proposer and the distinguished learner. The Paxos consensus algorithm is precisely the one described above, where requests and responses are sent as ordinary messages. (Response messages are tagged with the corresponding proposal number to prevent confusion.) Stable storage, preserved during failures, is used to maintain the information that the acceptor must remember. An acceptor records its intended response in stable storage before actually sending the response.

All that remains is to describe the mechanism for guaranteeing that no two proposals are ever issued with the same number. Different proposers choose their numbers from disjoint sets of numbers, so two different proposers never issue a proposal with the same number. Each proposer remembers (in stable storage) the highest-numbered proposal it has tried to issue, and begins phase 1 with a higher proposal number than any it has already used.

3 Implementing a State Machine

A simple way to implement a distributed system is as a collection of clients that issue commands to a central server. The server can be described as a deterministic state machine that performs client commands in some sequence. The state machine has a current state; it performs a step by taking as input a command and producing an output and a new state. For example, the clients of a distributed banking system might be tellers, and the state-machine state might consist of the account balances of all users. A withdrawal would be performed by executing a state machine command that decreases an account's balance if and only if the balance is greater than the amount withdrawn, producing as output the old and new balances.

An implementation that uses a single central server fails if that server fails. We therefore instead use a collection of servers, each one independently implementing the state machine. Because the state machine is deterministic, all the servers will produce the same sequences of states and outputs if they all execute the same sequence of commands. A client issuing a command can then use the output generated for it by any server.

To guarantee that all servers execute the same sequence of state machine commands, we implement a sequence of separate instances of the Paxos consensus algorithm, the value chosen by the i^{th} instance being the i^{th} state machine command in the sequence. Each server plays all the roles (proposer, acceptor, and learner) in each instance of the algorithm. For now, I assume that the set of servers is fixed, so all instances of the consensus algorithm

use the same sets of agents.

In normal operation, a single server is elected to be the leader, which acts as the distinguished proposer (the only one that tries to issue proposals) in all instances of the consensus algorithm. Clients send commands to the leader, who decides where in the sequence each command should appear. If the leader decides that a certain client command should be the 135^{th} command, it tries to have that command chosen as the value of the 135^{th} instance of the consensus algorithm. It will usually succeed. It might fail because of failures, or because another server also believes itself to be the leader and has a different idea of what the 135^{th} command should be. But the consensus algorithm ensures that at most one command can be chosen as the 135^{th} one.

Key to the efficiency of this approach is that, in the Paxos consensus algorithm, the value to be proposed is not chosen until phase 2. Recall that, after completing phase 1 of the proposer's algorithm, either the value to be proposed is determined or else the proposer is free to propose any value.

I will now describe how the Paxos state machine implementation works during normal operation. Later, I will discuss what can go wrong. I consider what happens when the previous leader has just failed and a new leader has been selected. (System startup is a special case in which no commands have yet been proposed.)

The new leader, being a learner in all instances of the consensus algorithm, should know most of the commands that have already been chosen. Suppose it knows commands 1–134, 138, and 139—that is, the values chosen in instances 1–134, 138, and 139 of the consensus algorithm. (We will see later how such a gap in the command sequence could arise.) It then executes phase 1 of instances 135–137 and of all instances greater than 139. (I describe below how this is done.) Suppose that the outcome of these executions determine the value to be proposed in instances 135 and 140, but leaves the proposed value unconstrained in all other instances. The leader then executes phase 2 for instances 135 and 140, thereby choosing commands 135 and 140.

The leader, as well as any other server that learns all the commands the leader knows, can now execute commands 1–135. However, it can't execute commands 138–140, which it also knows, because commands 136 and 137 have yet to be chosen. The leader could take the next two commands requested by clients to be commands 136 and 137. Instead, we let it fill the gap immediately by proposing, as commands 136 and 137, a special "no-op" command that leaves the state unchanged. (It does this by executing phase 2 of instances 136 and 137 of the consensus algorithm.) Once these

no-op commands have been chosen, commands 138–140 can be executed.

Commands 1–140 have now been chosen. The leader has also completed phase 1 for all instances greater than 140 of the consensus algorithm, and it is free to propose any value in phase 2 of those instances. It assigns command number 141 to the next command requested by a client, proposing it as the value in phase 2 of instance 141 of the consensus algorithm. It proposes the next client command it receives as command 142, and so on.

The leader can propose command 142 before it learns that its proposed command 141 has been chosen. It's possible for all the messages it sent in proposing command 141 to be lost, and for command 142 to be chosen before any other server has learned what the leader proposed as command 141. When the leader fails to receive the expected response to its phase 2 messages in instance 141, it will retransmit those messages. If all goes well, its proposed command will be chosen. However, it could fail first, leaving a gap in the sequence of chosen commands. In general, suppose a leader can get α commands ahead—that is, it can propose commands $i + 1$ through $i+\alpha$ after commands 1 through i are chosen. A gap of up to $\alpha-1$ commands could then arise.

A newly chosen leader executes phase 1 for infinitely many instances of the consensus algorithm—in the scenario above, for instances 135–137 and all instances greater than 139. Using the same proposal number for all instances, it can do this by sending a single reasonably short message to the other servers. In phase 1, an acceptor responds with more than a simple OK only if it has already received a phase 2 message from some proposer. (In the scenario, this was the case only for instances 135 and 140.) Thus, a server (acting as acceptor) can respond for all instances with a single reasonably short message. Executing these infinitely many instances of phase 1 therefore poses no problem.

Since failure of the leader and election of a new one should be rare events, the effective cost of executing a state machine command—that is, of achieving consensus on the command/value—is the cost of executing only phase 2 of the consensus algorithm. It can be shown that phase 2 of the Paxos consensus algorithm has the minimum possible cost of any algorithm for reaching agreement in the presence of faults [2]. Hence, the Paxos algorithm is essentially optimal.

This discussion of the normal operation of the system assumes that there is always a single leader, except for a brief period between the failure of the current leader and the election of a new one. In abnormal circumstances, the leader election might fail. If no server is acting as leader, then no new commands will be proposed. If multiple servers think they are leaders, then

they can all propose values in the same instance of the consensus algorithm, which could prevent any value from being chosen. However, safety is preserved—two different servers will never disagree on the value chosen as the i^{th} state machine command. Election of a single leader is needed only to ensure progress.

If the set of servers can change, then there must be some way of determining what servers implement what instances of the consensus algorithm. The easiest way to do this is through the state machine itself. The current set of servers can be made part of the state and can be changed with ordinary state-machine commands. We can allow a leader to get α commands ahead by letting the set of servers that execute instance $i + \alpha$ of the consensus algorithm be specified by the state after execution of the i^{th} state machine command. This permits a simple implementation of an arbitrarily sophisticated reconfiguration algorithm.

References

[1] Michael J. Fischer, Nancy Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[2] Idit Keidar and Sergio Rajsbaum. On the cost of fault-tolerant consensus when there are no faults—a tutorial. TechnicalReport MIT-LCS-TR-821, Laboratory for Computer Science, Massachusetts Institute Technology, Cambridge, MA, 02139, May 2001. also published in SIGACT News 32(2) (June 2001).

[3] Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2:95–114, 1978.

[4] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[5] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.