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.

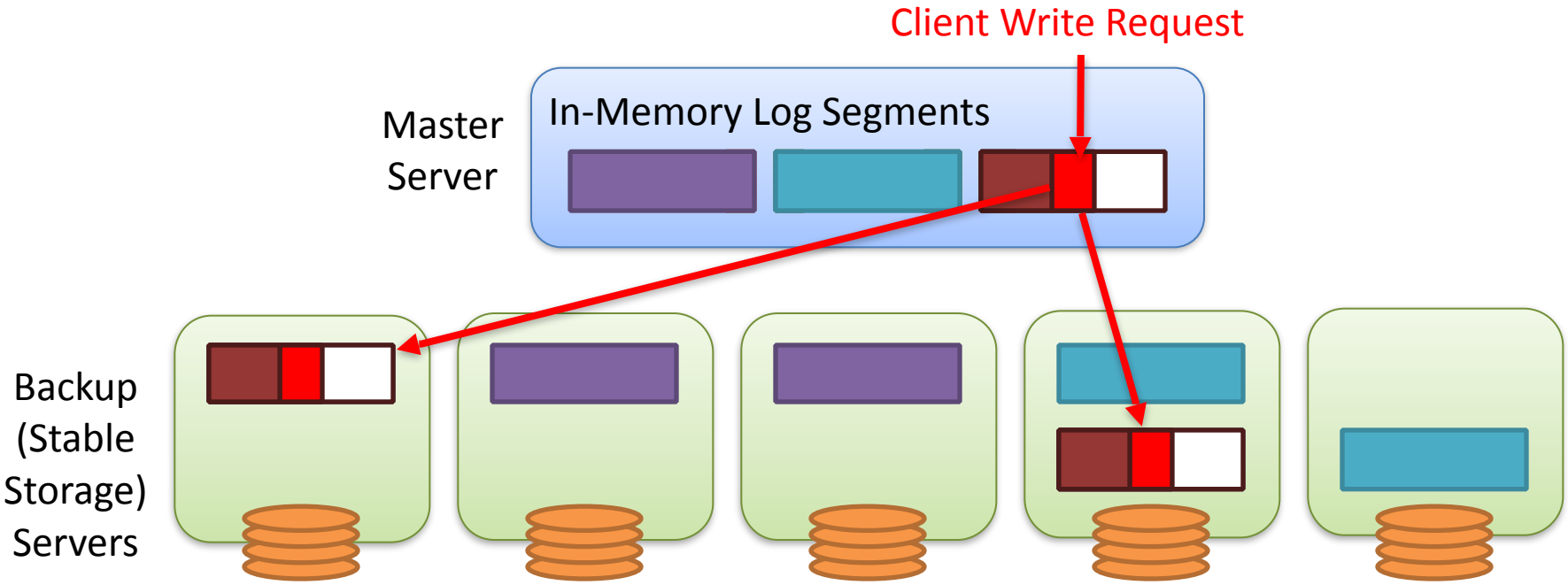
T. D. Chandra, R. Griesemer, and J. Redstone. *Paxos Made Live*, PODC '07.

Even more true when worrying about latency/performance

Outline

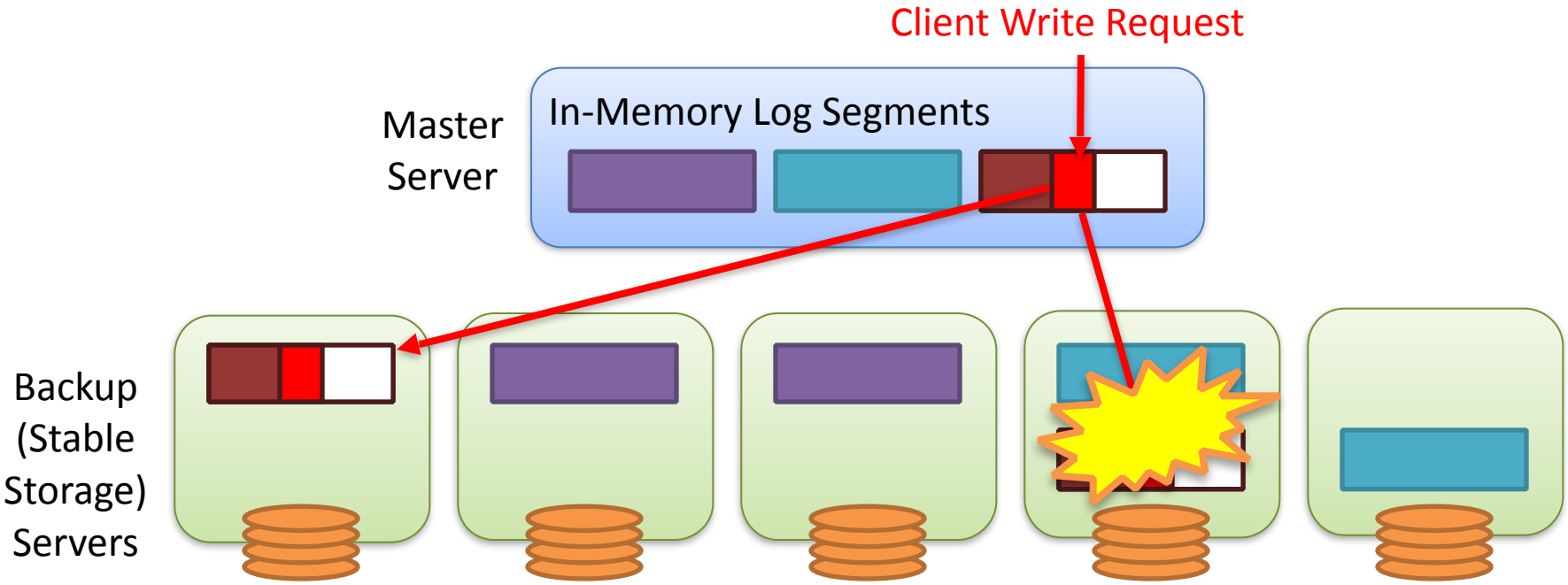
- **Example Module: Log Replication**
- Today: Event-Driven State Machines
- Our Approach: Rules-based Programming
- Structuring Rules
- Conclusion

Example: Distributed Log Replication



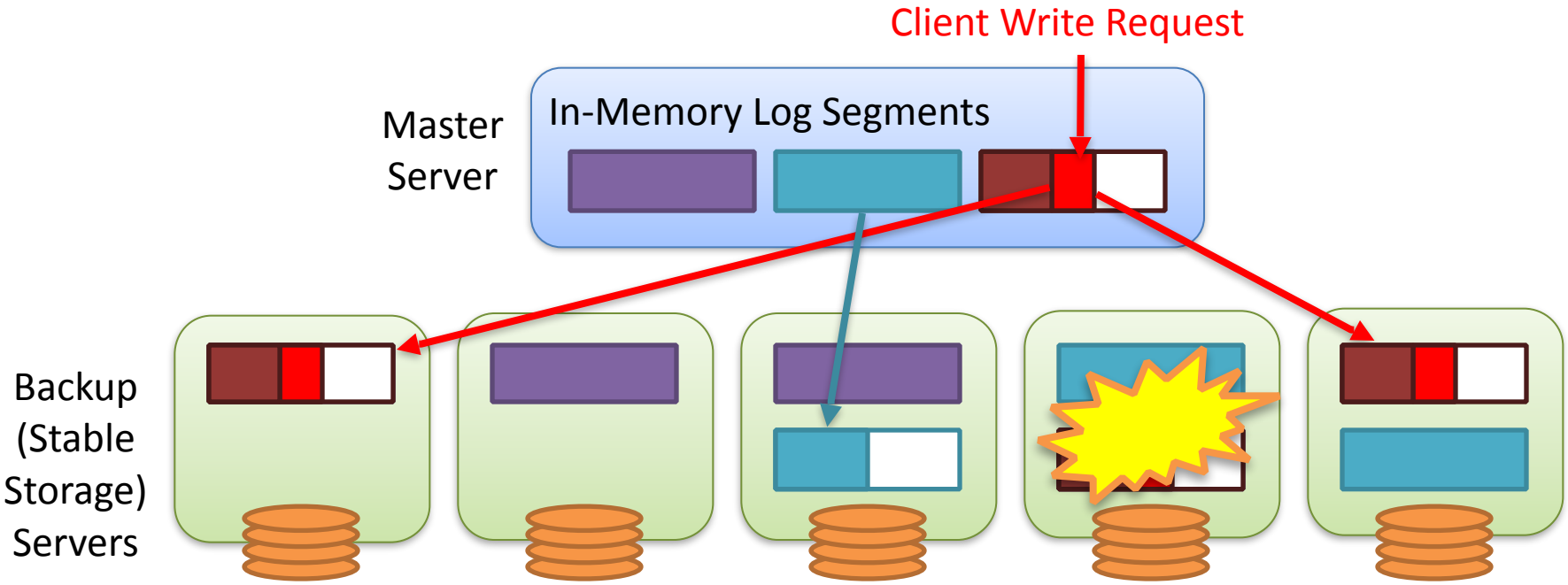
- **Goal: create and maintain on-disk replicas for segments**

Example: Distributed Log Replication



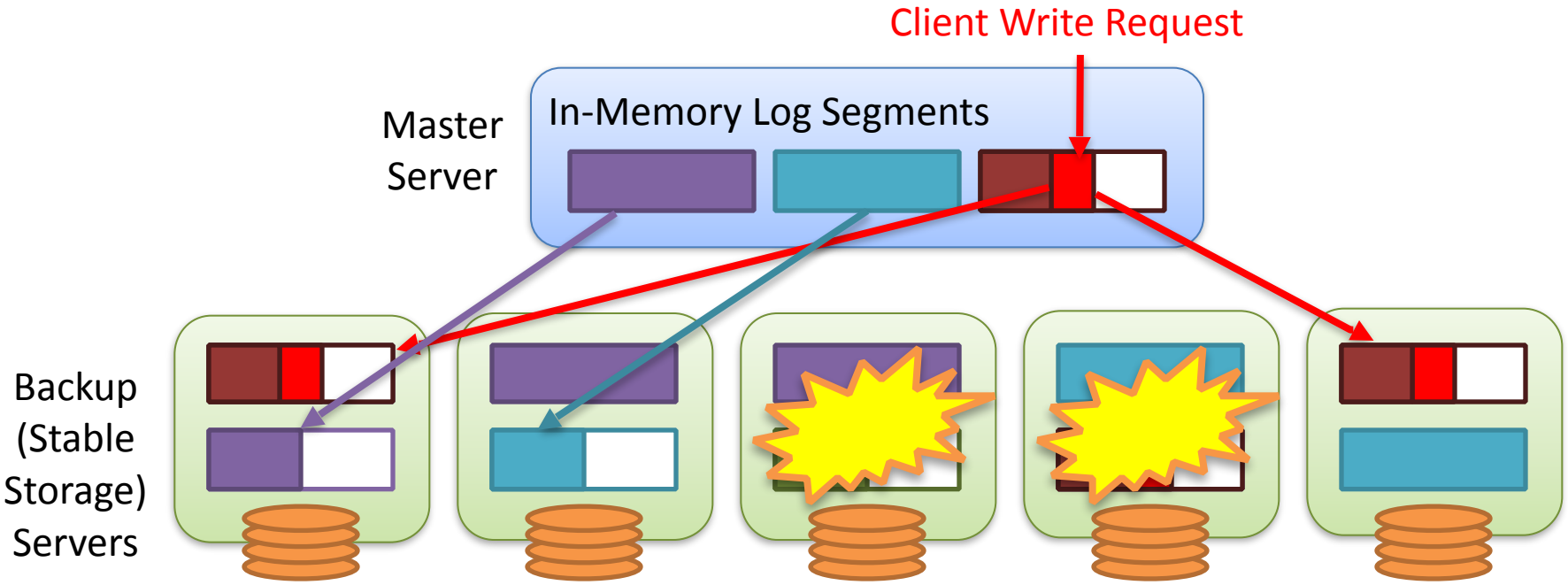
- Failures can occur at any time
- Must “go back” to recreate replicas; sometimes extra steps

Example: Distributed Log Replication



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

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



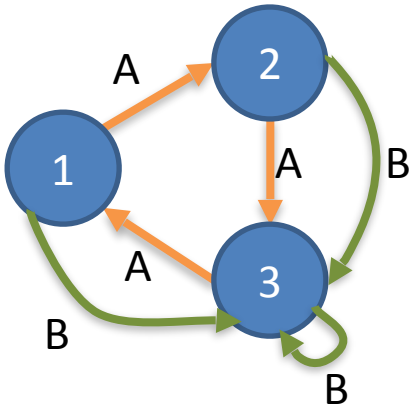
- 100s of 1000s of logical flows of execution
- Jumps at any point → **complex control flow**
- Hard to interrupt
  - Shared state, holding locks? Blocking?
- Imperative code creates expectation of ordering
  - Program counter does more harm than good
- **Result: spaghetti code, brittle, buggy**

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

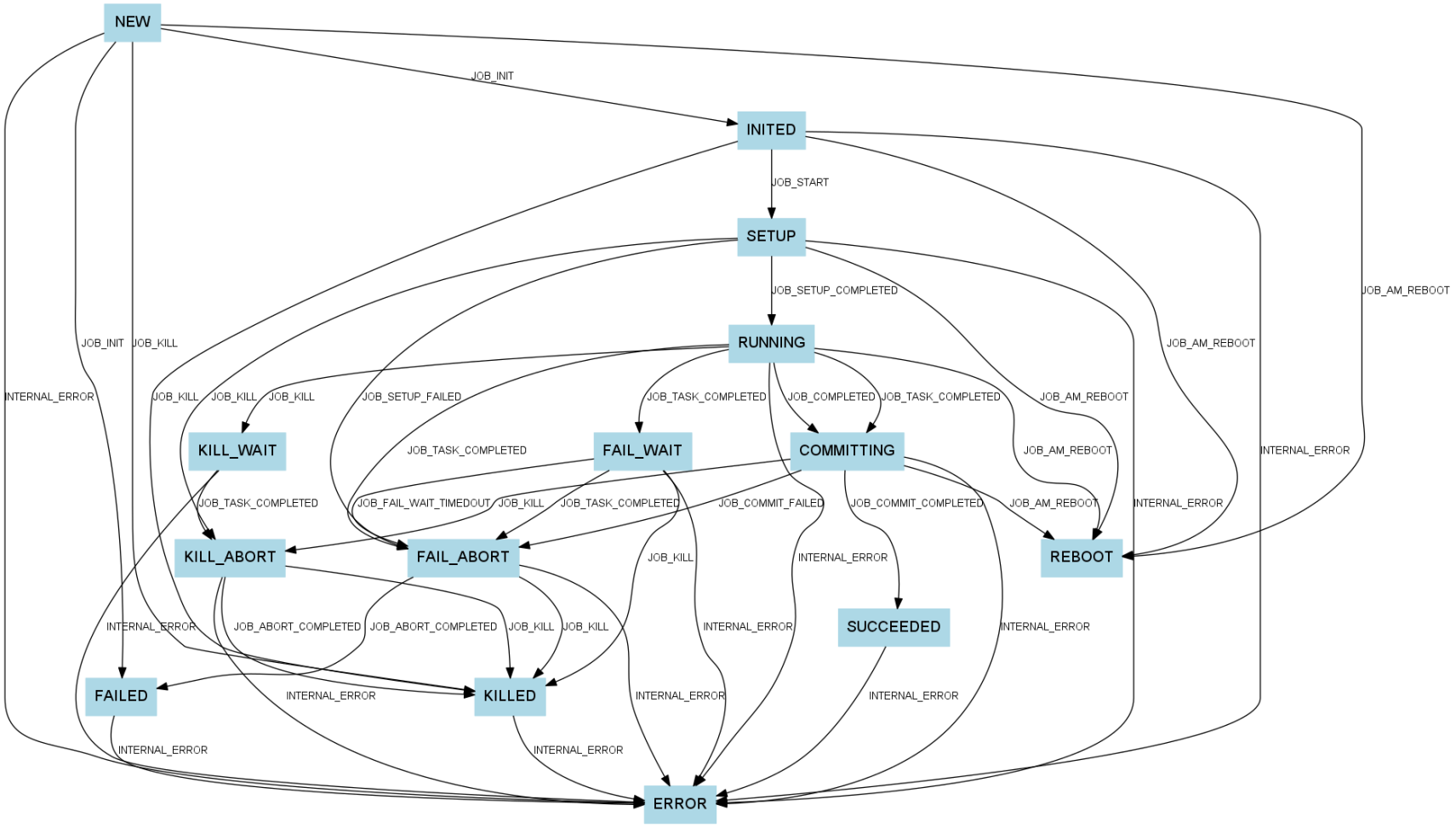
Hadoop Scheduler State Machine

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

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

- 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.KILLE
    TaskAttemptEventType.TA_KILL, new DeallocateContainerTransition(
        TaskAttemptStateInternal.KILLED, true))
```

TaskAttempt Transitions

```
private static class LaunchedContainerTransition implements
    SingleArcTransition<TaskAttemptImpl, TaskAttemptEvent> {
  @SuppressWarnings("unchecked")
  @Override
  public void transition(TaskAttemptImpl taskAttempt,
      TaskAttemptEvent evnt) {

    TaskAttemptContainerLaunchedEvent event =
      (TaskAttemptContainerLaunchedEvent) evnt;

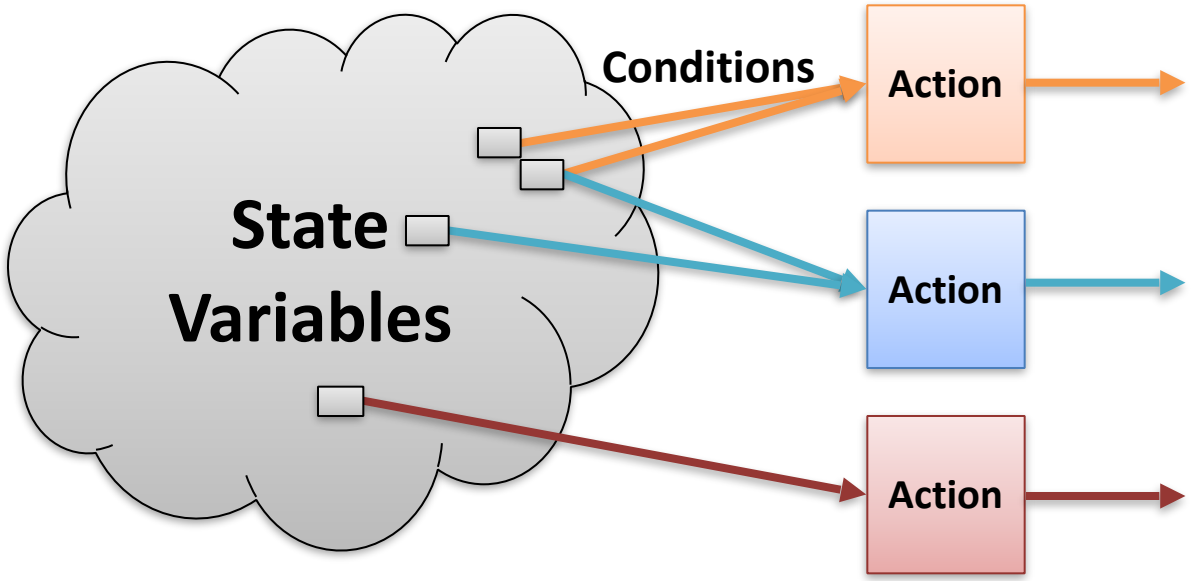
    //set the launch time
    taskAttempt.launchTime = taskAttempt.clock.getTime();
    taskAttempt.shufflePort = event.getShufflePort();

    // 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 nodeHttpInetAddr = // TODO: Costly to create soc
        NetUtils.createSocketAddr(taskAttempt.container.getNodeHttpAdd
    taskAttempt.trackerName = nodeHttpInetAddr.getHostName();
```

Outline

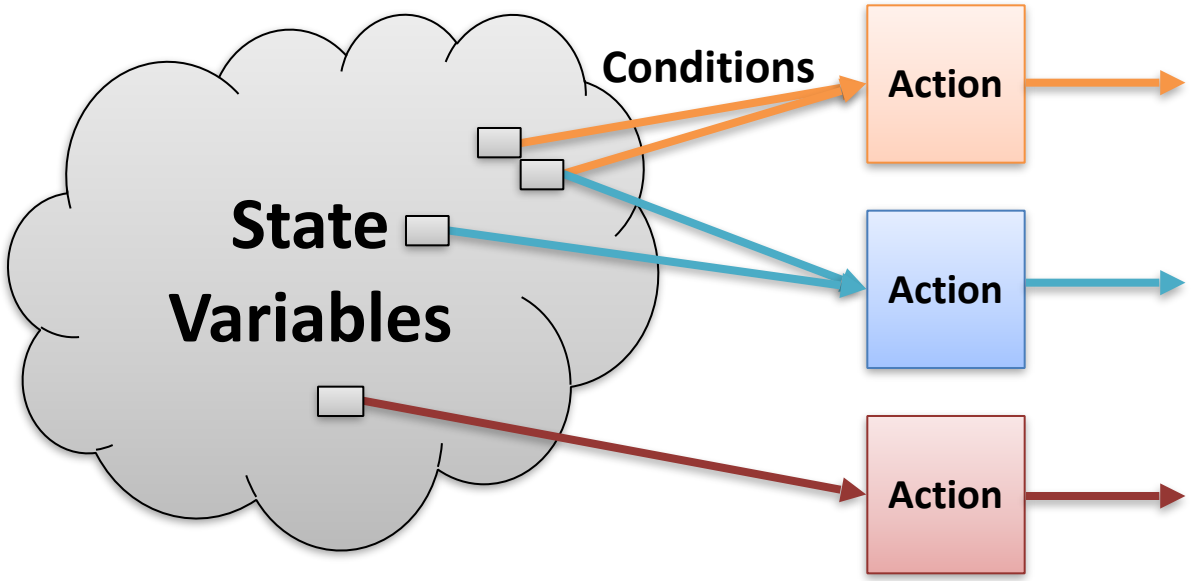
- Example Module: Log Replication
- Today: Event-Driven State Machines
- **Our Approach: Rules-based Programming**
- Structuring Rules
- Conclusion

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

Rules-based Programming

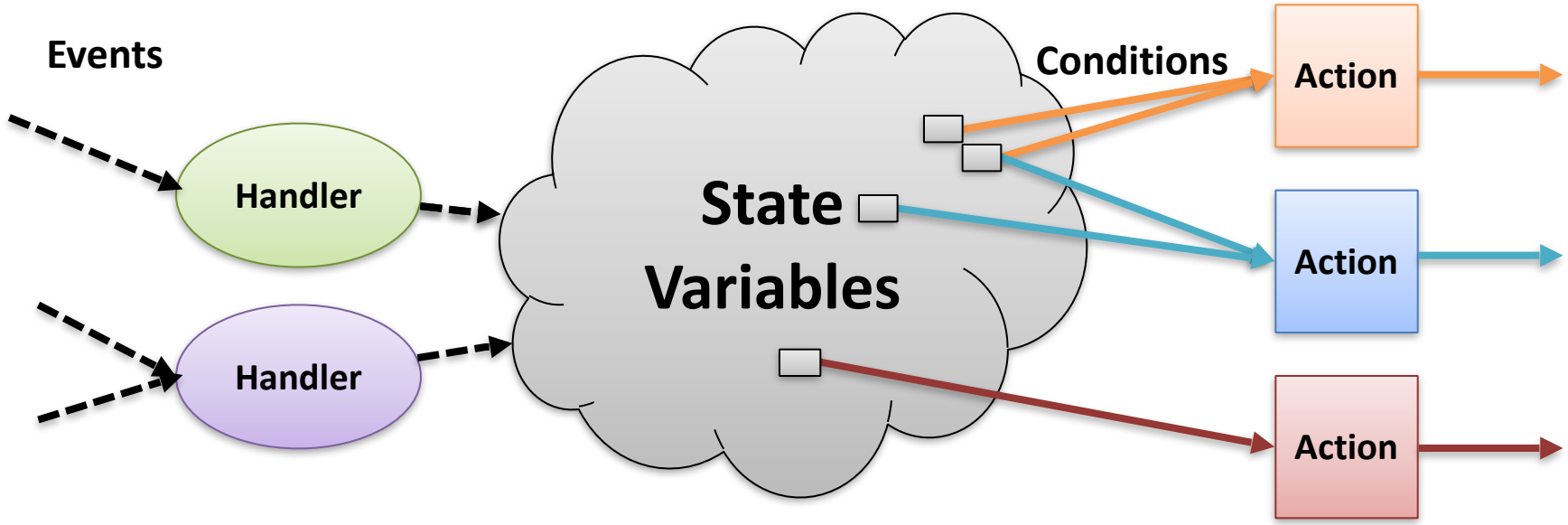


Rule: Condition + Action

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

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



On failure: reset sent/replicated bytes and RPCs

Mappy: Rules-based MapReduce

```
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

State Machine	States	Distinct Transitions	Rules
Job	14	27	**7**
Task	7	16	**4**
TaskAttempt	13	15	**8**
Total	34	58	**19**

- **< 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(|States × Events|)

Outline

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

Structuring Rules

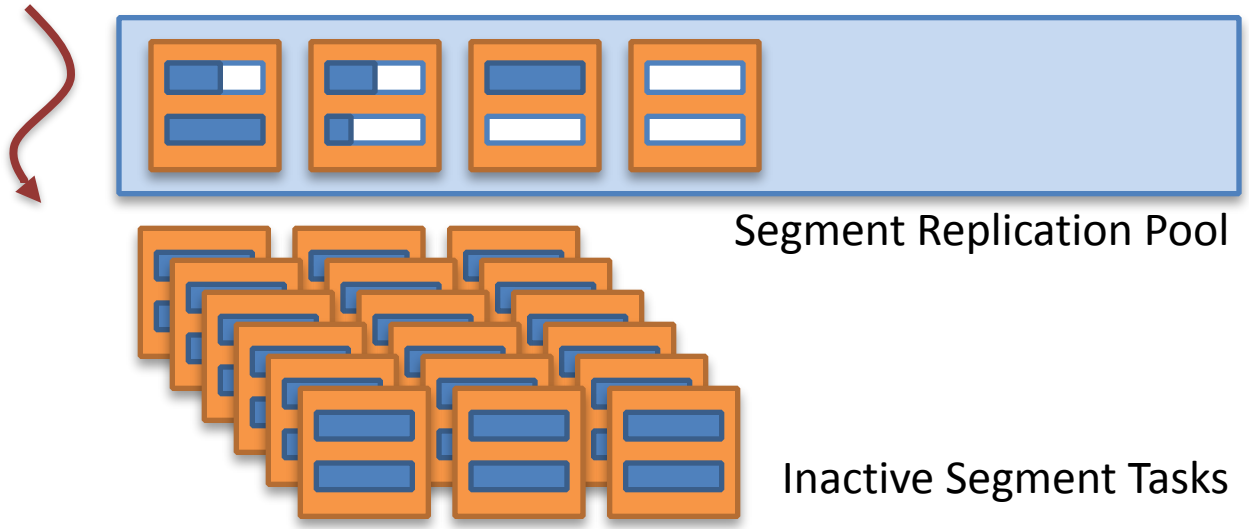
- **How should thousands of rules be organized?**
  - Need modularity and clear visualization
  - **Tasks:** group rules with the state they act on
- **How can rules be evaluated efficiently?**
  - Polling to test conditions for all rules won't scale
  - **Pools:** only check rules likely to fire

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

Module	Task Types	Rules	Event Handlers	Lines of Code
Log Replication	1	23	3	258
Recovery Coordination	4	12	2	299
Master Recovery on Masters	1	3	0	230
Master Recovery on Backups	1	4	2	31
Membership Notifier	1	3	3	36
Multi-Read	1	2	2	75
Indexed Read	1	14	2	131
Lines of Rules-based Code				**1,060**
Total Source Lines				**123,000**

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

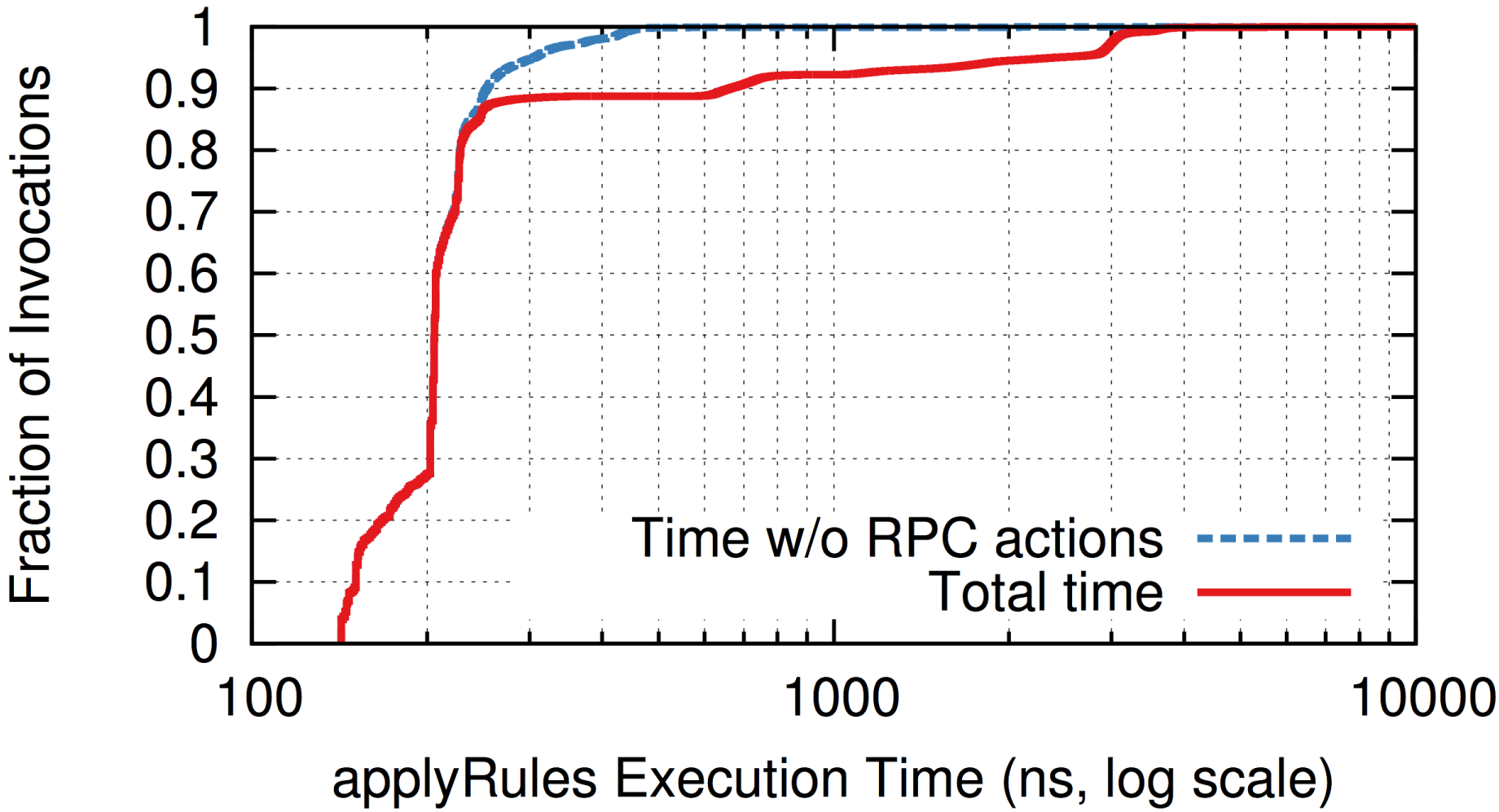
- **Reactive Programming** [Cooper &Krishnamurthi 06], [Maier 10], …
  - Like spreadsheet: recompute when inputs change
  - PL concepts, then UI programming, now in services

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



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