Experience with Rules-Based Programming for Distributed, Concurrent, Fault-Tolerant Code

Ryan Stutsman, *University of Utah;* Collin Lee and John Ousterhout, *Stanford University*

https://www.usenix.org/conference/atc15/technical-session/presentation/stutsman

This paper is included in the Proceedings of the
2015 USENIX Annual Technical Conference (USENIC ATC '15).

July 8–10, 2015 • Santa Clara, CA, USA

ISBN 978-1-931971-225

**Open access to the Proceedings of the
2015 USENIX Annual Technical Conference
(USENIX ATC '15) is sponsored by USENIX.**

Experience with Rules-Based Programming for Distributed, Concurrent, Fault-Tolerant Code

Ryan Stutsman[1], Collin Lee[2], and John Ousterhout[2]
University of Utah[1], Stanford University[2]

Abstract

This paper describes how a rules-based approach allowed us to solve a broad class of challenging distributed system problems in the RAMCloud storage system. In the rules-based approach, behavior is described with small sections of code that trigger independently based on system state; this provides a clean separation between the deterministic and nondeterministic parts of an algorithm. To simplify the implementation of rules-based modules, we developed a task abstraction for information hiding and complexity management, pools for grouping tasks and minimizing the cost of rule evaluation, and a polling-based asynchronous RPC system. The rules-based approach is a special case of an event-based state machine, but it encourages a cleaner factoring of code.

1 Introduction

Over the last decade more and more systems programmers have begun working on new and challenging software subsystems that manage distributed resources in a concurrent and fault-tolerant fashion. We call these subsystems *DCFT modules* (Distributed, Concurrent, Fault-Tolerant). DCFT modules are most common in systems that provide infrastructure for large-scale applications, such as Bigtable [8], Chubby [6], Hadoop [2], HDFS [25], RAMCloud [22], Sparrow [23], and ZooKeeper [15]. A DCFT module typically manages a collection of distributed servers, such as workers in a MapReduce application or replicas for a chunk of data; it issues remote requests in parallel to maximize performance, and it recovers from failures so that higher layers of software need not deal with them.

DCFT code is different from most systems code because it must describe behavior that is highly nondeterministic. As a result, DCFT modules are painfully difficult to implement. For example, the Chubby developers reported:

> Fault-tolerant algorithms are notoriously hard to express correctly, even as pseudo-code. This problem is worse when the code for such an algorithm is intermingled with all the other code that goes into building a complete system. [7]

The current state of development for DCFT modules resembles the situation in the mid-1960s for synchronizing concurrent processes. In both cases, a new and challenging style of programming was becoming more common; there were no widely accepted design patterns for implementing these modules, so each team developed its own set of ad hoc implementation techniques. In the case of synchronization, many different approaches were tried over a period of more than a decade, and there was considerable discussion about which approach was best. By the early 1980s the systems community had mostly settled on locks and condition variables, and this approach has been the dominant one for managing small-scale concurrency over the last three decades. We hope that this paper will provide useful data to fuel the discussion of DCFT modules, and that agreement will eventually emerge that makes it easier to implement these challenging systems.

This paper describes our experiences implementing several DCFT modules in the RAMCloud storage system [22, 24]. After struggling with our first implementations, we noticed that each of the DCFT modules ended up organized around a collection of rules. In this *rules-based approach*, the behavior of a module is described with a small set of code snippets that trigger independently based on the module's state. The order of execution is not determined a priori, but rather by the evolution of the module's state in response to events in the distributed system.

Since discovering this commonality, we have used the rules-based approach explicitly in more recent DCFT modules; these modules have been considerably easier to develop than the early DCFT modules. Rules provide a clean mechanism for expressing the nondeterminism of a DCFT module while allowing the vast majority of code to be written in a traditional imperative style.

This paper makes three contributions. First and foremost, it provides the first in-depth discussion of how to implement DCFT modules in a practical large-scale system. The paper introduces two of the DCFT modules in RAMCloud, describes why they were hard to implement, and discusses the design choices we made for RAMCloud along with their implications. We believe that the problems and solutions for RAMCloud are general enough to be relevant for a variety of other systems.

Second, the paper describes the implications of a rules-based approach on system structure. We found several abstractions useful in structuring rules and implementing efficient rules-based subsystems:

- A *task* structure combines a set of related rules with a collection of state variables. Each task uses its rules and state variables to achieve a particular goal, such as replicating an object. Tasks make it easier to manage rules and understand their behavior.
- A *pool* is a simple scheduler that improves the efficiency of rule evaluation. Each pool manages a collection of related tasks; it separates inactive tasks (those that are in their goal state) from active tasks and evaluates rules only for the active tasks.
- A polling-based asynchronous mechanism for remote procedure calls (RPCs) provides an efficient and convenient way to incorporate remote communication into a rules-based module. Asynchronous RPCs provide a better factoring than messages because they allow many error conditions to be handled entirely within the RPC system.

These facilities allowed rules to be incorporated simply and naturally into the RAMCloud system.

The paper's third contribution is to demonstrate the value of the rules-based approach. Rules allowed us to solve a wide range of problems in RAMCloud using a small amount of code (only 30-300 lines of rules-based code for each DCFT module). Rules are also efficient: when used in the critical path of RAMCloud's write operations, rules overheads account for only about 200-300 ns out of the total write time of 13.5 µs. The rules-based approach is a specialized form of an event-driven state machine, but it results in cleaner factoring and simpler code than the traditional approach to state machines. We reimplemented the scheduler for Hadoop MapReduce (which uses the traditional approach) using rules; our rules-based implementation replaced 163 state transitions with only 19 rules.

2 DCFT modules

A DCFT module is a piece of code that runs on a single machine but coordinates a collection of distributed resources. For example, the resources might be a group of worker machines, each of which will process a subset of the data in a scalable computation. Or, the resources might be storage servers, out of which a subset will be chosen to store replicas for a chunk of data. In many cases the management complexity is concentrated in a DCFT module on a single machine. The other machines are simply slaves that respond to requests; the slaves are simple enough that they do not require the DCFT approaches discussed in this paper. In other cases, such as consensus protocols, each machine runs an independent DCFT module.

Most of the work of a DCFT module involves communicating with other machines, and this introduces two challenges. First, communication is expensive enough that DCFT modules typically issue concurrent requests to improve performance. Second, distributed resources may fail. For example, a worker may crash before completing its computation, or a storage server may crash and lose all of its replicas. A DCFT module must detect failures and take recovery actions such as restarting a computation on a different worker or creating new replicas to replace the lost ones. Ideally, the complexities of distribution, concurrency, and fault tolerance are encapsulated within the DCFT module, so that it provides a simple and fault-free API for its clients.

The rest of this section describes two DCFT modules from the RAMCloud storage system, which will be used as examples in the remainder of the paper. RAMCloud contains several other DCFT modules besides these two; they are described in Table 2 in Section 6.

2.1 Membership notifier

RAMCloud's cluster membership notifier is a relatively simple DCFT module. Each server in a RAMCloud cluster needs to know about all of the other servers currently in the cluster. A special server called the *cluster coordinator* maintains the master copy of cluster membership information, called the *server list*, and it must notify all of the other servers whenever a server enters or leaves the cluster. The membership notifier runs on the coordinator and is responsible for propagating server list changes to the rest of the cluster.

The notifier uses RPCs to send updates to other servers. In order to update the cluster quickly, it sends updates to multiple servers concurrently. Additional server list updates may occur while the notifier is working; when this happens, the notifier batches multiple updates in future RPCs in order to minimize the total number of RPCs. The notifier must ensure that each server eventually receives all updates, and that all servers observe server list changes in the same order.

The membership notifier must handle a variety of faults. For example, a server may crash while a notification RPC to it is underway. If some servers are slow to respond to RPCs, this must not prevent other servers from receiving timely updates. Temporary network outages may cause update RPCs to fail; these RPCs must be retried.

2.2 Replica manager

The most complex DCFT module in RAMCloud, and the one that motivated much of our thinking about DCFT code, is the replica manager. The replica manager handles log replication for storage servers. Each RAMCloud storage server, called a *master*, organizes its DRAM as an append-only log of data, which is divided into tens

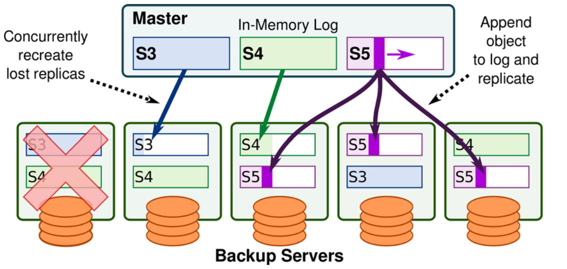
Master
In-Memory Log
S3
S4
S5
Concurrently recreate lost replicas
Append object to log and replicate
S3
S4
S3
S4
S4
S5
S5
S3
S4
S5
Backup Servers

Figure 1: The replica manager is responsible for ensuring that each of a master's log segments is properly replicated on backups. In this example new data has recently been appended to the log head (S5), so it is being replicated. In addition, a backup has crashed, so the replica manager is replacing lost replicas for segments S3 and S4.

of thousands of 8 MB *segments* (see Figure 1). Each segment must be replicated on the secondary storage of several other servers, called *backups*. An independent replica manager module runs on each master; its job is to ensure that the segments on that master are properly replicated. When new data is appended to the head segment, the replica manager must update the replicas for that segment. If a backup server crashes, the replica manager must create replacements for any replicas stored on that server.

In addition to the requirements above, the replica manager must also enforce constraints between segments. For example, in order to ensure that the log head can be identified unambiguously during crash recovery, an initial header must be written to a new head segment before the previous head is closed by writing a footer to it, and the footer must be written before any data can be written to the new head segment. See [26] for details on these constraints.

The replica manager is under particularly stringent timing constraints, since it is on the critical path for basic write operations. A master cannot respond to a write request from a client until the new data has been fully replicated. In order to minimize write latency (currently about 13.5 µs end-to-end for small objects) the replica manager must issue update requests in parallel for all of the replicas of the head segment.

3 How we ended up with rules

We did not consciously choose a rules-based approach for RAMCloud's DCFT modules: the code gradually assumed this form as we added functionality over a series of refactorings. When we built the first DCFT modules in RAMCloud, such as the ones described in the previous section, we had no particular point of view on how to write such code, and we did not know that DCFT modules would require an unusual approach. Thus, we initially tried to write each module as a monolithic piece of code that solved a problem from start to finish using a traditional imperative approach. However, this approach broke down almost immediately because of nondeterminism caused by concurrency and faults. To handle the nondeterminism, the code disintegrated into fragments that needed to execute relatively independently. None of our DCFT modules has reached anywhere near complete functionality with an imperative implementation.

For example, in the replica manager, the disintegration was initially caused by the desire to replicate segments concurrently. Different replicas could be in different states and could progress at different rates, so it didn't make sense to manage the replicas with a deterministic global algorithm. The most natural approach was to treat each replica independently.

Fault tolerance caused additional disintegration of the code. Failures have the effect of undoing work that was previously completed, thereby requiring earlier steps to be redone. For example, in the replica manager, if a backup crashes while receiving a replica, the replica manager must redo the process of selecting a server to store the replica. Failures can occur at many points, and different failures may require different amounts of work to be redone. As a result, it isn't possible to code an algorithm from start to finish. It makes more sense to think about the algorithm in terms of steps that make incremental progress, such as selecting a backup server or transmitting the segment header to the server for a particular replica. The execution order of the steps is nondeterministic, based on concurrency and failures.

Given a large set of relatively independent code fragments, we faced the question of how to manage their execution. We considered a fine-grained threaded approach, but quickly rejected this possibility. The replica manager must manage thousands of segments, with several replicas per segment, so using a separate thread per replica, or even per segment, would have been too inefficient [27]. Furthermore, multi-threading would only have handled the code disintegration caused by concurrency; it would not have addressed the disintegration caused by fault tolerance. In addition, threads were not needed from a performance standpoint: most of the work of a DCFT module consists of managing RPCs to other servers, with only a few RPCs typically outstanding at a time.

We also considered a coarse-grained approach to threading like SEDA [27], where tasks pass through a series of stages with each stage served by one or a few threads. However, the inter-thread communication required for this would have been unacceptable given our requirements for low latency (for example, using a condition variable to wake a thread takes about 2 µs; RAMCloud servers process simple requests in about 1 µs).

Thus, we decided to manage all of the code fragments for each DCFT module in a single thread. This left the problem of deciding the order in which fragments should execute in the thread. Writing an intelligent dispatcher that always knew what to do next was infeasible; the order depended on nondeterministic events such as RPC completions and failures, and there were complex dependencies between fragments (for example, the footer cannot be replicated for one segment until the header has been replicated for the following segment). As a result, we decided to let the fragments schedule themselves. Each fragment has an associated *condition*, which tests state variables to determine when it is appropriate for the fragment to run. The DCFT module operates by repeatedly testing conditions and executing the fragments for the conditions that are satisfied. Although this may appear to be expensive, we developed a few simple techniques that make this approach efficient (see Section 5).

Over time, we noticed that all of our DCFT modules disintegrated in the same way and ended up with similar features. Furthermore, these features resembled rules-based programming. Since then we have adopted the rules-based approach for all new DCFT modules, and we have developed infrastructure in RAMCloud to make the rules-based approach efficient and easy to use. Section 4 describes the approach at a conceptual level, and Section 5 describes how we implemented it in RAMCloud, and the supporting infrastructure that we developed.

In retrospect, we realized that DCFT modules require a highly unconstrained execution order; structures that restrict the order are likely to cause problems. For example, our first implementation of the membership notifier ran synchronously: each time the coordinator modified its server list to indicate the entry or exit of a server, the rest of the cluster was notified before returning from the modification. During this time, the server list lock was held. However, if a server crashed during the notification process, deadlock could result; the notifier couldn't complete without knowing about the crashed server (otherwise it would keep attempting to update that server), and it couldn't mark the server crashed without acquiring the lock. The rules-based version of this module never performs synchronous operations, which allows it to adapt to server list changes. We now use deadlock as a "canary in the coal mine:" if a module experiences deadlock, it may be a sign that its execution order is overly constrained and that we need to convert it to the rules-based approach.

As another example, we considered using the C++ exception mechanism to handle errors, but it is too restrictive to handle all of the error cases. Specifically, an exception handler can only catch exceptions in the calls nested beneath it, but the steps in a DCFT module do not follow a sequential or nested pattern. For example, the replica manager must replace segment replicas that are lost when a backup crashes. If a lost replica is for an older segment that is closed and inactive, then no thread is replicating it, and there is nowhere to deliver an exception.

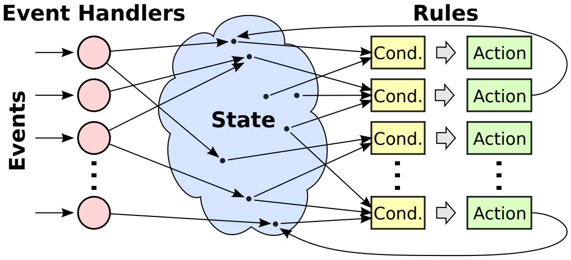

Figure 2: In a rules-based approach, code is divided into rules, each of which has a condition that tests state variables, and an action that is executed if the condition is satisfied. Rules can trigger in any order permitted by their conditions. The state is modified by handlers for external events and by the actions of some rules, which then allows new rules to trigger.

4 The rules-based approach

We use the term *rules-based* to describe a style of programming where there is not a crisp algorithm that progresses monotonically from beginning to end. Instead, the top-level controlling code of the module is divided into small chunks, called *actions*, which can potentially execute in any order (see Figure 2). Each action has an associated *condition* that determines when the action can execute; the condition is expressed as a predicate on the module's state variables. Together, an action and its associated condition constitute a *rule*.

A rules-based module operates by repeatedly selecting a rule whose condition is satisfied and then executing that rule's action. Each action makes incremental progress towards some *goal* (such as proper replication of a segment); the module executes rules repeatedly until it reaches the goal state. The goal is also described as a predicate on the module's state variables.

Actions can modify the state of the module or initiate external operations such as RPCs to other servers. Each action is nonblocking, and faults and external events have no effect on an action once it starts executing. If an action turns out to involve blocking or must handle nondeterminism due to faults, then it must be split into multiple actions in different rules. For example, an action cannot both initiate an RPC *and* wait for it to complete, since that would require the action to block and would expose it to nondeterministic failures of the RPC.

Nondeterminism manifests itself between actions, in the form of *events*. An event is an occurrence outside the direct control of the DCFT module that affects its behavior, such as:

Rules

Rule	Condition	Action
R1	No backup server selected.	Choose an available server to store replica.
R2	Header not committed, no RPC outstanding.	Start RPC containing segment header.
R3	Header RPC completed.	If backup rejected request, clear server assignment for replica. Otherwise, mark header committed and mark prior segment to allow footer replication.
R4	Uncommitted data, no RPC outstanding, prior footer is committed.	Start write RPC containing up to 1 MB of uncommitted data.
R5	RPC containing data completed.	Mark sent data as committed.
R6	Segment finalized, following header committed, footer not sent, no RPC outstanding.	Start RPC containing footer.
R7	Segment footer RPC completed.	Mark footer as committed and mark following segment to allow data replication.

Events

Event	What Happened	Handler
E1	RPC completed (or failed).	Update RPC object to indicate completion.
E2	New data added to segment.	Increment count of uncommitted bytes ready for replication.
E3	Backup server failed.	Cancel any RPCs outstanding to server. For all replicas stored on the failed server: cancel server selection; reset replica header, footer, and data to unsent and uncommitted.

Table 1: A partial list of the rules and events for managing one replica of a particular log segment. In the normal case, rules execute in order from R1 to R7 (R4 and R5 may trigger many times). Some rules test (R4 and R6) or modify (R3 and R7) state from multiple segments. If an RPC fails, no actions are taken other than to mark the state "no RPC outstanding"; the rules will automatically retry it. The handler for E1 is implemented by the RPC system.

- The completion of an RPC.
- The failure of a server.
- A new server joining the cluster (for the membership updater).
- The addition of new data to the head segment (for the replica manager).

When an event occurs, a handler updates state variables as shown in Figure 2; these state changes then allow new rules to trigger. For example, when an RPC completes, the RPC subsystem sets a state variable associated with the RPC.

The rules-based approach is similar in many ways to event-based programming. However, in event-based programming an event typically triggers actions directly. In the rules-based approach an event handler merely updates state variables; actions are then triggered based on the new state. In our experience, this two-step approach results in a cleaner factoring of code than the traditional event-based approach (see Section 8).

As an example, Table 1 shows some of the rules and events for managing a single segment replica in the replica manager described in Section 2. Rule R4 specifies the following predicate on a segment replica:

- some data appended to the segment has not been transmitted to the backup storing the replica, and
- no replication RPC is outstanding to the backup, and
- the preceding segment in the log has already committed its footer (so is safe to write to this replica).

If this condition is met, then the replica manager starts an RPC to send uncommitted data to the backup storing the replica. If the RPC completes successfully, a state variable is set, which allows R5 to execute. If a backup fails, event E3 executes: it cancels any RPCs outstanding to the failed backup, then iterates over the full list of segments in the log, resetting the replication state for any replica assigned to the failed backup. After the state is reset, recreation of the replicas happens automatically, just as it does during normal operation, starting with rule R1.

We found it natural to program with rules because they reflect the inherently nondeterministic structure of the problems being solved. Rules separate the deterministic parts of a module (actions) from the nondeterministic parts (events). Each action implements one of the basic steps of the module. In this problem domain it is difficult to describe all of the control flows from one step to another, so the rules-based approach does not even try. Instead, it describes the control flow in terms of the conditions that determine when each action executes, independently of how that state was reached. This results in a clean code factoring.

5 Implementing Rules

This section describes how we implemented the rules-based approach in RAMCloud, with emphasis on two issues: (a) achieving a clean code factoring, and (b) integrating rules-based DCFT modules cleanly and efficiently with the rest of RAMCloud. We introduced two new abstractions to manage rules: *tasks*, which provide modularity by associating a set of rules with a collection of state variables, and *pools*, which reduce the cost of rule evaluation by separating tasks into *active* and *inactive* groups. The rules-based approach requires an asynchronous communication mechanism; we chose to implement an asynchronous RPC system, which provides a cleaner factoring than a message-based approach. In addition to discussing these abstractions, this section also describes how events are handled and the role of threads in processing rules.

5.1 Tasks

The primary abstraction for implementing rules in RAMCloud is a *task*. Tasks provide modularity for rule sets, and they make it easy to use rules within a system mostly programmed in an imperative style. A task consists of three elements: a collection of state variables, a set of rules, and a goal. The collection of state variables is implemented as an instance of a class, and the rules are implemented by an `applyRules` method on the class. Each invocation of `applyRules` makes one pass over all the rules for that task, testing conditions and invoking actions for any conditions that are satisfied. In its simplest form, the body of `applyRules` consists of a sequence of `if` statements, one for each rule.

The goal of a task represents the outcome that the task is trying to achieve, such as ensuring proper replication of a single segment or updating all of the server lists in the cluster to reflect a change on the coordinator. A goal can be expressed as a predicate on the task's state variables. Goals are reminiscent of invariants, but we chose to use a different term because goals are unmet during much of the operation of a DCFT module, whereas invariants are almost always true.

A DCFT module contains one or more tasks. It operates by repeatedly calling the `applyRules` methods on its tasks until all tasks have achieved their goals. Events may cause a task to fall out of its goal state (for example, a server may crash, or new data may arrive that requires replication). If this happens, the DCFT module resumes processing rules until all tasks have once again reached their goal states. RAMCloud uses a polling approach, continually testing rules for tasks not in their goal state. Section 7 describes how to use rules in environments where sleeping is preferable to polling.

Many DCFT modules contain only a single task; the RAMCloud membership notifier is one example. Its state includes the coordinator's server list (including its version number), the version number of the server list stored on each server in the cluster, a list of recent updates, where each update mutates a server list from one version to the next, and a list of outstanding RPCs. The task contains three rules:

- **Condition**: there exists a server whose server list is out of date with respect to the coordinator's list, and for which there is currently no outstanding RPC.
  Action: initiate an RPC to that server, containing the updates not yet received by that server.
- **Condition**: there are updates that are no longer needed (they have been received by all servers in the cluster).
  Action: delete those updates.
- **Condition**: an outstanding RPC has completed.
  Action: if the RPC succeeded, update the version number stored for that server to reflect the updates it just received.

The goal of the membership notifier is to reach a state where the update list is empty. The notifier is implemented as a thread that repeatedly invokes the `applyRules` method until the goal is achieved, then sleeps until the coordinator's server list changes.

We try to structure our `applyRules` methods to make it easier for the programmer to reason about the overall behavior of the task. For example, we tend to order the rules in an `applyRules` method to match the order in which they will occur in the normal case without errors. This preserves enough ordering in the code for the developer to understand how the code is intended to progress.

For tasks with a complex state space, it can sometimes be difficult to ensure that the rules cover all possible states. In these cases, we write the task's rules using a set of nested `if` statements, each of which always has an unconditional `else` clause. This approach ensures that all possible states have been considered; exactly one action executes each time `applyRules` is called. Some of these cases may not contain code, which means that state is waiting for an event; the empty block serves as documentation that the state was considered.

5.2 Handling Events

In addition to invoking rules, a DCFT module must also handle events, which are occurrences outside the module that modify its state. Events are asynchronous with respect to the DCFT module, so they must synchronize with the DCFT module's rules engine in order to update state. In RAMCloud this is typically done with a traditional locking approach. For example, in the membership notifier, the only events other than RPC completions are modifications to the coordinator's server list, which also create new entries in the update list. A lock synchronizes these modifications with the execution of rules, and a condition variable is used to wake up the rules engine if it is sleeping when the server list is modified.

RPC completion events are handled specially using state variables; this mechanism is described in Section 5.5.

5.3 Threading

There are several ways we could have chosen to use threads in implementing rules. One possibility would be to employ threading on a fine-grain basis, with one thread for each task or perhaps even one thread for each rule; however, this is unnecessary and inefficient. For RAMCloud we chose a coarse-grain approach where each DCFT module executes in a single thread and evaluates its rules sequentially. There is little incentive to use multiple threads within a DCFT module because DCFT modules spend most of their time waiting for RPCs to complete. The concurrency of a DCFT module comes from concurrent execution of RPCs to other servers, not from concurrent execution of the module's internal code. If a DCFT module's functions include significant local processing then multiple threads might make sense for that module, and Section 7 describes how this can be implemented, but we have not yet encountered any modules where this is the case.

Using a single thread per DCFT module eliminates the need for most locks within a DCFT module, which makes the module both simpler and faster. For example, rules from one task can safely test and modify state variables from other tasks without synchronization (see rules R4 and R7 in Table 1). However, most DCFT modules must respond to some events generated outside the module, and locks are needed to synchronize these event handlers with the DCFT rules. This is typically implemented with a lock around each call to `applyRules`.

Different DCFT modules can execute concurrently in RAMCloud, since they use different threads.

5.4 Pools

Many of RAMCloud's DCFT modules are simple ones with only one task, but other modules have multiple tasks. For example, the replica manager uses one task for each segment stored on the server (typically tens of thousands). Evaluating all of the rules for all of these tasks is prohibitively expensive, so we introduced a *pool* abstraction to make rule evaluation efficient. A pool is a simple scheduler for a collection of related tasks. Pools reduce the overhead of rule application by dividing tasks into two groups: *active* tasks, whose rules must be evaluated, and *inactive* tasks, whose rules can be skipped. A task stays active until it achieves its goal, at which point it becomes inactive. Typically, only a small subset of tasks are active at a time, so testing rules is efficient. Over its life, a single task may be activated and deactivated many times, since failures and other state changes can return a task to a state where its goal is no longer met.

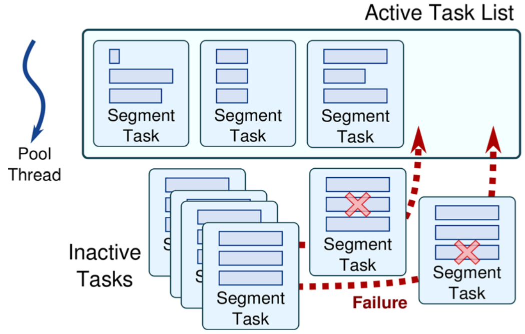

Figure 3: The replica manager's pool and its associated tasks. Each task manages the replication of a single segment. The pool thread continually evaluates rules for active tasks (those that have not yet met their replication goal). When a task meets its replication goal it becomes inactive and its rules are no longer evaluated. Server failures cause the state of some tasks to be reset, and the affected tasks are reactivated. Testing rules is efficient, since there is rarely more than one segment (the head) under active replication at a time.

Each pool is implemented as a thread associated with a list of active tasks. Whenever the list contains at least one task, the thread cycles through the tasks, invoking their `applyRules` methods. When the list is empty the thread sleeps.

For example, the replica manager contains a pool with one task for each segment; the task's goal is to maintain three complete replicas of the data in the segment. When new data is added to the head segment, the corresponding task is activated (see Figure 3). When the task catches up in replicating its data, its goal is reached, so it is deactivated. Once a segment is completely filled by the server and its task finishes replicating it, then its task is deactivated forever unless one of its replicas is lost due to a failure. In this case the task is reactivated to begin recreating the lost data. At any given time, most segments are fully replicated, so the replica manager pool usually only has to test rules for a few segment tasks at a time.

5.5 Asynchronous RPCs

The rules-based approach requires an asynchronous communication mechanism because actions that initiate remote requests cannot wait for completion (blocking would prevent other rules from firing; it would also expose actions to nondeterminism, since remote communication can fail). To meet this requirement, we implemented an asynchronous RPC mechanism based on polling. This section describes how the RAMCloud RPC system simplifies the implementation of rules, and why it results in a cleaner code factoring than alternatives such as callback-based RPCs or message-based programming.

In RAMCloud all RPCs are inherently asynchronous. Each RPC is represented with a C++ object; the constructor for the object forms the request message and initiates transmission of that message. The RPC object contains a state variable that can be tested to determine whether the RPC has completed. If the state variable indicates completion, another method may be invoked on the RPC object to retrieve results or failure information. Each RPC object also supports a synchronous `wait` method that polls the state variable until the RPC has finished.

This approach fits naturally with rules-based programming: DCFT modules keep RPC objects as part of their state, and the conditions for rules test the RPC objects for completion (see Table 1 for an example). An alternative approach for asynchronous RPCs is to invoke a callback function when an RPC completes. However, callbacks are awkward because they must synchronize their execution with rules that might be executing concurrently. The state variable provides a simpler form of synchronization between the completion of the RPC and the rules engine.

RAMCloud uses polling not only for asynchronous RPCs in DCFT rules engines, but also for synchronous RPCs invoked outside DCFT modules. Polling works well in RAMCloud because the expected completion time for RPCs is only a few microseconds. Blocking a thread to wait for an RPC serves little purpose: by the time the CPU could switch to another task, the RPC will probably have completed, and the polling approach eliminates the latency overhead for waking the blocked thread when the RPC completes.

An alternative to asynchronous RPCs would have been to use a messaging approach, with separate request and response messages. However, we found that RPCs produce a cleaner code factoring by allowing more functionality to be implemented transparently in the RPC mechanism; this simplifies the code in DCFT modules. Remote procedure calls automatically associate each request message with the corresponding response message. In a pure message-based approach, higher level software must make this association, which increases its complexity. Furthermore, the RPC approach allows some errors to be detected and handled transparently in the RPC system, whereas a message-based approach must expose these errors to higher-level software. RAMCloud's RPC system allows the creation of customized modules that recover automatically from many errors. For example, if a network connection fails, a recovery module will automatically open a new connection and retry; or, if an RPC fails with an error indicating that the target server no longer stores the desired object, a recovery module will automatically find the correct server and retry the request with that server. As a result, many of RAMCloud's RPCs return no errors except those caused by bad arguments: all system errors are handled internally by the RPC system. In a message-based approach, these problems must be handled by higher-level software.

6 Evaluation

This section discusses the strengths and weaknesses of rules, based on our experiences in RAMCloud.

6.1 Benefits

Thinking in terms of rules has allowed us to produce new DCFT modules more quickly, with fewer refactorings before reaching satisfactory solutions. Specifically:

- The task and pool abstractions simplify the development of DCFT modules. Tasks serve a purpose similar to that of monitors [19]: a monitor helps to modularize synchronization code by encapsulating a lock with a collection of state variables and a set of methods that manipulate those variables; a task helps to modularize DCFT code by encapsulating a collection of state variables with rules and events that manipulate those variables to achieve a goal. Both of these structures provide a framework that reduces the number of decisions a developer must make to produce a working module.
- The `applyRules` methods bring all of the rules for a task together in a few pages of code, making it easier to understand the task's behavior.
- It is relatively easy to add rules to an existing DCFT module when new issues are discovered.
- It is straightforward to integrate rules-based modules into the RAMCloud system. We use rules-based code surgically in only a few `applyRules` methods, while the vast majority of the system is programmed in a traditional imperative fashion.

Table 2 summarizes each of the seven DCFT modules in RAMCloud, with two overall conclusions. First, the table shows that the rules-based approach can be used to implement a variety of tasks, including different approaches to replication, coordinating workers executing in parallel, and crash recovery. Second, the rules-based approach allows the nondeterministic parts of the system to be concentrated in a small amount of code, so that the vast majority of the system can be written using a simpler imperative style. The `applyRules` methods in RAMCloud range in size from 30-300 lines, which is only a small fraction of the overall DCFT modules. All of the rules-based code in RAMCloud amounts to only about 1,100 lines, out of a total system size of more than 50,000 lines.

One potential problem with the rules-based approach is the cost of testing rule conditions, which happens repeatedly. We measured this cost for the RAMCloud replica manager, which is the most time-sensitive DCFT

DCFT Module	**Functionality**	**Task Types**	**Rules**	**Events**	`applyRules` **code**
Membership notifier	Notifies all servers of changes to coordinator's server list	1	3	3	36
Replica manager	Maintains a specified number of replicas of each segment on a master	1	23	3	258
Recovery manager	Executes on coordinator to recover crashed master: locates complete copy of log, splits the master's tablets, coordinates many masters to replay log and recover partitions	4	12	2	299
Recovery master replay	Executes on recovery masters during crash recovery: reads log segment replicas from backups, replays entries, replicates new data	1	3	0	230
Backup replica recovery	Executes on backups during crash recovery: reads segment replicas, divides log entries into buckets for different recovery masters	1	4	2	31
Multi-read	Executes on clients: reads many objects concurrently using batched requests to multiple servers	1	2	2	75
Indexed read	Executes on clients: retrieves index entries from one or more secondary index servers, then reads the corresponding objects from other servers	1	14	2	132

Table 2: Summary of DCFT modules in RAMCloud. "Task Types" counts the number of different kinds of task (not instances) in the module. "Rules" counts the total number of rules in all task types. "Events" counts only module-specific events (it excludes RPC completion events, which are handled automatically by the RPC subsystem). "`ApplyRules` code" counts lines of code (not including comments) in all `applyRules` methods. Some of the line counts include additional code not directly related to processing rules.

module in RAMCloud (it is on the critical path for all write operations). The replica manager also has the largest rule set of all the RAMCloud DCFT modules. As shown in Figure 4, only a few hundred nanoseconds are needed for evaluating conditions in each call to `applyRules`. When `applyRules` takes a significant amount of time to execute, it is because of actions that initiate RPCs and handle completions. Furthermore, only two invocations of `applyRules` are on the critical path for each write: the first (which issues replication RPCs) and the last (which receives the results from the last replication RPC). Based on Figure 4, we estimate that testing conditions accounts for 200-300 ns out of a total time of about 13.5 µs for writes.

6.2 Challenges

It is not always easy to identify modules that require the rules-based approach. The natural tendency is to code any new module in an imperative style (especially for programmers not already familiar with DCFT modules and rules), and it is easy to underestimate the implications of fault tolerance. Thus, we sometimes find ourselves attempting to implement new DCFT modules without rules. When this happens, corner cases result in refactorings that gradually break up the code flow, until eventually we realize that we need to switch to a rules-based approach. The introduction of rules usually simplifies the code, and seemingly intractable problems suddenly become tractable. For example, the introduction of rules in the membership notifier eliminated deadlocks that had plagued several previous versions of the code.

The greatest challenge in using rules is to get out of the traditional mental model where an algorithm is defined monolithically. Instead, the algorithm must be defined as a collection of independent small pieces, each of which makes incremental progress towards a goal. These pieces become the actions of rules, and conditions and event handlers are added to invoke the actions appropriately. Our experience is that once a developer adopts this mental model, the actual rule set follows fairly quickly, and it is straightforward to incorporate the rules into the overall system.

It can be difficult to visualize the behavior of a DCFT module from a collection of rules. However, we think this problem is inevitable, given the nondeterminism that the rules must capture: nondeterministic solutions will always be harder to understand than deterministic ones.

We do not advocate the use of rules as an overall architecture for applications. Asynchronous nondeterministic programming is fundamentally more difficult than traditional imperative programming, so it should only be used where it is absolutely necessary.

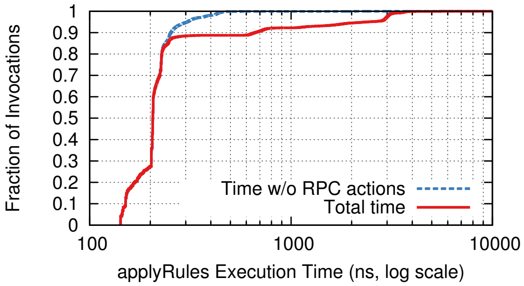

Figure 4: Cumulative distribution of the execution time of the `applyRules` method for the RAMCloud replica manager, measured using YCSB Workload A [9] to generate writes of 1000-byte objects. "Total time" includes the cost of actions as well as condition checks. "Time w/o RPC actions" excludes time spent in actions that initiate RPCs and process RPC results. Most invocations of `applyRules` occur while waiting for RPCs to complete; these invocations test conditions but no actions fire.

7 Rules Without RAMCloud

Although most of our experience with rules is in the context of RAMCloud, we believe that the rules-based approach also makes sense for other applications. This section discusses modifications of the rules approach that may be appropriate in environments other than RAMCloud.

A polling approach to rules evaluation makes sense in RAMCloud's low-latency environment, but in applications with high communication latency it makes more sense for the rules engine to sleep while waiting for events. The rules approach can accommodate sleeping with two modifications. First, the rules engine must be able to determine when it is safe to sleep. To do this, `applyRules` methods must return an indication of whether any rules triggered. If no rules triggered, then no state changes were made, so no rules will trigger in the future until an event occurs. In this situation, the pool can deactivate the task just as if it had reached its goal state. In fact, with this mechanism for sleeping, no special handling is needed for goal states: the mechanism for sleeping will handle them automatically. Once all of the tasks in a pool are inactive, the pool can sleep. The second modification for sleeping is that the rules engine must wake up again when necessary. To implement this, each event must be associated with a task; when the event occurs, the task is reactivated and its pool is awakened.

Most of the rest of the rules mechanism still applies, even in environments without polling. For example, the task abstraction still makes sense as a way of encapsulating a rule set with its state variables. Pools are still useful, both for minimizing the number of rules that must be tested and also as the mechanism for sleeping. Asynchronous RPCs retain their advantages over messages or synchronous RPCs, as described in Section 5.5.

One additional modification that may be appropriate in some environments is to relax RAMCloud's one-thread-per-DCFT-module restriction. If some actions involve significant local processing, then it may be desirable to allow other rules to execute concurrently with them. Concurrency can be implemented using an approach similar to that for asynchronous RPCs: the action dispatches its work to a separate worker thread and then returns, so that the rules engine can process other rules while the worker thread executes. When the worker thread completes, it sets a state variable just like an RPC completion, which can then cause other rules to trigger. If the worker thread needs to access state variables during its execution, then it must synchronize with the rules engine.

In summary, most of RAMCloud's rule mechanism carries over directly to other environments, and with a few small changes the mechanism can handle issues we have not yet experienced in RAMCloud, such as high-latency communication and long-running actions.

8 Event-driven state machines

The rules-based approach emerged so consistently in all of our DCFT modules that we initially thought it might be inevitable. However, we have since discovered that other systems use different approaches for DCFT modules. The most common alternative appears to be an event-driven state machine; Chubby [6] and Hadoop [2] are examples of this approach. In this section we compare the rules-based approach to this alternative, and we argue that the rules-based approach produces cleaner and simpler code.

An event-driven state machine is a system with one or more state variables, whose behavior is determined by events. When an event occurs, the state machine takes actions based on the current state and the event. The actions can alter the state, which affects the way that future events are handled.

The state machine definition is broad enough that it includes the rules-based approach as a special case. However, in most state machines the actions are determined directly from the events. Rules use a two-step approach where event handlers only modify state and never take actions. Rules then trigger based on the resulting state, which reflects the sum of the outside events.

The difference between these two approaches is subtle, but the rules-based approach results in a cleaner code factoring. In DCFT modules, the current state of the system is more important than how the system got to that state, so it is cleaner to structure code around state, not events. A single event may need to trigger many actions, and the same action might be triggered from multiple events. For example, the replica manager may

State Machine	States	Transitions	Distinct
Job	14	82	27
Task	7	24	16
TaskAttempt	13	57	15
Total	**34**	**163**	**58**

Table 3: Hadoop MapReduce 2.4 manages task scheduling using 3 state machines. For each state machine the table lists the number of explicitly named states, the total number of transitions between states, and the number of distinct actions among all the transitions for the state machine.

need to choose a backup for a particular replica either because a new segment is being created, or because an existing replica was lost in a crash. The traditional state machine approach results in considerable duplication of code, which is not present in the rules-based approach.

To demonstrate the advantages of the rules-based approach, we analyzed the job scheduler in Hadoop MapReduce 2.4, which uses the state machine approach described above. The overall goal of the job scheduler is to schedule a collection of tasks across a group of servers. The module manages a group of objects, with each object controlled at any given time by one of three state machines (see Table 3). In total, the three state machines contain 34 states, with 163 separately-defined *transitions*, where a transition describes the actions to take when a particular event occurs in a particular state.

Of the 163 transitions, only 58 have distinct actions: the other 105 transitions are duplicates. Furthermore, upon analysis of the actions, we found that many of the "distinct" transitions are near-duplicates. For example, rather than writing one error cleanup action that works across many states, MapReduce contains numerous nearly-identical cleanup actions, each specialized slightly for the state and event that trigger it.

For comparison, we reimplemented the MapReduce job scheduler using a rules-based approach, with each state machine replaced by one task. We used Python for the rules-based implementation because of its rapid-prototyping capabilities and verified by hand that each of the 163 transitions in the Java state machines is covered by the rules-based implementation. Our Python implementation is complete enough to schedule and run simple jobs. The source code for the Python implementation is available on GitHub along with the corresponding code for the Java state machine [3].

The rules-based implementation of the MapReduce scheduler is significantly simpler than the state machine implementation: a total of 19 rules in 3 tasks provided functionality equivalent to the 163 transitions in the state implementation. Each of the three `applyRules` methods fits in a screen or two of code (117 total lines of code and comments between the three `applyRules` methods), which makes it possible to view the entire behavior of each task at once. Furthermore, the order of the rules within each `applyRules` method shows the normal order of processing, which also helps visualization. In contrast, the state machine implementation required more than 750 lines of code just to specify the three transition tables, plus another 1,500 lines of code for the transition handlers.

Transition handler counts, rule counts, and lines of code are metrics that are easy to compare, but other metrics may provide more insight. For example, more elaborate code complexity metrics or a full user study could help highlight the differences between the approaches.

9 Other Related Work

Several formalisms exist for specifying and reasoning about concurrent code. For example, Dijkstra's Guarded Command Language (GCL) [11] provides nondeterministic conditional and loop constructs similar to the iterative conditional checks used in rules-based tasks. Hoare's Communicating Sequential Processes (CSP) [14] extends GCL to support specification of and reasoning about interconnected nondeterministic processes. GCL and CSP have been influential in the design of concurrency primitives of programming languages like the recent Go and Rust systems languages. The occam [5] programming language adheres to CSP even more closely; programs in occam tend to follow a rules-based style.

Lamport's Temporal Logic of Actions and his TLA+ specification language [18, 28] allow specification of concurrent systems. TLA+ supports a model checker and proof system. However, it is not a full programming language and cannot be used for implementation. Similar to our rules, TLA+ specifications use conditional atomic actions to transition from one state to the next; this may make it a good fit for verifying rules-based modules.

Rules are also used in other application domains to solve problems other than DCFT:

- Expert Systems [16] and the General Problem Solver [21, 20] are AI programs that reason using heuristic methods; they are implemented as a set of rules that iteratively transform a knowledge base to arrive at a solution.
- Make [12] is a utility that builds software based on user-provided rules. Each rule specifies how to rebuild a particular file, if the current version is out of date.
- Model Checkers are formal verification tools that, like rules-based modules, iteratively apply conditional transformations to state variables to ensure user-specified invariants are preserved.

Actors, originally conceived in the 1970s, have become popular in recent years for building distributed or concurrent applications [13, 1]. In the actors approach, a program is divided into independent modules with no shared state, called *actors*, which communicate using asynchronous messages. Actors often handle messages using an approach like that described in Section 8 for event-driven state machines (each actor is a state machine, and messages represent events). However, actors could also use a rules-based approach, and this would be advantageous for actors that implement DCFT modules. We also believe that an asynchronous RPC system would provide a better communication mechanism for DCFT actors than asynchronous messages, as discussed in Section 5.5.

Several groups have developed domain-specific languages and frameworks for specifying DCFT modules as a collection of event-driven state machines:

- Chubby [7] is a consensus-based configuration service that uses a custom state machine specification language to simplify the specification of its core algorithm.
- Mace [17] provides a restricted language for specifying state machines that allows specifications to be verified using a provided model checker. Unlike the rules-based approach, Mace programs only perform actions in response to events, though Mace also provides a construct that generates events in response to conditions on state.
- P [10] is a graphical language for specifying state machines that was used to implement the USB device driver stack of Microsoft Windows 8. P also allows model checking of state machines.
- SEDA [27] is an event-driven framework for building highly concurrent services that avoids the overhead of the thread-per-request approach. SEDA services are actor-like; they consist of pipelined stages interconnected by message queues. Each stage can optimize throughput by adjusting how many threads it uses. SEDA doesn't address fault-tolerance, and it increases response latency, making it inappropriate for RAMCloud.
- Bloom [4] is a language for building distributed systems in which programs are expressed using declarative rules over unordered sets of tuples. A key benefit of Bloom's "disorderly" approach is that it avoids artificial coordination compared to traditional imperative programs. Conditions on rules have something of a declarative style in rules-based code, but actions are programmed in an imperative style.

10 Conclusion

DCFT modules are becoming increasingly important in large-scale software systems, but they are difficult to implement and developers today have little guidance on how to implement them. In this paper we have described the problems we faced while implementing DCFT modules in RAMCloud and the rules-based solution that emerged from our experience. The rules-based approach results in a simple code factoring because it separates the deterministic and nondeterministic parts of a DCFT module. In addition, we found that a few other patterns and mechanisms encouraged a clean factoring of rules-based code, including tasks, pools, and an asynchronous RPC system. With this infrastructure, it was relatively easy to incorporate rules-based code into the RAMCloud system, and the rules-based approach has provided clean solutions to a variety of problems. In comparison to other approaches we considered, the rules-based approach encourages a cleaner factoring of code, which is particularly important given the inherent complexity of DCFT modules.

Experience with many more systems will be needed before agreement can be reached on the best way to implement DCFT modules. We hope that our experience can serve as a basis for additional discussion and experimentation.

11 Acknowledgments

Many people provided helpful feedback on this paper, including Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Eric Eide, Diego Ongaro, Henry Qin, David Silver, numerous anonymous conference reviewers, and our shepherd Ajay Gulati. This work was supported by STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and by grants from Facebook, Google, Huawei, Mellanox, NEC, NetApp, Samsung, and VMWare.

References

[1] Akka, 2014. http://akka.io/.

[2] Welcome to Apache Hadoop!, 2014. http://hadoop.apache.org/.

[3] PlatformLab/mappy Git Repository, May 2015. https://github.com/PlatformLab/mappy.git.

[4] P. Alvaro, N. Conway, J. Hellerstein, and W. R. Marczak. Consistency Analysis in Bloom: a CALM and Collected Approach. In *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*, pages 249–260, 2011.

[5] A. Burns. *Programming in Occam 2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[6] M. Burrows. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[7] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos Made Live: An Engineering Perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM.

[8] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 205–218, Berkeley, CA, USA, 2006. USENIX Association.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[10] A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferey. P: Safe Asynchronous Event-driven Programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 321–332, New York, NY, USA, 2013. ACM.

[11] E. W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM*, 18(8):453–457, Aug. 1975.

[12] S. I. Feldman. Make — A Program for Maintaining Computer Programs. *Software: Practice and Experience*, 9(4):255–265, 1979.

[13] P. Haller and M. Odersky. Event-Based Programming Without Inversion of Control. In D. Lightfoot and C. Szyperski, editors, *Modular Programming Languages*, volume 4228 of *Lecture Notes in Computer Science*, pages 4–22. Springer Berlin Heidelberg, 2006.

[14] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, Aug. 1978.

[15] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Annual Technical Conference*, USENIX ATC '10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[16] P. Jackson. *Introduction to Expert Systems*. Addison-Wesley Longman Publishing Co., Inc., 1990.

[17] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: Language Support for Building Distributed Systems. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 179–188, New York, NY, USA, 2007. ACM.

[18] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, 1994.

[19] B. W. Lampson and D. D. Redell. Experience with Processes and Monitors in Mesa. *Communications of the ACM*, 23(2):105–117, Feb. 1980.

[20] A. Newell, J. C. Shaw, and H. A. Simon. Report on a General Problem-solving Program. In *IFIP Congress*, pages 256–264, 1959.

[21] A. Newell and H. Simon. GPS, A Program That Simulates Human Thought. In *Computers & thought*, pages 279–293. 1995.

[22] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 29–41, New York, NY, USA, 2011. ACM.

[23] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, Low Latency Scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 69–84, New York, NY, USA, 2013. ACM.

[24] S. M. Rumble, A. Kejriwal, and J. Ousterhout. Log-structured Memory for DRAM-based Storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, pages 1–16, Berkeley, CA, USA, 2014. USENIX Association.

[25] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[26] R. S. Stutsman. *Durability and Crash Recovery in Distributed In-Memory Storage Systems*. PhD thesis, Stanford, CA, USA, 2013.

[27] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 230–243, New York, NY, USA, 2001. ACM.

[28] Y. Yu, P. Manolios, and L. Lamport. Model checking TLA+ specifications. In *Correct Hardware Design and Verification Methods*, pages 54–66. Springer, 1999.