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

Ryan Stutsman

and John Ousterhout

Stanford University

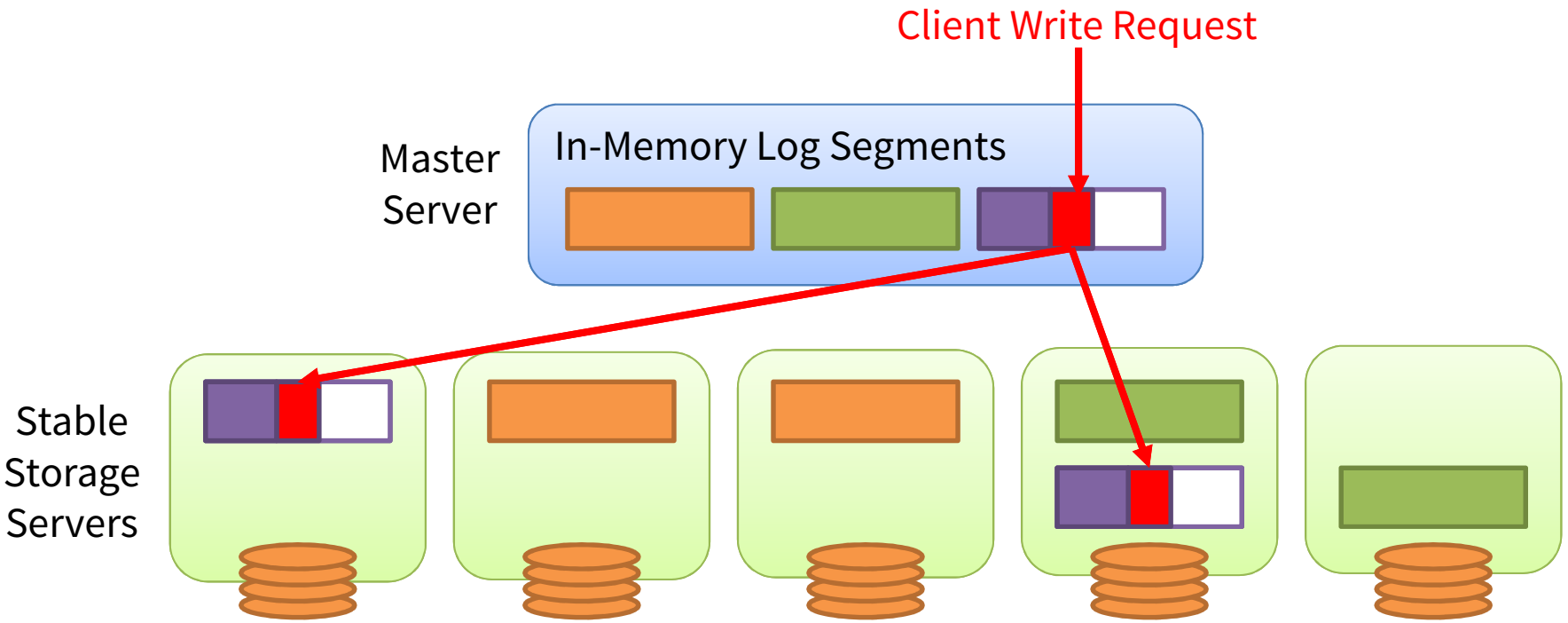
# Introduction

- More developers writing more code that is **distributed, concurrent, fault-tolerant (DCFT)**
  - Hard to get right
  - 1000s of logical threads of execution
  - Failures require highly adaptive control flow
  - **No commonly accepted patterns**
- Threads versus events but then what?
- A pattern from our experiences: "**rules**"
  - Small steps whose execution order is based on state
  - Potential for correct code more quickly
- Fumbling in the dark; interested in others' ideas

# DCFT Code Examples

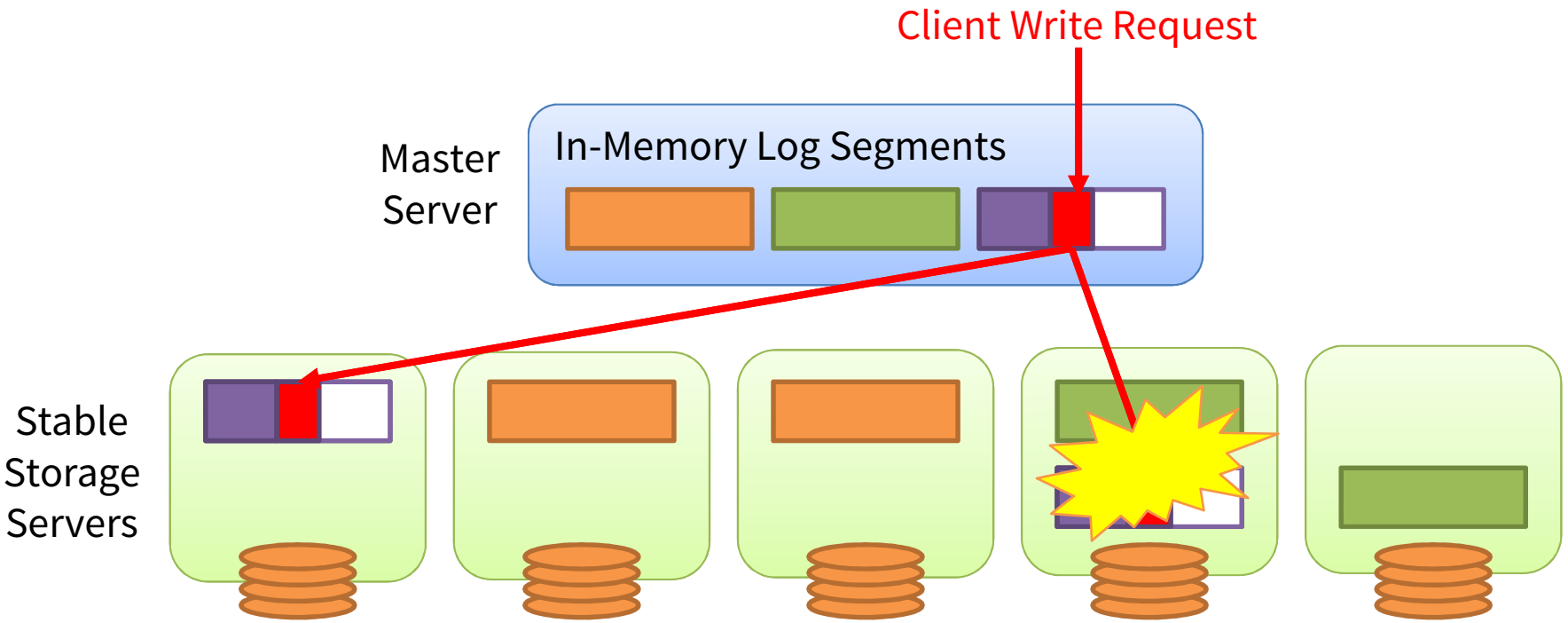
- HDFS chunk replication
- Coordinating  Hadoop jobs
- RAMCloud
  - Replicate 1000s of chunks across 1000s of servers
  - Coordinate 1000s servers working to recover failed server
  - Coordinate many ongoing recoveries at the same time

# Example: Distributed Log Replication



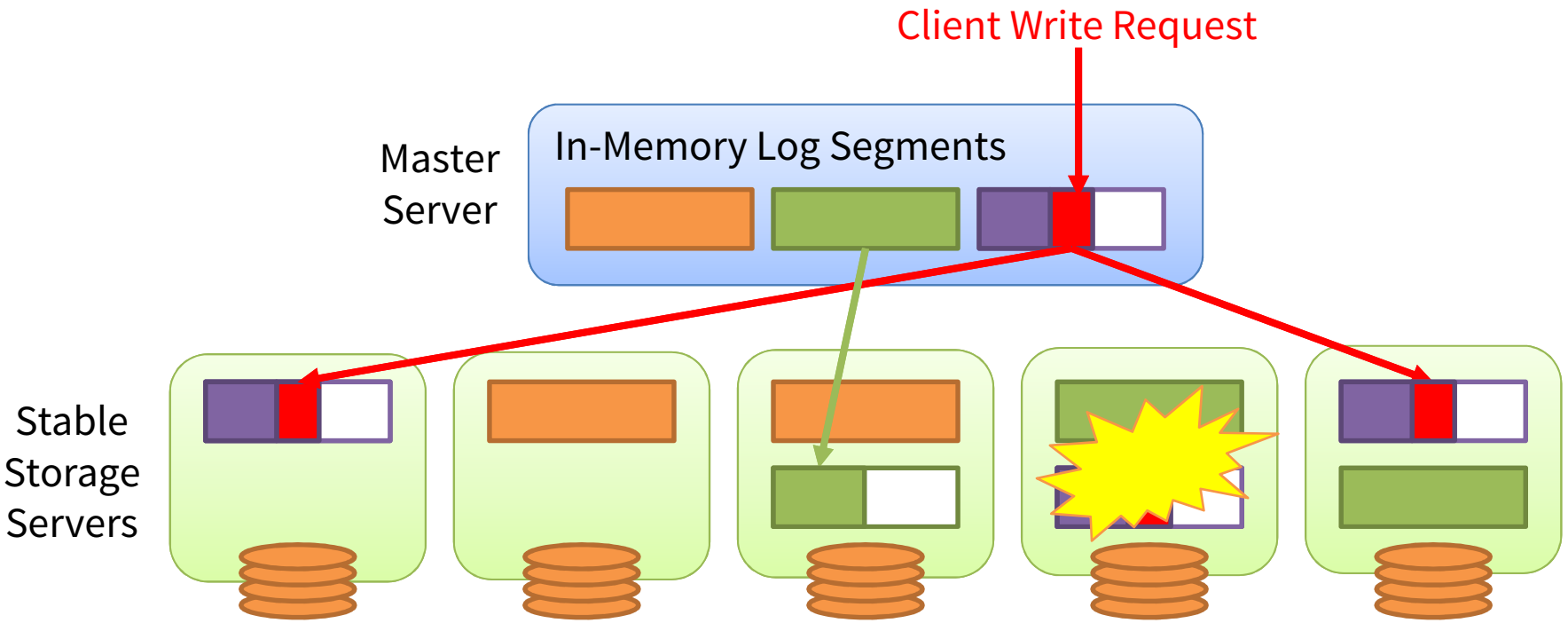

- Segmented log replicates client data

# Example: Distributed Log Replication



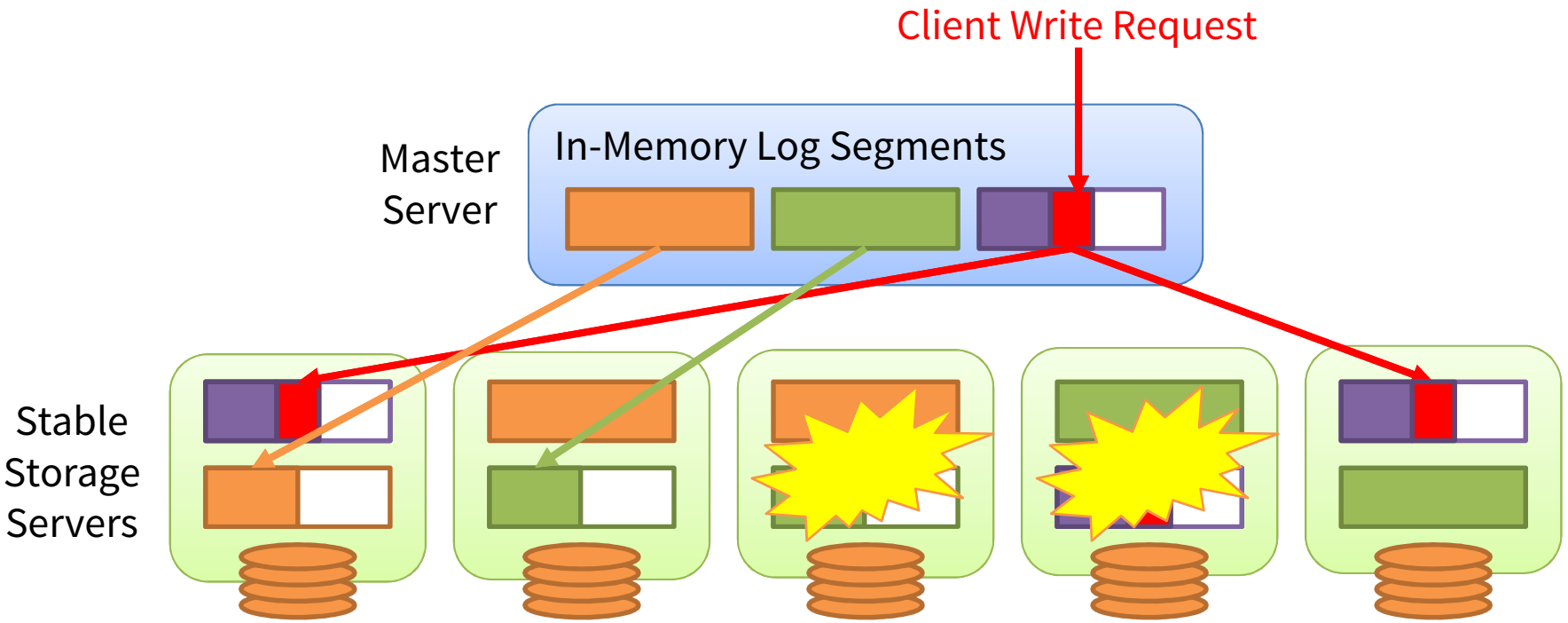

- Segmented log replicates client data
- Failures require recreation of lost segments

# Example: Distributed Log Replication




- Segmented log replicates client data
- Failures require recreation of lost segments

# Example: Distributed Log Replication



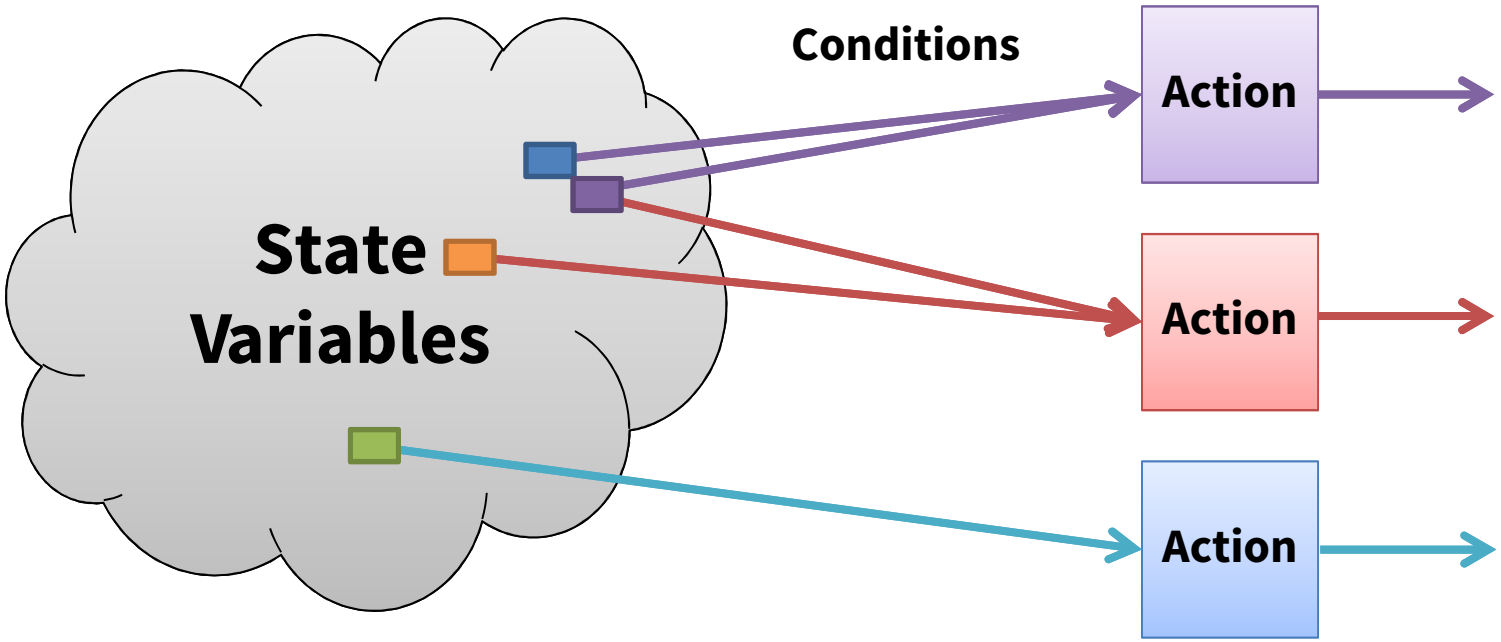

- Segmented log replicates client data
- Failures require recreation of lost segments
- Failures can occur at any time

# This Type of Code is Hard

- Traditional imperative programming doesn't work
- Result: spaghetti code, brittle, buggy


- PC doesn't matter, only state matters

# Our Approach: Rules

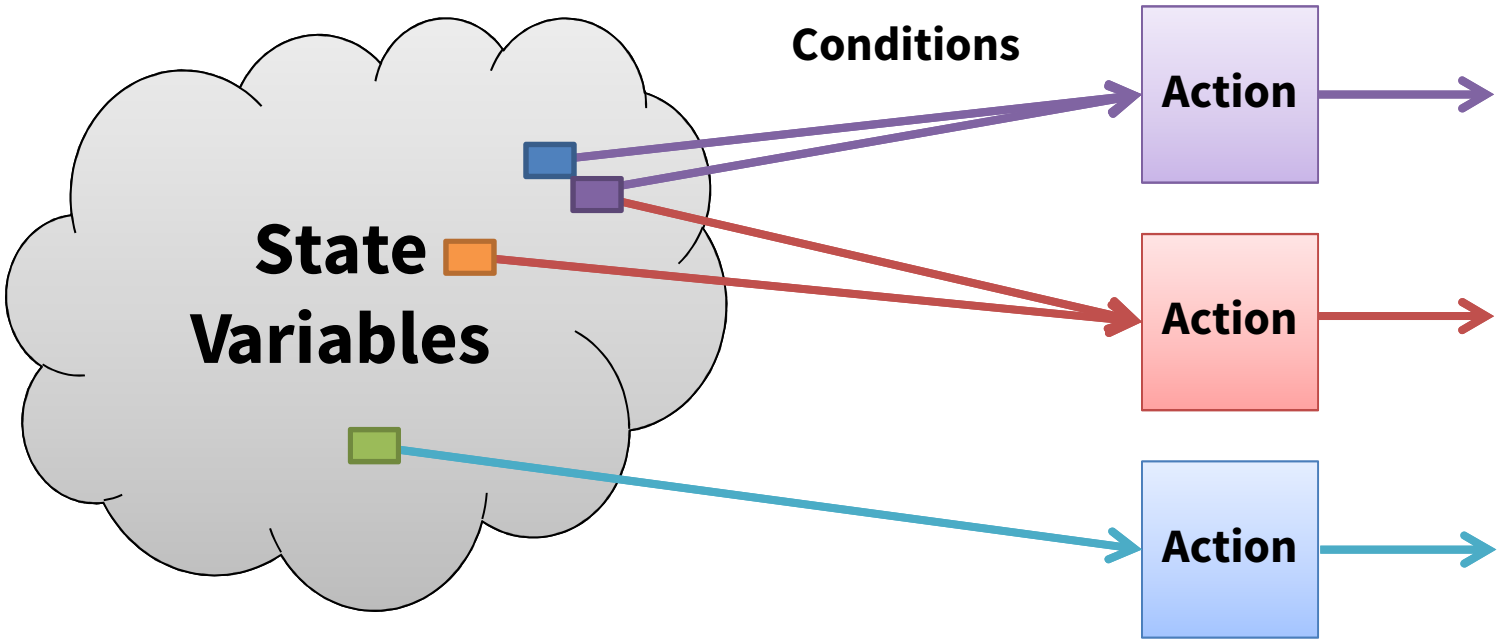



**Rule: Condition + Action**

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

Then **start write RPC** containing unreplicated data

# Our Approach: Rules




- Execution order determined by state
- Actions are short, non-blocking, atomic
- Failures handled between actions not within

# Segment Replication Rules

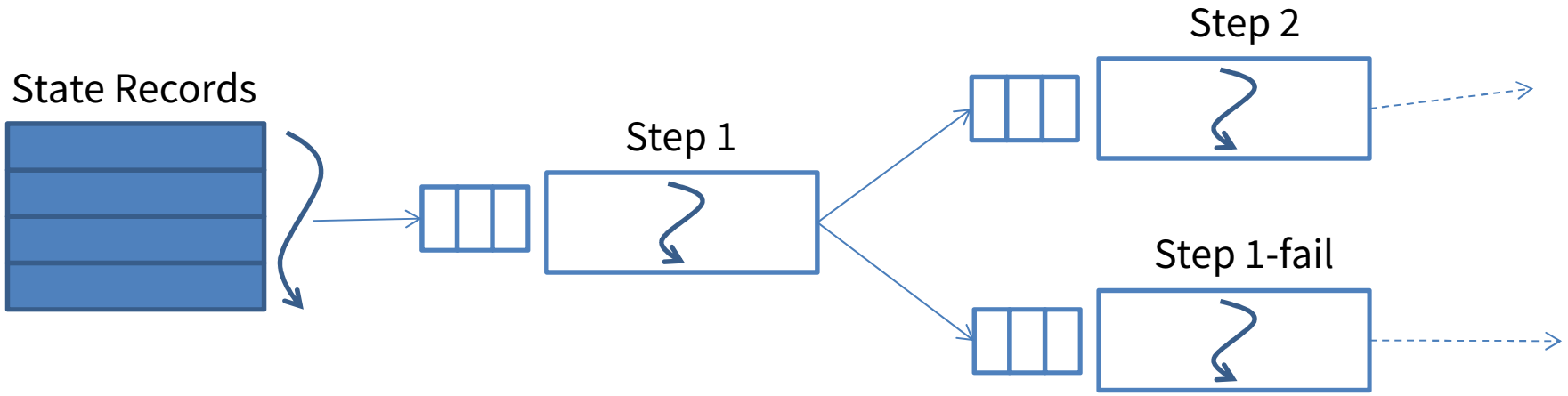
| # | Condition | Action |
|---|---|---|
| 1 | No backup server selected | Choose  available server to hold replica |
| 2 | Header unreplicated, no RPC outstanding | Start RPC containing the header |
| 3 | Header unreplicated, RPC completed | Mark header replicated; mark prior segment to allow footer 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**

# Structuring rules

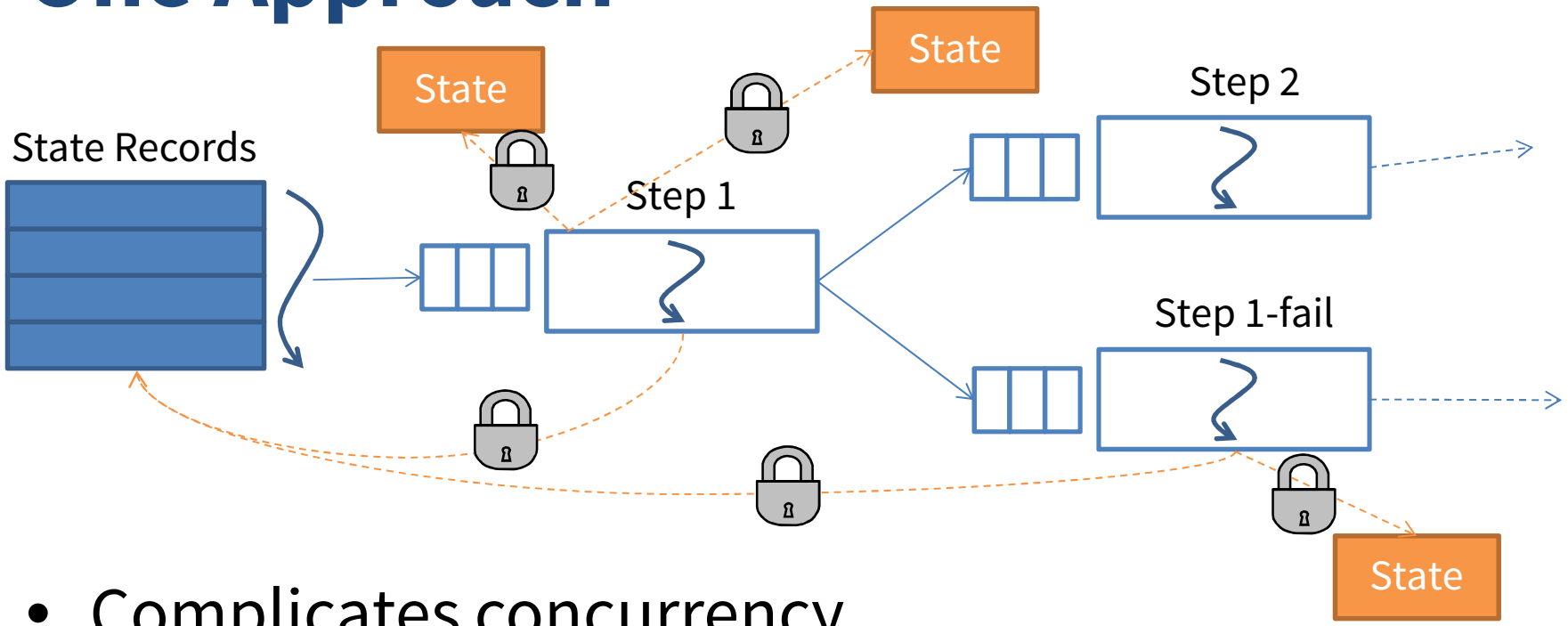
- How should hundreds of rules be organized?
  - Need modularity and clear visualization
- How can rules be evaluated efficiently?
  - Polling to test conditions for all rules won’t scale

# One Approach




- Used by HDFS chunk replication
- Structures paired with a thread that polls to find needed work due to failure
- State records queued between threads/actions
- No modularity; code scattered across threads

# One Approach




- Complicates concurrency
- CPU parallelism unneeded
- Need fine-grained locking on structures
- But must carefully perforated for liveness/responsiveness

# Tasks

- **Task:** Rules, state, and a goal
  - Similar to how monitors group locks with state
  - 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 3 complete replicas are made

# Pools

- **Pool:** Applies rules to tasks with unmet goals
- A pool for each independent set of tasks
- Serializes execution of rules in each pool
- Simplifies synchronization

# Benefits of Rules, Tasks, and Pools

- Directly determines code structure
- Eases local reasoning
  - Write a rule at a time
  - Condition clearly documents expected state
  - Actions are short with simple control flow
- Rules are amenable to model checking
  - But less restrictive than modeling languages
  - Flexible, easy to abuse with gross performance hacks and odd concurrency needs
- **Question:** fundamental or just a mismatch of kernel scheduling/concurrency abstractions to app?

# Conclusion

- Code pattern from our experiences: “**rules**”
  - Small steps whose order is based on state
  - Easy to adapt on failures
- PC doesn’t matter, only state matters
- In DCFT code non-linear flow is unavoidable
- Interesting question, how to structure rules
- This is one way, we’d love to hear others

# 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 just events?

- Rules take actions in 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

# 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 add causality tracking to log messages