Cheap Paxos

Leslie Lamport and Mike Massa
Microsoft

Abstract

Asynchronous algorithms for implementing a fault-tolerant distributed system, which can make progress despite the failure of any F processors, require $2F + 1$ processors. Cheap Paxos, a variant of the Paxos algorithm, guarantees liveness under the additional assumption that the set of nonfaulty processors does not "jump around" too fast, but uses only $F + 1$ main processors that actually execute the system and F auxiliary processors that are used only to handle the failure of a main processor. The auxiliary processors take part in reconfiguring the system to remove the failed processor, after which they can remain idle until another main processor fails.

1 Introduction

The state-machine approach consists of describing a system as a state machine that takes as input a sequence of client commands and produces a sequence of states and outputs [4, 10]. The state machine is implemented by a collection of servers. It reduces the problem of implementing a distributed system to that of having the servers choose a sequence of commands. Making the system reliable requires that all processors agree on each command in the sequence, despite the failure of some components. For asynchronous systems, we require that consistency be maintained in the face of any number of non-malicious (non-Byzantine) failures, and that progress be ensured when enough processors are nonfaulty and can communicate with one another in a timely manner [2]. The "classic" Paxos algorithm is an efficient, practical algorithm for achieving this [1, 5, 7].

Consider the problem of implementing a distributed system that can make progress if all but one processor is working. Previous algorithms, such as classic Paxos, require three processors. Only two of those processors need maintain the system state; but a third processor must participate in choosing the sequence of commands. The following argument shows that this third processor is necessary. Suppose the system is implemented by only two processors, p and q, and suppose that q fails. The requirement that the system continue to make progress despite a single failed processor means that p must continue operating the system. Now suppose that p fails and then q is repaired. Since there is only one failed processor, q must be able to resume operating the system. But this is clearly impossible, since q does not know the current state of the system because it does not know what p did after q failed. Some third processor is needed—for example, a disk that can be accessed by both p and q.

Suppose we are willing to weaken the liveness requirement, so that if q fails and then p fails before q is repaired, then the system may halt until p is repaired. Two processors are still not enough if we require that consistency be maintained despite communication failure. With only two processors p and q, one processor cannot distinguish failure of the other processor from failure of the communication medium. Since consistency is lost if each processor continues operating the system by itself, the system cannot allow each processor to continue just because it thinks that the other processor has failed. A third processor is needed. However, that third processor does not have to participate in choosing the sequence of commands. It must take action only in case p or q fails, after which it does nothing while either p or q continues to operate the system by itself. The third processor can therefore be a small/slow/cheap one, or a processor primarily devoted to other tasks.

This argument suggests that there exists a method of implementing a one-fault tolerant system, satisfying the consistency property of classic Paxos and a weaker liveness property, using two main processors plus a third auxiliary processor. This paper describes Cheap Paxos, a generalization of such an algorithm that tolerates F faults with $F + 1$ main processors and F auxiliary processors. It maintains liveness under a sort of "amoeba" assumption [3], under which the subnetwork of working main processors does not move around too quickly. The assumption can be described as follows. A nonfaulty processor maintains a certain knowledge

Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)
0-7695-2052-9/04 $ 20.00 © 2004 IEEE

of the system's state. When a faulty processor is repaired, it can, in a finite length of time, re-acquire this knowledge from any other processor that possesses it. Liveness is maintained as long as there is at least one main processor with knowledge of the system state and $F+1$ processors (main or auxiliary) that are nonfaulty and can communicate with one another in a timely manner. Consistency is always maintained (assuming non-malicious failures).

There are two threads of previous work that superficially resemble Cheap Paxos. The first is the use of main processors that are replaced by spares if they fail [8]. Indeed, classic Paxos requires only $F+1$ working processors to operate a system that tolerates F faults; the remaining F processors can be used as spares. However, unlike the auxiliary processors of Cheap Paxos, spares must have the necessary computing power to replace a failed main processor. The second thread is the use of dynamic quorum algorithms for maintaining multiple copies of a database. These algorithms can employ "witness" processors that need not maintain the data [9]. However, unlike the auxiliary processors of Cheap Paxos, these witnesses participate in each operation.

Two moderately recent developments in computing may make Cheap Paxos useful. First, improvements to hardware and operating systems make computers less likely to crash. The weaker liveness guarantee of Cheap Paxos may therefore still provide sufficient reliability. Second, the widespread use of computers makes it more likely that an organization will have additional machines from which cycles can be "stolen" to implement the auxiliary processors.

One might think that the low cost of computers would make Cheap Paxos uninteresting. However, we have observed that people are no more willing to use extra hardware to make a system simpler and more reliable than they were 40 years ago, even though that hardware has become orders of magnitude cheaper.

The following section reviews Paxos, and Section 3 describes Cheap Paxos. The obligatory conclusion follows.

2 A Review of Paxos

The Paxos algorithm for implementing a distributed state machine was introduced in [5]. We consider two versions of Paxos. In the basic version, to which we give the name Static Paxos, the set of servers is fixed. A variation that we call Dynamic Paxos, mentioned briefly in [5], uses state machine commands to change the set of servers. We begin by considering Static Paxos; Dynamic Paxos is explained in Section 2.3.

2.1 The Paxos Consensus Algorithm

To implement a distributed system as a state machine, the processors of the system must choose a sequence of commands. This is done by executing a sequence of instances of a consensus algorithm, the i^{th} instance choosing the i^{th} command in the sequence. We now review the Paxos consensus algorithm.

The goal of a consensus algorithm is for a collection of processes to agree upon a value. It is most convenient to phrase the consensus problem in terms of three classes of agents: *proposers* that propose values, *acceptors* that cooperate to choose a single proposed value, and *learners* that must learn what value has been chosen [6]. A single processor can act as more than one kind of agent. The safety properties that a consensus algorithm must satisfy are:

Nontriviality Only a value that has been proposed may be chosen,

Consistency Only a single value may be chosen.

Conservatism Only a chosen value may be learned.

There is also a liveness requirement that we do not try to state precisely; it is discussed informally below.

The Paxos consensus algorithm has been discussed elsewhere [1, 5, 6, 7], so we do not explain here exactly how it works. Instead, we just describe its actions.

Paxos assumes an underlying procedure for selecting a leader. Safety is guaranteed even if no leader or multiple leaders are selected, but a unique leader is required to ensure progress. Proposers send their proposals to the leader.

The consensus algorithm assumes predefined sets of acceptors called *quorums*. The only requirement on the quorums is that any two quorums have at least one acceptor in common. Paxos also assumes a set of *ballot numbers*, which for simplicity we take to be the natural numbers. The ballot numbers are partitioned among potential leaders, each possible leader having its own disjoint set of ballot numbers.

The consensus algorithm has two phases, each with two subphases. The algorithm's actions are described below. The algorithm sends messages between learners and acceptors, and from acceptors to learners. Since the same processor may be playing multiple roles, it can send messages to itself.

*Phase*1$a(l, b)$ Leader l chooses a number b from among its ballot numbers and sends $\langle$ "1a", $b\rangle$ messages to the acceptors.

*Phase*1$b(a, b)$ When acceptor a receives a $\langle$ "1a", $b\rangle$ message from a leader l, if it has not received any message with a ballot number greater than b, then

it replies to l with a $\langle$ "1b", $b, \ldots\rangle$ message, where the precise contents of the message do not concern us. If a has received a message with ballot number greater than b, it sends a reply to l indicating that it is ignoring the $\langle$ "1a", $b\rangle$ message. (Upon receiving that message, l will perform a $Phase1a(l, b')$ action for $b' > b$, if it still believes itself to be the leader.)

*Phase*2$a(l, b)$ If leader l has received $\langle$ "1b", $b, \ldots\rangle$ messages from a quorum of acceptors, then it sends a $\langle$ "2a", $b, v\rangle$ message to the acceptors where, depending on the contents of those "1b" messages, either:

- The value of v is determined by the "1b" messages, or
- l chooses v arbitrarily from among the proposals it has received.

This action may not be performed twice for different values of v (with the same b).

*Phase*2$b(a, b, v)$ If acceptor a receives a $\langle$ "2a", $b, v\rangle$ message and it has not already received any message with a ballot number greater than b, it sends a $\langle$ "2b", $b, v\rangle$ message to every learner.

Learn(r, v, b) If learner r has received $\langle$ "2b", $b, v\rangle$ messages from a quorum of acceptors, then it learns that the value v has been chosen.

In normal execution, the actions occur in the order listed above, starting with the leader's *Phase*1a action. However, processes may fail, messages may be lost or delivered out of order, and several processors could simultaneously think they are the leader, causing "1a" and "2a" messages for several different ballot numbers to be sent concurrently. Nevertheless, the algorithm maintains its three safety properties, nontriviality, consistency, and conservatism. (We are assuming non-Byzantine failures in which a process can halt, but does not perform incorrect actions.) Moreover, if there is a single working processor l that believes itself to be the leader, has received a proposal, and can communicate with a quorum of acceptors, then some value will eventually be chosen. Any learner that can communicate with this quorum of acceptors will then learn the chosen value.

We can allow failed processes to be restarted if they have stable storage that survives a failure. Processes must maintain the following amounts of information in stable storage: an acceptor must keep two ballot numbers and one proposed value, and a leader must keep one ballot number (the largest one for which it has performed a *Phase*2a action).

As described here, the algorithm never terminates. A leader can at any time perform a *Phase*1a action for a new ballot number. In an application, there will be some point at which enough processes have learned the chosen value, after which processes can forget all about this instance of the algorithm, erasing any information about it from their stable storage.

For later reference, we make the following observations.

O1. We can save messages at the cost of an extra message delay by having a single distinguished learner that informs the other learners when it finds out that a value has been chosen. Acceptors then send "2b" messages only to the distinguished learner. In most applications, the roles of leader and distinguished learner are performed by the same processor.

O2. A leader can send its "1a" and "2a" messages just to a quorum of acceptors. As long as all acceptors in that quorum are working and can communicate with the leader and the learners, there is no need for acceptors not in the quorum to do anything.

O3. Acceptors do not care what value is chosen. They simply respond to "1a" and "2a" messages, using their stable storage to ensure that, despite failures, only a single value can be chosen. However, if an acceptor does learn what value has been chosen, it can store the value in stable storage and erase any other information it has saved there. If the acceptor later receives a "1a" or "2a" message, instead of performing its *Phase*1b or *Phase*2b action, it can simply inform the leader of the chosen value.

O4. Instead of sending the value v, the leader can send a hash of v to some acceptors in its "2a" messages. (A hash is a function H from values to a smaller set such that there is a negligible chance that $H(v)$ equals $H(v')$ for two different values v and v'.) A learner will learn that v is chosen if it receives "2b" messages for either v or its hash from a quorum of acceptors, and at least one of those messages contains v rather than its hash. However, a leader could receive "1b" messages that tell it the hash of a value v that it must use in its *Phase*2a action without telling it the actual value of v. If that happens, the leader cannot execute its *Phase*2a action until it communicates with some process that knows v.

2.2 Implementing a State Machine

In the state machine approach, a set of servers execute commands submitted by clients. For simplicity, we assume that each server keeps in stable storage the entire sequence of state machine commands that have been chosen so far. In many applications, a server would keep only a recent checkpoint of the state machine's state and the commands after that checkpoint.

In the traditional Paxos algorithm, the clients are the proposers and each server acts as an acceptor, a learner, and a potential leader in each instance of the consensus algorithm. A quorum consists of a majority of the servers. The leader receives client commands, assigns each one a number, and tries to get the i^{th} command to be chosen by the i^{th} instance of the Paxos consensus algorithm.

To understand how Static Paxos works, suppose the system has been operating for a while when the leader fails. A new server l is then selected to be leader. Since l is a learner, it should know most of the commands that have already been chosen. Suppose it knows commands 1–134, 138, and 139—that is, the commands chosen in instances 1–134, 138, and 139 of the consensus algorithm. (Such a gap in its knowledge is possible because multiple instances of the consensus algorithm can be executed concurrently.) Server l chooses a ballot number b that it believes to be greater than any ballot number used by previous leaders. (The election algorithm can be used to choose b as well as l.) It then simultaneously executes $Phase1a(b, l)$ for instances 135–137 and for all instances greater than 139 of the consensus algorithm, sending "1a" messages to all the servers. (Some of those messages are to itself, since the leader is chosen from among the servers.) It can obviously send these infinitely many virtual messages in a single physical message.

Each server then simultaneously executes $Phase1b$ actions in response to those virtual "1a" messages, sending infinitely many virtual "1b" messages back to l. Since those "1b" messages contain information only for instances for which actions have been performed, those virtual messages will contain only a finite amount of information that can usually be fit into a single real message. By observation O3 above, if a server knows that a command was already chosen by some instance, it responds with the chosen command rather than a "1b" message for that instance.

Suppose that, from these messages, l learned:

- The command that was chosen in instance 135 (sent by some server instead of an instance 135 "1b" message).
- Commands v_{137} and v_{140} that it must use as the value v in its $Phase2a(l, b)$ actions for instances 137 and 140, respectively.
- For instance 136 and for all instances greater than 140, it can use any proposed command v in its $Phase2a(l, b)$ action.

Leader l then does the following:

- It performs $Phase2a(l, b)$ actions for instances 137 and 140, using the commands v_{137} and v_{140} determined by the "1b" messages it received.
- It performs the $Phase2a(l, b)$ action for instance 136, using as the command v a special *no-op* state machine command that does nothing.
- In some manner that does not concern us, it ensures that all servers know commands 1–135, 138, and 139.

If a majority of the servers are working, they will perform $Phase2b$ actions for instances 136, 137, and 140, and all servers will learn the commands chosen for all instances 1–140 of the consensus algorithm. However, even before that has happened, leader l can resume normal operation. It assigns the number 141 to the first client command it receives, and it executes $Phase2a(l, b)$ for instance 141 using that command as the value v. It assigns number 142 to the next client command and executes $Phase2a(l, b)$ for that instance and that command as value v. And so on.

Since each server is a learner, it learns the sequence of chosen commands. In most applications, the leader will act as the distinguished learner (mentioned in observation O1 above) to which "2b" messages are sent. Once a server has learned what command the i^{th} command is, it can delete all other information about the i^{th} instance of the consensus protocol from its storage.

When a failed server is repaired, it must be brought up to date so it knows all the commands that have already been chosen. In principle, this is a straightforward matter of having the newly repaired server obtain the information from some working server. If a server maintains only recent commands and a checkpoint of the state, then the repaired server must update its saved checkpoint. If the state machine maintains a large state, this must be done in such a way that only the part of the state that has changed is sent to the repaired server.

2.3 Dynamic Paxos

So far, we have described Static Paxos, in which the set of acceptors and the quorums are constant and fixed in advance. A system that must continue working despite the failure of any F processors then requires $2F + 1$ servers. For example, with Static Paxos, it takes seven

servers to tolerate three failures. In many systems, the best way to achieve the desired degree of fault tolerance is to reconfigure the system to replace failed servers by spares. With reconfiguration, a system that uses three active servers and two spares can tolerate a total of three failures, if a failed server can be replaced by a spare before another failure occurs. Reconfiguration therefore allows fewer processors to tolerate the same total number of failures, though not the same number of simultaneous failures. (In most systems, simultaneous failures are much less likely than successive ones.)

In Dynamic Paxos, the set of acceptors and the quorums are determined by the state machine itself. Reconfiguration is performed by state machine commands. To explain how this works, let state k be the state machine's state after executing command k. For $k \leq 0$, define state k to be the initial state. For some fixed constant α, we let the acceptors and quorums used for instance i of the consensus algorithm be determined by state $i-\alpha$. Before performing any action for instance i, a leader waits until it knows state $i-\alpha$. In other words, a leader must wait until it knows all commands through command number $i-\alpha$ before it knows to which acceptors it should send its "2a" messages for the i^{th} instance of the Paxos consensus algorithm.

As a simple example of how this might work, consider a system with a fixed set S of processors that can act as servers. Let the set of servers currently executing the system (and acting as acceptors) be G, and let a quorum consist of a majority of the processors in G. Suppose we want a processor to be declared to have failed or have been repaired if a majority of the processors in G believe it has. The state machine's state would contain the set G together with a Boolean array *good*, where $good[p, q]$ indicates whether processor p believes processor q is nonfaulty, for all $p, q \in S$. A processor r would issue a state machine command to change the value of $good[r, s]$ when it believes processor s has failed or been repaired. Such a command would set the new value $good'$ of the array *good* in the obvious way, and it would set the new value G' of G to equal the set of all processors $q \in S$ such that $good'[p, q]$ equals TRUE for a majority of processors $p \in G$. (Remember that a change to G caused by command number i takes effect beginning with instance $i+\alpha$ of the Paxos consensus algorithm.)

In practice, deciding when to reconfigure a system is not easy. Replacing servers that have not failed can cause the system to run out of servers; but not replacing a failed server lowers the system's tolerance to additional failures. One would probably not use so naive an algorithm as the one just described. Instead, one would use a more sophisticated algorithm, tuned for the particular system. Any desired algorithm can easily be implemented with the state machine.

When a new server is added to the system, it must learn the current state machine state. This is essentially the same problem as bringing a failed server up to date, which is discussed in Section 2.2 above.

3 Cheap Paxos

3.1 The Algorithm

We now develop Cheap Paxos as an instance of Dynamic Paxos. In Cheap Paxos, we posit a system of $F+1$ main processors and F auxiliary processors. The main processors act as the servers in a distributed state machine implementation. The auxiliary processors perform actions only in the event of the failure of a main processor, after which the main processors continue to operate the system by themselves.

The key to Cheap Paxos is observation O2. In normal operation of the Paxos consensus algorithm, the leader sends only "2a" messages. By O2, those messages need be sent to and acted upon by only a quorum of acceptors. Hence, to implement Cheap Paxos, we use Dynamic Paxos to configure the system so that the set of all working main processors forms a quorum. As long as these processors continue working, they can execute the system. If one of them fails, then the quorum consisting only of main processors can no longer succeed in choosing commands. A different quorum, containing one or more auxiliary processors, is then used to (i) complete the execution of any of the instances of the Paxos consensus algorithm that were in progress when the failure occurred, and (ii) propose and choose the necessary state machine commands to reconfigure the system. The reconfiguration removes the failed processor and modifies the set of quorums so the remaining main processors form a quorum. These main processors can then resume executing the system, while the auxiliary processors once more become idle.

Cheap Paxos uses Dynamic Paxos, where the set G of all processors that are currently acceptors is determined by the state machine state. Let M be the subset of G consisting of all the main processors in G. We want M to be a quorum. Since the only requirement on quorums is that any two of them have a non-empty intersection, we can let M be a quorum and let the other quorums consist of all sets containing a majority of the processors in G and at least one processor in M. (If M contains only a single processor p, then a quorum can consist of any set containing p, of course including M itself.) We require that G contain at least one main processor—a condition that is satisfied by any sensible reconfiguration algorithm because failure of all main processors implies that there

is no quorum of working processors, so no state machine commands can be chosen until a main processor is repaired. (A non-sensible reconfiguration algorithm could gratuitously remove the last working main processor from G.)

In normal operation, the processors in M execute phase 2 of successive instances of the Paxos consensus algorithm to choose the sequence of state machine commands. They can perform reconfiguration commands to add a repaired main server to G and M. However, if a main processor fails, then there will no longer be a working quorum consisting only of main processors. The following sequence of steps is then performed.

1. If the failed processor was the leader, a new leader is selected from among the processors in M that are still working.

2. The leader interrogates the other working main processors to learn all chosen commands that any main processor knows about.

3. The leader completes the execution of any instances of the Paxos consensus algorithm that were in progress when the main processor failed, using a quorum that contains one or more auxiliary processors. If a new leader was chosen in step 1, it does this as described in Section 2.2 above by choosing a new ballot number and initiating phase 1 with that ballot number for all relevant instances of the consensus algorithm. If the old leader is still working, it just sends to the auxiliary processors the same "2a" messages that it had already sent to the main processors.

4. As in standard Dynamic Paxos, the working processors propose and choose a sequence of state machine commands to reconfigure the system so the failed main processor is removed from the set G of acceptors (and hence from M).

5. The leader proposes and gets chosen a sequence of α *no-op* state machine commands. Let j be the command number of the last of these commands.

After step 5, the new set G of acceptors chosen in step 4 is in effect for subsequent instances of the consensus algorithm. This means that the set M of main processors in G constitute a quorum, so they can resume executing the system. However, remember that the Paxos consensus algorithm's ability to recover from failures rests on acceptors maintaining certain information in stable storage. To bound the amount of storage required by auxiliary processors, they need to be able to forget the information they saved in steps 3–5, where they participated in executing instances of the consensus algorithm. They can do that after the working main processors have learned the commands chosen in those instances. Therefore, before the system resumes normal execution, the following steps should also be performed.

6. The leader ensures that all processors in (the new) M know all commands through command number j (defined in step 5).

7. The leader instructs all auxiliary processors to remember in stable storage that instances 1 through j of the consensus algorithm have chosen commands, and that they have performed no actions for any other instances. (They need not record any of the chosen commands.) The auxiliary processors can then erase from stable storage all information relevant to any of the first j instances of the consensus algorithm.

This sequence of steps describes what normally happens when a main processor fails. A complete algorithm must handle abnormal cases as well—for example, if the leader fails while these steps are being performed, or if two processors both believe they are the leader and have different views of what processor has failed. But the actions performed in these steps are just implementing Dynamic Paxos. (Steps 2 and 6 simply disseminate knowledge of what commands have been chosen.) The precise definition of these actions is therefore the same as in ordinary Paxos. The only difference is that an auxiliary processor may not be able to respond appropriately to a "1a" or "2a" message because it has erased the needed information in step 7. In that case, instead of replying with the chosen command as indicated in observation O3, it must ignore the message. (It could report to the leader why it is ignoring the message, advising the leader to ask a main processor what command was chosen.)

The auxiliary processors are needed only in the event of failure of one of the main processors, at which time they must participate in the execution of only a small number of instances of the Paxos consensus algorithm. This would seem to imply that they do not need much processing power. However, the consensus algorithm requires them to write proposed commands to stable storage. In some applications, such commands could be quite big, and writing them to stable storage could be expensive. If this is the case, we can apply observation O4 and have auxiliary processors receive and store only hashes of proposed commands. Since every quorum contains a main processor, a learner receiving "2b" messages from a quorum must receive at least one that contains the command rather than its hash. However, we need to prevent the problem mentioned in O4, in which progress is prevented because a leader knows only the hash of a value without knowing the value

itself. This is done by having the leader delay sending a "2a" message with the hash of a value v to any auxiliary processor until all the working main processors have acknowledged receipt of their "2a" messages containing v.

As in ordinary Dynamic Paxos, we are not committed to any particular algorithm for determining that a processor has failed or has been repaired. Since the reconfiguration is performed by state machine commands, any algorithm can be used. In practice, the algorithm used by Cheap Paxos for determining whether a main processor has failed will be quite different from that used in traditional Dynamic Paxos. In traditional implementations of Dynamic Paxos, any majority of acceptors constitutes a quorum, so system can continue to make progress even if a server has failed. In that case, one can afford to wait to make sure the server has really failed before reconfiguring it out of the system. But Cheap Paxos stops making progress as soon as one main processor fails. It must therefore be more aggressive in removing processors that may have failed. Although this difference affects the details of deciding what processors are working, it does not change the basic state machine algorithm.

We have been tacitly assuming that the set of all processors (main plus auxiliary) is fixed. Reconfiguration can also be used to change the set of processors. We are already reconfiguring the set of acceptors to remove failed main processors and add repaired ones. Additional fault tolerance can be obtained by replacing failed auxiliary processors with spares, just as in ordinary Dynamic Paxos. This reconfiguration can be performed with state machine commands executed by only the main processors; auxiliary processors need only periodically inform the main processors that that are still working. A newly installed auxiliary processor needs to remember in its stable storage only that it has not performed any actions for any instances of the Paxos consensus algorithm.

An auxiliary processor acts only as an acceptor, not a learner, so it does not know what commands are chosen. Hence, if reconfiguration can change the set of auxiliary processors, an auxiliary processor does not know whether it is an acceptor. This makes no difference. As observed in O3, acceptors simply respond to "1a" and "2a" messages. Only leaders and learners, which are roles played by main processors, need to know the set of acceptors and the quorums.

3.2 Correctness of Cheap Paxos

Cheap Paxos uses the Paxos consensus algorithm to choose commands. Its safety properties follow directly from the safety properties of the consensus algorithm. In particular, two different servers can never disagree about the value of the i^{th} command, for any i.

The liveness properties of Cheap Paxos can also be inferred from those of the Paxos consensus algorithm. However, this is complicated because Cheap Paxos is an implementation of Dynamic Paxos, in which liveness depends on precisely how the reconfiguration is performed. For example, the system can make no progress if a reconfiguration selects a set of failed or nonexistent processors as acceptors. Moreover, simply being able to choose new commands doesn't ensure progress. To be able to execute the i^{th} command, a server needs to know not just that command, but also all previous commands. For example, the system could make no progress if command 1 had been chosen, but no working server knew its value. Cheap Paxos also has the additional complication that auxiliary processors forget information that can be used by ordinary Paxos to recover from certain failures.

To state the liveness property satisfied by Cheap Paxos, we need some definitions. We call a set of processors *nonfaulty* if they are all working and can communicate with one another in a timely fashion. Define command number i to be *recorded* if some auxiliary processor has stored in its stable storage the fact that i has been chosen. (That is, the auxiliary processor has recorded in step 7 of the reconfiguration procedure that all commands numbered through j have been chosen, for some $j \geq i$.) A command number i is said to be *active* if i has not been recorded, but a "2a" message has been sent for instance i of the consensus algorithm. We define a main processor to be *up-to-date* if it knows all recorded commands. (If auxiliary processors store only hashes of values, then for a main processor p to be up-to-date, it must also satisfy the following condition: For every active command number i and every "2a" message sent to an auxiliary processor in instance i of the consensus algorithm, p must have received its corresponding "2a" message.)

We can now state the liveness property satisfied by Cheap Paxos. Step 6 of the reconfiguration procedure assumes some method of propagating knowledge of chosen commands. We assume that knowledge is continually exchanged among the main processors so that, if p and q are main processors, p knows a command, and the set $\{p, q\}$ is nonfaulty for long enough, then q will learn that command. The liveness property satisfied by Cheap Paxos is then:

> The system makes progress if there is a nonfaulty set of processors containing a unique leader, at least one up-to-date main processor, and, for all active command numbers i, a quorum for instance i of the consensus algorithm.

Informally, we can view the set of nonfaulty main processors as an "amoeba" that withdraws a pseudopod when a processor fails and extends one when a processor is repaired. It takes time for knowledge of chosen commands to flow into a new pseudopod. If the amoeba were to move around too quickly, knowledge could be lost because processors failed before their knowledge could flow to newly repaired processors. Assuming that the total number of nonfaulty processors (main plus auxiliary) remains large enough, Cheap Paxos guarantees that the system will continue to make progress as long as the amoeba moves slowly enough that such knowledge is not lost.

4 Conclusion

Cheap Paxos is a variant of the Paxos algorithm that can make progress in the face of up to F failures by using $F + 1$ main processors plus F auxiliary processors. Unlike the spare processors used in previous systems, our auxiliary processors need do nothing except for a brief period after a main processor fails. The auxiliary processors therefore do not require nearly as much processing power or storage as the main processors. By using auxiliary processors, Cheap Paxos can lead to a system that achieves greater fault tolerance than other algorithms with the same number of main processors.

Acknowledgement

Butler Lampson pointed out to us the problem of large commands, and its solution through observation O4. He was also first to observe that, in a configuration with a single working main processor, every set of processors containing that processor is a quorum.

References

[1] R. De Prisco, B. Lampson, and N. Lynch. Revisiting the PAXOS algorithm. *Theoretical Comput. Sci.*, 243:35–91, 2000.

[2] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, Apr. 1988.

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

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

[5] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[6] L. Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4 (Whole Number 121)):18–25, Dec. 2001.

[7] B. W. Lampson. How to build a highly available system using consensus. In O. Babaoglu and K. Marzullo, editors, *Distributed Algorithms*, volume 1151 of *Lecture Notes in Computer Science*, pages 1–17, Berlin, 1996. Springer-Verlag.

[8] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, and L. Shrira. Replication in the harp file system. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 226–238. ACM Press, 1991.

[9] J.-F. Pâris and D. D. E. Long. Voting with regenerable volatile witnesses. In *Proceedings of the Seventh International Conference on Data Engineering, April 8-12, 1991, Kobe, Japan*, pages 112–119. IEEE Computer Society, 1991.

[10] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.