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

Ryan Stutsman, Collin Lee, and John Ousterhout

University of Utah and Stanford University
Introduction

• More developers writing more code that is distributed, concurrent, fault-tolerant
  – 1000s of logical threads of execution
  – Erratic control flow
  – No commonly accepted patterns

• Imperative programming breaks down

• A pattern from experiences in RAMCloud: “rules”
  – Small steps whose execution order is based on state

• Goal: a simple approach to develop fault-tolerant code quickly/correctly
Examples of Fault-Tolerant Systems

• RAMCloud
  – Chunk replication: millions on 1000s of nodes
  – Node recovery: 1000s of nodes working together
  – Coordinate simultaneous recoveries
  – Cluster membership, tablet ownership, and more

• Coordinating Hadoop jobs, HDFS replication

• Consensus-based coordination services
  – Chubby, ZooKeeper, LogCabin
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.


Even more true when worrying about latency/performance
Outline

• Example Module: Log Replication
• Today: Event-Driven State Machines
• Our Approach: Rules-based Programming
• Structuring Rules
• Conclusion
Example: Distributed Log Replication

- **Goal:** create and maintain on-disk replicas for segments
Example: Distributed Log Replication

- Failures can occur at any time
- Must “go back” to recreate replicas; sometimes extra steps
Example: Distributed Log Replication

- Failures can occur at any time
- Must “go back” to recreate replicas; sometimes extra steps

![Diagram showing distributed log replication process]
Example: Distributed Log Replication

- Failures can occur at any time
- Must “go back” to recreate replicas; sometimes extra steps
- Even failures while handling failures
- Transitions tricky: ordering constraints → shared state
Why Imperative Code Breaks Down

- 100s of 1000s of logical flows of execution
- Jumps at any point → complex control flow
- Hard to interrupt
  - Shared state, holding locks? Blocking?
- Imperative code creates expectation of ordering
  - Program counter does more harm than good
- Result: spaghetti code, brittle, buggy
Outline

• Example Module: Log Replication
• Today: Event-Driven State Machines
• Our Approach: Rules-based Programming
• Structuring Rules
• Conclusion
In Practice: Hadoop Job Scheduler

• Implemented as an explicit state machine

• Incoming event together with distinguished state variable gives a *transition*
  – Transactions take forward steps
Hadoop Scheduler State Machine

• Three main ‘tasks’: Job, Task, TaskAttempt

<table>
<thead>
<tr>
<th>State Machine</th>
<th>States</th>
<th>Transitions</th>
<th>Distinct Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job</td>
<td>14</td>
<td>82</td>
<td>27</td>
</tr>
<tr>
<td>Task</td>
<td>7</td>
<td>24</td>
<td>16</td>
</tr>
<tr>
<td>TaskAttempt</td>
<td>13</td>
<td>57</td>
<td>15</td>
</tr>
<tr>
<td>Total</td>
<td>34</td>
<td>163</td>
<td>58</td>
</tr>
</tbody>
</table>

• Conservatively 2,250 lines of code
  – 750 lines just to describe the transition diagram
‘Job’ State Machine Diagram
TaskAttempt Transition Setup

// Transitions from the NEW state.
.addTransition(TaskAttemptStateInternal.NEW, TaskAttemptStateInternal.UNASSIGNED,
   TaskAttemptEventType.TA_SCHEDULE, new RequestContainerTransition(false))
.addTransition(TaskAttemptStateInternal.NEW, TaskAttemptStateInternal.UNASSIGNED,
   TaskAttemptEventType.TA_RESCHEDULE, new RequestContainerTransition(true))
.addTransition(TaskAttemptStateInternal.NEW, TaskAttemptStateInternal.KILLED,
   TaskAttemptEventType.TA_KILL, new KilledTransition())
.addTransition(TaskAttemptStateInternal.NEW, TaskAttemptStateInternal.FAILED,
   TaskAttemptEventType.TA_FAILMSG, new FailedTransition())
.addTransition(TaskAttemptStateInternal.NEW,
   EnumSet.of(TaskAttemptStateInternal.FAILED,
   TaskAttemptStateInternal.KILLED,
   TaskAttemptStateInternal.SUCCEEDED),
   TaskAttemptEventType.TA_RECOVER, new RecoverTransition())
.addTransition(TaskAttemptStateInternal.NEW,
   TaskAttemptStateInternal.NEW,
   TaskAttemptEventType.TA_DIAGNOSTICS_UPDATE,
   DIAGNOSTIC_INFORMATION_UPDATE_TRANSITION)

// Transitions from the UNASSIGNED state.
.addTransition(TaskAttemptStateInternal.UNASSIGNED,
   TaskAttemptStateInternal.ASSIGNED, TaskAttemptEventType.TA_ASSIGNED,
   new ContainerAssignedTransition())
.addTransition(TaskAttemptStateInternal.UNASSIGNED, TaskAttemptStateInternal.KILLED,
   TaskAttemptEventType.TA_KILL, new DeallocateContainerTransition( 
   TaskAttemptStateInternal.KILLED, true))
private static class LaunchedContainerTransition implements SingleArcTransition<TaskAttemptImpl, TaskAttemptEvent> 
{
    @Override
    public void transition(TaskAttemptImpl taskAttempt, TaskAttemptEvent evnt) {

        TaskAttemptContainerLaunchedEvent event = 
            (TaskAttemptContainerLaunchedEvent) evnt;

        //set the launch time
        taskAttempt.launchTime = taskAttempt.clock.getTime();
        taskAttempt.shufflePort = event.getShufflePort();

        // register it to TaskAttemptListener so that it can start monitor
        taskAttempt.taskAttemptListener
            .registerLaunchedTask(taskAttempt.attemptId, taskAttempt.jvmID);

        // TODO Resolve to host / IP in case of a local address.
        InetSocketAddress nodeHttpInetAddress = // TODO: Costly to create soc
        NetUtils.createSocketAddr(taskAttempt.container.getNodeHttpAddr
        taskAttempt.trackerName = nodeHttpInetAddress.getHostName());
Outline

• Example Module: Log Replication
• Today: Event-Driven State Machines
• Our Approach: Rules-based Programming
• Structuring Rules
• Conclusion
Rules-based Programming

- Trigger actions based on shared state
  - Execution order determined by **state**
  - Divide into short, non-blocking, atomic actions
- Failures handled “between” actions, not within
  - Simple deterministic control flow within an action
Rule: Condition + Action

If *unreplicated data* and *no RPC outstanding* and *prior segment footer is replicated*

Then *start write RPC containing data*
Rules-based Programming

- Convert all events to changes on shared state
- Decouples event handling from actions
## Example: Segment Replication Rules

<table>
<thead>
<tr>
<th>#</th>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>No backup server selected</td>
<td>Choose available server to hold replica</td>
</tr>
<tr>
<td>2</td>
<td>Header unreplicated, no RPC outstanding</td>
<td>Start RPC containing the header</td>
</tr>
<tr>
<td>3</td>
<td>Header unreplicated, RPC completed</td>
<td>Mark header replicated; mark prior segment to allow footer replication</td>
</tr>
<tr>
<td>4</td>
<td>Unreplicated data, no RPC outstanding, prior footer is replicated</td>
<td>Start write RPC containing up to 1 MB of unreplicated data</td>
</tr>
<tr>
<td>5</td>
<td>Unreplicated data, RPC completed</td>
<td>Mark sent data as replicated</td>
</tr>
<tr>
<td>6</td>
<td>Segment finalized, following header replicated, footer not sent, no RPC outstanding</td>
<td>Start RPC containing the footer</td>
</tr>
<tr>
<td>7</td>
<td>Segment finalized, RPC completed</td>
<td>Mark footer replicated; mark following segment to allow data replication</td>
</tr>
</tbody>
</table>

### On failure: reset sent/replicated bytes and RPCs
Mappy: Rules-based MapReduce

```python
def applyRules():
    ...
    elif self.launch_rpc == None:
        # Rule 4: Attempt not launched; launch.
        self.launch_rpc = RPC(self.container, None, ("LAUNCH", self.work))
        self.rpcManager.send(self.launch_rpc)
        self.time = time.time()
    elif self.launch_rpc.status != "complete":
        # Placeholder for state with nothing to do (Attempt running).
        pass
    elif self.launch_rpc.reply == "failed":
        # Rule 5: Attempt failed; report container failure, goal reached.
        self.eventQueue.append(("CONTAINER_FAILED", (self, self.container)))
        self.status = "FAILED"
    elif self.commit_rpc == None:
        # Rule 6: Attempt complete but not committed; request commit.
        self.commit_rpc = RPC(self.container, None, ("COMMIT", self.work))
        self.rpcManager.send(self.commit_rpc)
```
Comparison with State Machine

<table>
<thead>
<tr>
<th>States Machine</th>
<th>States</th>
<th>Distinct Transitions</th>
<th>Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job</td>
<td>14</td>
<td>27</td>
<td>7</td>
</tr>
<tr>
<td>Task</td>
<td>7</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>TaskAttempt</td>
<td>13</td>
<td>15</td>
<td>8</td>
</tr>
<tr>
<td>Total</td>
<td>34</td>
<td>58</td>
<td>19</td>
</tr>
</tbody>
</table>

- <300 lines of code compared to 2,250
  - Each task fits on a single screen

- Eliminates redundancy
  - Explicit listing of transition diagram
  - Many events/event combinations may trigger same action
  - No need for code $O(|\text{States} \times \text{Events}|)$
Outline

• Example Module: Log Replication
• Today: Event-Driven State Machines
• Our Approach: Rules-based Programming
• Structuring Rules
• Conclusion
Structuring Rules

• How should thousands of rules be organized?
  – Need modularity and clear visualization
  – **Tasks:** group rules with the state they act on

• How can rules be evaluated efficiently?
  – Polling to test conditions for all rules won’t scale
  – **Pools:** only check rules likely to fire
Tasks: Modularizing Rules

• **Task:** Rules, state, and a goal
  – Implemented as a C++ object
  – **State:** fields of the object
  – **Rules:** applied via virtual method on the object
  – **Goal:** invariant the task is intended to attain/retain

• **Log segment replication**
  – One task per segment
  – Rules send/reap RPCs, reset state on failures
  – Goal is met when enough complete replicas are made
**Pools: Making Rules Efficient**

- **Pool:** Applies rules to tasks with unmet goals
  - Divides tasks into active/inactive set
- One pool for each independent set of tasks
- Serial execution within pool; easy synchronization
  - Parallelism only if desired; easy access to shared state
## Fault-Tolerant RAMCloud Modules

<table>
<thead>
<tr>
<th>Module</th>
<th>Task Types</th>
<th>Rules</th>
<th>Event Handlers</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Log Replication</td>
<td>1</td>
<td>23</td>
<td>3</td>
<td>258</td>
</tr>
<tr>
<td>Recovery Coordination</td>
<td>4</td>
<td>12</td>
<td>2</td>
<td>299</td>
</tr>
<tr>
<td>Master Recovery on Masters</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>230</td>
</tr>
<tr>
<td>Master Recovery on Backups</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>31</td>
</tr>
<tr>
<td>Membership Notifier</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>36</td>
</tr>
<tr>
<td>Multi-Read</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>75</td>
</tr>
<tr>
<td>Indexed Read</td>
<td>1</td>
<td>14</td>
<td>2</td>
<td>131</td>
</tr>
<tr>
<td><strong>Lines of Rules-based Code</strong></td>
<td></td>
<td></td>
<td></td>
<td><strong>1,060</strong></td>
</tr>
<tr>
<td><strong>Total Source Lines</strong></td>
<td></td>
<td></td>
<td></td>
<td><strong>123,000</strong></td>
</tr>
</tbody>
</table>
Outline

• Example Module: Log Replication
• Today: Event-Driven State Machines
• Our Approach: Rules-based Programming
• Structuring Rules
• Conclusion
Related Work

- **Threads versus events**
  - [Welsch 02], [Adya 02], [Dabek 02], [von Behren 03], [Zeldovich 03], [Krohn 07], ...
  - Neither answer how fault-handling shapes code
  - Mostly considered services with trivial fault-handling

- **Actors**
  - [Hewitt]..., [Agha]..., [Haller & Odersky 06], ...
  - Async communicating entities w/o shared state
  - Performance cost & don’t want unneeded async

- **Reactive Programming**
  - [Cooper & Krishnamurthi 06], [Maier 10], ...
  - Like spreadsheet: recompute when inputs change
  - PL concepts, then UI programming, now in services
Conclusion

• More developers writing more code that is distributed, concurrent, fault-tolerant
• No commonly accepted patterns
• A pattern from experiences in RAMCloud: “rules”
  – Small steps whose execution order is based on state
  – Easy to adapt on failures
• Simple approach to developing fault-tolerant code quickly/correctly
  – More understandable code
  – Suitable for high-performance, low-latency systems
github.com/PlatformLab/mappy
Segment Replication Latency Impact

![Graph showing the relationship between applyRules Execution Time (ns, log scale) and Fraction of Invocations. The graph compares Time w/o RPC actions and Total time.]
Isn’t this just state machines?

• Explicit states explode or hide detail
  – Similar to code flowcharts of the 70s
  – Mental model doesn’t scale well to complex code

• Collate on state rather than on events+state
  – Convert all events to state
  – Reason about next step based on state alone

• Conditions (implicit states) serve as documentation
  – Provide strong hint about what steps are needed
Isn’t this hard to debug?

• Loss of stack context makes debugging hard
• Yes, but it would be lost even with threads
• Fundamental limitation of the need to break code into reactive, reorderable blocks
• Best we’ve got so far
  – Dump state variables when a goal goes unmet for a long period
  – Log aggressively for debugging
• Can track causality in log (Grapevine)
Isn’t this just events?

• Rules take actions based on state
  – Rather than events+state
• Event-based code: handler triggers all needed actions
• Rules-based code: events just modify state
  – Decouples events from rules that react to them
  – Event handler unaware of needed reactive steps
  – Add reactions without modifying event handler
  – Improves modularity
Don’t user-level threads solve this?

• They help
  – Support 1000s of lightweight contexts
  – Limit interruption to well-defined points
    (cooperatively scheduled)

• Stack-trace is still of limited benefit, though
  – Threads must recheck for failures after resuming
  – Code devolves into small, non-blocking, atomic actions just as with rules

• Still need rules-like constructs for failures
What about actors?

• Push vs Pull
  – On failure, actor must push to all the right places
  – Breaks separation of concerns
• Unneeded parallelism
• No shared state
• Asynchronous whether you like it or not
• Hard to make efficient
  – Locking/pushing/popping/scanning queues
  – Copying instead of sharing