Reconfiguring a State Machine

Leslie Lamport Dahlia Malkhi Lidong Zhou

29 January 2010

Abstract

Reconfiguration means changing the set of processes executing a distributed system. We explain several methods for reconfiguring a system implemented using the state-machine approach, including some new ones. We discuss the relation between these methods and earlier reconfiguration algorithms—especially view changing in group communication.

Contents

1 Introduction **1**

2 State Machines **2**
2.1 Preliminaries . . . 2
2.2 Garbage Collection . . . 3
2.3 Reconfiguration Made Easy . . . 3
2.4 Correctness . . . 4

3 Reconfiguration Made Harder **4**
3.1 Stopping a State Machine . . . 5
3.1.1 The Stop Sign . . . 5
3.1.2 Padding . . . 5
3.1.3 The Brick Wall . . . 6
3.2 Choosing a New Configuration . . . 7
3.3 Combining the Command Sequences . . . 7
3.4 The Interface . . . 8

4 Reconfiguration in Group Communication and Other Works **9**

5 Conclusions **11**

References **11**

1 Introduction

A fault-tolerant system may need to change the set of processes that is executing it—a procedure called *reconfiguration*. Reconfiguration can be performed to reduce the vulnerability to further failures after a process has failed [25] or to replace hardware without shutting down the system. Distinguishing process failure from transient communication problems is often a challenging engineering problem. Waiting too long to reconfigure can reduce fault tolerance, but not waiting long enough can lead to thrashing. A reconfigurable system provides an interface for reconfiguring it, separating reconfiguration from the decision of when to reconfigure and what new configuration to use. That decision can be made by any desired algorithm or by a human operator.

A state machine (also called an *object* [12]) accepts commands and produces outputs. The functional behavior of any system can be specified by a state machine [16]. The state-machine approach consists of implementing a fault-tolerant distributed system (or subsystem) by describing it as a state machine and using a general fault-tolerant algorithm to implement that state machine [20, 24]. A state machine's definition and the correctness of its implementation directly imply the correctness properties of the system.

An important property provided by the state-machine approach is *irrevocability* of output. For a stock exchange, irrevocability means that if a broker receives output from the system confirming that a customer has bought 100 shares of Nocturnal Aviation, then she really owns them. Even if the broker's computer is destroyed by an asteroid, the customer can go to any other broker and sell those shares. Without irrevocability, it is hard to explain what fault tolerance means for a system.

With the state-machine approach, it is easy to make a system reconfigurable by letting the state machine itself specify the configuration [18, 24]. Although this basic idea was described more than twenty years ago [17] and is implicit in earlier process-control applications [25], it still appears not to be well understood. We therefore review it along with the state-machine approach in Section 2.

Another approach that may seem more intuitive is to reconfigure the system by terminating the state machine, letting it choose the new configuration, then resuming execution with a new state machine that uses the new configuration. This is the way reconfiguration is viewed in group communication [6, 8]. We show in Section 3 that, while it is not as simple as the classic one described in Section 2, this approach does lead to new reconfigurable state-machine implementations. These two approaches to reconfiguration are different paths to obtaining reconfiguration algorithms, but they do not imply any fundamental difference in the resulting algorithms.

The relation between reconfiguration in the state-machine approach, in group communication, and in some particular systems is discussed in Section 4. Although most of our methods for state-machine reconfiguration work in the presence of malicious processes, the body of this paper considers only crash faults. Malicious faults are discussed in the conclusion.

2 State Machines

2.1 Preliminaries

A state machine is described by a set of states, an initial state, and a function that maps command-state pairs to output-state pairs. If the pair $\langle c, s \rangle$ is mapped to the pair $\langle o, s' \rangle$, then we say that executing command c in state s produces output o and changes the state to s'. Execution of a state machine consists of executing a sequence of commands in the obvious way, starting with the initial state, to produce a sequence of outputs and new states. A *noop* is a special command that produces a null output and leaves the state unchanged.

A state-machine implementation provides an interface by which clients propose commands to the system and receive outputs from it. Outputs may be sent to clients other than the proposer, but the proposer usually receives output at least informing it that the command was executed. A command can include the identity of the client that proposes it, and the state machine can be specified to treat the command as a *noop* if the client is not authorized to propose it.

The safety requirement for a state-machine implementation is that the outputs received by all clients are generated by a single sequence of *chosen* commands, each of which has been proposed by a client. (A more complete specification also requires the state machine/object to be linearizable [12].) The safety requirement implies irrevocability.

As with any fault-tolerant distributed algorithm, the precise liveness property satisfied by a state-machine implementation depends on the details of how it is implemented. It states approximately that, if enough servers are nonfaulty, and eventualy partial synchrony is satisfied, then proposed commands are added to the sequence of chosen commands and their outputs delivered to nonfaulty clients.

There are two (usually non-disjoint) sets of servers called *acceptors* and *learners*. Acceptors choose the commands to be executed; learners maintain the state, learn what commands are chosen, and execute them, generating the outputs. Acceptors essentially provide stable storage and do not care what commands are chosen. Making progress despite the failure of any f servers requires at least $2f + 1$ acceptors and $f + 1$ learners [19].

The classic way of implementing a state machine is with a consensus algorithm for choosing a single command. The implementation runs a sequence of logically separate instances of the consensus algorithm, using instance i to choose the i^{th} command. Most state-machine implementations use a special subset of learners called *leaders*. A client proposes a command c by sending it to a leader, which assigns it a number i and proposes c as the command to be chosen by consensus instance i. The number i assigned to command c can be included in the output generated by executing c. How client commands are delivered to a leader is an instance of the general problem of how clients of a distributed system locate the servers executing the system. We do not discuss this problem.

A leader does not have to wait until command i is chosen before proposing another command as number $i+1$. Different commands can be chosen concurrently.

In general, command i cannot be executed and its output determined until all commands numbered less than i have been chosen. However, there are important cases in which the output of a command can be generated as soon as the command is chosen—for example, if the output reveals only the command's number and the fact that it has been executed. Because the choice is irrevocable, choosing a command is tantamount to executing it. (Linearizability is satisfied if command numbers are consistent with the order in which the commands are proposed.)

2.2 Garbage Collection

In a non-reconfigurable system, we could allow the acceptors to maintain forever their information for each instance of the consensus algorithm. At any time, a learner could then learn the entire sequence of chosen commands by communicating with enough acceptors, where enough generally means a majority. In practice, consensus instances must usually be garbage collected. Periodically, learners checkpoint the state at some point in the execution sequence and instruct the acceptors to forget about consensus instances for earlier commands. Although command indices continue growing monotonically, the memory taken by lower numbered commands may be reclaimed. Exactly how this is done is an engineering detail that does not concern us.

2.3 Reconfiguration Made Easy

A configuration is the set of processes (clients, acceptors, etc.) that are executing the system. The consensus algorithm used to implement a state machine assumes a fixed configuration, so we must ensure that each instance of that algorithm is executed by a single configuration. (It is the set of acceptors that is important; adding or removing clients or learners while executing a consensus algorithm is not a problem.) In the *easy* approach, reconfiguration is achieved by using different configurations for different instances. We define the configuration at command number i to be the one used to execute consensus instance i.

To prevent chaos, processes must agree on the configuration at command i. The easy way to obtain a reconfigurable state-machine implementation is to introduce a component *cfg* of the state-machine state that specifies the current configuration. In other words, the configuration at command i is determined by the value of *cfg* in the state immediately following execution of command $i - 1$, or by its initial value if $i = 1$. (We use ordinal numbers for commands, so the first command is number 1.) We add reconfiguration commands of the form $\mathit{rcfg}(\mathcal{C})$, which specifies a new configuration $\mathcal{C}$.

The obvious way to define the state machine is to let executing $\mathit{rcfg}(\mathcal{C})$ set *cfg* to $\mathcal{C}$, so reconfiguration occurs immediately. We call this method $\mathcal{R}_1$. The problem with $\mathcal{R}_1$ is that it prevents concurrent processing of different proposed commands. Since the configuration used to execute instance $i + 1$ of the consensus algorithm can be changed by executing command i, the state-machine implementation cannot begin choosing command $i + 1$ until command i has been chosen.

To allow concurrent processing of commands, we define the state machine (by introducing additional state) so that executing $\mathit{rcfg}(\mathcal{C})$ as command number i causes cfg to change after executing command $i + \alpha - 1$, for some positive integer α. A reconfiguration command thus takes effect α commands later, allowing the concurrent processing of up to α commands. (This was originally described for $\alpha = 3$ in the mistaken belief that the generalization would be obvious [18]. A practical implementation was later presented in [21]). We call this method $\mathcal{R}_\alpha$. As the notation implies, $\mathcal{R}_1$ is the $\alpha = 1$ case of $\mathcal{R}_\alpha$.

To make the reconfiguration happen right away, a client that proposes a reconfiguration command can propose *noop*s as the next $\alpha - 1$ commands. A sequence of successive commands, with successive command numbers, can be batched so that they are proposed, chosen, and executed as efficiently as (using no more messages than) a single command. Thus, α can be made arbitrarily large, permitting the concurrent processing of any desired number of commands. To maintain correctness, the implementation must produce the same result as if each instance of the consensus algorithm were executed separately.

With method $\mathcal{R}_\alpha$, reconfiguration is performed using the ordinary state-machine interface for proposing commands. Only a subset of the clients might be authorized to propose reconfiguration commands, effectively separating the reconfiguration interface from that used to propose other commands.

2.4 Correctness

A simple induction argument shows that $\mathcal{R}_\alpha$ maintains the safety property of a state-machine implementation. In particular, it satisfies irrevocability. Liveness for a state-machine implementation states that it makes progress if enough servers are nonfaulty. Reconfiguration raises the question, enough of which servers?

The purpose of reconfiguration is to eliminate the dependence on servers that have been reconfigured out of the system. Information maintained by those servers must be transferred to servers in the new configuration. This is done by garbage collecting consensus instances executed under the old configuration and transferring the state to the new configuration's learners. During the transition to the new configuration, progress is guaranteed only if there are enough nonfaulty servers from both the old configuration and the new one. The liveness property satisfied by the resulting algorithm is not easy to state precisely, but its general nature should be intuitively clear.

3 Reconfiguration Made Harder

Algorithm $\mathcal{R}_\alpha$ of Section 2.3 has the practical drawback that a reconfiguration introduces a sequence of *noop* commands of length about α. This can be inconvenient for large values of α—for example, $\alpha = 2^{64}$. The bound on concurrency implied by α, even if not a practical concern, may also be considered inelegant. We now present reconfiguration algorithms that permit any number of commands to be chosen concurrently during normal execution, when no reconfiguration is in progress.

We obtain a reconfigurable state-machine implementation by combining the executions of a sequence of separate non-reconfigurable state-machine implementations. To reconfigure when executing state-machine implementation number v, we stop that execution, choose the new configuration, and start the execution of state-machine implementation number $v + 1$ by that configuration. The starting state of state machine $v + 1$ is the final state of state machine v.

This approach requires solving three largely orthogonal problems: (i) stopping the current state machine, (ii) choosing the configuration to implement the next state machine, and (iii) combining the sequence of commands chosen for each separate state machine into a single sequence. We consider them separately, starting with three basic methods for implementing a stoppable state machine.

3.1 Stopping a State Machine

3.1.1 The Stop Sign

The Stop-Sign method adds to the state machine a *stop* command that turns every subsequent state-machine command into a *noop*. Unlike algorithm $\mathcal{R}_1$ of Section 2.3, this method allows multiple commands to be chosen concurrently. Any command chosen after the *stop* command simply has no effect. However, the Stop-Sign method has the following problem. As explained in Section 2.1, an ordinary state-machine implementation allows an output like "this command was executed" to be generated as soon as the command has been chosen. The Stop-Sign method does not allow any command's output to be generated before every lower-numbered command has been chosen, since one of those commands could turn out to be *stop*.

The *delayed* Stop-Sign method allows command number i to be executed, if possible, as soon as it and all commands numbered at most $i - \alpha$ have been chosen. Just as $\mathcal{R}_\alpha$ generalizes $\mathcal{R}_1$ to make the reconfiguration command take effect α commands later, the delayed Stop-Sign method generalizes the *stop* command to take effect α commands later. In other words, if command number i is a *stop*, then all commands starting with number $i + \alpha$ are treated as *noop*s. As with $\mathcal{R}_\alpha$, the client proposing the *stop* command can at the same time propose $\alpha - 1$ consecutive *noop* commands.

3.1.2 Padding

We have already observed that we can batch a sequence of commands and process them essentially as easily as a single command. This applies equally well to an infinite sequence of consecutive commands, if all but a finite number of them are *noop* commands. (It is easy to devise a finite encoding of the information needed to execute the infinite number of consensus instances.) In the Padding method, a client stops the state machine by proposing an infinite number of *noop* commands. More precisely, it proposes *noop*s for all commands numbered greater than some i. When command number j has been chosen for all positive integers j, the state machine has stopped.

3.1.3 The Brick Wall

The Stop-Sign and Padding methods use the underlying state-machine implementation, so their correctness follows immediately from the correctness of that implementation. However, they may seem unintuitive—the Stop-Sign method because it can choose commands that are never executed by the state machine and the Padding method because it fills the sequence with infinitely many *noop* commands. We now describe the *Brick-Wall* method that is conceptually simpler but requires a new state-machine implementation called *Stoppable Paxos*.

Like the Stop-Sign method, Stoppable Paxos assumes a special *stop* command. However, it guarantees that if a *stop* command is chosen as command number i, then no command can ever be chosen for any command number greater than i. A precise description and rigorous correctness proof of Stoppable Paxos appears in [14]. Here, we briefly sketch the algorithm, starting with a description of classic Paxos [18].

As described in Section 2.1, Paxos implements a state machine by using logically separate instances of a consensus algorithm. It assumes a method for selecting a leader, guaranteeing progress only when there is a unique leader. (Safety is preserved despite multiple leaders.) The Paxos consensus algorithm uses numbered ballots, each initiated by at most one leader. (Do not confuse ballot numbers with consensus-instance/command numbers.) A newly selected leader chooses a ballot number b it believes to be greater than any already used, and it begins ballot b by sending a *phase 1a* message to the acceptors, who respond with *phase 1b* messages. If the leader has chosen b large enough and it receives phase 1b messages from a majority of the acceptors, then it will learn that either:

P1. Command c (and no other command) might have been chosen by a lower-numbered ballot, or

P2. No command has been chosen by a lower-numbered ballot.

The leader then proposes a command in ballot b by sending a *phase 2a* message to the acceptors, proposing c in case P1 and any command in case P2. If no higher-numbered ballot is begun, the proposed command will be chosen when a majority of acceptors receive the phase 2a message. The Paxos state-machine algorithm is efficient because the phase 1a and 1b messages sent by any one process for all consensus instances are bundled into a single physical message.

Stopping Paxos prevents a leader from proposing a command if *stop* could be chosen as a lower-numbered command. To describe how this is done, we first define $\xi(b, i)$, for ballot b of consensus instance i, to equal the command c of case P1 above, and to equal $\perp$ in case P2. Stopping Paxos places the following two additional constraints on what command a leader can propose in ballot b of instance i:

S1. It cannot propose any command if, for some $j < i$, it proposed a reconfiguration command in ballot b of instance j or $\xi(b, j)$ is a reconfiguration command.

S2. It cannot propose a reconfiguration command if, for some $k > i$, it has proposed a command in ballot b of instance k or $\xi(b, k) \neq \perp$.

It is not hard to show that S1 and S2 prevent instance i from choosing any command if a lower-numbered instance chooses *stop*. However, S1 can prevent progress if a reconfiguration command is proposed but not chosen by a ballot numbered less than b in instance $j < i$. To eliminate this problem, we modify the definition of ξ approximately as follows: in case P1, $\xi(b, i)$ equals $\perp$ if c is a reconfiguration command proposed in ballot $b' < b$ and some command was proposed in ballot b'' of instance k with $b'' > b'$ and $k > i$. The precise definition and the proof that this works are not trivial.

3.2 Choosing a New Configuration

When changing from state-machine implementation v to state-machine implementation $v+1$, there are two basic ways to choose the new configuration that executes implementation $v + 1$:

R1. Let it be chosen by a reconfiguration command executed by state machine v.

R2. Use a special instance of the consensus algorithm, executed by the same configuration that executes state machine v.

Option R2 has the potential advantage of allowing the processes in configuration $v + 1$ to be determined and commands to be chosen for state machine $v + 1$ before state machine v has been stopped. It is not clear if this is ever a good idea in practice. For option R1 to work, we must:

(a) Make sure that a reconfiguration command is passed before configuration v is terminated, and

(b) Decide what to do if multiple reconfiguration commands are passed.

The solution to (a) is obvious for the Stop-Sign and Brick-Wall methods—namely, let the *stop* command specify the new configuration. For the Padding method, a client should make sure that a reconfiguration command has been chosen before it proposes an infinite sequence of *noop*s. (If no reconfiguration is chosen, then configuration $v + 1$ is the same as configuration v.)

Problem (b) does not exist for the Brick-Wall method. For the other methods, there are two obvious choices: use either the first reconfiguration command that was passed or the last one. If we use the first one, then the system can start choosing commands for configuration $v+1$ as soon as all chosen commands through the first reconfiguration command are known.

3.3 Combining the Command Sequences

We have shown how to initiate and execute a sequence of state-machine implementations, all but possibly the last one being stopped. State machine number v is executed with configuration v.

If a state machine has been stopped, it has a unique "last interesting" command. For the Stop-Sign or Brick-Wall method, that command is the (first) *stop* command.

For the delayed Stop-Sign method, it is $\alpha - 1$ commands after the *stop* command. For the Padding method, it is the last non-*noop* command. Let $\eta(v)$ be the number of the last interesting command of state machine number v, and define $\eta(0)$ to equal 0.

Let us assign the "number" $\langle v, i \rangle$ to the i^{th} command chosen for state machine number v. With the usual lexicographical ordering of pairs, this provides a linearly ordered numbering of the sequence of all chosen commands. However, it is not a useful numbering scheme because it does not tell a client whether there are any commands between the ones numbered $\langle 4, 417 \rangle$ and $\langle 5, 1 \rangle$. It is better to number commands with consecutive positive integers.

To give commands integer numbers, we observe that there is no reason why a state machine's command numbers should start with 1. Let us make the starting number a parameter of the state machine, and let the first command of state machine number $v + 1$ be one greater than that of the last interesting command of state machine v. In other words the first command of state machine $v + 1$ has number $\eta(v)+1$. Thus, command i is the one chosen as command number i of state machine number v, where $\eta(v-1) < i \leq \eta(v)$. To implement this, the choosing of commands in state machine $v + 1$ cannot begin until the value of $\eta(v)$ is known.

In a naive implementation, each process would maintain a two-dimensional array of data, where element $\langle v, i \rangle$ contains the data maintained for consensus instance i of state machine number v. However, for state machine v, there is no consensus instance i if $i \leq \eta(v-1)$, and it does not matter what value is chosen by instance i if $i > \eta(v)$. In any state-machine implementation that uses a sequence of completely separate consensus instances, a process need maintain information for instance i only for the largest state machine number v for which processing of command i has been initiated. To make this true using a state-machine implementation like Stoppable Paxos, in which the execution of other consensus instances can depend on the state of consensus instance i, the implementation must be modified so it can forget the state of instance i when $i > \eta(v)$. For Stoppable Paxos, the modification is simple.

3.4 The Interface

In the reconfiguration methods based on a sequence of state machines, a state machine is stopped by proposing either a *stop* command or an infinite sequence of *noop* commands. If the new configuration is specified by a state-machine command (option R1), then the entire reconfiguration is performed using the ordinary state-machine interface. If the new configuration is determined by a special consensus instance that is not part of a state-machine execution (option R2), then a separate interface is required to specify the configuration.

In practice, any of these methods would use a separate interface for issuing reconfiguration requests. The state-machine commands needed to perform the reconfiguration would be proposed by a leader, which would also initiate the special consensus instance in option R2.

4 Reconfiguration in Group Communication and Other Works

Group communication (GC) is an alternate method for implementing a fault-tolerant distributed system, in which a group of processes called a *view* execute a broadcast mechanism so that all nonfaulty processes in the view receive the same set of messages [6, 5]. Reconfiguration is an integral part of GC, being required to achieve fault tolerance. There are a number of different versions of GC that provide different guarantees. Here, we consider only a few of the versions that produce a consistent total ordering of delivered messages, referring the reader to the survey by Chockler et al. [8] for a more detailed account of previous work.

A GC service provides an interface in which the processes in a view send messages. An ordered sequence of sent messages is *delivered* to the processes. Each delivered message is usually chosen by the view's processes with an unreliable consensus algorithm—an algorithm that guarantees at most one message is chosen, but may be prevented from making progress by the failure of even one process. (Unreliable consensus is usually implemented with a leader whose failure prevents progress.) In the event of failure, the current view is ended and a new one is begun. The view change is performed by using a fault-tolerant consensus algorithm to determine (i) the sequence of messages that were delivered in the old view and (ii) the members of the new view. The interface provided by a GC service typically allows processes to enter and leave a view and to learn what processes belong to the view, but the implementation may spontaneously change the view in response to a failure.

There is an obvious correspondence between GC and the state-machine approach in which a view corresponds to a configuration. We can consider a state-machine implementation to be based on a GC service whose messages are commands, where the delivered messages are the chosen commands. This correspondence is most obvious for the implementations of Section 3 based on a sequence of state machines. Conversely, we can consider a GC service to implement something like a state machine whose commands are of the form *send message m* and whose state is the sequence of delivered messages.

If the GC service does implement a real message-sending state machine with irrevocable message delivery, then using it to implement an arbitrary state machine is straightforward. Whether or not it does depends on exactly how it performs a view change. The property required of the GC view-changing operation was called Virtual Synchrony by Birman and Joseph [6]. As originally defined, Virtual Synchrony requires that the new view contain a majority of the old view's processes, and that the messages delivered to any process in that majority be declared to have been delivered in the old view. This requirement is not strong enough to ensure irrevocability, since a broker's computer will not be in the new view if it has been hit by an asteroid, so a message committing a stock sale need not be declared by the view change to have been delivered. (This has been observed before [23, 13].)

It is not hard to implement the stronger requirement that messages delivered by *any* process in the old view are declared to have been delivered. We need only require that before a process delivers a message, it learns that a majority of processes in the view know that the message was sent. A value proposed to the view-changing consensus algorithm as the old view's sequence of delivered messages must include all messages that some majority of processes knows to have been sent. With the stronger requirement, it is easy to make the GC service implement a state machine. Such stronger services do exist, including Isis's GBCAST protocol [6] and the "Safe" messages of Transis [9] and Totem [2]. A GBCAST/Safe message is delivered after all members of the view received it. These services have been used to provide consistent totally ordered broadcast by implementing a view-changing decision that forces delivery of any received GBCAST/Safe message [10, 13, 23]. Babaoglu et al. [4] take the direct approach of delaying delivery of a message until a majority of processes acknowledge having received it.

State-machine implementations seem quite different from GC. They use a reliable consensus algorithm to choose each command, while GC implementations use an unreliable consensus algorithm to broadcast a message and invoke a reliable consensus algorithm only on a view change. However, an "asynchronous" consensus algorithm executes a sequence of unreliable consensus algorithms—for example, the individual ballots of Paxos—stopping when a command is chosen. The unreliable consensus algorithm used by GC to broadcast a message can be viewed as just the first one of a reliable consensus algorithm, the rest of the consensus algorithm being performed simultaneously for all messages at the view change. This is effectively what happens in Paxos when a leader fails and a new ballot is begun simultaneously for all instances of the consensus algorithm. A state-machine implementation and a strong GC service perform essentially the same actions; those actions are just viewed differently. The state-machine view makes correctness obvious; the GC view makes the implementation more obvious.

All protocols that provide a strong GC service prevent any messages from being delivered after a view change has begun. They therefore essentially use the Brick-Wall method for stopping a state machine. Totem [2] chooses the number of the last message delivered in the old configuration and the new reconfiguration at the same time, a procedure analogous to option R1 of Section 3.2. CoRel [13] first selects a new configuration and only later determines which messages are delivered in the old configuration, which is analogous to option R2.

In traditional GC, the view-changing consensus algorithm is executed by the processes in the new view, while in $\mathcal{R}_\alpha$ and in the methods of Section 3, the new configuration is chosen with a consensus algorithm executed by the old configuration. (In $\mathcal{R}_\alpha$ and in option R1 of Section 3.2, that consensus algorithm is the one that chooses an ordinary state-machine command.)

Maintaining a unique sequence of configurations requires the new configuration to be chosen by the acceptors of the old configuration, but making progress requires it to be learned by the learners of the new configuration. Because the original work on GC did not distinguish between acceptors and learners, it ensured progress by having the processes in the new view execute the consensus algorithm and ensured uniqueness of views by requiring that the new view contain a majority of the processes from the old view. As our algorithms show, there is no need for any overlap between the old and new views. The processes in the new view just have to learn the new view chosen by the old one.

There are also reconfigurable systems that do not provide the power of a state-machine implementation. RAMBO [7, 11, 22] is a fault-tolerant system that provides simple read and write operations. It is based on the ABD algorithm [3], which plays the role that a consensus algorithm does in a state-machine implementation. In Rambo, reconfiguration is done using a separate state machine to choose a new configuration that can take effect in the middle of executing an individual read or write. This is similar to an alternative approach to reconfiguration in Paxos that we have not discussed [15]. Interestingly, it was shown recently that it is possible to construct reconfigurable atomic read/write objects without relying on a consensus for reconfiguration decisions [1].

5 Conclusions

In the state-machine approach, a fault-tolerant distributed system is built on top of a fault-tolerant state-machine implementation. The correctness properties of the system, including irrevocability, follow from the definition of the state machine and correctness of the state-machine implementation. A reconfigurable state-machine implementation provides an interface for choosing the configuration that executes the system. Section 2 reviewed the obviously correct method $\mathcal{R}_\alpha$ for making a state-machine implementation reconfigurable. Section 3 introduced other methods for doing this. Except for the Brick-Wall method, their correctness follows directly from the correctness of the underlying consensus algorithm.

We have considered only non-malicious faults. The state-machine implementation obtained with any of our algorithms except the Brick-Wall method (Section 3.1.3) tolerates malicious failures if the underlying consensus algorithms does. We must also design the state machine to tolerate malicious clients—including ones that propose reconfigurations. This is done by requiring that a critical operation such as stopping the current configuration or choosing a new one be performed only when requested by enough different clients. In the Stop-Sign method (Section 3.1.1), the *stop* command is the request that triggers stopping. For option R2 in which the new view is chosen by a separate consensus algorithm (Section 3.2), that algorithm is replaced by a fault-tolerant state machine.

References

[1] M. K. Aguilera, I. Keidar, D. Malkhi, and A. Shraer. Dynamic atomic storage without consensus. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, August 2009.

[2] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4):311–342, 1995.

[3] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, 1995.

[4] Ozalp Babaoglu, Renzo Davoli, and Alberto Montresor. Group communication in partitionable systems: Specification and algorithms. *Software Engineering*, 27(4):308–336, 2001.

[5] Kenneth Birman and Tommy Joseph. Exploiting virtual synchrony in distributed systems. In *Eleventh ACM Symposium on Operating Systems Principles*, pages 123–138, 1987.

[6] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[7] Gregory Chockler, Seth Gilbert, Vincent C. Gramoli, Peter M. Musial, and Alex A. Shvartsman. Reconfigurable distributed storage for dynamic networks. In *9th International Conference on Principles of Distributed Systems (OPODIS)*, 2005.

[8] Gregory Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4):427–469, 2001.

[9] Danny Dolev and Dalia Malki. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4):64–70, 1996.

[10] Alan Fekete, Nancy Lynch, and Alex Shvartsman. Specifying and using a partitionable group communication service. *ACM Transactions on Computer Systems*, 19(2):171–216, 2001.

[11] Seth Gilbert, Nancy A. Lynch, and Alex A. Shvartsman. RAMBO II: Rapidly reconfigurable atomic memory for dynamic networks. In *International Conference on Dependable Systems and Networks (DSN)*, 2003.

[12] M.P. Herlihy and J.M. Wing. Axioms for concurrent objects. In *Proceedings of the Fourteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 13–26, Munich, January 1987. ACM.

[13] Idit Keidar and Danny Dolev. Efficient message ordering in dynamic networks. In *Fifteenth ACM Symp. on Principles of Distributed Computing (PODC)*, pages 68–76, 1996.

[14] L. Lamport, D. Malkhi, and L. Zhou. Stoppable paxos. Technical report, Microsoft Research, April 2008.

[15] L. Lamport, D. Malkhi, and L. Zhou. Brief announcement: Vertical paxos and primary-backup replication. In *The ACM Symposium on Principles of Distributed Computing (PODC 2009)*, August 2009.

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

[17] Leslie Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems*, 6(2):254–280, April 1984.

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

[19] Leslie Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125, October 2006.

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

[21] J. R. Lorch, A. Adya, W. J. Bolosky, R. Chaiken, J. R. Douceur, and J. Howell. The smart way to migrate replicated stateful services. In *Proceedings of ACM Eurosys*, 2006.

[22] Nancy A. Lynch and Alex A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *5th International Symposium on Distributed Computing (DISC)*, 2002.

[23] Louise E. Moser, Yair Amir, P. Michael Melliar-Smith, and Deborah A. Agarwal. Extended virtual synchrony. In *The 14th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 56–65, 1994.

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

[25] J. Wensley et al. SIFT: Design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10):1240–1254, October 1978.