How to Build a Highly Available System Using Consensus

Butler W. Lampson[1]

Microsoft
180 Lake View Av., Cambridge, MA 02138

WDAG 1996: Workshop on Distributed Algorithms, Bologna, Italy

Abstract. Lamport showed that a replicated deterministic state machine is a general way to implement a highly available system, given a consensus algorithm that the replicas can use to agree on each input. His Paxos algorithm is the most fault-tolerant way to get consensus without real-time guarantees. Because general consensus is expensive, practical systems reserve it for emergencies and use leases (locks that time out) for most of the computing. This paper explains the general scheme for efficient highly available computing, gives a general method for understanding concurrent and fault-tolerant programs, and derives the Paxos algorithm as an example of the method.

1 Introduction

A system is available if it provides service promptly on demand. The only way to make a highly available system out of less available components is to use redundancy, so the system can work even when some of its parts are broken. The simplest kind of redundancy is replication: make several copies or 'replicas' of each part.

This paper explains how to build efficient highly available systems out of replicas, and it gives a careful specification and an informal correctness proof for the key algorithm. Nearly all of the ideas are due to Leslie Lamport: replicated state machines [5], the Paxos consensus algorithm [7], and the methods of specifying and analyzing concurrent systems [6]. I wrote the paper because after I had read Lamport's papers, it still took me a long time to understand these methods and how to use them effectively. Surprisingly few people seem to know about them in spite of their elegance and power.

In the next section we explain how to build a replicated state machine that is both efficient and highly available, given a fault-tolerant algorithm for consensus. Section 3 gives some background on the consensus problem and its applications. Section 4 reviews the method we use for writing specifications and uses it to give a precise specification for consensus in several forms. In section 5 we introduce the basic idea behind the Paxos algorithm for consensus and derive the algorithm from this idea and the specification. Finally we explain some important optimizations and summarize our conclusions.

[1] Email: blampson@microsoft.com. This paper is at http://www.research.microsoft.com.

2 Replicated State Machines

Redundancy is not enough; to be useful it must be coordinated. The simplest way to do this is to make each non-faulty replica do the same thing. Then any non-faulty replica can provide the outputs; if the replicas are not fail-stop, requiring the same output from f replicas will tolerate $f - 1$ faults. More complicated kinds of redundancy (such as error-correcting codes) are cheaper, but they depend on special properties of the service being provided.

In this section we explain how to coordinate the replicas in a fully general and highly fault tolerant way. Then we explore an optimization called 'leases' that makes the coordination very efficient in nearly all cases.

2.1 Coordinating the Replicas

How can we arrange for each replica to do the same thing? Adopting a scheme first proposed by Lamport [5], we build each replica as a deterministic state machine; this means that the transition relation is a function from (state, input) to (new state, output). It is customary to call one of these replicas a 'process'. Several processes that start in the same state and see the same sequence of inputs will do the same thing, that is, end up in the same state and produce the same outputs. So all we need for high availability is to ensure that all the non-faulty processes see the same inputs. The technical term for this is 'consensus' (sometimes called 'agreement' or 'reliable broadcast'). Informally, we say that several processes achieve consensus if they all agree on some value; we give a formal definition later on.

So if several processes are implementing the same deterministic state machine and achieve consensus on the values and order of the inputs, they will do the same thing. In this way it's possible to replicate an *arbitrary* computation and thus make it highly available. Of course we can make the order a part of the input value by defining some total order on the set of inputs, for instance by numbering them 1, 2, 3, ...

In many applications the inputs are requests from clients to the replicated service. For example, a replicated storage service might have *Read*(*a*) and *Write*(*a, d*) inputs, and an airplane flight control system might have *ReadInstrument*(*i*) and *RaiseFlaps*(*d*) inputs. Different clients usually generate their requests independently, so it's necessary to agree not only on what the requests are, but also on the order in which to serve them. The simplest way to do this is to number them with consecutive integers, starting at 1. This is done in 'primary copy' replication, since it's easy for one process (the primary) to assign consecutive numbers. So the storage service will agree on Input 1 = *Write*(*x, 3*) and Input 2 = *Read*(*x*).

There are many other schemes for achieving consensus on the order of requests when their total order is not derived from consecutive integers; see Schneider's survey [11]. These schemes label each input with some value from a totally ordered set (for instance, (client UID, timestamp) pairs) and then devise a way to be certain that you have seen all the inputs that can ever exist with labels smaller than a given value. This is complicated, and practical systems usually use a primary to sequence the inputs instead.

2.2 Leases: Efficient Highly Available Computing

Fault-tolerant consensus is expensive. Exclusive access by a single process (also known as locking) is cheap, but it is not fault-tolerant—if a process fails while it is holding a lock, no one else can access the resource. Adding a timeout to a lock makes a fault-tolerant lock or ‘lease’. Thus a process holds a lease on a state component or ‘resource’ until an expiration time; we say that the process is the ‘master’ for the resource while it holds the lease. No other process will touch the resource until the lease expires. For this to work, of course, the processes must have synchronized clocks. More precisely, if the maximum skew between the clocks of two processes is ε and process *P*’s lease expires at time *t*, then *P* knows that no other process will touch the resource before time $t - \varepsilon$ on *P*’s clock.

While it holds the lease the master can read and write the resource freely. Writes must take bounded time, so that they can be guaranteed either to fail or to precede any operation that starts after the lease expires; this can be a serious problem for a resource such as a SCSI disk, which has weak ordering guarantees and a long upper bound on the time a write can take.

Locks in transaction processing systems are usually leases; if they expire the transaction is aborted, which means that its writes are undone and the transaction is equivalent to **skip**. A process that uses leases outside the scope of a transaction must take care to provide whatever atomicity is necessary, for example, by ensuring that the resource is in a good state after every atomic write (this is called ‘careful writes’), or by using standard redo or undo methods based on logs [4]. The latter is almost certainly necessary if the resource being leased is itself replicated.

A process can keep control of a resource by renewing its lease before it expires. It can also release its lease, perhaps on demand. If you can’t talk to the process that holds the lease, however (perhaps because it has failed), you have to wait for the lease to expire before touching its resource. So there is a tradeoff between the cost of renewing a lease and the time you have to wait for the lease to expire after a (possible) failure. A short lease means a short wait during recovery but a higher cost to renew the lease. A long lease means a long wait during recovery but a lower cost to renew.

A lease is most often used to give a process the right to cache some part of the state, for instance the contents of a cache line or of a file, knowing that it can’t change. Since a lease is a kind of lock, it can have a ‘mode’ which determines what operations its holder can do. If the lease is exclusive, then its process can change the leased state freely. This is like ‘owner’ access to a cache line or ownership of a multi-ported disk.

2.3 Hierarchical Leases

In a fault-tolerant system leases must be granted and renewed by running consensus. If this much use of consensus is still too expensive, the solution is hierarchical leases. Run consensus once to elect a czar *C* and give *C* a lease on a large part of the state. Now *C* gives out sub-leases on *x* and *y* to masters. Each master controls its own resources. The masters renew their sub-leases with the czar. This is cheap since it doesn’t require any coordination. The czar renews its lease by consensus. This costs more, but there’s only one czar lease. Also, the czar can be simple and less likely to fail, so a longer lease may be acceptable.

Hierarchical leases are commonly used in replicated file systems and in clusters.

By combining the ideas of consensus, leases, and hierarchy, it's possible to build highly available systems that are also highly efficient.

3 Consensus

Several processes achieve consensus if they all agree on some allowed value called the 'outcome' (if they could agree on any value the solution would be trivial: always agree on 0). Thus the interface to consensus has two actions: allow a value, and read the outcome. A consensus algorithm terminates when all non-faulty processes know the outcome.

There are a number of specialized applications for consensus in addition to general replicated state machines. Three popular ones are:

Distributed transactions, where all the processes need to agree on whether a transaction commits or aborts. Each transaction needs a separate consensus on its outcome.

Membership, where a group of processes cooperating to provide a highly available service need to agree on which processes are currently functioning as members of the group. Every time a process fails or starts working again there must be a new consensus.

Electing a leader of a group of processes without knowing exactly what the members are.

Consensus is easy if there are no faults. Here is a simple implementation. Have a fixed *leader* process. It gets all the *Allow* actions, chooses the outcome, and tells everyone. If it were to fail, you would be out of luck. Standard two-phase commit works this way: the allowed value is *commit* if all participants are prepared, and *abort* if at least one has failed. If the leader fails the outcome may be unknown.

Another simple implementation is to have a set of processes, each choosing a value. If a majority choose the same value, that is the outcome (the possible majorities are subsets with the property that any two majorities have a non-empty intersection). If the processes choose so that there is no majority for a value, there is no outcome. If some members of the majority were to fail the outcome would be unknown.

Consensus is tricky when there are faults. In an asynchronous system (in which a non-faulty process can take an arbitrary amount of time to make a transition) with perfect links and even one faulty process, there is no algorithm for consensus that is guaranteed to terminate [3]. In a synchronous system consensus is possible even with processes that have arbitrary or malicious faults (Byzantine agreement), but it is expensive in messages sent and in time [9].

4 Specifications

We are studying systems with a state which is an element of some (not necessarily finite) state space, and a set of actions (not necessarily deterministic) that take the system from one state to another. Data abstractions, concurrent programs, distributed systems, and fault-tolerant systems can all be modeled in this way. Usually we describe the state spaces as the Cartesian product of smaller spaces called 'variables'.

4.1 How to Specify a System with State

In specifying such a system, we designate some of the actions or variables as 'external' and the rest as 'internal'. What we care about is the sequence of external actions (or equivalently, the sequence of values of external variables), because we assume that you can't observe internal actions or variables from outside the system. We call such a sequence a 'trace' of the system. A specification is a set of traces, or equivalently a predicate on traces. Such a set is called a 'property'.

We can define two special kinds of property. Informally, a 'safety' property asserts that nothing bad ever happens; it is the generalization of partial correctness for sequential programs. A 'liveness' property asserts that something good eventually happens; it is the generalization of termination. You can always tell that a trace does not have a safety property by looking at some finite prefix of it, but you can never do this for a liveness property. Any property (that is, any set of sequences of actions) is the intersection of a safety property and a liveness property [2].

In this paper we deal only with safety properties. This seems appropriate, since we know that there is no terminating algorithm for asynchronous consensus. It is also fortunate, because liveness properties are much harder to handle.

It's convenient to define a safety property by a state machine whose actions are also divided into external and internal ones. All the sequences of external actions of the machine define a safety property. Do not confuse these specification state machines with the replicated state machines that we implement using consensus.

We define what it means for one system Y to implement another system X as follows:

- Every trace of Y is a trace of X; that is, X's safety property implies Y's safety property.
- Y's liveness property implies X's liveness property.

The first requirement ensures that you can't tell by observing Y that it isn't X; Y never does anything bad that X wouldn't do. The second ensures that Y does all the good things that X is supposed to do. We won't say anything more about liveness.

Following this method, to specify a system with state we must first define the state space and then describe the actions. We choose the state space to make the spec clear, not to reflect the state of the implementation. For each action we say what it does to the state and whether it is external or internal. We model an action with parameters and results such as *Read*(x) by a family of actions, one of which is *Read*(x) **returning** 3; this action happens when the client reads x and the result is 3.

Here are some helpful hints for writing these specs.

- Notation is important, because it helps you to think about what's going on. Invent a suitable vocabulary.
- Less is more. Fewer actions are better.
- More non-determinism is better, because it allows more implementations.

I'm sorry I wrote you such a long letter; I didn't have time to write a short one.

Pascal

4.2 Specifying Consensus

We are now ready to give specifications for consensus. There is an *outcome* variable initialized to nil, and an action *Allow*(*v*) that can be invoked any number of times. There is also an action *Outcome* to read the *outcome* variable; it must return either nil or a *v* which was the argument of some *Allow* action, and it must always return the same *v*.

More precisely, we have two requirements:

Agreement: Every non-nil result of *Outcome* is the same.

Validity: A non-nil *outcome* equals some allowed value.

Validity means that the outcome can't be any arbitrary value, but must be a value that was allowed. Consensus is reached by choosing some allowed value and assigning it to *outcome*. This spec makes the choice on the fly as the allowed values arrive.

Here is a precise version of the spec, which we call C. It gives the state and the actions of the state machine. The state is:

outcome	: *Value* ∪ {nil}	**initially** nil

The actions are:

Name	Guard	Effect
**Allow*(*v*)		**choose if** *outcome* = nil **then** *outcome* := *v* **or** **skip**
**Outcome*		**choose return** *outcome* **or** **return** nil

Here the external actions are marked with a *. The guard is a precondition which must be true in the current state for the action to happen; it is true (denoted by blank) for both of these actions. The **choose** ... **or** ... denotes non-deterministic choice, as in Dijkstra's guarded commands.

Note that *Outcome* is allowed to return nil even after the choice has been made. This reflects the fact that in an implementation with several replicas, *Outcome* is often implemented by talking to just one of the replicas, and that replica may not yet have learned about the choice.

Next we give a spec T for consensus with termination. Once the internal *Terminate* action has happened, the outcome is guaranteed not to be nil. You can find out whether the algorithm has terminated by calling *Done*. We mark the changes from C by boxing them.

State:	*outcome*	: *Value* ∪ {nil}	**initially** nil
	done	: Bool	**initially** false

Name	Guard	Effect
**Allow*(*v*)		**choose if** *outcome* = nil **then** *outcome* := *v* **or** **skip**
**Outcome*		**choose return** *outcome* **or** **if not** *done* **then** **return** nil
**Done*		**return** *done*
Terminate	outcome ≠ nil	*done* := true

Note that the spec T says nothing about whether termination will actually occur. An implementation in which *Outcome* always returns nil satisfies T. This may seem unsatisfactory, but it's the best we can do with an asynchronous implementation. In other words, a stronger spec would rule out an asynchronous implementation.

Finally, here is a more complicated spec D for 'deferred consensus'. It accumulates the allowed values and then chooses one of them in the internal action *Agree*.

State:	*outcome*	: *Value* ∪ {nil}	**initially** nil
	done	: Bool	**initially** false
	allowed	: **set** *Value*	**initially** {}

Name	Guard	Effect
**Allow*(*v*)		*allowed* := *allowed* ∪ {*v*}
**Outcome*		**choose return** *outcome* **or** **if not** *done* **then return** nil
**Done*		**return** *done*
Agree(*v*)	*outcome* = nil **and** *v* **in** *allowed*	*outcome* := *v*
Terminate	outcome ≠ nil	*done* := true

It should be fairly clear that D implements T. To prove this using the abstraction function method described in the next section, however, requires a prophecy variable or backward simulation [1, 10], because C and T choose the outcome as soon as the see the allowed value, while an implementation may make the choice much later. We have two reasons for giving the D spec. One is that some people find it easier to understand than T, even though it has more state and more actions. The other is to move the need for a prophecy variable into the proof that D implements T, thus simplifying the much more subtle proof that Paxos implements D.

5 Implementation

In this section we first explain the abstraction function method for showing that an implementation meets a specification. This method is both general and practical. Then we discuss some hints for designing and understanding implementations, and illustrate the method and hints with abstraction functions for the simple implementations given earlier. The next section shows how to use the method and hints to derive Lamport's Paxos algorithm for consensus.

5.1 Proving that Y implements X

The definition of 'implements' tells us what we have to do (ignoring liveness): show that every trace of Y is a trace of X. Doing this from scratch is painful, since in general each trace is of infinite length and Y has an infinite number of traces. The proof will therefore require an induction, and we would like a proof method that does this induction once and for all. Fortunately, there is a general method for proving that Y implements X without reasoning explicitly about traces in each case. This method was originally invented by Hoare to prove the correctness of data abstractions. It was generalized by Lamport [6] and others to arbitrary concurrent systems.

The method works like this. First, define an *abstraction function f* from the state of Y to the state of X. Then show that Y *simulates* X:

1) *f* maps an initial state of Y to an initial state of X.

2) For each Y-action and each reachable state *y*
there is a sequence of X-actions (perhaps empty)
that is the same externally,
such that this diagram commutes.

$$\begin{tikzcd}
f(y) \arrow[r, "\text{X-actions}"] & f(y') \\
y \arrow[u, "f"] \arrow[r, "\text{Y-action}"] & y' \arrow[u, "f"]
\end{tikzcd}$$

A sequence of X-actions is the same externally as a Y-action if they are the same after all internal actions are discarded. So if the Y-action is internal, all the X-actions must be internal (perhaps none at all). If the Y-action is external, all the X-actions must be internal except one, which must be the same as the Y-action.

A straightforward induction shows that Y implements X: For any Y-behavior we can construct an X-behavior that is the same externally, by using (2) to map each Y-action into a sequence of X-actions that is the same externally. Then the sequence of X-actions will be the same externally as the original sequence of Y-actions.

If Y implements X, is it always to possible to prove it using this method. The answer is "Almost". It may be necessary to modify Y by adding auxiliary 'history' and 'prophecy' variables according to certain rules which ensure that the modified Y has exactly the same traces as Y itself. With the right history and prophecy variables it's always possible to find an abstraction function [1]. An equivalent alternative is to use an abstraction relation rather than an abstraction function, and to do 'backward' simulation as well as the 'forward' simulation we have just described [10]. We mention these complications for completeness, but avoid them in this paper.

In order to prove that Y simulates X we usually need to know what the reachable states of Y are, because it won't be true that every action of Y from an arbitrary state of Y simulates a sequence of X-actions; in fact, the abstraction function might not even be defined on an arbitrary state of Y. The most convenient way to characterize the reachable states of Y is by an *invariant*, a predicate that is true of every reachable state. Often it's helpful to write the invariant as a conjunction; then we call each conjunct an invariant. It's common to need a stronger invariant than the simulation requires; the extra strength is a stronger induction hypothesis that makes it possible to establish what the simulation does require.

So the structure of a proof goes like this:

- Define an abstraction function.
- Establish invariants to characterize the reachable states, by showing that each action maintains the invariants.
- Establish the simulation, by showing that each Y-action simulates a sequence of X-actions that is the same externally.

This method works only with actions and does not require any reasoning about traces. Furthermore, it deals with each action independently. Only the invariants connect the actions. So if we change (or add) an action of Y, we only need to verify that the new action maintains the invariants and simulates a sequence of X-actions that is the same externally.

In the light of this method, here are some hints for deriving, understanding, and proving the correctness of an implementation.

- Write the specification first.
- Dream up the idea of the implementation. This is the crucial creative step. Usually you can embody the key idea in the abstraction function.
- Check that each implementation action simulates some spec actions. Add invariants to make this easier. Each action must maintain them. Change the implementation (or the spec) until this works.
- Make the implementation correct first, then efficient. More efficiency means more complicated invariants. You might need to change the spec to get an efficient implementation.

An efficient program is an exercise in logical brinksmanship
Dijkstra

In what follows we give abstraction functions for each implementation we consider, and invariants for the Paxos algorithm. The actual proofs that the invariants hold and that each Y-action simulates a suitable sequence of X-actions are routine, and we omit them.

5.2 Abstraction Functions for the Simple Implementations

Recall our two simple, non-fault-tolerant implementations. In the first a single *leader* process, with the same state as the specification, tells everyone else the outcome (this is how two-phase commit works). The abstraction function to C is:

outcome = the *outcome* of the coordinator.

done = everyone has gotten the outcome.

This is not fault-tolerant—it fails if the leader fails.

In the second there is set of processes, each choosing a value. If a majority choose the same value, that is the outcome. The abstraction function to C is:

outcome = the choice of a majority, or nil if there's no majority.

done = everyone has gotten the outcome.

This is not fault-tolerant—it fails if a majority doesn't agree, or if a member of the majority fails.

6 The Paxos Algorithm

In this section we describe Lamport's Paxos algorithm for implementing consensus [7]. This algorithm was independently invented by Liskov and Oki as part of a replicated data storage system [8]. Its heart is the best known asynchronous consensus algorithm. Here are its essential properties:

It is run by a set of *leader* processes that guide a set of *agent* processes to achieve consensus.

It is correct no matter how many simultaneous leaders there are and no matter how often leader or agent processes fail and recover, how slow they are, or how many messages are lost, delayed, or duplicated.

It terminates if there is a single leader for a long enough time during which the leader can talk to a majority of the agent processes twice.

It may not terminate if there are always too many leaders (fortunate, since we know that guaranteed termination is impossible).

To get a complete consensus algorithm we combine this with a sloppy timeout-based algorithm for choosing a single leader. If the sloppy algorithm leaves us with no leader or more than one leader for a time, the consensus algorithm may not terminate during that time. But if the sloppy algorithm ever produces a single leader for long enough the algorithm will terminate, no matter how messy things were earlier.

We first explain the simplest version of Paxos, without worrying about the amount of data stored or sent in messages, and then describe the optimizations that make it reasonably efficient. To get a really efficient system it's usually necessary to use leases as well.

6.1 The Idea

First we review the framework described above. There is a set of agent processes, indexed by a set *I*. The behavior of an agent is deterministic; an agent does what it's told. An agent has 'persistent' storage that survives crashes. The set of agents is fixed for a single run of the algorithm (though it can be changed using the Paxos algorithm itself). There are also some leader processes, indexed by a totally ordered set *L*, that tell the agents what to do. Leaders can come and go freely, they are not deterministic, and they have no persistent storage.

The key idea of Paxos comes from the non-fault-tolerant majority consensus algorithm described earlier. That algorithm gets into trouble if the agents can't agree on a majority, or if some members of the majority fail so that the rest are unsure whether consensus was reached. To fix this problem, Paxos has a sequence of *rounds* indexed by a set *N*. Round *n* has a single leader who tries to get a majority for a single value v_n. If one round gets into trouble, another one can make a fresh start. If round *n* achieves a majority for v_n then v_n is the outcome.

Clearly for this to work, any two rounds that achieve a majority must have the same value. The tricky part of the Paxos algorithm is to ensure this property.

In each round the leader

- *queries* the agents to learn their status for past rounds,
- chooses a value and *commands* the agents, trying to get a majority to accept it, and
- if successful, distributes the value as the outcome to everyone.

It takes a total of 2½ round trips for a successful round. If the leader fails repeatedly, or several leaders fight it out, it may take many rounds to reach consensus.

6.2 State and Abstraction Function

The state of an agent is a persistent 'status' variable for each round; persistent means that it is not affected by a failure of the agent. A status is either a *Value* or one of the special symbols *no* and *neutral*. The agent actions are defined so that a status can only change if it is *neutral*. So the agent state is defined by an array *s*:

State: $s_{i,\,n}$: *Value* ∪ {*no*, *neutral*} **initially** *neutral*

A round is *dead* if a majority has status *no*, and *successful* if a majority has status which is a *Value*.

The state of a leader is the round it is currently working on (or nil if it isn't working on a round), the value for that round (or nil if it hasn't been chosen yet), and the leader's idea of the *allowed* set.

State:	n_l	: *N* ∪ {nil}	**initially** nil
	u_l	: *Value* ∪ {nil}	**initially** nil
	allowed$_l$	: **set** *Value*	**initially** {}

The abstraction function for *allowed* is just the union of the leaders' sets:

AF: $allowed = \bigcup_{l \in L} allowed_l$

We define the value of round n as follows:

$v_n \equiv$ **if** some agent i has a *Value* in $s_{i,\,n}$ **then** $s_{i,\,n}$
else nil

For this to be well defined, we must have

Invariant 1: A round has at most one value.

That is, in a given round all the agents with a value have the same value. Now we can give the abstraction function for *outcome*:

AF: $outcome = v_n$ for some successful round n
or nil if there is no successful round.

For this to be well defined, we must have

Invariant 2: Any two successful rounds have the same value.

We maintain invariant 1 (a round has at most one value) by ensuring that a leader works on only one round at a time and never reuses a round number, and that a round has at most one leader. To guarantee the latter condition, we make the leader's identity part of the round number by using (sequence number, leader identity) pairs as round numbers. Thus $N = (J, L)$, where J is some totally ordered set, usually the integers. Leader l chooses (j, l) for n_l, where j is a J that l has not used before. For instance, j might be the current value of local clock. We shall see later how to avoid using any stable storage at the leader for choosing j.

6.3 Invariants

We introduce the notion of a *stable* predicate on the state, a predicate which once true, remains true henceforth. This is important because it's safe to act on the truth of a stable predicate. Anything else might change because of concurrency or crashes.

Since a non-*neutral* value of $s_{i,\,n}$ can't change, the following predicates are stable:

$s_{i,\,n} = no$
$s_{i,\,n} = v$
$v_n \ = v$

n is dead
n is successful

Here is a more complex stable predicate:

n is anchored $\equiv$ **for all** $m \le n$, m is dead **or** $v_n = v_m$

In other words, n is anchored iff when you look back at rounds before n, skipping dead rounds, you see the same value as n. If all preceding rounds are dead, n is anchored no matter what its value is. For this to be well-defined, we need a total ordering on *N*'s, and we use the lexicographic ordering.

Now all we have to do is to maintain invariant 2 while making progress towards a successful round. To see how to maintain the invariant, we strengthen it until we get a form that we can easily maintain with a distributed algorithm, that is, a set of actions each of which uses only the local state of a process.

Invariant 2: Any two successful rounds have the same value.

follows from

Invariant 3: **for all** n and $m \le n$, **if** m is successful **then** v_n = nil **or** $v_n = v_m$

which follows from

Invariant 4: **for all** n and $m \le n$, **if** m is not dead **then** v_n = nil **or** $v_n = v_m$

and we rearrange this by predicate calculus

≡ **for all** n and $m \le n$, m is dead **or** v_n = nil **or** $v_n = v_m$

≡ **for all** n, v_n = nil **or** (**for all** $m \le n$, m is dead **or** $v_n = v_m$)

≡ **for all** n, v_n = nil **or** n is anchored

So all we have to do is choose each non-nil v_n so that
there is only one, and
n is anchored.
Now the rest of the algorithm is obvious.

6.4 The Algorithm

A leader has to choose the value of a round so that the round is anchored. To accomplish this, the leader l chooses a new n_l and *queries* all the agents to learn their status in *all* rounds with numbers less than n_l (and also the values of those rounds). Before an agent responds, it changes any *neutral* status for a round earlier than n_l to *no*, so that the leader will have enough information to anchor the round. Responses to the query from a majority of agents give the leader enough information to make round n_l anchored, as follows:

> The leader looks back from n_l, skipping over rounds for which no *Value* status was reported, since these must be dead (remember that l has heard from a majority, and the reported status is a *Value* or *no*). When l comes to a round n with a *Value* status, it chooses that value v_n as u_l. Since n is anchored by invariant 4, and all the rounds between n and n_l are dead, n_l is also anchored if this u_l becomes its value.
>
> If all previous rounds are dead, the leader chooses any allowed value for u_l. In this case n_l is certainly anchored.

Because 'anchored' and 'dead' are stable properties, no state change can invalidate this choice.

Now in a second round trip the leader *commands* everyone to accept u_l for round n_l. Each agent that is still neutral in round n_l (because it hasn't answered the query of a later round) *accepts* by changing its status to u_l in round n_l; in any case it reports its status to the leader. If the leader collects u_l reports from a majority of agents, then it knows that round n_l has succeeded, takes u_l as the agreed outcome of the algorithm, and sends this fact to all the processes in a final half round. Thus the entire process takes five messages or 2½ round trips.

Note that the round succeeds (the *Agree* action of the spec happens and the abstract *outcome* changes) at the instant that some agent forms a majority by accepting its value, even though no agent or leader knows at the time that this has happened. In fact, it's possible for the round to succeed without the leader knowing this fact, if some agents fail after accepting but before getting their reports to the leader, or if the leader fails.

An example may help your intuition about why this works. The table below shows three rounds in two different runs of the algorithm with three agents *a*, *b*, and *c* and *allowed* = {7, 8, 9}. In the left run all three rounds are dead, so if the leader hears from all three agents it knows this and is free to choose any allowed value. If the leader hears only from *a* and *b* or *a* and *c*, it knows that round 3 is dead but does not know that round 2 is dead, and hence must choose 8. If it hears only from *b* and *c*, it does not know that round 3 is dead and hence must choose 9.

In the right run, no matter which agents the leader hears from, it does not know that round 2 is dead. In fact, it was successful, but unless the leader hears from *a* and *c* it doesn't know that. Nonetheless, it must choose 9, since it sees that value in the latest non-dead round. Thus a successful round such as 2 acts as a barrier which prevents any later round from choosing a different value.

				Status				
	v_n	$s_{a,n}$	$s_{b,n}$	$s_{c,n}$	v_n	$s_{a,n}$	$s_{b,n}$	$s_{c,n}$
Round 1	7	7	no	no	8	8	no	no
Round 2	8	8	no	no	9	9	no	9
Round 3	9	no	no	9	9	no	no	9
Leader's choices for round 4	7, 8, or 9 if *a, b, c* report 8 if *a, b* or *b, c* report 9 if *b, c* report				9 no matter what majority reports			

Presumably the reason there were three rounds in both runs is that at least two different leaders were involved, or the leader failed before completing each round. Otherwise round 1 would have succeeded. If leaders keep overtaking each other and forcing the agents to set their earlier status to *no* before an earlier rounds reach the command phase, the algorithm can continue indefinitely.

Here are the details of the algorithm. Its actions are the ones described in the table together with boring actions to send and receive messages.

Leader l	**Message**	**Agent** i
Choose a new n_l		

Query a majority of agents for their status	query(n_l) →	for all $m < n_l$,
		if $s_{i,m} = neutral$
		then $s_{i,m} := no$

	← report(i, s_i)	
Choose u_l to keep n_l anchored. If all $m < n_l$ are dead, choose any v in *allowed*$_l$		
Command a majority of agents to accept u_l	command(n_l, u_l) →	**if** $s_{i,n_l} = neutral$ **then** $s_{i,n_l} := u_l$
	← report(i, n_l, s_{i,n_l})	
If a majority accepts, publish the outcome u_l	outcome(u_l) →	

The algorithm makes minimal demands on the properties of the network: lost, duplicated, or reordered messages are OK. Because nodes can fail and recover, a better network doesn't make things much simpler. We model the network as a broadcast medium from leader to agents; in practice this is usually implemented by individual messages to each agent. Both leaders and agents can retransmit as often as they like; in practice agents retransmit only in response to the leader's retransmission.

A complete proof requires modeling the communication channels between the processes as a set of messages which can be lost or duplicated, and proving boring invariants about the channels of the form "if a message report(i, s) is in the channel, then s_i agrees with s except perhaps at some *neutral* components of s."

6.5 Termination: Choosing a Leader

When does the algorithm terminate? If no leader starts another round until after an existing one is successful, then the algorithm definitely terminates as soon as the leader succeeds in both querying and commanding a majority. It doesn't have to be the same majority for both, and the agents don't all have to be up at the same time. Therefore we want a single leader, who runs one round at a time. If there are several leaders, the one running the biggest round will eventually succeed, but if new leaders keep starting bigger rounds, none may ever succeed. We saw a little of this behavior in the example above. This is fortunate, since we know from the Fischer-Lynch-Paterson result [3] that there is no algorithm that is guaranteed to terminate.

It's easy to keep from having two leaders at once if there are no failures for a while, the processes have clocks, and the *usual* maximum time to send, receive, and process a message is known:

Every potential leader that is up broadcasts its name.

You become the leader one round-trip time after doing a broadcast, unless you have received the broadcast of a bigger name.

Of course this algorithm can fail if messages are delayed or processes are late in responding to messages. When it fails, there may be two leaders for a while. The one running the largest round will succeed unless further problems cause yet another leader to arise.

7 Optimizations

It's not necessary to store or transmit the complete agent state. Instead, everything can be encoded in a small fixed number of bits, as follows. The relevant part of s_i is just the most recent *Value* and the later *no* states:

$s_{i,\, last_i} = v$
$s_{i,\, m} = no$ for all m between $last_i$ and $next_i$
$s_{i,\, m} = neutral$ for all $m \geq next_i$.

We can encode this as $(v, last_i, next_i)$. This is all that an agent needs to store or report.

Similarly, all a leader needs to remember from the reports is the largest round for which a value was reported and which agents have reported. It is not enough to simply count the reports, since report messages can be duplicated because of retransmissions.

The leaders need not be the same processes as the agents, although they can be and usually are. A leader doesn't really need any stable state, though in the algorithm as given it has something that allows it to choose an n that hasn't been used before. Instead, it can poll for the $next_i$ from a majority after a failure and choose an n_l with a bigger j. This will yield an n_l that's larger than any from this leader that has appeared in a command message so far (because n_l can't get into a command message without having once been the value of $next_i$ in a majority of agents), and this is all we need.

If Paxos is used to achieve consensus in a non-blocking commit algorithm, the first round-trip (query/report) can be combined with the prepare message and its response.

The most important optimization is for a sequence of consensus problem, usually the successive steps of a replicated state machine. We try to stay with the same leader, usually called the 'primary', and run a sequence of instances of Paxos numbered by another index p. The state of an agent is now

State: $s_{p,\, i,\, n}$: $Value \cup \{no, neutral\}$ **initially** *neutral*

Make the fixed size state $(v, last, next, p)$ for agent i encode

$s_{q,\, i,\, m} = no$ for all $q \leq p$ and $m < next$.

Then a query only needs to be done once each time the leader changes. We can also piggyback the outcome message on the command message for the next instance of Paxos. The result is 2 messages (1 round trip) for each consensus.

8 Conclusion

We showed how to build a highly available system using consensus. The idea is to un a replicated deterministic state machine, and get consensus on each input. To make it efficient, use leases to replace most of the consensus steps with actions by one process.

We derived the most fault-tolerant algorithm for consensus without real-time guarantees. This is Lamport's Paxos algorithm, based on repeating rounds until you get a majority, and ensuring that every round after a majority has the same value. We saw how to implement it with small messages, and with one round-trip for each consensus in a sequence with the same leader.

Finally, we explained how to design and understand a concurrent, fault-tolerant system. The recipe is to write a simple spec as a state machine, find the abstraction function from the implementation to the spec, establish suitable invariants, and show that the implementation simulates the spec. This method works for lots of hard problems.

9 References

1. M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science* **82**, 2, May 1991.
2. B. Alpern and F. Schneider. Defining liveness. *Information Processing Letters* **21**, 4, 1985.
3. M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM* **32**, 2, April 1985.
4. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
5. L. Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks* **2**, 1978.
6. L. Lamport. A simple approach to specifying concurrent systems. *Comm. ACM,* **32, 1,** Jan. 1989.
7. L. Lamport. The part-time parliament. Technical Report 49, Systems Research Center, Digital Equipment Corp., Palo Alto, Sep. 1989.
8. B. Liskov and B. Oki. Viewstamped replication, *Proc. 7th PODC*, Aug. 1988.
9. N Lynch. *Distributed Algorithms.* Morgan Kaufmann, 1996.
10. N. Lynch and F. Vaandrager. Forward and backward simulations for timing-based systems. *Lecture Notes in Computer Science* 600, Springer, 1992.
11. F. Schneider. Implementing fault-tolerant services using the state-machine approach: A tutorial. *Computing Surveys* **22** (Dec 1990).