**1. The Paxos algorithm**

The algorithm runs in a message-passing model with asynchrony and less than n/2 crash failures (but not Byzantine failures, at least in the original algorithm). As always, we want to get agreement, validity, and termination. The Paxos algorithm itself is mostly concerned with guaranteeing agreement and validity while allowing for the possibility of termination if there is a long enough interval in which no process restarts the protocol.

Processes are classified as *proposers*, *accepters*, and *learners* (a single process may have all three roles). The idea is that a proposer attempts to ratify a proposed decision value (from an arbitrary input set) by collecting acceptances from a majority of the accepters, and this ratification is observed by the learners. Agreement is enforced by guaranteeing that only one proposal can get the votes of a majority of accepters, and validity follows from only allowing input values to be proposed. The tricky part is ensuring that we don't get deadlock when there are more than two proposals or when some of the processes fail. The intuition behind how this works is that any proposer can effectively restart the protocol by issuing a new proposal (thus dealing with lockups), and there is a procedure to release accepters from their old votes if we can prove that the old votes were for a value that won't be getting a majority any time soon.

To organize this vote-release process, we attach a distinct proposal number to each proposal. The safety properties of the algorithm don't depend on anything but the proposal numbers being distinct, but since higher numbers override lower numbers, to make progress we'll need them to increase over time. The simplest way to do this in practice is to make the proposal number be a timestamp plus the proposer's id to break ties. We could also have the proposer poll the other processes for the most recent proposal number they've seen and add 1 to it.

The revoting mechanism now works like this: before taking a vote, a proposer tests the waters by sending a *prepare*(n) message to all accepters where n is the proposal number. An accepter responds to this with a promise never to accept any proposal with a number less than n (so that old proposals don't suddenly get ratified) together with the highest-numbered proposal that the accepter has accepted (so that the proposer can substitute this value for its own, in case the previous value was in fact ratified). If the proposer receives a response from a majority of the accepters, the proposer then does a second phase of voting where it sends an *accept*(n, v) to all accepters and wins if receives a majority of votes.

So for each proposal, the algorithm proceeds as follows:

1. The proposer sends a message prepare(n) to all accepters. (Sending to only a majority of the accepters is enough, assuming they will all respond.)
2. Each accepter compares n to the highest-numbered proposal for which it has responded to a prepare message. If n is greater, it responds with ack(n, v, nv) where v is the highest-numbered proposal it has accepted and nv is the number of that proposal (or ⊥, 0 if there is no such proposal). (An optimization at this point is to allow the accepter to send back nack(higher number) to let the proposer know that it's doomed and should back off and try again—this keeps a confused proposer who thinks it's the future from locking up the protocol until 2037.)
3. The proposer waits (possibly forever) to receive ack from a majority of accepters. If any ack contained a value, it sets v to the most recent (in proposal number ordering) value that it received. It then sends accept(n, v) to all accepters (or just a majority). You should think of *accept* as a command ("Accept!") rather than acquiescence ("I accept")—the accepters still need to choose whether to accept or not.
4. Upon receiving accept(n, v), an accepter accepts v unless it has already received prepare(n') for some n' > n. If a majority of acceptors accept the value of a given proposal, that value becomes the decision value of the protocol.

Note that acceptance is a purely local phenomenon; additional messages are needed to detect which if any proposals have been accepted by a majority of accepters. Typically this involves a fourth round, where accepters send accepted(n, v) to all learners (often just the original proposer).

There is no requirement that only a single proposal is sent out (indeed, if proposers can fail we will need to send out more to jump-start the protocol). The protocol guarantees agreement and validity no matter how many proposers there are and no matter how often they start.

**2. Informal analysis: how information flows between rounds**

Call a *round* the collection of all messages labeled with some particular proposal n. The structure of the algorithm simulates a sequential execution in which higher-numbered rounds follow lower-numbered ones, even though there is no guarantee that this is actually the case in a real execution.

When an acceptor sends ack(n, v, nv), it is telling the round n proposer the last value preceding round n that it accepted. The rule that an acceptor only acknowledges a proposal higher than any proposal it has previously acknowledged prevents it from sending information "back in time"—the round nv in an acknowledgment is always less than n. The rule that an acceptor doesn't accept any proposal earlier than a round it has acknowledged means that the value v in an ack(n, v, nv) message never goes out of date—there is no possibility that an acceptor might retroactively accept some later value in round n' with nv < n' < n. So the ack message values tell a consistent story about the history of the protocol, even if the rounds execute out of order.

The second trick is to use the overlapping-majorities mechanism that makes ABD work (see [SharedMemoryVsMessagePassing](http://www.cs.yale.edu/homes/aspnes/pinewiki/SharedMemoryVsMessagePassing.html)). If the only way to decide on a value in round n is to get a majority of acceptors to accept it, and the only way to make progress in round n' is to get acknowledgments from a majority of acceptors, these two majorities overlap. So in particular the overlapping process reports the round n proposal value to the proposer in round n', and we can show by induction on n' that this round n proposal value becomes the proposal value in all subsequent rounds that proceed past the acknowledgment stage. So even though it may not be possible to detect that a decision has been reached in round n (say, because some of the acceptors in the accepting majority die without telling anybody what they did), no later round will be able to choose a different value. This ultimately guarantees agreement.

**3. Safety properties**

We now present a more formal analysis of the Paxos protocol. We consider only the safety properties of the protocol, corresponding to validity and agreement; without additional assumptions, Paxos does *not* guarantee termination.

Call a value *chosen* if it is accepted by a majority of accepters. The safety properties of Paxos are:

* No value is chosen unless it is first proposed. (This gives validity.)
* No two distinct values are both chosen. (This gives agreement.)

The first property is immediate from examination of the algorithm.

For the second property, we need some invariants. The intuition is that if some value is chosen, then a majority of accepters have accepted it for some proposal number n. Any proposal sent in an accept message with a higher number n' must be sent by a proposer that has seen an overlapping majority respond to its prepare(n') message. If we consider the process that overlaps, this process must have accepted v before it received prepare(n'), since it can't accept afterwards, and unless it has accepted some other proposal since, it responds with ack(n', v, n). If these are the only values that the proposer receives with number n or greater, it chooses v as its new value.

Worrying about what happens in rounds between n and n' is messy, so we'll use two formal invariants (taken more or less directly from Lamport's paper):

**Invariant 1**

An accepter accepts a proposal numbered n if and only if it has not responded to a prepare message with a number n' > n.

**Invariant 2**

For any v and n, if a proposal with value v and number n has been issued (by sending *accept* messages), then there is a majority of accepters S such that either (a) no accepter in S has accepted any proposal numbered less than n, or (b) v is the value of the highest-numbered proposal among all proposals numbered less than n accepted by at least one accepter in S.

The proof of the first invariant is immediate from the rule for issuing acks.

The proof of the second invariant follows from the first invariant and the proposer's rule for issuing proposals: it can only do so after receiving ack from a majority of accepters—call this set S—and the value it issues is either the proposal's initial value if all responses are ack(n, ⊥, 0), or the maximum value sent in by accepters in S if some responses are ack(n, v, nv). In the first case we have case (a) of the invariant: nobody accepted any proposals numbered less than n before responding, and they can't afterwards. In the second case we have case (b): the maximum response value is the maximum-numbered accepted value within S at the time of each response, and again no new values numbered < n will be accepted afterwards. Amazingly, none of this depends on the temporal ordering of different proposals or messages: the accepters enforce that their acks are good for all time by refusing to change their mind about earlier rounds later.

So now we suppose that some value v is eventually accepted by a majority T with number n. Then we can show by induction on proposal number that all proposals issued with higher numbers have the same value (even if they were issued earlier). For any proposal accept(v', n') with n' > n, there is a majority S (which thus overlaps with T) for which either case (a) holds (a contradiction—once the overlapping accepter finally accepts, it violates the requirement that no proposal less than n' has been accepted) or case (b) holds (in which case by the induction hypothesis v' = the value of some earlier proposal numbered ≥n = v).

**4. Learning the results**

Somebody has to find out that a majority accepted a proposal in order to get a decision value out. The usual way to do this is to have a fourth round of messages where the accepters send chose(v, n) to some designated learner (usually just the original proposer), which can then notify everybody else if it doesn't fail first. If the designated learner does fail first, we can restart by issuing a new proposal (which will get replaced by the previous successful proposal because of the safety properties).

**5. Liveness properties**

We'd like the protocol to terminate eventually. Suppose there is a single proposer, and that it survives long enough to collect a majority of acks and to send out accepts to a majority of the accepters. If everybody else cooperates, we get termination in 3 message delays.

If there are multiple proposers, then they can step on each other. For example, it's enough to have two carefully-synchronized proposers alternate sending out prepare messages to prevent any accepter from every accepting (since an accepter promises not to accept accept(n, v) once it has responded to prepare(n+1)). The solution is to ensure that there is eventually some interval during which there is exactly one proposer who doesn't fail. One way to do this is to use exponential random backoff (as popularized by Ethernet): when a proposer decides it's not going to win a round (e.g. by receiving a nack or by waiting long enough to realize it won't be getting any more acks soon), it picks some increasingly large random delay before starting a new round; thus two or more will eventually start far enough apart in time that one will get done without interference.

A more abstract solution is to assume some sort of weak leader election mechanism, which tells each accepter who the "legitimate" proposer is at each time. The accepters then discard messages from illegitimate proposers, which prevents conflict at the cost of possibly preventing progress. Progress is however obtained if the mechanism eventually reaches a state where a majority of the accepters bow to the same non-faulty proposer long enough for the proposal to go through.

Such a weak leader election method is an example of a more general class of mechanisms known as [FailureDetectors](http://www.cs.yale.edu/homes/aspnes/pinewiki/FailureDetectors.html), in which each process gets hints about what other processes are faulty that eventually converge to reality. (The particular failure detector in this case is known as the Ω failure detector; there are other still weaker ones that we will talk about later that can also be used to solve consensus.)

**Source**: <http://www.cs.yale.edu/homes/aspnes/pinewiki/Paxos.html>