Locks support the implementation of reliable storage for loosely coupled distributed systems; they enable

controlled access to shared storage and ensure atomicity of read and write operations. Furthermore,

critically important to the design of reliable distributed storage systems are distributed consensus problems,

such as the election of a master from a group of data servers. A master has an important role

in system management; for example, in GFS the master maintains state information about all system

components.

Locking and the election of a master can be done using a version of the Paxos algorithm for asynchronous

consensus, discussed in Section 2.11. The algorithm guarantees safety without any timing

assumptions, a necessary condition in a large-scale system when communication delays are unpredictable.

Nevertheless, the algorithm must use clocks to ensure liveliness and to overcome the impossibility

of reaching consensus with a single faulty process [123]. Coordination using the Paxos algorithm

is discussed in Section 4.5.

Distributed systems experience communication problems such as lost messages, messages out of

sequence, or corrupted messages. There are solutions for handling these undesirable phenomena; for

example, one can use virtual time, that is, sequence numbers, to ensure that messages are processed

in an order consistent with the time they were sent by all processes involved, but this complicates the

algorithms and increases the processing time.

Advisory locks are based on the assumption that all processes play by the rules. Advisory locks do

not have any effect on processes that circumvent the locking mechanisms and access the shared objects

directly. Mandatory locks block access to the locked objects to all processes that do not hold the locks,

regardless of whether they use locking primitives.

Locks that can be held for only a very short time are called fine-grained, whereas coarse-grained

locks are held for a longer time. Some operations require meta-information about a lock, such as the

name of the lock, whether the lock is shared or held in exclusivity, and the generation number of the

lock. This meta-information is sometimes aggregated into an opaque byte string called a sequencer.

The question of how to most effectively support a locking and consensus component of a large-scale

distributed system demands several design decisions. A first design decision is whether the locks should

be mandatory or advisory. Mandatory locks have the obvious advantage of enforcing access control; a

traffic analogy is that a mandatory lock is like a drawbridge. Once it is up, all traffic is forced to stop.

An advisory lock is like a stop sign; those who obey the traffic laws will stop, but some might not. The

disadvantages of mandatory locks are added overhead and less flexibility. Once a data item is locked,

even a high-priority task related to maintenance or recovery cannot access the data unless it forces the

application holding the lock to terminate. This is a very significant problem in large-scale systems where

partial system failures are likely.

A second design decision is whether the system should be based on fine-grained or coarse-grained

locking. Fine-grained locks allow more application threads to access shared data in any time interval,

but they generate a larger workload for the lock server.Moreover, when the lock server fails for a period

of time, a larger number of applications are affected. Advisory locks and coarse-grained locks seem

to be a better choice for a system expected to scale to a very large number of nodes distributed in data

centers that are interconnected via wide area networks.

A third design decision is how to support a systematic approach to locking. Two alternatives come

to mind: (i) delegate to the clients the implementation of the consensus algorithm and provide a library

of functions needed for this task, or (ii) create a locking service that implements a version of the

asynchronous Paxos algorithm and provide a library to be linked with an application client to support

service calls. Forcing application developers to invoke calls to a Paxos library is more cumbersome and

more prone to errors than the service alternative. Of course, the lock service itself has to be scalable to

support a potentially heavy load.

Another design consideration is flexibility, the ability of the system to support a variety of applications.

A name service comes immediately to mind because many cloud applications read and write

small files. The names of small files can be included in the namespace of the service to allow atomic

file operations. The choice should also consider the performance; a service can be optimized and clients

can cache control information. Finally, we should consider the overhead and resources for reaching

consensus. Again, the service seems to be more advantageous because it needs fewer replicas for high

availability.

In the early 2000s, when Google started to develop a lock service called Chubby [61], it was decided

to use advisory and coarse-grained locks. The service is used by several Google systems, including the

GFS discussed in Section 8.5 and BigTable (see Section 8.9).

The basic organization of the system is shown in Figure 8.9. A Chubby cell typically serves one

data center. The cell server includes several replicas, the standard number of which is five. To reduce

the probability of correlated failures, the servers hosting replicas are distributed across the campus of a

data center.

The replicas use a distributed consensus protocol to elect a new master when the current one fails.

The master is elected by a majority, as required by the asynchronous Paxos algorithm, accompanied by

the commitment that a new master will not be elected for a period called a master lease. A session is a

connection between a client and the cell server maintained over a period of time; the data cached by the

client, the locks acquired, and the handles of all files locked by the client are valid for only the duration

of the session.

Clients use RPCs to request services from the master. When it receives a write request, the master

propagates the request to all replicas and waits for a reply from a majority of replicas before responding.

When it receives a read request the master responds without consulting the replicas. The client interface

of the system is similar to, yet simpler than, the one supported by the Unix File System. In addition, it

includes notification of events related to file or system status. A client can subscribe to events such as

file content modification, change or addition of a child node, master failure, lock acquired, conflicting

lock requests, and invalid file handle.

The files and directories of the Chubby service are organized in a tree structure and use a naming

scheme similar to that of Unix. Each file has a file handle similar to the file descriptor. The master of a

cell periodically writes a snapshot of its database to a GFS file server.

Each file or directory can act as a lock. To write to a file the client must be the only one holding

the file handle, whereas multiple clients may hold the file handle to read from the file. Handles are

created by a call to open () and destroyed by a call to close (). Other calls supported by the service

are GetContentsAndStat (), to get the file data and meta-information, SetContents, and Delete () and

several calls allow the client to acquire and release locks. Some applications may decide to create and

manipulate a sequencer with calls to SetSequencer (), which associates a sequencer with a handle, Get-

Sequencer () to obtain the sequencer associated with a handle, or check the validity of a sequencer with

CheckSequencer ().

The sequence of calls SetContents(), SetSequencer(), GetContentsAndStat(), and CheckSequencer()

can be used by an application for the election of a master as follows: all candidate threads attempt to

open a lock file, call it lfile, in exclusive mode; the one that succeeds in acquiring the lock for lfile

becomes the master, writes its identity in lfile, creates a sequencer for the lock of lfile, call it lfseq, and

passes it to the server. The other threads read the lfile and discover that they are replicas; periodically

they check the sequencer lfseq to determine whether the lock is still valid. The example illustrates the

use of Chubby as a name server. In fact, this is one of the most frequent uses of the system.

We now take a closer look at the actual implementation of the service. As pointed out earlier, Chubby

locks and Chubby files are stored in a database, and this database is replicated. The architecture of

these replicas shows that the stack consists of the Chubby component, which implements the Chubby

protocol for communication with the clients, and the active components, which write log entries and

files to the local storage of the replica see (Figure 8.10).

Recall that an atomicity log for a transaction-processing system allows a crash recovery procedure

to undo all-or-nothing actions that did not complete or to finish all-or-nothing actions that committed but did not record all of their effects. Each replica maintains its own copy of the log; a new log entry is

appended to the existing log and the Paxos algorithm is executed repeatedly to ensure that all replicas

have the same sequence of log entries.

The next element of the stack is responsible for the maintenance of a fault-tolerant database – in

other words, making sure that all local copies are consistent. The database consists of the actual data,

or the local snapshot in Chubby speak, and a replay log to allow recovery in case of failure. The state

of the system is also recorded in the database.

The Paxos algorithm is used to reach consensus on sets of values (e.g., the sequence of entries in

a replicated log). To ensure that the Paxos algorithm succeeds in spite of the occasional failure of a

replica, the following three phases of the algorithm are executed repeatedly:

1. Elect a replica to be the master/coordinator. When a master fails, several replicas may decide to

assume the role of a master. To ensure that the result of the election is unique, each replica generates

a sequence number larger than any sequence number it has seen, in the range (1, r ), where r is the

number of replicas, and broadcasts it in a propose message. The replicas that have not seen a higher

sequence number broadcast a promise reply and declare that they will reject proposals from other

candidate masters. If the number of respondents represents a majority of replicas, the one that sent

the propose message is elected master.

2. The master broadcasts to all replicas an accept message, including the value it has selected, and

waits for replies, either acknowledge or reject.

3. Consensus is reached when the majority of the replicas send an acknowledge message; then the

master broadcasts the commit message.

Implementation of the Paxos algorithm is far from trivial. Although the algorithm can be expressed in

as few as ten lines of pseudocode, its actual implementation could be several thousand lines of C++ code

[71]. Moreover, practical use of the algorithm cannot ignore the wide variety of failure modes, including

algorithm errors and bugs in its implementation, and testing a software system of a few thousands lines

of codes is challenging.