18. Consensus

Consensus (sometimes called 'reliable broadcast' or 'atomic broadcast') is a fundamental building block for distributed systems. Informally, we say that several processes achieve consensus if they all agree on some value. Three obvious applications are:

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

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

Electing a leader of a group of processes.

A less obvious, but much more powerful application is to replicate that state machines, which are discussed in detail below and in handout 28.

There are four important things to learn from this part of the course:

The idea of replicated state machines as a completely general method for building highly available, fault tolerant systems. In handout 28 we will discuss replicated state machines and other methods for fault tolerance in more detail.

The Paxos algorithm for distributed, fault tolerant consensus: how and why it works .

Paxos as an example of the best style for distributed, fault tolerant algorithms.

The correctness of Paxos as an example of the abstraction functions and simulation proofs applied to a very subtle algorithm.

Replicated state machines

There is a much more general way to use consensus, as the mechanism for coding a highly available state machine, which is the basic tool for building a highly available system. The way to get availability is to have either perfect components or redundancy. Perfect components are too hard, which leaves redundancy. The simplest form of redundancy is replication: have several copies or *replicas* of each component, and make sure that all the non-faulty components do the same thing. Since any computation can be expressed as a state machine, a replicated state machine can make any computation highly available.

Recall the basic idea of a replicated state machine:

If the transition relation is deterministic (in other words, is a function from (state, input) to (new state, output)), then several copies of the state machine 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 if several processes are implementing the same 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 value of the input by defining some total order on the set of possible inputs;[1] the easiest way to do this is simply to number them 1, 2, 3, ... We have already seen one application of this replicated state machine idea, in the code for transactions; there the replication takes the form of redoing a sequence of actions that is remembered in a log.

Suppose, for example, that we want to build a highly available file system. The transitions are read and write operations on the files (and rename, list, … as well). We make several copies of the file system and make sure that they process read and write operations in the same order. A client sends its operation to some copy, which gets consensus that it is the next operation. Then all the copies do the operation, and one of them returns the result to the client.

In many applications the inputs are requests from clients to the replicated service. Typically different clients 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 especially easy in the usual implementation, 'primary copy' replication, since there's one place (the primary) to assign consecutive numbers. As we shall see, however, it's straightforward in any consensus scheme: you get consensus on input 1, then on input 2, etc.

You might think that a read could be handled by any copy with no need for consensus, since it doesn't change the state of the file system. Without consensus, however, a read might fail to see the result of a write that finished before the read started, since the read might be handled by a copy whose state is behind the current state of the file system. This result violates "external consistency", which is a formal expression of the usual intuition about state machines. In some applications, however, it is acceptable to get a possibly old result from a read, and then any copy can satisfy it without consensus. Another possibility is to notice that a given copy will have done all the operations up to *n*, and define a read operation that returns *n* along with the result value, and possibly the real time of operation *n* as well. Then it's up to the client to decide whether this is recent enough.

The literature is full of other schemes for achieving consensus on the order of requests when their total order is not derived from consecutive integers. These schemes label each input with some label from a totally ordered set (for instance, (client UID, timestamp) pairs) and then devise some way to be certain that you have seen all the inputs that can ever exist with labels smaller than a given value. They are complicated, and of doubtful utility.[2] People who do it for money use primary copy.[3]

[1] This approach was first proposed in a classic paper by Leslie Lamport: Time, clocks, and the ordering of events in a distributed system, *Comm. ACM* **21**, 7, July 1978, pp 558-565. This paper is better known for its analysis of the partial ordering of events in a distributed system, which is too bad.
[2] For details, see F. Schneider, Implementing fault-tolerant services using the state-machine approach: A tutorial, *ACM Computing Surveys* **22** (Dec 1990). This paper is reprinted in the book *Distributed Systems*, 2nd edition, ed. S. Mullender, Addison-Wesley, 1993, pp 169-197.
[3] Jim Gray said this about using locks for concurrency control; see handout 20.

Unfortunately, consensus is expensive. The section on optimizations at the end of this handout explains a variety of ways to make a replicated state machine run efficiently: leases, transactions, and batching.

Spec for consensus

Here is the spec for consensus; we have seen it already in handout 8 on history and prophecy variables. The idea is that the outcome of consensus should be one and only one of the allowed values. In the spec 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 if it doesn't return `nil` 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.

```
MODULE Consensus [V] EXPORT Allow, Outcome =         % data value to agree on

VAR  outcome   : (V + Null) := nil

APROC Allow(v) = << outcome = nil => outcome := v [] SKIP >>
APROC Outcome() -> (V + Null) = << RET outcome [] RET nil >>

END Consensus
```

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

If only one `Allow` action occurs, there's no need to choose a `v`, and the code's only problem is to ensure termination. An algorithm that does so is said to implement 'reliable' or 'atomic' broadcast; there is only one sender, and either everyone or no one gets the message. The single `Allow` might not set `outcome`, which corresponds to failure of the sender of the broadcast message; in this case no one gets the message.

Here is an equivalent spec, slightly more complicated but perhaps more intuitive, and certainly closer to an implementation. It accumulates the allowed values and then chooses one of them in the internal action `Agree`.

```
MODULE LateConsensus [V] EXPORT Allow, Outcome =

VAR outcome     :  (V + Null) := nil
    allowed     :  SET V := {}
```

```
APROC Allow(v) = << allowed \/ := {v} >>

APROC Outcome() -> (V + Null) = << RET outcome [] RET nil >>
% Only outcome is visible

APROC Decide() = << VAR v :IN allowed | outcome = nil => outcome := v >>

END LateConsensus
```

It should be fairly clear that `LateConsensus` implements `Consensus`. An abstraction function to prove this, however, requires a prophecy variable, because `Consensus` decides on the outcome (in the `Allow` action) before `LateConsensus` does (in the `Decide` action). We saw these specs in handout 8 on generalized abstraction functions, where prophecy variables are explained.

In the code we have in mind, there are some processes, each with its own `outcome` variable initialized to `nil`. The `outcome` variables are supposed to reach consensus, that is, become equal to the argument of some `Allow` action. An `Outcome` can be directed to any process, which returns the value of its `outcome` variable. The tricky part is to ensure that two non-nil `outcome` variables are always equal, so that the agreement property is satisfied.

We would also like to have the property that eventually `Outcome` stops returning `nil`. In the code, this happens when every process' `outcome` variable is non-`nil`. However, this could take a long time if some process is very slow (or down for a long time).

We can change `Consensus` to express this with an internal action `Done`:

```
MODULE TerminatingConsensus [V] EXPORT Allow, Outcome =

VAR outcome      :  (V + Null) := nil
    done         :  Bool := false

APROC Allow(v) = << outcome = nil => outcome := v [] SKIP >>
APROC Outcome() -> (V + Null) = << RET outcome [] ~ done => RET nil >>

THREAD Done() = << outcome # nil => done := true >>

END TermConsensus
```

Note that this spec does not say anything about the processes in the assumed code; the abstraction function will say that `done` is true when all the processes have `outcome` ≠ `nil`.

An even stronger spec returns an `outcome` only when it's done:

```
APROC Outcome() -> (V + Null) = << done => RET outcome [] ~ done => RET nil >>
```

This is usually too strong for distributed code. It means that a process may not be able to respond to an `Outcome` request, since it can't return a value if it doesn't know the outcome yet, and it can't return `nil` if anyone else has already returned a value. If either the processes or the communication are asynchronous, it won't be possible in general for one process to know whether another one no longer matters because it has failed, or is just slow.

Facts about consensus

In this section we summarize the most important facts about when consensus is possible and what it costs. You can learn more about this in Nancy Lynch's course on distributed algorithms, 6.852J, or in her book cited in handout 2.

Fault models

To devise code for `Consensus` we need a precise model for the general setting of processes connected by links that can communicate messages from one process to another. In particular, the model must define what faults are possible. There are lots of ways to do this, and we have space to describe only the models that are most popular and closest to reality.

There are two broad classes of models:

- *Synchronous*, in which a non-faulty component makes its state transitions within a known amount of time. Usually this is coded by using a timeout, and declaring a component faulty if it fails to perform within the specified time.
- *Asynchronous*, in which nothing is known about the relative rate of progress of different components. In particular, a process can take an arbitrary amount of time to make a transition, and a link can take an arbitrary amount of time to deliver a message.

In general a process can send a message only to certain other processes; this "can send message" relation defines a graph whose edges are the links. The graph may be directed (it's possible that *A* can talk to *B* but *B* can't talk to *A*), but we will assume that communication is full-duplex so that the graph is undirected. Links are either working or faulty; a faulty link delivers no messages. Even a working link may lose messages, and in some models may lose any number of messages; it's helpful to think of such a system as one with totally asynchronous communication.

Processes are either working or faulty. There are two models for a faulty process:

- *Stopping* faults: a faulty process stops making transitions and doesn't start again. In an asynchronous model there's no way for another process to distinguish a stopped process or link from one that is simply very slow.
- *Byzantine* faults: a faulty process makes arbitrary transitions; these are named after the Byzantine Empire, famous for treachery. The motivation for this model is usually not fear of treachery, but ignorance of the ways in which a process might fail. Clearly Byzantine failure is an upper bound on how bad things can be.

Is consensus possible (will it terminate)?

A consensus algorithm terminates when the outcome variables of all non-faulty processes equal some allowed value. Here are the basic facts about consensus in some of these models.

- There is no consensus algorithm that is guaranteed to terminate in an asynchronous system with perfect links and even one process that has a stopping fault. This startling result is due to Fischer, Lynch, and Paterson.[4] It holds even if the communication system provides reliable broadcast that delivers each message either to all the non-faulty processes or to none of them. Real systems get around it by using timeout to make the system synchronous, or by using randomness.
- Even in a synchronous system with perfect processes there is no consensus algorithm that is guaranteed to terminate if an unbounded number of messages can be lost (that is, if communication is effectively asynchronous). The reason is that the last message sent must be pointless, since it might be lost. So it can be dropped to get a shorter algorithm. Repeat this argument to drop all the messages. But clearly an algorithm with no messages can't achieve consensus. The simplest case of this problem, with just two processes, is called the "two generals problem".
- In a system with both synchronous processes and synchronous communication, terminating consensus is possible. If *f* faults are allowed, then:

  For processes with stopping faults, consensus requires *f*+1 processes and an *f*+1-connected[5] network (that is, at least one good process and a connected subnet of good processes after all the allowed faults have happened). Even if the network is fully connected, it takes *f*+1 rounds to reach consensus in the worst case.

  For processors with Byzantine faults, consensus requires 3*f*+1 processes, a 2*f*+1-connected network, at least *f*+1 rounds of communication, and 2^f bits of data communicated.

  For processors with Byzantine faults and digital signatures (so that a process can present unforgeable evidence that another process sent it a message), consensus requires *f*+1 processes. Even if the network is fully connected, it takes *f*+1 rounds to reach consensus in the worst case.

The amount of communication required depends on the number of faults, the complexity of the algorithm, etc. Randomized algorithms can achieve better results with arbitrarily high probability.

Warning: In many applications the model of no more than *f* faults may not be realistic if the system is allowed to do the wrong thing when the number of faults exceeds *f*. It's often more important to do either the right thing or nothing.

The simplest consensus algorithms

There are two simple and popular algorithms for consensus. Both have the problem that they are not very fault-tolerant.

[4] Fischer, M., Lynch, N., and Paterson, M., Impossibility of distributed consensus with one faulty process, *J. ACM* **32**, 2, April 1985, pp 374-382.

[5] A graph is connected if there is a path (perhaps traversing several links) between any two nodes, and disconnected otherwise. It is *k*-connected if *k* is the smallest number such that removing *k* links can leave it disconnected.

- A fixed 'leader', 'master', or 'coordinator' process that works like the `Consensus` spec: it gets all the `Allow` actions, chooses the outcome, and tells everyone. If it fails, you are out of luck. The abstraction function is just the identity on the leader's state; `TerminatingConsensus.done` is true iff everyone has gotten the outcome (or failed permanently). Standard two-phase commit for distributed transactions works this way.
- Simple majority voting. The abstraction function for `outcome` is the value that has a majority, or `nil` if there isn't one. This fails if you don't get a majority, or if enough members of a majority fail that it isn't a majority any more. In the latter case you can't determine the outcome. Example: `a` votes for `l1`, `b` and `c` vote for `l2`, and `b` fails. Now all you can see is one vote for `l1` and one for `l2`, so you can't tell that `l2` had a majority.

The Paxos algorithm: The idea

In the rest of this handout, we describe Lamport's Paxos algorithm for coding asynchronous consensus; Liskov and Oki independently invented this algorithm as part of a replicated data storage system.[6] Its heart is the best asynchronous algorithm known, which is

> run by a set of *proposer* processes that guide a set of *acceptor* processes to achieve consensus,
>
> correct no matter how many simultaneous proposers there are and no matter how often proposer or acceptor processes fail and recover or how slow they are, and
>
> guaranteed to terminate if there is a single proposer for a long enough time during which each member of a majority of the acceptor processes is up for long enough, but
>
> possibly non-terminating if there are always too many proposers (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 proposer. If the sloppy algorithm leaves us with no proposer or more than one proposer for a time, the consensus algorithm may not terminate during that time. But if the sloppy algorithm ever produces a single proposer for long enough the algorithm will terminate, no matter how messy things were earlier.

Paxos is the way to do consensus if you want a high degree of fault-tolerance, don't have any real-time requirements, and can't tightly control the time to transmit and process a message. There isn't any simpler algorithm that has the same fault-tolerance. There is lots of code for consensus that doesn't work.

[6] L. Lamport, The part-time parliament, Technical report 49, Systems Research Center, Digital Equipment Corp, Palo Alto, Sep. 1989, finally published in *ACM Transactions on Computer Systems* **16**, 2 (May 1998), pp 133-169. Unfortunately, the terminology of this paper is confusing. B. Liskov and B. Oki, Viewstamped replication: A new primary copy method to support highly available distributed systems, *Proc. 7th ACM Conference on Principles of Distributed Computing*, Aug. 1988. In this paper the consensus algorithm is intertwined with the replication algorithm. See also B. Lampson, The ABCDs of Paxos, at http://research.microsoft.com/lampson/65-ABCDPaxos/Abstract.html.

The grand plan of the algorithm is to have a sequence of *rounds*, each with a single proposer. This attacks the problem with simple majority voting, which is that a single attempt to get a majority may fall victim to failure. Each Paxos round is a distinct attempt to get a majority. Each acceptor has a state variable `s(a)` that is a function of the round; that is, there's a state value `s(a)(n)` for each round `n`. To reduce clutter, we write this s_n^a. In each round the proposer:

> *queries* the acceptors to learn their state for past rounds,
>
> chooses a *safe* value `v`,
>
> *commands* the acceptors, trying to get a majority to *accept* `v`, and
>
> if it gets a majority, that's a *decision*, and it distributes `v` as the *outcome* to everyone.

The outcome is the value accepted by a majority in some round. The tricky part of the algorithm is to ensure that there is only one such value, even though there may be lots of rounds.

Most descriptions of Paxos call the acceptors 'voters'. This is unfortunate, because the acceptors do not make any decisions; they do whatever a proposer requests, unless they have already done something inconsistent with that. In fact, an acceptor can be coded by a memory that has a compare-and-swap operation, as we shall see later. Of course, the proposers and acceptors can run on the same machine, and even in the same process. This is usually the way it's coded, but the algorithm with separate proposers and acceptors is easier to explain.

It takes a total of 21/2 round trips for a deciding round. If there's only one proposer that doesn't fail, Paxos reaches consensus in one round. If the proposer fails repeatedly, or several proposers fight it out, it may take arbitrarily many rounds to reach consensus. This may seem bad, but actually it is a good thing, since we know from Fischer-Lynch-Paterson that we can't have an algorithm that is guaranteed to terminate.

The rounds are numbered (not necessarily consecutively) with numbers of type `N`, and the numbers determine a total ordering on the rounds. Each round has a single value, which starts out `nil` and then may change to one of the allowed values; we write v_n for the value of round `n`. In each round an acceptor starts out `neutral`, and it can only change to v_n or `no`. A v_n or `no` state can't change. Note that different rounds can have different values. A round is *dead* if a majority has state `no`, and *decides* if a majority has state v_n. If a round decides, that round's value is the outcome.

The state of Paxos that contributes to the abstraction function to `LateConsensus` is

```
MODULE Paxos [                                          % implements Consensus
               V,                                       % data Value to decide on
               P WITH {"<=": (P, P)->Bool},             % Proposer; <= a total order
               A WITH {majority : SET A->Bool} ]        % Acceptor; majorities must intersect

TYPE I      = Int
     N      = [i, p] WITH {"<=":=LEqN}                  % round Number; <= is total
     Z      = (ENUM[no, neutral] + V)                   % one acceptor's state in one round
     S      = A -> N -> Z                               % Acceptors' states

VAR  s      : S            := {*->{*->neutral}          % acceptor working States
```

```
    outcome : A -> (V+Null) := {*->nil}               % acceptor outcome values
    allowed : P -> SET V    := {*->{}}                % proposer states
```

% Agreement: `(outcome.rng – {nil}).size <= 1`

% Validity: `(outcome.rng – {nil}) <= (\/ : allowed.rng)`

Paxos ensures that a round has a single value by having at most one proposer process per round, and making the proposer's identity part of the round number. So `N = [i, p]`, and proposer `p` proposes `(i, p)` for `n`, where `i` is an `I` that `p` has not used before, for instance, a local clock. The proposer keeps v_n in a volatile variable; rather than resuming an old round after a crash, it just starts a new one.

To understand why the algorithm works, it's useful to have the notion of a *stable* predicate on the state, a predicate that once true, remains true henceforth. Since the non-`nil` value of a round can't change, "v_n = v" is stable. Since a state value in an acceptor once set can't change, s_n^a = `v` and are stable; hence "round `n` is dead" and "round `n` decides" are stable as well. Note that s_n^a = `neutral` is not stable, since it can change to `v` or to `no`. Stable predicates are important, since they are the only kind of information that can usefully be passed from one process to another in a distributed system. If you get a message from another process telling you that `p` is true, and `p` is stable, then you know that `p` is true now (assuming that the other process is trustworthy).

We would like to have the idea of a *safe* value at round `n`: `v` is safe at `n` if any previous round that decided, decided on `n`. Unfortunately, this is not stable as it stands, since a previous round might decide later. In order to make it stable, we need to prevent this. Here is a more complex stable predicate that does the job:

`v` is safe at `n` = `(ALL n′ | n′ <= n ==> n′ is dead \/` $v_{n'}$ `= v)`

In other words, if you look back at rounds before `n`, skipping dead rounds, you see v_n. If all preceding rounds are dead, any v_n makes `n` safe. For this to be well-defined, we need an ordering on `N`'s, and for it to be useful we need a total ordering. We get this as the lexicographic ordering on the `N = [i, p]` pairs. This means that we must assume a total ordering on the proposers `P`.

With these preliminaries, we can give the abstraction function from `Paxos` to `LateConsensus`. For `allowed` it is just the union of the proposers' `allowed` sets. For `outcome` it is the value of a deciding round.

```
ABSTRACTION FUNCTION
    LateConsensus.allowed = \/ : allowed.rng
    LateConsensus.outcome = {n | Decides(n) | Value(n)}.choose

FUNC Decides(n) -> Bool = RET {a | s_n^a IS V}.majority
FUNC Value(n) -> (V + Null) = IF VAR a, v | s_n^a = v => RET v [*] RET nil FI
```

For this to be an abstraction function, we need an invariant:

(I1) Every deciding round has the same value.

It's easy to see that this follows from a stronger invariant: If round `n′` decides, then any later round's value is the same or `nil`.

(I2) `(ALL n′, n | n′ <= n /\ n′ deciding ==>` v_n `= nil \/` $v_n = v_{n'}$`)`

This in turn follows easily from something with a weaker antecedent:

(I3) `(ALL n′, n | n′ <= n /\ n′` is not dead `==>` v_n `= nil \/` $v_n = v_{n'}$`)`
= `(ALL n′, n |` v_n `= nil \/ (n′ <= n ==> n′` is dead `\/` $v_n = v_{n'}$`)`
= `(ALL n |` v_n `= nil \/ (ALL n′ | n′ <= n ==> n′` is dead `\/` $v_n = v_{n'}$`))`
= `(ALL n |` v_n `= nil \/` v_n is safe`)`

For validity, we also need to know that every round's value is allowed:

(I4) `(ALL n |` v_n `= nil \/` v_n `IN (\/ : allowed.rng))`

Initially all the v_n are `nil` so that (I3) and (I4) hold trivially. The Paxos algorithm maintains (I3) by choosing a safe value for a round. To accomplish this, the proposer chooses a new `n` and *queries* all the acceptors to learn their state in *all* rounds with numbers less than `n`. Before an acceptor responds, it *closes* a round earlier than `n` by setting any `neutral` state to `no`. Responses to this query from a majority of acceptors give the proposer enough information to find a safe value at round `n`, as follows:

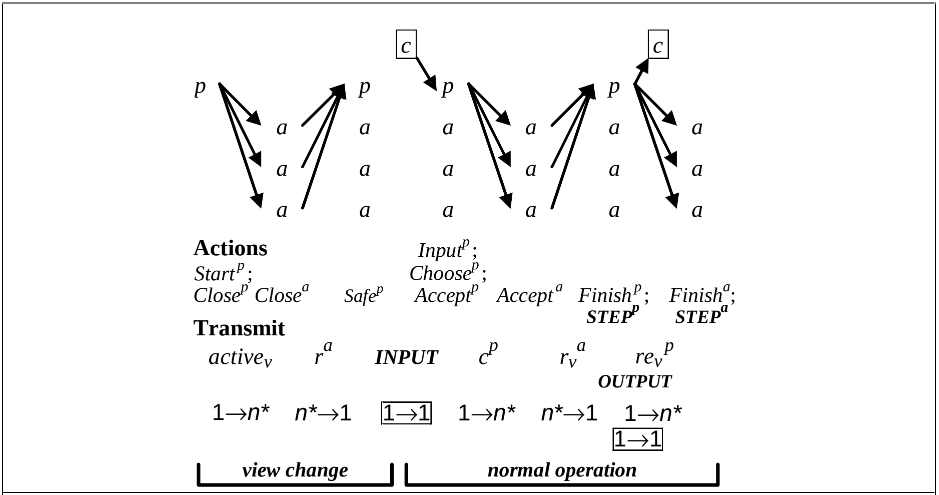
> It looks back from `n`, skipping over rounds with no `V` state, since these must be dead (remember that the reported state is a `V` or `no`). When it comes to a round `n′` with $s_{n'}^a$ = `v` for some acceptor `a`, it takes `v` as safe. Since $v_{n'}$ is safe by (I3), and all the rounds between `n′` and `n` are dead, `v` is also safe.
>
> If all previous rounds are dead, any allowed value is safe.

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

Another way of looking at this is that because a deciding round is not dead and can never become dead, it forms a barrier that the proposer can't get past in choosing a safe value for a later round. Thus the deciding round forces any later safe value to be the same. A later round can propose differently from an earlier one only if the earlier one is dead.

An example may help to clarify the idea. Assume the allowed set is `{x, y, w}`. Given the following two sets of states in rounds `1` through `3`, with three acceptors `a`, `b`, and `c`, the safe values for round `4` are as indicated. In the middle column the proposer can see a majority of acceptors in each round without seeing any `V`'s, because all the rounds are dead. In the right-hand column there is a majority for `w` in round `2`.

	State							
Value, acceptors	value	a	b	c	value	a	b	c
round 1	x	x	no	no	y	y	no	no
round 2	x	x	no	no	w	$\boxed{\texttt{w}}$	$\boxed{\texttt{no}}$	$\boxed{\texttt{w}}$
round 3	y	no	no	y	w	no	no	w
safe values for round 4	x, y, or w				w			



Message flow in Paxos

Proposer `p`	**Message**	**Acceptor** `a`
Choose a new n_p, `n`		

Query a majority of acceptors for their status	`query(n)` ⟶
	⟵ `report(a,` s^a`)`
		for all `n′ < n`,
		if $s_{n'}^a$ `= neutral`
		then $s_{n'}^a$ `:= no`

Find a safe `v` at `n`. If all `n′ < n` are dead, any `v` in $allowed_p$ is safe		
Command a majority of acceptors to accept `v`	`command(n, v)` ⟶ ⟵ `report(a,` s^a`)`	**if** s_n^a `= neutral` **then** s_n^a `:= v`
If a majority *accepts*, publish the outcome `v`	`outcome(v)` ⟶	

Note that only the latest `v` state from each acceptor is of interest, so only that state actually has to be transmitted.

Now in a second round trip the proposer *commands* everyone for round `n`. Each acceptor that is still neutral in round `n` (because it hasn't answered the query of a round later than `n`) *accepts* by changing its state to v_n in round `n`; in any case it reports its state to the proposer. If the proposer collects v_n reports from a majority of acceptors, then it knows that round `n` has succeeded, takes v_n 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.

When does a round succeed, that is, what action simulates the `Decide` action of the spec? It succeeds at the instant that some acceptor forms a majority by accepting its value, *even though* no acceptor or proposer knows at the time that this has happened. In fact, it's possible for the round to succeed without the proposer knowing this fact, if some acceptors fail after accepting but before getting their reports to the proposer, or if the proposer fails. In this case, some proposer will have to run another round, but it will have the same value as the invisibly deciding round.

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

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

> Every potential proposer that is up broadcasts its name.
>
> You become the proposer one round-trip time after doing a broadcast unless you have received the broadcast of a bigger name.

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 proposer to acceptors; in practice this is usually coded by individual messages to each acceptor. We describe continuous retransmission; in practice acceptors retransmit only in response to the proposer's retransmission.

A process acting as a proposer uses messages to communicate with the same process acting as an acceptor, so we describe the two roles of each process completely independently. In fact, the proposer need not be an acceptor at all.

Note the structure of the algorithm as a collection of independent atomic actions, each taking place at a single process. There are no non-atomic procedures except for the top-level scheduler, which simply chooses an enabled atomic action to run next. This structure makes it much easier to reason about the algorithm in the presence of concurrency and failures.

The next section gives the algorithm in detail. You can skip this, but be sure to read the following section on optimizations, which has important remarks about using Paxos in practice.

The Paxos algorithm: The details

We give a straightforward version of the algorithm in detail, and then describe encodings that reduce the information stored and transmitted to small fixed-size amounts. The first set of types and variables is copied from the earlier declarations.

```
MODULE Paxos[                                        % implements Consensus
    V,                                               % data Value to decide on
    P WITH {"<=": (P, P)->Bool SUCHTHAT IsTotal},    % Proposer; <= a total order
    A WITH {majority: SET A->Bool} SUCHTHAT IsMaj ]  % Acceptor

TYPE I          =  Int
     N          =  [i, p] WITH {"<=":=LEqN}          % round Number; <= total
     Z          =  (ENUM[no, neutral] + V)           % one acceptor's state in one round
     S          =  A -> N -> Z                       % Acceptors' states

VAR  outcome    :  A -> (V+Null) :={*->nil}          % the acceptors' state, in
     s          :  S             :={*->{*->neutral}} % two parts.
     allowed    :  P -> SET V    :={*->{}}           % the proposers' state
     % The rest of the proposers' state is the variables of ProposerActions(p).
     % All volatile except for n, which we make stable for simplicity.

TYPE K          =  ENUM[query,command,outcome,report] % Kind of message
     M          =  [k,                               % Message; kind of message,
                    n,                               % about round n
                    x: (Null + S + V) ]              % acceptor state or outcome.
                                                     % S defined for just one a
     Phase      =  ENUM[idle, querying, commanding]  % of a proposer process

CONST n0        := N{i:=0, p:={p | true}.min}        % smallest N

% Actions for handling messages

APROC Send(k, n, x: ANY) = << UnreliableCh.Put({M{k, n, x}}) >>
APROC Receive(k) -> M = << VAR m | m := UnreliableCh.Get(); m.k = k => RET m >>

% External actions. Can happen at any proposer or acceptor.

APROC Allow(v)       = << VAR p | allowed(p) := allowed(p) \/ {v} >>
APROC Outcome() -> V = << VAR a | RET outcome(a) >>

THREAD ProposerActions(p) =
  VAR                                                % proposer state (volatile except n)
    n           := N{i := 1, p := p},                % last round started
    phase       := idle,                             % proposer's phase
    pS          := S{},                              % Proposer info about acceptor State
    v: (V+Null) := nil                               % used iff phase = commanding
    |

DO <<  % Pick an enabled action and do it.
    % Crash. We don't care about the values of pS or v.
    phase := idle; allowed := {}; pS := {}; v := nil

[]  % New round. Hope n becomes largest N in use so this round succeeds.
    phase = idle => VAR i | i > n.i => n.i := i; pS := {}; phase := querying

[]  % Send query message.
    phase = querying => Send(query, n, nil)

[]  % Receive report message for the current round.
    << phase = querying => VAR m := Receive(report) |
          m.n = n => pS + := m.x >>
```

```
[]  % Start command. Note that Dead can't be true unless pS has a majority.
    << phase = querying =>
      IF  VAR n1 |    (ALL n' | n1 < n' /\ n' < n ==> Dead(pS, n'))
                   /\ Val(pS, n1) # nil => v := Val(pS, n1)
      [*] (ALL n'| n' < n ==> Dead(pS, n')) => VAR v' :IN allowed(p) | v := v'
      FI;
      pS := S{}; phase := commanding >>

[]  % Send command  message.
    phase = commanding => Send(command, n, v)

[]  % Receive report message for the current round with non-neutral state.
    << phase = commanding => VAR m := Receive(report), s1 := m.x, a |
          m.n = n /\ s1!a /\ $s1_n^a$ # neutral => $pS_n^a$:= $s1_n^a$ >>

[]  % Send outcome message if a majority has V state.
    Decides(pS, n) => Send(outcome, n, v)

[]  % This round is dead if a majority has no state. Try another round.
    Dead(pS, n) => phase := idle

>> OD

THREAD AcceptorActions(a) = % State is in $s^a$ and $outcome^a$, which are stable.

DO <<  % Pick an enabled action and do it.

    % Receive query message, change neutral state to no for all before m.n, and
    % send report. Note that this action is repeated each time a query message arrives.
    VAR m := Receive(query) |
          DO VAR n | n < m.n /\ $s_n^a$ = neutral => $s_n^a$:= no OD;
          Send(report, m.n, s.restrict({a}))

[] % Receive command message, change neutral state to $v_n$, and send state message.
   % Note that this action is repeated each time a command message arrives.
    VAR  m := Receive(command) |
          IF $s_{m.n}^a$ = neutral => $s_{m.n}^a$:= m.x [*] SKIP FI;
          Send(report, m.n, s.restrict({a}).restrict({n}))

[]  % Receive outcome message.
    VAR m := Receive(outcome) | $outcome^a$:= m.x

>> OD
```

=============Useful functions for the proposer choosing a value=============

```
FUNC LEqN(n1, n2) -> Bool =                               % lexicographic ordering
    RET n1.i < n2.i \/ (n1.i = n2.i /\ n1.a <= n2.a)

FUNC Dead(s', n)       -> Bool = RET {a | s'!a /\ s'(a)(n) = no}.majority
FUNC Decides(s', n) -> Bool = RET {a | s'!a /\ s'(a)(n) IS V}.majority

FUNC Val(s', n) -> (V+Null) = IF VAR a, v | s'(a)(n) = v => RET v [*] RET nil FI
% The value of round n according to s': if anyone has v then v else nil.
```

==================Useful functions for the invariants==================

% We write x_1 for `ProposerActions(p).x` to make the formulas more readable.

```
FUNC IsTotal(le: (P, P) -> Bool) -> Bool =              % Is le a total order?
    RET ( ALL p1, p2, p3 |    (le(p1, p2) \/ le(p2, p1))
                          /\ (le(p1, p2) /\ le(p2, p3) ==> le(p1, p3)) )

IsMaj(m: SET A->Bool) -> Bool =                         % any two must intersect
    RET (ALL aa1: SET A, aa2: SET A | (m(aa1) /\ m(aa2) ==> aa1 /\ aa2 ≠ {})

FUNC Allowed() -> SET V = RET \/ : allowed.rng

FUNC Decides(n) -> Bool = RET Decides(s, n)
FUNC Value(n) -> (V+Null)  = RET Val(s, n)              % The value of round n

FUNC ValAnywhere(n) -> SET V =
% Any trace of the value of round n, in messages, in s, or in a proposer.
     RET    {m, a, s1 | m IN UnreliableCh.q /\ m.x = s1 /\ s1!a /\ s1(a)!n
                  | $s1_n^a$ }
         \/ {Value(n)}
         \/ ($phase_{n.1}$ = commanding => {$v_{n.1}$} [*] {})

FUNC SafeVals(n) -> SET V =
% The safe v's at n.
     RET {v | ( ALL n' | n' < n ==> Dead(s, n') \/ v = Val(s, n') )}

FUNC GoodVals(n) -> SET V =
% The good values for round n: if there's no v yet, the ones that satisfy
% (I3) and validity. If there is a v, then v.
     RET ( Value(n) = nil => SafeVals(n) /\ Allowed() [*] {Value(n)} )
```

============================ Invariants ============================

```
% Proofs are straightforward except as noted. Observe that s changes only when an acceptor receives query
% (when it may add no state) or when  an acceptor receives command (when it may add V state).

% (1) A proposer's n.p is always the proposer itself.
( ALL p | $n_p$.p = p )                                % once p has taken an action
% (2) Ensure that there's no value yet for any round that a proposer might start.
( ALL p, n |    n.p = p /\ (n > $n_p$ \/ (n = $n_p$ /\ $phase_p$ = querying))
             ==> Value(n) = nil )

% (3)  s always has a most one value per round, because a only changes $s^a$ (when a receives query or command)
% from neutral, and either to no (query) or to agree with the proposer (command).
(ALL n | {a | s!a /\ $s_n^a$ IS V | $s_n^a$}.size <= 1)

% (4)  All the S's in the channel or in any pS agree with s.
( ALL s1 :IN ({m | m IN UnreliableCh.q /\ m.x IS S | m.x} \/ {p | $pS_p$}) |
    (ALL a, n | s1!a /\ s1(a)!n /\ $s1_n^a$ # neutral ==> $s1_n^a$ = $s_n^a$))

% (5) Every round value is allowed
( ALL n | Value(n) IN (Allowed()\/ {nil}) )

% (6)  If anyone thinks v is the value of a round, it is a good round value .
( ALL n | ValAnywhere(n) <= v IN GoodVals(n)) )
% (7) A round has only one non-nil value.
( ALL n | ValAnywhere(n).size <= 1 )
```

```
% Major invariant (I3).
( ALL n | Value(n) IN ({nil} \/ SafeVals(n)) )
% Easy consequence (I2)
( ALL n1, n2 | n1 < n2 /\ Decides(n1) ==> Value(n2) IN {nil, Value(n1)} )
% Further easy consequence (I1)
( ALL a | outcome(a) # nil ==> (EXISTS n | Decides(n) /\ outcome(a) = Value(n))

END Paxos
```

Optimizations

It's possible to reduce the size of the proposer and acceptor state and the messages transmitted to a few bytes, and in many cases to reduce the latency and the number of messages sent by combining rounds of the algorithm.

Reducing state and message sizes

It's not necessary to store or transmit the complete acceptor state s^a. Instead, everything can be encoded in a small fixed number of `N`'s, `A`'s, and `V`'s, as follows.

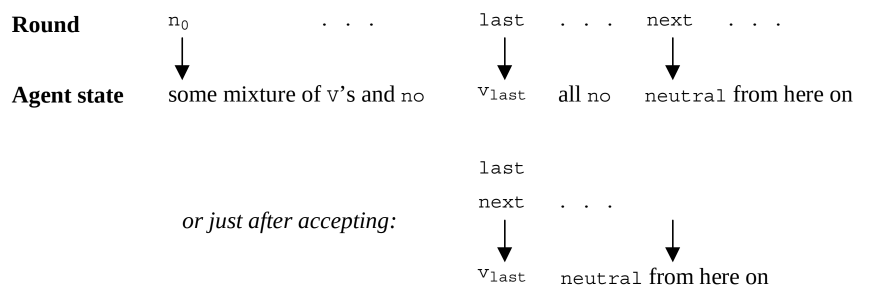
The relevant part of `S` in an acceptor or in a `report` message consists of v_{last} in some round `last`, plus `no` in all rounds strictly between `last` and some later round `next`, and `neutral` in any round after `last` and at or after `next`. Hence `S` can be encoded simply as (v_{last}, `last`, `next`). The part of `S` before `last` is just history and is not needed for the algorithm to run, because we know that v_{last} is safe. Making this precise, we have

```
     VAR y: [z, last: N, next: N] := {no, n0, n0}
```

and s^a is just

```
     (\ n | (n <= y.last => y.z [*] n < y.next => no [*] neutral))
```

Note that this encoding loses the details of the state for rounds before `last`, but the algorithm doesn't need this information Here is a picture of this coding scheme.



In a proposer, there are two cases for `pS`.

- If `phase = querying`, `pS` consists of the s^a's, for rounds less than `n`, transmitted by a set of acceptors `a`. Hence it can be encoded as a set of 'last state' tuples (`a`, $last_a$, `v`). From this we care only about the number of `a`'s (assuming that `A.majority` just counts acceptors), the biggest `last`, and its corresponding `v`. So the whole thing can be coded as `(`count of `A`'s, $\texttt{last}_{\texttt{max}}$, `v)`.

- If `phase = commanding`, `pS` consists of a set of $\texttt{v}_\texttt{n}$ or `no` in round `n`, so it can be encoded as the set of acceptors responding. We only care about a majority, so we only need to count the number of acceptors responding.

The proposers need not be the same processes as the acceptors. A proposer doesn't really need any stable state, though in the algorithm as given it has `n`. Instead, it can poll for the `next`'s from a majority after a failure and choose an `n` with a bigger `n.i`. This will yield an `n` that's larger than any `n` from this proposer that has appeared in a `command` message so far (because `n` can't get into a `command` message without having once been the value of `next` in a majority of acceptors), and this is all we need to keep `s` good. In fact, if a proposer ever sees a report with a `next` bigger than its own `n`, it should either stop being a proposer or immediately start another round with a new, larger `n`, because the current round is unlikely to succeed.

Combining rounds

In the most important applications of Paxos, we can combine the first round trip (`query/report`) with something else. For a commit algorithm, we can combine the first round-trip with the prepare message and its response; see handout 27 on distributed transactions.

The most important application is a state machine that needs to decide on a sequence of actions. We can number the actions $\texttt{a}_0$, $\texttt{a}_1$, …, $\texttt{a}_\texttt{k}$, run a separate instance of the algorithm for each action, and combine the `query/report` messages for all the actions. Note that these action numbers, which we are calling `K`'s, are *not* the same as the Paxos round numbers, the `N`'s; each action has its own instance of Paxos and therefore its own set of round numbers. In this application, the proposer is usually called the *primary*. The following trick allows us to do the first round trip only once after a new primary starts up: interpret the `report` messages for action `k` as applying not just to consensus on $\texttt{a}_\texttt{k}$, but to consensus on $\texttt{a}_\texttt{j}$ for *all* `j >= k`.

As long as the same process continues to be proposer, it can keep track of the current `K` in its private state. A new proposer needs to learn the first unused `K`. If it tries to get consensus using a number that's too small, it will discover that there's already an outcome for that action. If it uses a number `k` that's too big, however, it can get consensus. This is tricky, since it leads to a gap in the action numbers. Hence you can't apply the decided action `k`, since you don't know the state at `k` because you don't know all of the preceding actions that are applied to make that state. So a new proposer must find the first unused `K` for its first action `a`. A clumsy way to do this is to start at `k = 0` and try to get consensus on successive $\texttt{a}_\texttt{k}$'s until you get consensus on `a`.

You might think that this is silly. Why not just poll the acceptors for the largest `K` for which one of them knows the outcome? This is a good start, but it's possible that consensus was reached on the last $\texttt{a}_\texttt{k}$ (that is, there's a majority for it among the acceptors) but the outcome was not published before the proposer crashed. Or it was not published to a majority of the acceptors, and all the ones that saw it have also failed. So after finding that `k` is the largest `K` with an outcome, the proposer may still discover that the consensus on $\texttt{a}_{\texttt{k+1}}$ is not for its action `a`, but for some earlier action whose deciding outcome was never broadcast. If proposers follow the rule of not starting consensus on `k+1` until a majority knows the outcome for `k`, then this can happen at most

once. It may be convenient for a new proposer to start by getting consensus on a `SKIP` action in order to get this complication out of the way before trying to do real work.

Further optimizations are possible in distributing the actions already decided, and handout 28 on primary copy replication describes some of them.

Note that since a state machine is completely general, one of the things it can do is to change the set of acceptors. So acceptors can be added or dropped by getting consensus among the existing acceptors. This means that no special algorithm is needed to change the set of acceptors. Of course this must be done with care, since once you have gotten consensus on a new set of acceptors you have to get a majority of *them* in order to make any further changes.

Leases

In a synchronous system, if you want to avoid running a full-blown consensus algorithm for every action that reads the state, you can instead run consensus to issue a *lease* on some component of the state. The lease guarantees that the state component won't change (unless you have an exclusive lease and change it yourself) until some expiration time, a point in real time. Thus a lease is just like a lock, except that it times out. Provided you have a clock that has a known maximum difference from real time, you can be confident that the value of a leased state component hasn't changed (that is, that you still hold the lock). To keep control of the state component (that is, to keep holding the lock), you can renew the lease before it expires. If you can't talk to all the processes that have the lease, you have to wait for it to expire before changing the leased state component. So there is a tradeoff between the cost of renewing a lease and the time you have to wait for it to expire after a (possible) failure.

There are several variations:

- If you issue the lease to some known set of processes, you can revoke it provided they all acknowledge the revocation.
- If you know the maximum transmission time for a message, you can get by with clocks that have known differences in running rate rather than known differences in absolute time.
- Since a lease is a kind of lock, it can have different lock modes, for instance, 'read' and 'write'.

The most common application is to give some set of processes the right to cache some part of the state, for instance the contents of a cache line or of a file, without having to worry about the possibility that it might change. If you don't have a lease, you have to do an expensive state machine action to get a current result; otherwise you might get an old result from some replica that isn't up to date. (Of course, you could settle for a result as of action `k`, rather than a current one. Then you would need neither a lease nor a state machine action, but the client has to interpret the `k` that it gets back along with the result it wanted. Usually this is too complicated for the client.)

If the only reason for running consensus is to issue a lease, you don't need stable state in the acceptors. If an acceptor fails, it can recover with empty state after waiting long enough that any previous lease has expired. This means that the consensus algorithm can't reliably tell you the owner of such a lease, but you don't care because it has expired anyway. Schemes for electing a proposer usually depend on this observation to avoid disk writes.

You might think that an exclusive lease allows the process that owns it to change the leased state as well, as with 'owner' access to a cache line or ownership of a multi-ported disk. This is not true in general, however, because the state is replicated, and at least a majority of the replicas have to be changed. There's no reliable way to do this without running consensus.

In spite of this, leases are not completely irrelevant to updates. With a lease you can use a simple read-write memory as an acceptor for consensus, rather than a fancier one that can do compare-and-swap, since the lease allows you do the necessary read-modify-write operation atomically under the lease's mutual exclusion. For this to work, you have to be able to bound the time that the write takes, so you can be sure that it completes before the lease expires. Actually, the requirement is weaker: a read must see the atomic effect of any write that is started earlier than the read. This ensures that if the write is started before the lease expires, a reader that starts after the lease expires will see the write.

This observation is of great practical importance, since it lets you run a replicated state machine where the acceptors are 'dual-ported' disk drives that can be accessed from more than one machine. One of the machines becomes the master by taking out a lease, and it can then write state changes to the disks.

Compare-and-swap acceptors

An alternative to using simple memory and leases is to use memory that implements a compare-and-swap or conditional store operation. The spec for compare-and-swap is

```
APROC CAS(a, old: V, new: V) -> V =
    << IF m(a) = old => m(a) := new; RET old [*] RET m(a) FI >>
```

Many machines, including the IBM 370 and the DEC Alpha, have such an operation (for the Alpha, you have to program it with Load Locked and Store Conditional, but this is easy and cheap).

To use a CAS memory as an acceptor, we have to code the state so that we can do the query and command actions as single CAS operations. Recall that the coded state of an acceptor is y = (v_{last}, last, next), representing the value v_{last} for round last, no for all rounds strictly between last and next, and neutral for all rounds starting at next. A query for round n changes the state to (v_{last}, last, n) provided next <= n. A command for round n changes the state to (v_n, n, n) provided next = n. So we need a representation that allows us to atomically test the current value of next and change the state in one of these ways. This is possible if an N fits in a single memory cell that CAS can read and update atomically. We can store the rest of the triple in a data structure on the side that is indexed by next. If each possible proposer has its own piece of this data structure, they won't interfere with each other when they are updating it.

Since this is hard concurrency, the details of such a representation are tricky.

Complex updates

The actions of a state machine can be arbitrarily complex. For example, they can be complete transactions. In a replicated state machine the replicas must decide on the sequence of actions, and then each replica must do each action atomically. In handouts 19 and 20, on sequential and concurrent transactions, we will see how to make arbitrary actions atomic using redo recovery and locking. So in a general state machine each acceptor will write a redo log, in which each committed transaction corresponds to one action of the state machine. The acceptors must reach consensus on the complete sequence of actions that makes up the transaction. In practice, this means that each acceptor logs all the updates, and then they reach consensus on committing the transaction. When an acceptor recovers from a failure, it runs redo recovery in the usual way. Then it has to find out from other acceptors about any actions that they agreed on while it was down.

Of course, if several proposers are trying to run transactions at the same time, you have to make sure that the log entries don't get mixed up. Usually this is done by funneling everything through a single master called the primary; this master also acts as the proposer for consensus.

Another way of doing this is to use a single master with passive acceptors that just implement simple memory; usually these are disk drives that record redundant copies of the log. The previous section on leases explains how to run Paxos with such passive acceptors. When a master fails, the new one has to sort out the consensus on the most recent transaction as part of recovery.

Batching

Another way to avoid paying for consensus each time the state machine does an action is to batch several actions, possibly from different clients, into one super-action. They you get consensus on the super-action, and each replica can apply the individual actions of the super-action. You still have to pay to send the information that describes all the actions to each replica, but all the per-message overhead, plus the cost of the acknowledgements, is paid only once.