Toward Common Patterns for Distributed, Concurrent, Fault-Tolerant Code

Ryan Stutsman and John Ousterhout
Stanford University
{*stutsman, ouster*}*@cs.stanford.edu*

Abstract

There are no widely accepted design patterns for writing distributed, concurrent, fault-tolerant code. Each programmer develops her own techniques for writing this type of complex software. The use of a common pattern for fault-tolerant programming has the potential to produce correct code more quickly and increase shared understanding between developers.

We describe *rules*, *tasks*, and *pools*, patterns extracted from the development of RAMCloud, a fault-tolerant datacenter storage system. We illustrate their application and discuss their relationship to concurrent programming models. Our goal is to generate discussion that will ultimately lead to common techniques for fault-tolerant programming.

1 Introduction

As datacenters and large-scale applications have become prevalent, more programmers are developing code modules that must be distributed, concurrent, and fault-tolerant (DCFT). These systems must manage hundreds or thousands of machines while retaining correctness, consistency, and availability in the face of frequent and unpredictable failures. This type of programming is notoriously difficult to get right [2]. Failures can occur at the most inopportune times; they may even occur in the middle of handling other failures.

In developing RAMCloud [7], we struggled to express DCFT code in several subsystems, such as

- replicating a distributed log and restoring durability when replicas are lost due to server failures;
- coordinating the recovery of the contents of the DRAM of a failed server;
- disseminating cluster membership information in a consistent way to failure-prone servers;
- collecting recovery data from distributed logs, detecting inconsistencies, and replaying data;
- and reclaiming distributed storage and resources.

There are no widely accepted design patterns for implementing DCFT systems. Each programmer develops his own set of ad hoc implementation techniques.

Much of the prior discussion on structuring concurrent servers has focused on threads [10, 11] and events [3, 5, 12, 13], but neither model on its own gives insight on how to structure code for fault-tolerance. For example, the traditional serial programming model using threads [1] does not work for these types of systems. In a large scale system, the state of the cluster is constantly changing. Programs must react to failures quickly, and, as a result, control flow in fault-tolerant modules must be able to adapt radically to unpredictable events. A simple serial programming style presumes a sequence of steps in achieving a goal, but faults break that assumption.

Our hope is that our reflections on developing RAMCloud will generate discussion and lead to common techniques for developing distributed, concurrent, fault-tolerant systems. A common, formulaic approach would allow developers to produce software quickly and correctly and would benefit from a shared understanding between developers.

As independent developers have implemented RAMCloud subsystems, a pattern has emerged. Fault-tolerant code, by nature, must be composed of short blocks that may have a desired order, but that can be quickly redirected based on outside events. Consequently, these modules are structured as a set of conditional *rules* iteratively applied to state. Conceptually, rules specify small steps in achieving some goal or attaining some invariant. At each iteration, rules are selected for execution based solely on state and not on any static ordering. As a result, rules-based code can react to outside events easily with simple state changes which, in turn, determine future actions.

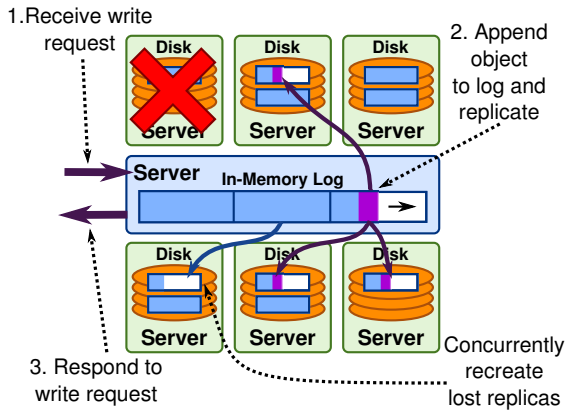

Figure 1: Servers execute client write requests by appending object data to an in-memory log. Log data is replicated and scattered across the cluster in units of segments. When a server fails (upper left), some replicas are lost. The log module overlaps recreation of lost replicas with normal replication RPCs (lower left).

Rules are a basic construct; the challenge is in determining the best way to structure programs comprised of hundreds of rules on thousands of objects. We group independent sets of rules into *tasks* which provide modularity. Tasks are organized into *pools* to ease synchronization and to make the application of rules efficient.

In extracting a simple pattern for distributed, concurrent, fault-tolerant code from our experiences, the "productive ignorance" of the monitor pattern has served as our inspiration. Using a simple structure, monitors [4, 6] eliminate programmer decisions about concurrency, which produces correct code faster. When writing a monitor, a programmer need only decide which methods should be synchronized. Details about which fields should be protected by a mutex, how fine-grained mutexes should be, and what locking patterns to use are all prescribed. We believe a similarly prescriptive approach to DCFT programming will result in better code and higher developer productivity.

We illustrate rules, tasks, and pools through an example taken from RAMCloud's data replication module, discuss the trade-offs of our approach, and discuss its relationship to existing concurrent programming models.

2 Log Replication in RAMCloud

RAMCloud is a fault-tolerant datacenter storage system that runs on clusters of thousands of machines. During normal operation, each RAMCloud server stores objects by appending them to a log that is replicated across other servers in the cluster (Figure 1). Each server's log is divided into fixed-size *segments*. Each segment is replicated on a different set of servers. When a server fails, some segment replicas are lost. The log manager on each server reacts to server failure messages from the cluster coordinator and creates new replicas for segments that were affected by a server failure. The log manager typifies many subsystems in RAMCloud.

- **It is highly distributed.** Each server's log is comprised of tens of thousands of segment replicas scattered across thousands of machines. Whenever any server fails all servers' logs are affected.

- **It is highly concurrent for performance.** Client observed end-to-end latency for a triplicated 100 byte write operation in RAMCloud is 15 µs. Replication RPCs must be overlapped to synchronously replicate data with the lowest possible latency. Additionally, on server failure, lost replicas must be recreated elsewhere in the cluster in parallel with normal operation and with each other. Recreating replicas uses the full output bandwidth of our machines: 25 Gbps.

- **It must be fault-tolerant.** Server failures can occur at any time and can make ongoing replication operations impossible or unsafe to complete. The log manager must quickly handle failures that affect segments that are fully replicated, under active replication, or already under repair due to prior failures.

Under these requirements, traditional serial programming with threads is difficult because handling server failure notifications requires unpredictable changes to execution order. For example, when a server failure notification is received by a server, several of its segments are affected, and each affected segment replica can be in a different state. Some replicas may have an RPC outstanding to the failed server and must abort the RPC and restart replication elsewhere instead of expecting a response. Other affected replicas may not be consistent with their counterparts; such replicas require contacting the cluster coordinator before starting recreation in order to prevent inconsistencies. If each replica of each segment were managed by a separate thread they would have to be interrupted to abort and redirect operations. In general, faults require the flow of execution to change in radical and unpredictable ways.

3 Rules, Tasks, and Pools

When we started the RAMCloud project we had no particular strategy for implementing DCFT code. We also had no idea how many different subsystems would require this type of code. Over time, several different

Segment Replication Task

Rule	Condition	Action
R1	No backup server selected.	Choose an available server on which to create replica.
R2	Header not committed, no RPC outstanding.	Start RPC containing the header.
R3	Header not committed, RPC completed.	Mark header committed; 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	Uncommitted data, RPC completed.	Mark sent data as committed.
R6	Segment finalized, following header committed, footer not sent, no RPC outstanding.	Start RPC containing the footer.
R7	Segment finalized, RPC completed.	Mark footer as committed; mark following segment to allow data replication.

Server Failure Task

Rule	Condition	Action
F1	True	For all replicas using the failed server: deselect server; reset replica header, footer, and data to unsent and uncommitted.

Figure 2: Rules for managing one replica of a particular log segment. Server failures are handled with the same rules as normal operation. Not all rules are isolated to using the state for a single segment; some rules test (R4 and R6) or modify (R3 and R7) state from multiple segments.

developers implemented many such modules independently. Although the implementations were different in many respects, we eventually noticed a common theme; each of these modules contained a set of rules that could trigger in any order. We gradually developed a pattern for DCFT code based on three layers: *rules*, *tasks*, and *pools*. This particular pattern has worked for a variety of problems in RAMCloud. We believe that this pattern, or something like it, might provide a convenient way of structuring DCFT modules in general.

A *rule* describes an *action* to be taken when a *condition* is met. An action is a block of code. A condition is predicate on state variables. Rules-based code has two interesting properties. First, actions are small and non-blocking so that control flow within an action is predictable; faults need not and do not affect the course of an executing action. Typical actions start operations such as asynchronous RPCs, check for the completion of operations, and update state. Second, the execution order of rules is unpredictable; changes to the state determine the order in which rules execute. As a result, execution adapts automatically in the face of concurrency and faults. Major changes in control flow happen between rules, not within an action.

Figure 2 shows the log manager's rules for creating segment replicas. As an example, rule R4 specifies the following predicate on a segment replica:

- some data appended to the segment has not been committed on the server storing the replica, and
- no replication RPC is outstanding to the server, and
- the preceding segment in the log has already committed its footer.

If this condition is met, then the log manager starts a replication RPC with the newly appended data to the server storing the replica. If the RPC completes successfully without intervening server failures, then rule R5 will eventually execute. If the target server fails, then replication will be redirected by rule F1 (bottom of Figure 2). When F1 is executed, it iterates over the full list of segments in the log. The rule resets the replication state for any replica stored on or in the process of being replicated to the failed server. Any RPCs outstanding to the failed server are canceled. After the state is reset, recreation of the replica happens automatically, just as it does during normal operation restarting with rule R1.

Rules address the unpredictability of fault handling, but a full system may consist of many rules and state records; how rules are organized and applied affects the complexity and efficiency of the resulting code. In RAMCloud, we group rules using a structure we call a *task*. A task has three elements: a state record, a set of rules, and a goal. Tasks are represented as instances of a class that uses its fields as the state record. For example, for log replication each segment is represented as a task whose fields describe which server each replica is stored on and how much data has been sent and acknowledged for each replica. Rules for a task are specified statically as a set of nested "if/else" statements in a single method. Its rules are applied to its state by invoking the method. Lastly, each task has a goal that its rules are intended to achieve. A segment replication task meets its goal when it creates three complete replicas of the segment.

Finally, we use a third layer we call a *pool* to group the tasks for a subsystem. Pools provide isolation between subsystems by partitioning tasks into independent sets and allow tasks from different pools to execute concurrently. For example, all of the segment replication tasks reside in a single pool in the log manager. Tasks for other subsystems like server recovery reside in separate pools and are run in parallel with log replication tasks.

Pools reduce the overhead of rule application by dividing tasks into two groups: *active* tasks, whose rules have to be evaluated, and *inactive* tasks, which can be skipped without evaluating their rules. A task remains active until it achieves its goal, at which point it becomes inactive.

For typical subsystems, only a small subset of tasks are active at any one time, so testing rules is efficient. For example, segments are usually only active for a short period when they are first added to the log, while they are transmitting new data for replication. Most segments are fully replicated and are ignored by the log manager pool. Failures can return a task to a state where its goal is no longer met, at which point it is reactivated.

In RAMCloud, each task pool has a single thread that cycles through the active tasks, executing their rules. Because rule execution is serialized, no synchronization is needed when testing rules or executing actions; rules from one task can safely test and modify the state from other tasks in the same pool. For example, rule R6 prevents a segment from replicating its footer before the header of the following segment in the log has been replicated; the condition on R6 tests the replication state of the following segment without any synchronization. Similarly, the action from rule R7 modifies state in the following segment task to allow it to start replication after a footer is committed.

High concurrency is achieved in RAMCloud by overlapping long-running operations using asynchronous RPCs and IO. Tasks are well-suited to managing this type of concurrency by using actions to start operations and conditions to poll for their completion. Using multiple threads for performance provides little benefit for these types of modules since they do not perform long running computations that could be run in parallel. RAMCloud does use threads for performance in code where fault-tolerance is ensured by underlying rules-based modules.

4 Discussion

As with monitors, one of the primary benefits of rules, tasks, and pools is that they free the developer from making complex decisions about how to organize code. In developing a rules-based software module a programmer iteratively applies a simple line of thinking.

1. Determine a circumstance under which a step must be taken to meet a goal. Define a condition for it.

2. Create an action that makes progress toward that goal. Subdivide actions that block. Try to implement actions that require the fewest additional rules. Wherever possible actions should leave the state such that existing rules will apply to it.

3. If the new action does not leave the task in the goal state, then add it to the set of active tasks in its pool.

Rules have two key benefits that simplify writing fault-tolerant code.

- Outside events and exceptions can redirect execution to account for new information by modifying state. Typically, exceptional cases are handled by reverting state to meet the condition of a rule that is logically an "earlier" step.

- Rules are selected only based on explicit state rather than a prespecified order. The programmer only needs to reason about states when creating rules and is freed from worrying about the history of computation that led to the state.

Tasks provide a structure that

- modularizes rules and state into relatively independent sets and eases reasoning about their interactions;

- is well-suited to managing IO and RPC concurrency;

- is inexpensive and requires no per-task kernel state;

- and relieves programmers from making mundane structuring decisions.

Finally, pools organize tasks

- to make rules efficient by allowing tasks to be ignored that do not need immediate work;

- and to ease synchronization complexity by serializing the application of rules within a subsystem.

4.1 Issues

Programming with rules decomposes a problem and eases implementation of individual steps, but managing and debugging hundreds of rules across thousands of instances of tasks can be challenging.

First, tracing execution history and causality can be difficult when programming with rules. The static code structure of a set of rules in the programmer's editor typically provides little information about the order in which rules get applied. Furthermore, there is no runtime stack to produce a trace of calls when debugging anomalous behavior. In practice, we have found aggressive logging of the state of tasks and the actions performed makes debugging tractable. Others have augmented log messages to track causality to assist in this type of debugging [8]. This problem is fundamental to any solution for DCFT code because execution order in is unpredictable.

A second complication with tasks are actions that leave a task in a non-goal state but fail to add it to the active task list for its pool. This leads to stalls in work

which can be difficult to detect. In our code, it has been helpful to dump the state of a task along with a warning message if a task is taking an unexpectedly long time to reach its goal state.

Similarly, a developer must ensure that all possible states that a task can get into are covered by the rules for the task. To trivially ensure all states are covered by a set of rules, for every clause in the condition we also include the "else" case for that condition. Unexpected states are marked with an error message. Statically enumerating all possible cases by construction has been adequate for us to avoid problems. We have only encountered unexpected states a few times, and they were always caught quickly using our error messages.

Finally, outside threads that want to extract information from tasks require synchronization. Though this is a problem in general rather than a problem with rules and tasks, our mechanism does not provide a built-in solution. In RAMCloud, different modules extract information from a set of tasks in various ways depending on the performance sensitivity of the outside code. One general approach is to create a special task that, when invoked, makes a copy of the requested state and notifies a condition variable to inform the outside code of its presence.

Despite these issues, this rules-based approach has already proven useful in our own development, and it exemplifies the type of pattern programmers need to develop fault-tolerant applications quickly and correctly.

5 Related Work

Rules are not new; others have arrived at rules-based code in writing fault-tolerant systems. Breaking code into small conditionally-applied non-blocking blocks is fundamental to systems where control flow must adapt to unpredictable events. For example, while not built on an explicit notion of rules, TCP implementations maintain per-connection state that is modified in reaction to timers and incoming and outgoing data. Actions are triggered based on the resulting state. Our goal is to extract a general pattern for implementing these types of systems.

Others have struggled with the difficulty of expressing fault-tolerant systems as well. For example, Chandra et al. [2] express a Paxos-based replication algorithm as a pair of state machines in a custom-made specification language to increase understandability. A task can be seen as a form of state machine with implicitly defined states and with rules on data fields driving transitions rather than events. We have found the entire set of rules for our most complex tasks can be expressed in a few hundred lines of C++; a task is compact enough for a developer to thoroughly reason about its rules.

Prior discussion on structuring concurrent servers has focused on performance rather than fault-tolerance. The two most prevalent models are threads [10, 11] and events [3, 5, 12, 13]; however, while fault handling has a substantial impact on how code and concurrency are organized, neither approach on its own addresses the complexity of fault handling in a distributed system.

5.1 Threads

Threads are the standard model for writing concurrent software; they allow for CPU parallelism and preserve the appearance of a simple serial programming model.

HDFS [9] is an example of a system that uses pervasive threading for DCFT code. It spawns threads that watch data structures and take specific actions when they change. For example, one thread periodically scans a "heartbeat" table removing cluster members that have failed to check in recently. Another thread polls replicated block state to ensure blocks are fully replicated, and it queues needed replication operations. Despite the ad hoc nature of these threads, at its heart this approach is rules-based; the threads perform actions in response to changes in state, divide work into small non-blocking actions, and schedule execution across many high-level state records (thousands of replicated block records, for example).

Compared to tasks, modularity suffers in HDFS since threads are assigned to specific rules rather being grouped with state records. Threads typically perform just a few actions before queueing the state record for other threads to perform additional steps. There is no single location for code operating on a particular record; the steps are scattered across the code text of many threads. Additionally, because many threads perform steps on a record, fine-grained locking is needed both on records themselves and on the many shared data structures consulted in processing them. As a result, the code contains precarious ad hoc locking patterns that are carefully structured to avoid starvation and deadlock. Just as monitors [4, 6] simplify concurrency by grouping synchronization with state records and the code that operates on them, tasks simplify code by grouping related state records and their associated rules together.

One major benefit of threads is that they can preserve code that follows a simple serial programming model; however, HDFS demonstrates how the need for fault handling interferes with this benefit. HDFS's structure loses the simplicity of stack-based code and deep, informative debugging backtraces. Handling outside faults makes this unavoidable; the code must allow frequent well-defined interruption points for failure notification which naturally fragments code into small, shallow blocks. The high cost of kernel threads amplifies this problem since they are too expensive to support the thousands of per-block contexts that would be most natural.

5.2 Event-based Programming

Programming with rules, tasks, and pools shares similarities with event-based programming [12, 3, 13, 5]. Both manage concurrency with asynchronous operations, and both serialize the execution of code to simplify synchronization. RAMCloud's RPC system uses an event-based model internally to manage asynchrony; however, these events are never exposed to higher-level code.

However, there is a key difference between events and rules; rules select actions for execution based solely on state, whereas events select handlers in response to occurrences, such as the completion of an RPC or a fault. As a result, with rules-based code, an event is handled only by modifying state, and rules that monitor the state are triggered indirectly when it changes. Rules use a "pull" model for reacting to events; event handlers remain decoupled from the actions which respond to the event. This improves modularity since the rules remain part of the tasks that react to the event. For example, adding a new task that reacts to an event requires no modification to the event handler. Event-based programming encourages a "push" model for reacting to events. An event handler must trigger all of the actions that are needed to respond to the event, so the details of how independent subsystems react to the event leak together.

By incorporating a simple form of events, a task could be inactivated while waiting on long-running operations for efficiency. Upon operation completion, the task would be reactivated; however, it could also be prematurely activated due to failures. This is similar to how notifications on condition variable are regarded as hints with monitors [6]. Since rules make no assumptions about execution order, no change would be needed in programming semantics. So far this optimization has been unnecessary.

6 Conclusion

We have described rules, tasks, and pools as a pattern for writing distributed, concurrent, fault-tolerant code. Rules-based code easily adapts to handle cluster-wide events like server faults. Tasks and pools organize rules for modularity, efficiency, and easy synchronization. We believe using some form of rules-based code is unavoidable in writing fault-tolerant systems, though investigating other patterns for organizing rules would be valuable.

These patterns have been a benefit to our own project already, and we plan to hone them as we develop new software systems. Hopefully, as we gain experience, we will find other useful ways to organize this type of complex code. We plan to explore and compare the patterns of other fault-tolerant systems to our own, and we encourage others to publish and relay their experiences in developing similar large-scale systems. Our hope is that widely accepted design patterns for writing distributed, concurrent, fault-tolerant code will emerge and simplify the development of large-scale systems.

Acknowledgments

Several people provided helpful feedback on the paper, including Steve Rumble, Diego Ongaro, and Ankita Kejriwal, and the anonymous HotOS reviewers. This work was supported by the National Science Foundation under Grant No. 0963859 and by STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

References

[1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative Task Management Without Manual Stack Management. In *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, ATEC '02, pages 289–302, Berkeley, CA, USA, 2002. USENIX Association.

[2] 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.

[3] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazières, and R. Morris. Event-driven Programming for Robust Software. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, EW 10, pages 186–189, New York, NY, USA, 2002. ACM.

[4] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Commun. ACM*, 17(10):549–557, Oct. 1974.

[5] M. Krohn, E. Kohler, and M. F. Kaashoek. Events Can Make Sense. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, pages 7:1–7:14, Berkeley, CA, USA, 2007. USENIX Association.

[6] B. W. Lampson and D. D. Redell. Experience with processes and monitors in Mesa. *Commun. ACM*, 23(2):105–117, Feb. 1980.

[7] 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.

[8] M. D. Schroeder, A. D. Birrell, and R. M. Needham. Experience with Grapevine: The Growth of a Distributed System. *ACM Transactions on Computer Systems*, 2(1):3–23, Feb. 1984.

[9] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[10] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the 9th conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03, pages 4–4, Berkeley, CA, USA, 2003. USENIX Association.

[11] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: scalable threads for internet services. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 268–281, New York, NY, USA, 2003. ACM.

[12] 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.

[13] N. Zeldovich, A. Yip, F. Dabek, R. Morris, D. Mazières, and M. F. Kaashoek. Multiprocessor Support for Event-Driven Programs. In *USENIX Annual Technical Conference, General Track*, pages 239–252, 2003.