a master has a lease most of the time. With our system, masters successfully maintain leases for several days at a time.

In our implementation, all replicas implicitly grant a lease to the master of the previous Paxos instance and refuse to process Paxos messages from any other replica while the lease is held. The master maintains a shorter timeout for the lease than the replicas – this protects the system against clock drift. The master periodically submits a dummy "heartbeat" value to Paxos to refresh its lease.

The Multi-Paxos optimization exhibits the following stability problem when there are intermittent network outages. When a master temporarily disconnects, Paxos will elect a new master. The new master will maintain a fixed sequence number across instances of Paxos. In the mean time, when the disconnected old master tries to run the Paxos algorithm, if it manages to connect with another replica, it may increase its sequence number. When it reconnects, it may have a higher sequence number than the new master and be able to replace the new master. Later it may disconnect again, and the cycle can repeat itself.

This behavior is undesirable as Chubby master changes have a negative impact on some of its users. Furthermore, this behavior can degenerate into rapid master changes in a network with poor connectivity.

[2]This mechanism will not detect files that roll back to an old state. We believe this is an unlikely scenario and chose not to handle it explicitly. Our distributed checksum mechanism, described later, may detect this type of problem.

[3]For example, if the operating system and the hardware underlying each replica rarely fail and failures at different replicas are independent of each other, it is possible to modify our system so it does not need to flush writes to disk.

In our implementation the master periodically boosts its sequence number by running a full round of the Paxos algorithm, including sending propose messages[4]. Boosting with the right frequency avoids this type of master churn in most cases.

Note that it is possible to extend the concept of leases to all replicas. This will allow any replica with a lease to serve read requests from its local data structure. This extended lease mechanism is useful when read traffic significantly exceeds write traffic. We have examined algorithms for replica leases, but have not implemented them yet.

5.3 Epoch numbers

Requests (by a Chubby client) submitted to a Chubby cell are directed to the current Chubby master replica. From the time when the master replica receives the request to the moment the request causes an update of the underlying database, the replica may have lost its master status. It may even have lost master status and regained it again. Chubby requires an incoming request to be aborted if mastership is lost and/or re-acquired during the handling of the request. We needed a mechanism to reliably detect master turnover and abort operations if necessary.

We solved this problem by introducing a global epoch number with the following semantics. Two requests for the epoch number at the master replica receive the same value iff that replica was master continuously for the time interval between the two requests. The epoch number is stored as an entry in the database, and all database operations are made conditional on the value of the epoch number.

5.4 Group membership

Practical systems must be able to handle changes in the set of replicas. This is referred to as the group membership problem in the literature [3]. Some Paxos papers point out that the Paxos algorithm itself can be used to implement group membership [8]. While group membership with the core Paxos algorithm is straightforward, the exact details are non-trivial when we introduce Multi-Paxos, disk corruptions, etc. Unfortunately the literature does not spell this out, nor does it contain a proof of correctness for algorithms related to group membership changes using Paxos. We had to fill in these gaps to make group membership work in our system. The details – though relatively minor – are subtle and beyond the scope of this paper.

5.5 Snapshots

As described thus far, the repeated application of a consensus algorithm to create a replicated log will lead to an ever growing log. This has two problems: it requires unbounded amounts of disk space; and perhaps worse, it may result in unbounded recovery time since a recovering replica has to replay a potentially long log before it has fully caught up with other replicas. Since the log is typically a sequence of operations to be applied to some data structure, and thus implicitly (through replay) represents a persistent form of that data structure, the problem is to find an alternative persistent representation for the data structure at hand. An obvious mechanism is to persist – or *snapshot* – the data structure directly, at which point the log of operations leading to the current state of the data structure is no longer needed. For example, if the data structure is held in memory, we take a snapshot by serializing it on disk. If the data structure is kept on disk, a snapshot may just be an on-disk copy of it.

By itself, the Paxos framework does not know anything about the data structure we are trying to replicate; its only concern is the consistency of the replicated log. It is the particular application using the Paxos framework that has all the knowledge about the replicated data structure. Thus the application must be responsible for taking snapshots. Our framework provides a mechanism that allows client applications, e.g. our fault-tolerant database, to inform the framework that a snapshot was taken; the client application is free to take a snapshot at any point. When the Paxos framework is informed about a snapshot, it will truncate its log by deleting log entries that precede the snapshot. Should the replica fail, during subsequent recovery

[4]In a loaded system, under one percent of the Paxos instances run the full Paxos algorithm.

it will simply install the latest snapshot and then replay the truncated log to rebuild its state. Snapshots are not synchronized across replicas; each replica independently decides when to create a snapshot.

This mechanism appears straightforward at first and is mentioned briefly in the literature [8]. However, it introduces a fair amount of complexity into the system: the persistent state of a replica now comprises a log *and* a snapshot that have to be maintained consistently. The log is fully under the framework's control, while the snapshot format is application-specific. Some aspects of the snapshot machinery are of particular interest:

- The snapshot and log need to be mutually consistent. Each snapshot needs to have information about its contents relative to the fault-tolerant log. In our framework we introduced the concept of a *snapshot handle* for this purpose. The snapshot handle contains all the Paxos-specific information related to a particular snapshot. When creating a snapshot (which is under control of the application) the corresponding snapshot handle (provided by the framework) needs to be stored by the application as well. When recovering a snapshot, the application must return the snapshot handle to the framework, which in turn will use the information in the handle to coordinate the snapshot with the log.

  Note that the handle is really a snapshot of the Paxos state itself. In our system, it contains the Paxos instance number corresponding to the (log) snapshot and the group membership at that point.

- Taking a snapshot takes time and in some situations we cannot afford to freeze a replica's log while it is taking a snapshot. In our framework, taking a snapshot is split into three phases. First, when the client application decides to take a snapshot, it requests a snapshot handle. Next, the client application takes its snapshot. It may block the system while taking the snapshot, or – more likely – spawn a thread that takes a snapshot while the replica continues to participate in Paxos. The snapshot must correspond to the client state at the log position when the handle was obtained. Thus if the replica continues to participate in Paxos while taking a snapshot, special precautions may have to be taken to snapshot the client's data structure while it is actively updated.[5] Finally, when the snapshot has been taken, the client application informs the framework about the snapshot and passes the corresponding snapshot handle. The framework then truncates the log appropriately.

- Taking a snapshot may fail. Our framework only truncates the log when it is informed that a snapshot has been taken and has received the corresponding snapshot handle. Thus, as long as the client application does not inform the framework, from the framework's viewpoint, no snapshot has been taken. This allows the client application to verify a snapshot's integrity and discard it if necessary. If there is a problem with the snapshot, the client doesn't ask the framework to truncate its log. A client application may even attempt to take several snapshots at the same time using this mechanism.

- While in catch-up, a replica will attempt to obtain missing log records. If it cannot obtain them (because no replica has old-enough log entries readily available), the replica will be told to obtain a snapshot from another replica. This snapshot's handle contains information about the Paxos instance up to which the snapshot captured the state. Once the snapshot has been received and installed, under most circumstances the lagging replica will be close to the leading replica. In order to completely catch-up, the lagging replica asks for and receives the remaining log records from the leading replica to bring it fully up-to-date.

  Note that a leading replica may even create a new snapshot while a lagging replica is installing an older snapshot – in a fault-tolerant system this cannot be avoided. In this scenario, the lagging replica may not be able to obtain any outstanding log records because the snapshot provider (and any other replicas) may have moved ahead in the meantime. The lagging replica will need to obtain a more recent snapshot.

[5] Our first implementation of the fault-tolerant database blocked the system very briefly while making an in-memory copy of the (small) database. It then stored the copied data on disk via a separate thread. Subsequently we implemented virtually pause-less snapshots. We now use a "shadow" data structure to track updates while the underlying database is serialized to disk.

Furthermore, the leading replica may fail after sending its snapshot. The catch-up mechanism must be able to recover from such problems by having the lagging replica contact another leading replica.

- We needed a mechanism to locate recent snapshots. Some applications may choose to transfer snapshots directly between leading and lagging replicas while others may ask a lagging replica to look up a snapshot on GFS. We implemented a general mechanism that allows an application to pass snapshot location information between leading and lagging replicas.

5.6 Database transactions

The database requirements imposed by Chubby are simple: the database needs to store key-value pairs (with keys and values being arbitrary strings), and support common operations such as `insert`, `delete`, `lookup`, an atomic `compare and swap` (`cas`), and iteration over all entries. We implemented a log-structured design using a snapshot of the full database, and a log of database operations to be applied to that snapshot. The log of operations is the Paxos log. The implementation periodically takes a snapshot of the database state and truncates the log accordingly.

The `cas` operation needed to be atomic with respect to other database operations (potentially issued by a different replica). This was easily achieved by submitting all `cas`-related data as a single "value" to Paxos. We realized that we could extend this mechanism to provide transaction-style support without having to implement true database transactions. We describe our solution in more detail because we believe it to be useful in other contexts.

Our implementation hinges around a powerful primitive which we call `MultiOp`. All other database operations except for iteration are implemented as a single call to `MultiOp`. A `MultiOp` is applied atomically and consists of three components:

1. A list of tests called **guard**. Each test in `guard` checks a single entry in the database. It may check for the absence or presence of a value, or compare with a given value. Two different tests in the guard may apply to the same or different entries in the database. All tests in the guard are applied and `MultiOp` returns the results. If all tests are true, `MultiOp` executes `t_op` (see item 2 below), otherwise it executes `f_op` (see item 3 below).

2. A list of database operations called **t_op**. Each operation in the list is either an `insert`, `delete`, or `lookup` operation, and applies to a single database entry. Two different operations in the list may apply to the same or different entries in the database. These operations are executed[6] if **guard** evaluates to true.

3. A list of database operations called **f_op**. Like `t_op`, but executed if **guard** evaluates to false.

Late in our development (and after we had implemented the database and `MultiOp`), we realized that we also needed epoch numbers to implement database operations for Chubby. With this additional requirement, all Chubby operations became associated with an epoch number and were required to fail if the Paxos epoch number changed. `MultiOp` proved useful in accomodating this new requirement. After we incorporated the Paxos epoch as a database entry, we were able to modify all previous calls to our database to include an additional guard to check for the epoch number.

6 Software Engineering

Fault-tolerant systems are expected to run continuously for long periods of time. Users are much less likely to tolerate bugs than in other systems. For instance, a layout bug exhibited by a document editor may be annoying to a user, but often it is possible to "work around" the issue, even though the bug is really at the

[6]Each `MultiOp` operation is serialized atomically with respect to other operations. The individual operations in the list are executed sequentially on the database.

core of what the software is supposed to do. A bug of similar gravity in a fault-tolerant system may make the system unusable.

We adopted several software engineering methods to give us confidence in the robustness of our implementation. We describe some of the methods we used in this section.

6.1 Expressing the algorithm effectively

Fault-tolerant algorithms are notoriously hard to express correctly, even as pseudo-code. This problem is worse when the code for such an algorithm is intermingled with all the other code that goes into building a complete system. It becomes harder to see the core algorithm, to reason about it, or to debug it when a bug is present. It also makes it difficult to change the core algorithm in response to a requirement change.

We addressed this problem by coding the core algorithm as two explicit state machines. For that purpose, we designed a simple state machine specification language and built a compiler to translate such specifications into C++. The language was designed to be terse so that a full algorithm can be rendered on a single screen. As an additional benefit, the state machine compiler also automatically generates code to log state transitions and measure code coverage to assist in debugging and testing.

We believe that choosing a specification language makes it easier to reason about and modify our state machines than an explicitly coded implementation that is intermingled with the rest of the system. This is illustrated by the following experience. Towards the end of our development of the fault-tolerant log, we had to make a fundamental change in our group membership algorithm. Prior to this change, a replica roughly went through three states. Initially it waited to join the group, then it joined the group, and finally it left the group. Once a replica left the group, it was not allowed to rejoin the group. We felt this approach was best because an intermittently failing replica would not be able to join the group and disrupt it for long. Intermittent failure turned out to be more common than originally anticipated because normal replicas exhibit intermittent failures from time to time. Thus, we needed to change the algorithm to have two states. Either a replica was in the group or it was out. A replica could switch between these two states often during the lifetime of the system. It took us about one hour to make this change and three days to modify our tests accordingly. Had we intermingled our state machines with the rest of the system, this change would have been more difficult to make.

6.2 Runtime consistency checking

The chance for inconsistencies increases with the size of the code base, the duration of a project, and the number of people working simultaneously on the same code. We used various active self-checking mechanisms such as the liberal use of `assert` statements, and explicit verification code that tests data structures for consistency.

For example, we used the following database consistency check. The master periodically submits a `checksum` request to the database log. On receipt of this request, each replica computes a checksum of its local database[7]. Since the Paxos log serializes all operations identically on all replicas, we expect all replicas to compute the same checksum. After the master completes a checksum computation, it sends its checksum to all replicas which compare the master's checksum with their computed checksum.

We have had three database inconsistency incidents thus far:

- The first incident was due to an operator error.
- We have not found an explanation for the second incident. On replaying the faulty replica's log we found that it was consistent with the other replicas. Thus it is possible that this problem was caused by a random hardware memory corruption.
- We suspect the third was due to an illegal memory access from errant code in the included codebase (which is of considerable size). To protect against this possibility in the future, we maintain a second database of checksums and double-check every database access against the database of checksums.

[7]We use a shadow datastructure to handle database operations concurrently with the checksum computation.

In all three cases manual intervention appeared to resolve the problem before it became visible to Chubby.

6.3 Testing

Given the current state of the art, it is unrealistic to prove a real system such as ours correct. To achieve robustness, the best practical solution in addition to meticulous software engineering is to test a system thoroughly. Our system was designed to be testable from the onset and now contains an extensive suite of tests. In this section we describe two tests that take the system through a long sequence of random failures and verify that it behaves as expected. Both tests can run in one of two modes:

1. **Safety mode**. In this mode, the test verifies that the system is consistent. However, the system is not required to make any progress. For example, it is acceptable for an operation to fail to complete or to report that the system is unavailable.

2. **Liveness mode**. In this mode, the test verifies that the system is consistent and is making progress. All operations are expected to complete and the system is required to be consistent.

Our tests start in safety mode and inject random failures into the system. After running for a predetermined period of time, we stop injecting failures and give the system time to fully recover. Then we switch the test to liveness mode. The purpose for the liveness test is to verify that the system does not deadlock after a sequence of failures.

One of our tests verifies the fault-tolerant log. It simulates a distributed system consisting of a random number of replicas and takes our fault-tolerant log through a random sequence of network outages, message delays, timeouts, process crashes and recoveries, file corruptions, schedule interleavings, etc. We wanted this test to be repeatable to aid in debugging. To this end, we use a random number generator to determine the schedule of failures. The seed for the random number generator is given at the beginning of the test run. We ensure that two test runs with the same random number seed are identical by running the test in a single thread to remove unwanted non-determinism from multi-threading. This is possible because the fault-tolerant log does not create its own threads and can run in a single-threaded environment (even though it normally runs in a multi-threaded environment).

Each test execution reports success or failure. If a test fails, we rerun that test with the failing random number seed and with detailed logging turned on in a debugger to determine what went wrong. This is possible because these tests are repeatable.

This test proved useful in finding various subtle protocol errors, including errors in our group membership implementation, and our modifications to deal with corrupted disks. In order to measure the strength of this test, we left some protocol bugs found during code and design reviews in the system, and verified that our test system detected these bugs. After a number of bug fixes, the test became very stable. In order to improve its bug yield, we started running this test on a farm of several hundred Google machines at a time. We found additional bugs, some of which took weeks of simulated execution time (at extremely high failure rates) to find.

Another test verifies robustness of the new Chubby system against lower-level system and hardware failures. We implemented several hooks in our fault-tolerant log to inject failures. The test randomly invokes these hooks and verifies that higher levels of the system can cope. Our hooks can be used to crash a replica, disconnect it from other replicas for a period of time or force a replica to pretend that it is no longer the master. This test found five subtle bugs in Chubby related to master failover in its first two weeks. In the same vein, we built a filesystem with hooks to programmatically inject failures and are using it to test our ability to deal with filesystem failures.

In closing we point out a challenge that we faced in testing our system for which we have no systematic solution. By their very nature, fault-tolerant systems try to mask problems. Thus they can mask bugs or configuration problems while insidiously lowering their own fault-tolerance. For example, we have observed the following scenario. We once started a system with five replicas, but misspelled the name of one of the replicas in the initial group. The system appeared to run correctly as the four correctly configured replicas

were able to make progress. Further, the fifth replica continunously ran in catch-up mode[8] and therefore appeared to run correctly as well. However in this configuration the system only tolerates one faulty replica instead of the expected two. We now have processes in place to detect this particular type of problem. We have no way of knowing if there are other bugs/misconfigurations that are masked by fault-tolerance.

6.4 Concurrency

At the onset of the project we were concerned about the problem of testing concurrent fault-tolerant code. In particular, we wanted our tests to be repeatable. As described earlier, our fault-tolerant log doesn't contain any of its own threads (even though it can handle concurrent requests on different threads). Threading is introduced at the edges of the code – where we receive calls from the networking layer. By making our tests repeatable, we were able to hunt down several obscure protocol errors during testing.

As the project progressed, we had to make several subsystems more concurrent than we had intended and sacrifice repeatability. Chubby is multi-threaded at its core, thus we cannot run repeatable tests against the complete system. Next we had to make our database multi-threaded so it could take snapshots, compute checksums and process iterators while concurrently serving database requests. Finally, we were forced to make the code that handles the local copy of the log multi-threaded as well (the exact reason why is beyond the scope of this paper).

In summary, we believe that we set ourselves the right goals for repeatability of executions by constraining concurrency. Unfortunately, as the product needs grew we were unable to adhere to these goals.

7 Unexpected failures

So far, our system has logged well over 100 machine years of execution in production. In this period we have witnessed the following unexpected failure scenarios:

- Our first release shipped with ten times the number of worker threads as the original Chubby system. We hoped this change would enable us to handle more requests. Unfortunately, under load, the worker threads ended up starving some other key threads and caused our system to time out frequently. This resulted in rapid master failover, followed by en-masse migrations of large numbers of clients to the new master which caused the new master to be overwhelmed, followed by additional master failovers, and so on.

  When this problem first appeared, the precise cause was unknown and we had to protect ourselves from a potentially dangerous bug in our system. We decided to err on the side of caution and to rollback our system to the old version of Chubby (based on 3DB) in one of our data centers. At that point, the rollback mechanism was not properly documented (because we never expected to use it), its use was non-intuitive, the operator performing the roll-back had no experience with it, and when the rollback was performed, no member of the development team was present. As a result, an old snapshot was accidentally used for the rollback. By the time we discovered the error, we had lost 15 hours of data and several key datasets had to be rebuilt.

- When we tried to upgrade this Chubby cell again a few months later, our upgrade script failed because we had omitted to delete files generated by the failed upgrade from the past. The cell ended up running with a months-old snapshot for a few minutes before we discovered the problem. This caused us to lose about 30 minutes of data. Fortunately all of Chubby's clients recovered from this outage.

- A few months after our initial release, we realized that the semantics provided by our database were different from what Chubby expected. If Chubby submitted an operation to the database, and the database lost its master status, Chubby expected the operation to fail. With our system, a replica

[8] In our implementation, a replica that is not (yet) a group member runs in catch-up mode to stay up-to-date. This allows us to keep a future group member "warm" so it can become an active member immediately after joining the group.

could be re-installed as master during the database operation and the operation could succeed. The fix required a substantial rework of the integration layer between Chubby and our framework (we needed to implement epoch numbers). `MultiOp` proved to be helpful in solving this unexpected problem – an indication that `MultiOp` is a powerful primitive.

- As mentioned before, on three occasions we discovered that one of the database replicas was different from the others in that Chubby cell. We found this problem because our system periodically takes checksums of all replicas and then compares them.
- Our upgrade script which is responsible for migrating cells from the 3DB version of Chubby to the Paxos version has failed several times for a variety of reasons. For example, it once failed because a basic Google program was not installed on one of our cells.
- We have encountered failures due to bugs in the underlying operating system. For example in our version of the Linux 2.4 kernel, when we try to flush a small file to disk, the call can hang for a long time if there are a lot of buffered writes to other files. This happens immediately after we write a database snapshot to disk. In this case, we observed that it could take several seconds for the kernel to flush an unrelated small write to the Paxos log. Our workaround is to write *all* large files in small chunks, with a flush to disk after each small chunk. While this hurts the performance of the write slightly, it protects the more critical log writes from unexpected delays.

A small number of failures in 100 machine years would be considered excellent behavior for most production systems. However, we consider the current failure rate too high for Chubby and we have determined that we need to reduce it further.

Three of the failures occurred during upgrade (or rollback). Every time we encountered a problem during upgrade, we updated our upgrade script accordingly. Once a cell is upgraded, this type of failure will disappear.

Two of the failures were from bugs that have since been fixed. To reduce the probability of other bugs, we continue to improve and run the Chubby verification test outlined earlier.

Two of our unexpected problems relate to operator error during rollout of a new release and caused us to lose data. At Google, the day-to-day monitoring and management of our systems is done by system operators. While they are very competent, they are usually not part of the development team that built the system, and therefore not familiar with its intricate details. This may lead to the occasional operator error in unforseen situations. We now rely on carefully written and well-tested scripts to automate rollout and minimize operator involvement. As a result our most recent major release of Chubby was rolled out across hundreds of machines without incident, while serving life traffic.

One of the failures was due to memory corruption. Because our system is log-structured and maintains several days of log data and snapshots, it was possible to replay the database upto the exact point at which the problem appears. We were able to verify that our logs were correct and conclude that the memory corruption occurred from errant software or due to hardware problems. We added additional checksum data to detect this type of problem in the future and will crash a replica when it detects this problem.

8 Measurements

The initial goal of our system was to replace 3DB with our own database. Thus our system had to demonstrate equal or superior performance relative to 3DB. We measured the performance of a complete Chubby system (clients, server, including network latency) using our fault-tolerant replicated database. We also benchmarked this system against an identical system based on 3DB (see Table 1). For our tests, we ran two copies of Chubby on the same set of 5 servers (typical Pentium®-class machines). One copy of Chubby used our database while the other copy used 3DB. We ran Chubby clients on workstations to generate load on the servers. For our tests, we measured total system throughput. Each call includes the Chubby client, the network, the Chubby server and our fault-tolerant database. While this test underestimates the performance of our database, it gives a sense of the full system throughput of a system based on Paxos.

Test	# workers	file size (bytes)	Paxos-Chubby (100MB DB)	3DB-Chubby (small database)	Comparison
Ops/s Throughput	1	5	91 ops/sec	75 ops/sec	1.2x
Ops/s Throughput	10	5	490 ops/sec	134 ops/sec	3.7x
Ops/s Throughput	20	5	640 ops/sec	178 ops/sec	3.6x
MB/s Throughput	1	8 KB	345 KB/s	172 KB/s	2x
MB/s Throughput	4	8 KB	777 - 949 KB/s	217 KB/s	3.6 - 4.4x
MB/s Throughput	1	32 KB	672 - 822 KB/s	338 KB/s	2.0 - 2.4x

Table 1: Comparing our system with 3DB (higher numbers are better).

Even though read requests to Chubby dominate in practice, we designed our tests to be write intensive. This is because read requests are completely handled on the master, which typically has a lease, and do not exercise the Paxos algorithm.

In our test, each worker repeatedly creates a file in Chubby and waits for Chubby to return before creating the file again. Thus each operation makes one write call to the underlying database. If the contents of the file are small and there is a single worker, the test measures the latency of the system. If the contents of the file are large, the test measures the throughput of the system in MB/s. By using multiple concurrent workers, we were also able to measure the throughput of the system in submissions/s.

All tests with more than one worker show the effect of batching a collection of submitted values. It should be possible to achieve some speedup with 3DB by bundling a collection of updates in a database transaction. The last two throughput tests show the effect of taking snapshots. This system was configured to take a snapshot whenever the replicated log size exceeded 100 MB. In these two tests, the system takes snapshots roughly every 100 seconds. When taking a snapshot, the system makes another copy of the database and writes it to disk. As a result, its performance temporarily drops off.

Our system is by no means optimized for performance, and we believe that there is a lot of room to make it faster. However, given the performance improvement over 3DB, further optimizations are not a priority at this time.

9 Summary and open problems

We have described our implementation of a fault-tolerant database, based on the Paxos consensus algorithm. Despite the large body of literature in the field, algorithms dating back more then 15 years, and experience of our team (one of us has designed a similar system before and the others have built other types of complex systems in the past), it was significantly harder to build this system then originally anticipated. We attribute this to several shortcomings in the field:

- There are significant gaps between the description of the Paxos algorithm and the needs of a real-world system. In order to build a real-world system, an expert needs to use numerous ideas scattered in the literature and make several relatively small protocol extensions. The cumulative effort will be substantial and the final system will be based on an unproven protocol.
- The fault-tolerance computing community has not developed the tools to make it easy to implement their algorithms.
- The fault-tolerance computing community has not paid enough attention to testing, a key ingredient for building fault-tolerant systems.

As a result, the core algorithms work remains relatively theoretical and is not as accessible to a larger computing community as it could be. We believe that in order to make a greater impact, researchers in the field should focus on addressing these shortcomings.

In contrast, consider the field of compiler construction. Though concepts in that field are complex, they have been made accessible to a wide audience. Industrial-strength parsing tools such `yacc` [6] appeared not too long after the theory of parsing was well-understood. Not only are there now many front-end tools such as `ANTLR` [15] or `CoCo/R` [13]; but there are also tree-rewriting tools helping with optimizations and instruction selection, assemblers helping with binary code generation, and so forth. Thus, in this area of software engineering, an entire family of tools has emerged, making the construction of a compiler significantly easier or at least less error-prone. Disciplines within the field of compiler construction, such as parsing, which were once at the cutting edge of research, are now considered "solved" and are routinely taught at the undergraduate level in many schools.

It appears that the fault-tolerant distributed computing community has not developed the tools and know-how to close the gaps between theory and practice with the same vigor as for instance the compiler community. Our experience suggests that these gaps are non-trivial and that they merit attention by the research community.

10 Acknowledgments

Many people at Google helped us with this project. Mike Burrows who implemented Chubby suggested that we replace 3DB with a Paxos-based system. He and Sharon Perl reviewed our designs and provided excellent feedback. They introduced us to the mechanism for handling disk corruptions and suggested that we implement master leases. Michal Cierniak ported the original state machine compiler from Perl to C++ and made substantial modifications (it is now being used elsewhere at Google as well). Vadim Furman helped us write the Chubby verification test. Salim Virji and his team were responsible for the roll-out of our system across Google data centers.

Mike Burrows, Bill Coughran, Gregory Eitzman, Peter Mckenzie, Sharon Perl, Rob Pike, David Presotto, Sean Quinlan, and Salim Virji reviewed earlier versions of this paper and provided valuable feedback.

References

[1] Burrows, M. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pp. 335-350

[2] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pp. 205-218

[3] Cristian, F. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing 4*, 4 (1991), 175–188.

[4] Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Dec. 2003), pp. 29–43.

[5] Gray, C., Cheriton, D. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles* (1989), pp. 202–210.

[6] Johnson, S. C. Yacc: Yet another compiler-compiler.

[7] Lamport, Shostak, and Pease. The byzantine generals problem. In *Advances in Ultra-Dependable Distributed Systems, N. Suri, C. J. Walter, and M. M. Hugue (Eds.), IEEE Computer Society Press*. 1995.

[8] Lamport, L. The part-time parliament. *ACM Transactions on Computer Systems 16*, 2 (1998), 133–169.

[9] Lamport, L. Paxos made simple. *ACM SIGACT News 32*, 4 (Dec. 2001), 18–25.

[10] Lampson, B. W. How to build a highly available system using consensus. In *10th International Workshop on Distributed Algorithms (WDAG 96)* (1996), Babaoglu and Marzullo, Eds., vol. 1151, Springer-Verlag, Berlin Germany, pp. 1–17.

[11] Lee, E. K., and Thekkath, C. A. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems* (Cambridge, MA, 1996), pp. 84–92.

[12] MacCormick, J., Murphy, N., Najork, M., Thekkath, C. A., and Zhou, L. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (2004), pp. 105–120.

[13] Moessenboeck, H. A generator for production quality compilers. In *Proceedings of the 3rd International Workshop on Compiler Compilers - Lecture Notes in Computer Science 477* (Berlin, Heidelberg, New York, Tokyo, 1990), Springer-Verlag, pp. 42–55.

[14] Oki, Brian M., and Liskov, Barbara H. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the 7th annual ACM Symposium on Principles of Distributed Computing* (1988), pp. 8–17.

[15] Parr, T. J., and QUONG, R. W. Antlr: A predicated-ll(k) parser generator. *Software–Practice and Experience 25*, 7 (JULY 1995), 789–810.

[16] Prisco, R. D., Lampson, B. W., and Lynch, N. A. Revisiting the paxos algorithm. In *11th International Workshop on Distributed Algorithms (WDAG 96)* (1997), pp. 111–125.

[17] Schneider, F. B. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys 22*, 4 (1990), 299–319.

[18] von Neumann, J. Probabilistic logics and synthesis of reliable organisms from unreliable components. *Automata Studies* (1956), 43–98.