Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.826 — Principles of Computer Systems

Handout 26 April 28, 1997

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.

State machines

There is a much more general way to use consensus, as the mechanism for implementing a highly available state machine. The way to get availability is to have either perfect components or redundancy. The simplest form of redundancy is replication: have several copies of each component, and make 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] We have already seen one application of this replicated state machine idea, in the implementation of transactions; there the replication takes the form of redoing a sequence of actions which is remembered in a log.

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 what is done in 'primary copy' replication, since it's easy for one place (the primary) to assign consecutive numbers.

The literature contains many 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.[2]

The section on leases at the end of this handout explains practical methods for minimizing the number of times you need to use consensus in implementing a reliable state machine.

Specification for consensus

Here is the specification 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.

```
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 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.

If only one `Allow` action occurs, there's no need to choose a `v`, and the implementation'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 a slightly more complicated, but perhaps more intuitive, spec. It accumulates the allowed values and then chooses one of them in the internal action `Agree`.

[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.

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

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

APROC Allow(v) = << allowed := allowed + {v} >>
APROC Outcome() -> (V+Null) = << RET outcome [] RET nil >>
% Only outcome is visible

APROC Agree() = << 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 `Agree` action). We saw these specs in handout 8, where prophecy variables are explained.

In the implementations 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 implementations, 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 TermConsensus [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
```

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 a distributed implementation. 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.

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 an implementation of `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 implemented 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 this isn't very interesting, since there's no way 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.[3] 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.
- 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. 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, consensus is possible. If *f* faults are allowed, then:

  For processes with stopping faults, consensus requires f+1 processes and an f+1-connected[4] network (that is, at least one good process and a connected subnet of good

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

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 3f+1 processes, a 2f+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.

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.

- A fixed 'leader' 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; `TermConsensus.done` is true iff everyone has gotten the outcome (or failed permanently). Standard two-phase commit 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 `11`, `b` and `c` vote for `12`, and `b` fails. Now all you can see is one vote for `11` and one for `12`, so you can't tell that `12` had a majority.

The Paxos algorithm: The idea

In the rest of this handout, we describe Lamport's Paxos algorithm for implementing consensus; this algorithm was independently invented by Liskov and Oki as part of a replicated data storage system.[5] Its heart is the best known asynchronous algorithm, which is

run by a set of *leader* processes that guide a set of *agent* processes to achieve consensus,

correct no matter how many simultaneous leaders there are and no matter how often leader or agent processes fail and recover or how slow they are, and

guaranteed to terminate if there is a single leader for a long enough time during which each member of a majority of the agent processes is up for long enough, but

possibly non-terminating if there are always too many leaders (fortunate, since we know that guaranteed termination is impossible).

[5] L. Lamport, The part-time parliament, Technical report 49, Systems Research Center, Digital Equipment Corp, Palo Alto, Sep. 1989. 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.

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.

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.

The grand plan of the algorithm is to have a sequence of *rounds*, each with a single leader. 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. In each round the leader

queries the agents to learn their state for past rounds,

chooses a value `v` and *commands* the agents, trying to get a majority to accept `v`, and

if successful, 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.

The agents do not make any decisions; they do whatever a leader requests, unless they have already done something inconsistent with that. In fact, an agent can be implemented by a memory that has a compare-and-swap operation, as we shall see later. Of course, the leaders and agents can run on the same machine, and even in the sample process. This is usually the way it's implemented, but the algorithm with separate leaders and agents is easier to explain.

It takes a total of 2 1/2 round trips for a successful round. If there's only one leader that doesn't fial, Paxos reaches consensus in one round. If the leader fails repeatedly, or several leaders fight it out, it may take many rounds to reach consensus.

The rounds are numbered (not necessarily consecutively), 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 agent 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 *successful* if a majority has state v_n. If a round is successful, 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 agree on
         L WITH {"<=": (L, L)->Bool},                 % Leader; <= a total order
         A WITH {majority : SET A->Bool} ]            % Agent; majorities
                                                      % must intersect

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

VAR  agents  : Z              := {*->{*->neutral}     % agent working states
     outcome : A -> (V+Null) := {*->nil}             % agent outcome values
     allowed : L -> SET V    := {*->{}}              % leader states

% Agreement: ( EXISTS v | outcome.rng <= {v, nil} )

% Validity:  outcome.rng <= (+ : allowed.rng) + {nil}
```

Paxos ensures that a round has a single value by having at most one leader process per round, and making the leader's identity part of the round number. So N = [i, l], and leader l chooses (i, l) for n, where i is an I that l has not used before, for instance, a local clock. The leader 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 which 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 once set can't change, agents(a)(n) = v and agents(a)(n) = no are stable; hence "round n is dead" and "round n is successful" are stable as well.

Here is a more complex stable predicate:

round n is anchored = (ALL n′ | n′ <= n ==> n′ is dead \/ $v_n = v_{n'}$)

In other words, if you look back at rounds before n, skipping dead rounds, you see the same value as n. If all preceding rounds are dead, round n is anchored no matter what its value is. 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, L] pairs. This means that we need a total ordering on the leaders L.

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

```
ABSTRACTION FUNCTION
    LateConsensus.allowed = + : allowed.rng
    LateConsensus.outcome = {n | Successful(n) | Value(n)}.choose

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

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

(I1) Every successful round has the same value.

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

(I2) (ALL n′, n | n′<=n /\ n′ is successful ==> 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 | v_n = nil \/ (ALL n′ | n′ <= n ==> n′ is dead \/ $v_n = v_{n'}$)
== (ALL n | v_n = nil \/ n is anchored)

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) holds trivially. The Paxos algorithm maintains (I3) by choosing the v value of a round so that the round is anchored. To accomplish this, the leader chooses a new n and *queries* all the agents to learn their state in *all* rounds with numbers less than n. Before an agent responds, it changes any neutral state for a round earlier than n to no. Responses to this query from a majority of agents give the leader enough information to make round n anchored, 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 $v_{n'}$ in some agent in the majority, it chooses $v_{n'}$ as v_n. Since n′ is anchored by (I3), and all the rounds between n′ and n are dead, n is also anchored.
>
> If all previous rounds are dead, the leader chooses any allowed value for v_n. In this case n is certainly anchored.

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

Another way of looking at this is that because a successful round is not dead and can never become dead, it forms a barrier that the leader can't get past in choosing a value that will make a later round anchored. Thus the successful round forces the leader of the later round to choose the same value. A later round can choose 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, z}. Given the following two sets of states in rounds 1 through 3, with three agents a, b, and c, the leader's possible choices for round 4 are as indicated. In the right-hand column there is a majority for z in round 2.

	State							
Value, agents	value	a	b	c	value	a	b	c
round 1	x	x	no	no	y	y	no	no
round 2	x	x	no	no	z	z	no	z
round 3	y	no	no	y	z	no	no	z
leader's choices for round 4	x, y, or z				z			

Note that only the latest V state is of interest, so only that state actually has to be transmitted.

Now in a second round trip the leader *commands* everyone for round n. Each agent 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 leader. If the leader collects v_n reports from a majority of agents, 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.

Leader l	**Message**	**Agent** a
Choose a new n_l		
Query a majority of agents for their status	query(n_l) → ← report(a, s_a)	for all n′ < n_l, **if** $s_{a,n'}$ = neutral **then** $s_{a,n'}$:= no
Choose v_l to keep n_l anchored. If all n′ < n_l are dead, choose any v in $allowed_l$		
Command a majority of agents to accept v_l	command(n_l, v_l) → ← report(a, s_a)	**if** s_{a,n_l} = neutral **then** s_{a,n_l} := v_l
If a majority accepts, publish the outcome v_l	outcome(v_l) →	

When does a round succeed, that is, what action simulates the Agree action of the spec? It succeeds 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. In this case, some leader will have to run another round, but it will have the same value as the invisibly successful round.

When does Paxos 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. 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 leaders 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 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.

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. We describe continuous retransmission; in practice agents retransmit only in response to the leader's retransmission.

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

The next section gives the algorithm in detail. You can skip this, but be sure to read the 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 agree on
    L WITH {"<=": (L, L)->Bool SUCHTHAT IsTotal},     % Leader; <= a total order
    A WITH {majority: SET A->Bool} SUCHTHAT IsMaj ]   % Agent

TYPE I          =  Int
     N          =  [i, l] WITH {"<=":=LEqN}           % round Number; <= total
     S          =  (ENUM[no, neutral] + V)            % State
     Z          =  A -> N -> S                        % Agents' states

VAR  outcome    :  A -> (V+Null) :={*->nil}           % the agents' state, in
     agents     :  Z             :={*->{*->neutral}}  % two parts.
     allowed    :  L -> SET V    :={*->{}}            % the leaders' state
     % The rest of the leaders' state is the variables of LeaderActions(l).
     % 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 + Z + V) ]              % agent state or outcome.
                                                      % Z defined for just one a
     Phase      =  ENUM[idle, querying, commanding]   % of a leader process

VAR  n0         := N{i:=0, l:={l | true}.min}         % constant: 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 leader or agent.
APROC Allow(v)       = << VAR l | allowed(l) := allowed(l) + {v} >>
APROC Outcome() -> V = << VAR a | RET outcome(a) >>

THREAD LeaderActions(l) =
  VAR                                                     % leader state
                                                          % (volatile except n)
    n           := N{i := 1, l := l},                     % last round started
    phase       := idle,                                  % leader's phase
    reports     := Z{},                                   % info about agents so far
    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 reports or v.
    phase := idle; allowed := {}; reports := {}; 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; reports := {}; 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 => reports := reports + m.x >>

[] % Start command. Note that Dead can't be true unless reports has a majority.
    << phase = querying =>
      IF  VAR n1 |   (ALL n' | n1 < n' /\ n' < n ==> Dead(reports, n'))
                  /\ Val(reports, n1) # nil => v := Val(reports, n1)
      [*] VAR v' :IN allowed(l) | v := v'
      FI;
      reports := Z{}; 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), z := m.x, a |
        m.n = n /\ z!a /\ z(a)(n) # neutral => reports(a)(n) := z(a)(n) >>

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

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

>> OD

THREAD AgentActions(a) = % State is in agents(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 /\ agents(a)(n) = neutral => agents(a)(n) := no OD;
    Send(report, m.n, agents.restrict({a}))
```

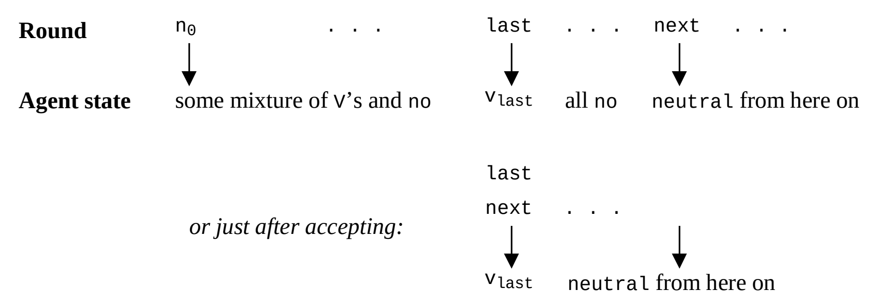
```
[] % Receive command message, change neutral state to v_n, and send state message.
   % Note that this action is repeated each time a command message arrives.
VAR m := Receive(command) |
        IF agents(a)(m.n) = neutral => agents(a)(m.n) := m.x [*] SKIP FI;
        Send(report, m.n, agents.restrict({a}).restrict({n}))


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

>> OD

=============Useful functions for the leader 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(z, n)       -> Bool = RET {a | z!a /\ z(a)(n) = no}.majority
FUNC Successful(z, n) -> Bool = RET {a | z!a /\ z(a)(n) IS V}.majority

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

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

% We write x_l for LeaderActions(l).x to make the formulas more readable.

FUNC IsTotal(le: (L, L) -> Bool) -> Bool =           % Is le a total order?
    RET ( ALL l1, l2, l3 |   (le(l1, l2) \/ le(l2, l1))
                          /\ (le(l1, l2) /\ le(l2, l3) ==> le(l1, l3)) )

IsMaj(m: SET A->Bool) -> Bool =                      % any two must intersect
    RET (ALL s1: SET A, s2: SET A | (m(s1) /\ m(s2) ==> s1 * s2 ≠ {})

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

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

FUNC ValAnywhere(n) -> SET V =
% Any trace of the value of round n, in messages, in agents, or in a leader.
    RET   {m, a, z | m IN UnreliableCh.q /\ m.x = z /\ z!a /\ z(a)!n
                   | z(a)(n)}
        + {Value(n)}
        + (phase_n.l = commanding => {v_n.l} [*] {})

FUNC AnchoredVals(n) -> SET V =
% The v's that would make n an anchored round.
    RET {v | ( ALL n' | n' < n ==> Dead(agents, n') \/ v = Val(agents, 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 => AnchoredVals(n) * Allowed() [*] {Value(n)} )

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

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

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

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

% (4) All the Z's in the channel or in any reports agree with agents.
( ALL z :IN {m | m IN UnreliableCh.q /\ m.x IS Z | m.x} + {l | reports_l} |
    (ALL a, n | z!a /\ z(a)!n /\ z(a)(n) # neutral ==> z(a)(n) = agents(a)(n)))

% (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} + AnchoredVals(n)) )
% Easy consequence (I2)
( ALL n1, n2 | n1 < n2 /\ Successful(n1) ==> Value(n2) IN {nil, Value(n1)} )
% Further easy consequence (I1)
( ALL a | outcome(a) # nil ==> (EXISTS n | Successful(n) /\ outcome(a) = Value(n))

END Paxos
```

Optimizations

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

- The relevant part of `Z` in an agent 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 `Z` can be encoded simply as (v_{last}, `last`, `next`). The part of `Z` before `last` is just history and is not needed for the algorithm to run, because we know that round `last` is anchored. Making this precise, we have
  `VAR y: [s, last: N, next: N] := {no, n0, n0}`
  and `z(a)` is just
  `(\ n | (n <= y.last => y.s [*] 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 leader, there are two cases for `reports`.

  If `phase = querying`, `reports` consists of the `Z`'s, for rounds less than `n`, transmitted by a set of agents `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 agents), the biggest `last`, and its corresponding `v`. So the whole thing can be coded as (count of `A`'s, $last_{max}$, `v`).

If phase = commanding, reports consists of a set of v_n or no in round n, so it can be encoded as the set of agents responding. We only care about a majority, so we only need to count the number of agents responding.

The leaders need not be the same processes as the agents. A leader 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 leader 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 agents), and this is all we need to keep agents good. In fact, if a leader ever sees a report with a next bigger than its own n, it should either stop being a leader or immediately start another round with a new, larger n, because the current round is unlikely to succeed.

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.

The most important application is a state machine that needs to agree on a sequence of actions. We can number the actions a_0, a_1, …, a_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 leader 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 a_k, but to consensus on a_j for *all* j >= k.

As long as the same process continues to be leader, it can keep track of the current K in its private state. A new leader 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 that's too big, however, it can get consensus. This is bad, since it leads to a gap in the action numbers that might later be filled by some other leader, with disastrous consequences. We want to maintain the invariant that if outcome k is decided, then either k=0 or outcome k-1 is decided as well. So a new leader 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 a_k's until you get consensus on a.

You might think that this is silly. Why not just poll the agents 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 a_k (that is, there's a majority for it among the agents) but the outcome was not published before the leader crashed. Or it was not published to a majority of the agents, and all the ones that saw it have also failed. So after finding that k is the largest K with an outcome, the leader may still discover that the consensus on a_{k+1} is not for its action a, but for some earlier action whose successful outcome was never broadcast. If leaders follow the rule of not starting consensus on k+1 until the outcome for k is known to a majority, then this can happen at most once. It may be convenient for a new leader 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 agreed, 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 agents. So agents can be added or dropped by getting consensus among the existing agents. Of course this must be done with care, since once you have gotten consensus on a new set of agents 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 until some expiration time, a point in real time. 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. To keep control of the state component, 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 the lease is issued to some known set of processes, it can be revoked 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.

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 agents. If an agent 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 leader 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 agent for consensus, since it 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 agents 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 agents

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: D, new: D) -> D =
    << 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.

To use a `CAS` memory as an agent, 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 agent 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` and 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 leader 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 agree on the sequence of actions, and then each replica must do each action atomically. In handouts 18 and 19 we saw how to make arbitrary actions atomic using redo recovery and locking. So in a general state machine each agent will write a redo log, in which each committed transaction corresponds to one action of the state machine. The agents must reach consensus on the complete sequence of actions that makes up the transaction. In practice, this means that each agent logs all the updates, and then they reach consensus on committing the transaction. When an agent recovers from a failure, it runs redo recovery in the usual way. Then it has to find out from other agents about any actions that they agreed on while it was down.

Of course, if several leaders 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 leader for consensus.

Another way of doing this is to use a single master with passive agents 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 agents. When a master fails, the new one has to sort out the consensus on the most recent transaction as part of recovery.