Reliable State Machines: A Framework for Programming Reliable Cloud Services

Suvam Mukherjee
Microsoft Research, Bangalore, India
t-sumukh@microsoft.com

Nitin John Raj
International Institute of Information Technology, Hyderabad, India
nitinjohnraj@gmail.com

Krishnan Govindraj
Microsoft Research, Bangalore, India
t-krgov@microsoft.com

Pantazis Deligiannis
Microsoft Research, Redmond, USA
pdeligia@microsoft.com

Chandramouleswaran Ravichandran
Microsoft Azure, Redmond, USA
chanravi@microsoft.com

Akash Lal
Microsoft Research, Bangalore, India
akashl@microsoft.com

Aseem Rastogi
Microsoft Research, Bangalore, India
aseemr@microsoft.com

Raja Krishnaswamy
Microsoft Azure, Redmond, USA
rajak@microsoft.com

Abstract

Building reliable applications for the cloud is challenging because of unpredictable failures during a program's execution. This paper presents a programming framework, called *Reliable State Machines* (RSMs), that offers fault-tolerance by construction. In our framework, an application comprises several (possibly distributed) RSMs that communicate with each other via messages, much in the style of actor-based programming. Each RSM is fault-tolerant by design, thereby offering the illusion of being "always-alive". An RSM is guaranteed to process each input request exactly once, as one would expect in a failure-free environment. The RSM runtime automatically takes care of persisting state and rehydrating it on a failover. We present the core syntax and semantics of RSMs, along with a formal proof of *failure-transparency*. We provide a .NET implementation of the RSM framework for deploying services to Microsoft Azure. We carry out an extensive performance evaluation on micro-benchmarks to show that one can build high-throughput applications with RSMs. We also present a case study where we rewrite a significant part of a production cloud service using RSMs. The resulting service has simpler code and exhibits production-grade performance.

2012 ACM Subject Classification Software and its engineering → Software reliability; Software and its engineering → Cloud computing; Software and its engineering → Software fault tolerance

Keywords and phrases Fault tolerance, Cloud computing, Actor framework

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2019.18

Related Version A full version of the paper is available at `https://arxiv.org/abs/1902.09502`.

Acknowledgements We thank the anonymous reviewers for suggesting several ways to improve our work. Nitin John Raj's work was done, in part, during an internship at Microsoft Research, India.

33rd European Conference on Object-Oriented Programming (ECOOP 2019).
Editor: Alastair F. Donaldson; Article No. 18; pp. 18:1–18:29
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

The industry trend in Cloud Computing is increasingly moving towards companies building and renting *cloud services* to provide software solutions to their customers [19]. A cloud service in this context refers to a software application that runs on multiple machines in the cloud, making use of the available resources – both compute and storage – to offer a scalable service to its customers. In this paper, we consider the problem of programming *reliable, fault-tolerant* cloud services.

Cloud services are essentially distributed systems consisting of concurrently running, communicating processes or *agents*[1]. Agents typically maintain state, and process user requests as they arrive, which may cause their state to get updated. Consider a word-counting application: the application receives a stream of words (or strings) as input and continuously produces output in the form of the highest frequency word that it has received so far. Programming such an application for the single-machine scenario is easy – the application maintains a map from words to their frequencies seen so far. For each new word, it updates the map and outputs the word if it is the new highest frequency word.

However, to design a more scalable application, this map can be split across multiple distributed agents. More specifically, the distributed word count application can be designed as follows. A *main* agent receives input words from clients, and sends each word to one of the several *counting* agents (based on some criteria, such as the hash of a word) for processing. Every counting agent maintains its own word-frequency map and the local maximum; whenever the local maximum changes, it sends a message to the *max* agent. The max agent collates the local maxima from all the counting agents and outputs the global maximum.

A reliable cloud service must be resilient to hardware and software failures that can cause the agents to crash, and to network failures that can cause message duplications, reorderings, and drops. To handle crashes in the word-counting service, the programmer needs to use some form of persistent storage for the input stream and the word-frequency maps, and write boilerplate code to read and write this state, while carefully orchestrating it with the rest of the computation. The programmer must also handle network message drops (to avoid missing a word) and duplications (to avoid counting the same occurrence of a word twice). While some existing programming frameworks and languages for distributed systems, such as Orleans [12], Kafka [25], Akka [1], Azure Service Fabric [35], among others, provide the necessary building blocks of persistent storage, transactions, etc., the programmer still has to carefully put them all together. Thus, an application that is quite simple to program in the single-machine scenario, quickly becomes a non-trivial task in the distributed setting.

In this paper, we present a novel programming framework, called *Reliable State Machines* (RSMs), to program reliable, fault-tolerant cloud services. The RSM framework enables the programmer to focus only on the application-specific logic, while providing resilience against failures – both machines and network – through language design and runtime.

At a high-level, RSMs are based on the actor style of programming, where the unit of concurrency is a communicating state machine. The programmer defines the types of events that the RSM can receive, and handlers for each event type. Optionally, the programmer can declare some RSM-local state to be persistent. The event handlers can manipulate state, send messages to other RSMs, and create new RSMs. Issues of orchestrating reads and writes of the persistent state with event handlers, handling network failures, etc., are left to

[1] We use the term agents in this paper as a programming construct to distinguish from Systems constructs like processes or physical/virtual machines.

the RSM runtime. The runtime ensures that the effects of an event handler are committed atomically in an all-or-nothing fashion, making it appear that an RSM processes an input message *exactly once*. In addition, the runtime provides a networking module for *exact once* delivery of messages. RSMs are built on top of the P# framework [15], which provides convenient .NET syntax for programming state machines [31] and enables programmers to systematically test their applications against functional specifications [16]. Section 2 provides an overview of the RSM framework and the word-counting application written using RSMs.

We formalize the syntax and semantics of RSMs and prove a *failure transparency* theorem. The theorem states that the semantics of RSMs that includes runtime failures is a refinement of the failure-free semantics in terms of the observable behavior of an RSM. As a result, programmers can program and test their applications assuming failure-free semantics, while the failure transparency theorem guarantees the same behavior even in the presence of runtime failures. Section 3 contains details of our formalization.

We have developed two different implementations of our framework – one using the Azure Service Fabric platform [4] and the other using Apache Kafka [2, 25] – demonstrating that the basic concepts behind RSMs are general and can be implemented on different platforms (Section 4). Our evaluation (Section 6) shows that performance-wise RSMs are competitive with other production cloud programming frameworks, even with the additional guarantees of failure transparency, and it is possible to build high-throughput applications using RSMs. To evaluate the programming and testing experience, we present a case study where we re-implement an existing production-scale backend service of Microsoft Azure. We show that the RSM implementation of the service is simple, easier to reason about, amenable to systematic testing via the P# framework, and meets its scalability requirements (Section 5).

2 Overview

This section presents an overview of the RSM framework. We show how to program the word-count application with RSMs, followed by the details of the RSM runtime and failure transparency. In the rest of the paper, we use *events* and *messages* interchangeably.

2.1 Programming and testing the word-count application

As mentioned in Section 1, we design the distributed word-count application using three types of RSMs: (a) a main RSM that sets up other RSMs, receives words from the client, and forwards them to the word-count RSMs, (b) word-count RSMs that maintain the highest frequency word they have *individually* seen so far, and (c) a max RSM that aggregates local maxima from the word-count RSMs, and outputs the global maximum.

Listing 1 shows the source code for the main RSM using an abbreviated C#-like syntax. RSMs are programmed as state machines. The programmer first declares the three event types to use in the program: `WordEvent`, `WordFreqEvent`, and `InitEvent`, each carrying the mentioned payloads. Values of type **`rsmId`** (e.g., used in the payload of `InitEvent`) are RSM instance ids. We will explain the use of these events as we go along.

The main RSM has two states: `Init` is the start state, and `Receive` is the state in which it receives the input words. The machine declares two persistent fields: a `WordCountMachines` dictionary to maintain the **`rsmId`** of each word count RSM, and a `MaxMachineId` for the **`rsmId`** of the max machine. Fields declared with the "`Persistent`" types denote persistent local state of the RSM. In the `Init` state, the main RSM creates an instance of max RSM and `N` instances of word-count RSMs (using the **`create`** API), and sends the **`rsmId`** of the max RSM instance to every word count RSM as payload in the `InitEvent` event (using the **`send`**

```
event WordEvent: (word: string); // Event types with their payloads
event WordFreqEvent: (word: string, freq: int);
event InitEvent: (target: rsmId);

machine MainMachine {
  PersistentDictionary<rsmId, int> WordCountMachines; // Set of word count machines
  PersistentRegister<rsmId> MaxMachineId; // The rsmId of the aggregator machine

  start state Init { do Initialize }
  state Receive { on WordEvent do ForwardWord }

  void Initialize () {
    var max_id = create (MaxMachine); // First create the max machine
    store (MaxMachineId, max_id); // Store its rsmId
    for (var i = 0; i < N; ++i) { // Create the word count machines
      var id = create (WordCountMachine); store (WordCountMachine[id], 1);
      send (id, new InitEvent (max_id)); // Send max−machine's rsmId to each word count machine
    }
    jump (Receive); // Begin receiving words
  }

  rsmId GetTargetMachine (string s) { return load (WordCountMachines[hash(s) mod N]); }

  void ForwardWord (WordEvent e) { send (GetTargetMachine (e.word), e); } // Forward the event
}
```

Listing 1 Main RSM for the word count example.

API). The persistent fields are also updated (using `store`). The machine then transitions to the `Receive` state (using the `jump` API). In the `Receive` state, when the machine receives a `WordEvent` from the environment, which contains the next word, it forwards the word to the appropriate max count machine. Since the `Receive` state specifies no transitions, the RSM remains in the `Receive` state, ready to receive the next word.

Listing 2 shows the code for a word-count RSM. It maintains, in its persistent state, a running map of word frequencies (`WordFreq`) and the highest frequency (`HighFreq`) that it has seen so far. Whenever the highest frequency changes, it forwards the corresponding word to the max machine, using the `rsmId` stored in the `TargetMachine` field. This RSM also shows the use of volatile state in the form of the field `WordsSeenSinceLastCrash`; this field is reset every time the RSM fails. Such variables can be used for gathering information such as program statistics that are not required to survive failures. Note that the execution of each handler (its call stack, all local variables, etc.) is also carried out on volatile memory.

A word count machine has two states: `Init` and `DoCount`. In the `Init` state, it waits for the `InitEvent` (from the main machine). The rest of the code is straightforward.

The max RSM, shown in Listing 3, simply takes a maximum over the frequencies that it receives, and forwards the maximum one to an external service (which may print to console or write to an output file).

Implementation. The RSM programming framework is embedded in `C#`, and uses the `P#` state machine programming model. Each RSM is defined as a `C#` class, with local-state as class fields, and event handlers as class methods. Using RSMs does not require the user to learn a new programming language. We provide more implementation details in Section 4.

Testing the application. Having written the application in our framework, the programmer can also test it by supplying a specification and asking the `P#` tester to validate it. In the word count application, for example, a functional correctness specification – eventually the word with the highest frequency is output by the `MaxMachine` RSM – can be tested. The `P#` tester uses state-of-the-art algorithms to search over the space of possible executions of an RSM program [18, 16] and can help catch many bugs. For instance, changing any of the

```
machine WordCountMachine {
  PersistentDictionary<string, int> WordFreq; // Local map for words to their frequencies
  PersistentRegister<int> HighFreq; // The highest frequency seen so far
  PersistentRegister<rsmId> TargetMachine; // The max machine rsmId, forwarded by the main machine
  int WordsSeenSinceLastCrash; // A volatile variable to count words seen since last crash

  start state Init { on InitEvent do Initialize }
  state DoCount { on WordEvent do Count }

  void Initialize (InitEvent e) { // Wait for the init event from the main machine
    store (TargetMachine, e.target);
    jump (DoCount);
  }

  void Count (WordEvent e) { // Receive the word from the main machine
    WordsSeenSinceLastCrash++;
    var f = load (WordFreq[e.word]) + 1; // Increment the frequency of the word by 1
    store (WordFreq[e.word], f);  // And store it back
    if (f > load (HighFreq)) { // Update the highest frequency, if required
      store (HighFreq, f);
      send (load (TargetMachine), new WordFreqEvent (e.word, f)); // And send it to the max machine
    }
  }
}
```

Listing 2 Word count RSM for the word count example.

```
machine MaxMachine {
  PersistentRegister<int> HighFreq; // Highest frequency seen so far

  state DoCount { on WordFreqEvent do CheckMax }

  void CheckMax (WordFreqEvent e) { // Update the current highest frequency if needed
    if (e.freq > load (HighFreq)) {
      store (HighFreq, e.freq);
      send (env, e);
    }
  }
}
```

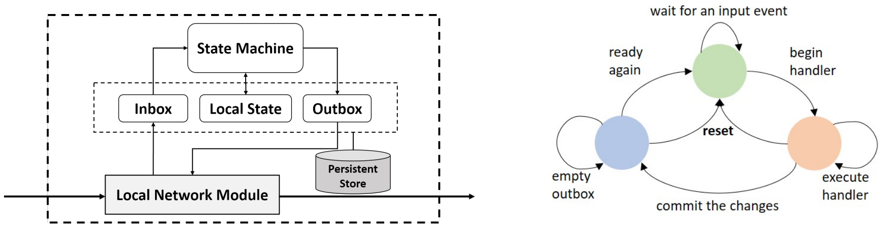
Listing 3 max RSM for the word count example.

persistent variables of the RSMs to volatile will render the program incorrect; indeed if the `MaxMachine`s don't persist their word frequency maps, upon restart, their output may not be correct. If the `MainMachine` does not use the same hash function inside `GetTargetMachine` for all input words, then too the specification fails to hold (because it may forward two different occurrences of the same word to two different RSMs), etc. We confirmed that the `P#` tester is able to find all these errors very quickly.

Summary. Our framework frees the programmer from the burden of designing and programming for failures. In the word count application, as we can see above, the source code *only* contains the application-specific logic, and no boilerplate code for handling failures, restarts, etc. There is still concurrency in the program that may be hard to reason about, which is why we provide `P#` testing. The next section describes the RSM runtime that provides resilience from machine and network failures.

2.2 RSM runtime

Figure 1a shows the runtime architecture of a single RSM. The runtime ensures that each RSM has a unique `rsmId`. An RSM is associated with its own *inbox* of input events, an *outbox* of output events (the events that it sends out), and local state that consists of both persistent and volatile (in-memory) components. The inbox, outbox, persistent fields, and the current state of the RSM state machine (e.g. `Init` or `Receive` in Listing 1) are backed by a persistent store (e.g. a replicated storage system). Each RSM also has a local networking module that



(a) Internals of an RSM.

(b) Flow of operations for an RSM.

Figure 1 Internals and flow of operations for an RSM.

is responsible for communicating with other RSMs or to clients or external services. The inbox and outbox are queues, following the standard FIFO enqueue and dequeue semantics.

The execution of an RSM consists of three operations.

- **Input.** The networking module receives messages over the network and enqueues them to the inbox.
- **Processing.** The processing inside an RSM is single-threaded. It iteratively dequeues an event from the inbox and processes it by executing its corresponding event handler. The handler can create other RSMs or send events to the existing ones. Each of these requests are enqueued to the outbox. The handler can also mutate the persistent and volatile local state of the RSM.
- **Output.** The networking module dequeues messages from the outbox and sends them over the network to their destination.

These operations can execute in any order. In our implementation of RSMs (Section 4), we run them in parallel using background tasks; we ensure that the enqueue and dequeue operations on the queues (inbox and outbox) are linearizable [22], and thus, safe to execute concurrently.

Exact-once processing. The RSM runtime ensures that the effects of an event handler are committed to the persistent storage atomically. In particular, the dequeue of an event e from the inbox, and the result of processing e (including all updates made to persistent fields as well as all enqueues to the outbox) are committed to the persistent storage in a single transaction. Thus, if the RSM fails before committing, then on restarting the RSM, e would still be at the head of the inbox, and none of its effects would have been propagated to the rest of the system. If the RSM fails after committing, then the event e has been processed and will not appear in the inbox on restart. The RSM only sends out those events that have been committed successfully to the outbox.

Networking module. The networking modules work with each other to ensure exact-once delivery of events between RSMs, i.e., an event is dequeued from the outbox of an RSM and enqueued to the inbox of the target RSM atomically. While exact-once delivery is default, the programmer can choose more relaxed delivery semantics. All our examples (Section 4) use the stricter exact-once implementation. To communicate with external non-RSM services, the RSM framework has the notion of an *environment* that acts as an interface to the outside world. The environment can supply input by enqueueing to the inbox of an RSM. The RSMs can in turn send events to a special `rsmId` called `env`, which references the environment. Such events still get enqueued to the outbox of the RSM. When committed, they are forwarded to their intended destination through plug-ins to the networking module supplied by the user.

Non-determinism. We allow RSM handlers to be non-deterministic, i.e., two executions of an event handler on the same event and starting from the same local state may produce different output. For instance, consider an extension to the word-count example where each input word is associated with a timestamp and the main-RSM forwards only those words with a timestamp not older than 24 hours. This requires main-RSM to look up the current time of day and make the decision of forwarding the word or not. This action is non-deterministic because it may not replay exactly on failover. Non-determinism does not change the RSM guarantees in any way: all state changes made by an event handler are first committed locally. This ensures that all non-deterministic choices are resolved and recorded before they are propagated outside the RSM.

Progress. The RSM runtime ensure global progress under the assumptions that: (i) each handler terminates in the absence of failures, (ii) for each handler, the system eventually recovers from failures in order to complete its execution, and (iii) a message that is repeatedly sent over the network is eventually delivered to its destination.

Using `P#` for testing. We chose `P#` for two reasons. First, it provides various programming conveniences for writing state machines and is already in use for writing production code [31, 17]. Second, `P#` offers means of writing end-to-end specifications of a collection of communicating state machines. The specifications (both safety and liveness) can then be validated using powerful systematic search over the space of all interleavings of the program. This method has been shown to be very effective at finding concurrency bugs [18, 15, 16, 28]. In our work, we provide an automatic way of lowering an RSM program to a `P#` program. A programmer can write the specification of an RSM program, then validate the specification using `P#` systematic testing. We provide more details in Section 6.2.

Failure transparency. Using the exact-once processing and exact-once delivery, the RSM framework provides a failure transparency property. The property essentially says that the observable behavior of an RSM is independent of the failures of host machines and the network. This enables the programmers to focus only on the application-specific logic when programming RSMs, and to test only for the failure-free executions. The property relies on the non-interference of the persistent storage from the volatile class fields. Intuitively, the volatile class fields are reset on failures, and so, if they leak into event payloads, for example, the crashes can be observed.

3 Formalization of RSMs

In this section, we formalize a core of the RSM programming model, denoted as R_{SM}, and its operational semantics. We state and prove the *failure transparency* theorem in Section 3.2.

3.1 Syntax and semantics

Figure 2 shows the R_{SM} syntax. For simplicity, we present the syntax in A-normal form [33], where most of the sub-expressions are values. An RSM C in R_{SM} is declared as a class definition D, consisting of `persistent` and `volatile` fields, and an event handler statement $\{\overline{x := n}; s\}$, with local variables $\overline{x}$ and statement s (handlers for specific event types can be encoded in R_{SM} using `if` statements in s). All the variables and fields in R_{SM} are integer-typed.

Statements in the language include local variable assignment ($x := e$), assignment to `volatile` fields ($f := e$), `persistent` field updates (`store` f v), conditional statements

$$
\begin{array}{rlll}
\text{Field} & f & & \text{Class name} \quad C \qquad \text{Integer} \quad n, r \\
\text{Value} & v & ::= & n \mid x \mid f \\
\text{Expression} & e & ::= & v \mid \texttt{load}\ f \mid v_1 \oplus v_2 \mid \star \\
\text{Statement} & s & ::= & x := e \mid f := e \mid \texttt{store}\ f\ v \mid \texttt{if}\ v\ s_1\ s_2 \mid s_1; s_2 \mid \texttt{create}\ x\ C \mid \texttt{send}\ v_1\ v_2\ v_3 \\
\text{Class defn.} & D & ::= & \texttt{class}\ C\ \{\overline{\texttt{persistent}\ f := n}; \overline{\texttt{volatile}\ f := n}; \{\overline{x := n}; s\}\}
\end{array}
$$

Figure 2 R_{SM} syntax.

$$
\begin{array}{rlll}
\text{Field map} & F & ::= & \cdot \mid f \mapsto n, F \\
\text{Local environment} & L & ::= & \cdot \mid x \mapsto n, L \\
\text{Event list} & E & ::= & \cdot \mid (n_r, n_e, n_p), E
\end{array}
$$

Figure 3 Runtime configuration syntax for local evaluation.

($\texttt{if}\ v\ s_1\ s_2$) and sequencing ($s_1; s_2$). While the **`volatile`** fields can be operated upon directly (e.g. adding two of them), to work on the **`persistent`** fields, they first need to be **`load`**ed into local variables, and then **`store`**d back. The form **`create`** x C creates a new RSM C and binds its RSM id to the variable x (the ids are also integer-valued). Finally the statement form **`send`** v_1 v_2 v_3 is used to send an event of event type v_2 (an integer) with payload v_3 to the destination machine with RSM id v_1. Expressions e in the language include values v, reading a **`persistent`** field (**`load`** f), and binary operations $v_1 \oplus v_2$. We also model non-determinism in the language – the expression form $\star$ evaluates to a random integer at runtime.

3.1.1 Local evaluation judgment

Operational semantics of R_{SM} consists of two judgments, a local evaluation judgment for reducing the event handler statement to process an event, and a global judgment where the configuration consists of all the RSMs executing concurrently. We first present the local evaluation judgment.

Local evaluation judgments are of the form $E; F; L; s \to E_1; F_1; L_1; s_1$, where the syntax

$$\boxed{F; L \vdash e \Downarrow n} \qquad \boxed{E; F; L; s \to E_1; F_1; L_1; s_1}$$

$$
\frac{}{F; L \vdash x \Downarrow L[x]}\ \textsc{E-var}
\qquad
\frac{}{F; L \vdash f \Downarrow F[f]}\ \textsc{E-volatile}
\qquad
\frac{}{F; L \vdash \texttt{load}\ f \Downarrow F[f]}\ \textsc{E-persistent}
$$

$$
\frac{\begin{array}{c} F; L \vdash v_i \Downarrow n_i \\ n = n_1 \oplus n_2 \end{array}}{F; L \vdash v_1 \oplus v_2 \Downarrow n}\ \textsc{E-binop}
\qquad
\frac{}{F; L \vdash \star \Downarrow n}\ \textsc{E-star}
$$

$$
\frac{F; L \vdash e \Downarrow n}{E; F; L; \texttt{store}\ f\ e \to E; F[f \mapsto n]; L; \texttt{skip}}\ \textsc{L-store}
\qquad
\frac{F; L \vdash v \Downarrow n \quad (n \neq 0 \Rightarrow s = s_1) \wedge (n = 0 \Rightarrow s = s_2)}{E; F; L; \texttt{if}\ v\ s_1\ s_2 \to E; F; L; s}\ \textsc{L-if}
$$

$$
\frac{\mathsf{fresh}\ n_r \qquad E_1 = (n_r, n_C, 0), E}{E; F; L; \texttt{create}\ x\ C \to E_1; F; L[x \mapsto n_r]; \texttt{skip}}\ \textsc{L-create}
\qquad
\frac{F; L \vdash v_i \Downarrow n_i \qquad E_1 = (n_1, n_2, n_3), E}{E; F; L; \texttt{send}\ v_1\ v_2\ v_3 \to E_1; F; L; \texttt{skip}}\ \textsc{L-send}
$$

Figure 4 R_{SM} local semantics.

for E, F, and L is shown in Figure 3. F and L are field map and local environment, mapping fields and local variables to values. L contains three special variables x_s, x_e, and x_p that map to the source RSM, event type, and the payload of the *current* event that is being processed; these fields are initialized in the global judgment.

E is a list of output events. An event is a triple of the form (r, n_e, n_p), where r is the destination RSM id, n_e is the event type, and n_p is the event payload. Notably F, L, and E are all non-persistent. Their interaction with the persistent state happens in the global judgment. Statement reduction uses an auxiliary expression evaluation judgment of the form $F; L \vdash e \Downarrow n$. Statements at runtime include an additional `skip` form to denote the terminal statement.

Figure 4 shows the selected rules for statement reduction and expression evaluation. The expression rules are all standard, notably rule E-STAR non-deterministically evaluates the $\star$ expression to some integer n. Most of the statement reduction rules are also standard. For example, rule L-STORE uses the expression evaluation form to evaluate e, and stores the result in the field map F. Rule L-IF branches based on the evaluated value of v. Rule L-CREATE simply records the creation request in the output events list with a special event type n_C. Finally, rule L-SEND evaluates each of the **send** arguments, and updates the output event list E.

3.1.2 Global evaluation judgment

Global evaluation judgment has the form $S \vdash M; \Pi \longrightarrow M_1; \Pi_1$. M and Π are maps with RSM ids as domains. The map M maps the RSM ids to local configurations $E; F; L; s; b$, where E, F, L, and s come from the local judgment, and b is a (volatile) bit that is 1 if the machine is currently processing an event or 0 otherwise. We will also write F_p and F_v to denote the F map components for **persistent** and **volatile** fields respectively. The map Π maps each RSM id to its class C and persistent storage, i.e. $C; I; O; P; T$, where I is the inbox persisting the incoming events, O is the outbox persisting the outgoing events, P is the persistent fields map, and T is the *trace* of the RSM that records its observable behavior; the trace T is ghost and is only used to state and prove the failure transparency theorem. The grammar for I, O, and T is same as that of the event list E, while persistent field map P is a field map like F. Finally, S is the signature that maps class C to its definition.

As shown in Figure 1b, each RSM (a) reads an event from its input queue, (b) processes it using its handler statement, (c) commits the events generated and the persistent field map in its persistent store, (d) empties the outbox in the persistent store, and starts from (a) again. At each of these steps, the machine can crash and recover, where all of its non-persistent data (including the local state E, F, L) is lost. The global semantics essentially implements this state machine for each RSM, while executing the RSMs concurrently with each other.

Figure 5 shows the global semantics judgment. In all the rules, one of the machines r takes the step. Using the Rule G-START, a machine r enters the event handler for processing the head event in the input event queue. The local state of the machine currently is *at rest*, i.e. $M(r).s =$ `skip` and $M(r).b = 0$, as well as the outbox $\Pi(r).O$ is empty. The rule creates the local environment L (using the `initL` auxiliary function, shown in the same figure), by initializing the local variables as per the RSM definition $S(C)$, and also adding the mappings for event source, event type and event payload (x_s, x_e and x_p). The local state of the machine is changed to process the handler statement s and the bit b is set to 1. The persistent store Π is left unchanged.

Rule G-LOCAL shows the local evaluation rule, where a machine r takes a local step by executing the event handler. The rule uses the local semantics judgment in the premise, and updates $M(r)$ accordingly.

$$\textsc{G-start}\quad \begin{array}{c} M(r) = \cdot; F; \_; \texttt{skip}; 0 \quad \Pi(r) = C; \_, (n_s, n_e, n_p); \cdot; P; \cdot \\ F = F_p \cup F_v \quad F_p = P \\ L = \mathsf{initL}(C, n_s, n_e, n_p) \quad s = \mathsf{handler}(C) \\ \hline S \vdash M; \Pi \longrightarrow M[r \mapsto \cdot; F; L; s; 1]; \Pi \end{array}$$

$$\textsc{G-local}\quad \begin{array}{c} M(r) = E; F; L; s; 1 \quad \Pi(r).O = \cdot \\ E; F; L; s \rightarrow E_1; F_1; L_1; s_1 \\ M_1 = M[r \mapsto E_1; F_1; L_1; s_1; 1] \\ \hline S \vdash M; \Pi \longrightarrow M_1; \Pi \end{array}$$

$$\textsc{G-commit}\quad \begin{array}{c} M(r) = E; F_p \cup F_v; L; \texttt{skip}; 1 \quad \Pi(r) = C; I, \_; \cdot; \_; T \\ M_1 = M[r \mapsto \cdot; F_p \cup F_v; L; \texttt{skip}; 0] \\ \hline S \vdash M; \Pi \longrightarrow M_1; \Pi[r \mapsto C; I; E; F_p; T] \end{array}$$

$$\textsc{G-create}\quad \begin{array}{c} M(r) = \_; \_; \_; \texttt{skip}; 0 \\ \Pi(r).O = \_, (r_1, n_C, \_) \\ \hline S \vdash M; \Pi \longrightarrow M; \mathsf{create}(r, r_1, C, \Pi) \end{array}$$

$$\textsc{G-send}\quad \begin{array}{c} M(r) = \_; \_; \_; \texttt{skip}; 0 \quad \Pi(r).O = \_, (r_1, n_e, n_p) \\ \hline S \vdash M; \Pi \longrightarrow M; \mathsf{send}(r, r_1, n_e, n_p, \Pi) \end{array}$$

$$\textsc{G-reset}\quad \begin{array}{c} \Pi(r) = C; \_; \_; P; \_ \\ M_1 = M[r \mapsto \cdot; \mathsf{resetF}(C, P); \cdot; \texttt{skip}; 0] \\ \hline S \vdash M; \Pi \longrightarrow M_1; \Pi \end{array}$$

$$\begin{array}{lcl}
\mathsf{initL}(C, n_s, n_e, n_p) & = & \texttt{let } S(C) = \texttt{class } C\ \{\_; \_; \{\overline{x := n}; \_\}\}\ \texttt{in} \\
 & & (\overline{x \mapsto n}, x_S \mapsto n_s, x_e \mapsto n_e, x_p \mapsto n_p) \\
\mathsf{handler}(C) & = & \texttt{let } S(C) = \texttt{class } C\ \{\_; \_; \{\_; s\}\}\ \texttt{in } s \\
\mathsf{create}(r, r_1, C, \Pi) & = & \texttt{let } S(C) = \texttt{class } C\ \{\overline{\texttt{persistent } f := n}; \_; \_\}\ \texttt{in} \\
 & & \texttt{let } \Pi(r) = C_r; I; O; P; T\ \texttt{in} \\
 & & \texttt{let } \Pi_1 = \Pi[r \mapsto C_r; I; \mathsf{tail}\ O; P; (r_1, n_C, 0), T]\ \texttt{in} \\
 & & \Pi_1[r_1 \mapsto C; \cdot; \cdot; \overline{f \mapsto n}; \cdot] \\
\mathsf{send}(r, r_1, n_e, n_p, \Pi) & = & \texttt{let } \Pi(r) = C; I; O; P; T\ \texttt{in let } \Pi(r_1) = C_1; I_1; O_1; P_1; T_1\ \texttt{in} \\
 & & \texttt{let } \Pi_1 = \Pi[r \mapsto C; I; \mathsf{tail}\ O; P; (r_1, n_e, n_p), T]\ \texttt{in} \\
 & & \Pi_1[r_1 \mapsto C_1; (r, n_e, n_p), I_1; O_1; P_1; T_1] \\
\mathsf{resetF}(C, P) & = & \texttt{let } S(C) = \texttt{class } C\ \{\_; \overline{\texttt{volatile } f := n}; \_\}\ \texttt{in } P, \overline{f \mapsto n}
\end{array}$$

Figure 5 R_{SM} global semantics.

Once a machine r has finished executing the event handler for the head input event, it uses the rule G-COMMIT to commit the persistent state. In the rule, the local state of the machine has reached the end of handler execution ($M(r).s = \texttt{skip}$ and $M(r).b = 1$). $M(r)$ is changed by setting the bit b to 0 and the local event list is reset to empty. The changes to $\Pi(r)$ are: (a) the head event is removed from $\Pi(r).I$, (b) the output event list E from the local state is committed to the outbox $\Pi(r).O$, and (c) the new values of the `persistent` variables from the local state are committed to $\Pi(r).P$. The (ghost) trace of the machine $\Pi(r).T$ remains unchanged; the machine next proceeds to send the events out of the outbox, and append the trace accordingly.

Rule G-CREATE handles the create event (rule L-CREATE, Figure 4). The auxiliary function create updates the persistent store Π. For the creator machine r, it removes the create event from the outbox $\Pi(r).O$, and adds it to the ghost trace $\Pi(r).T$. For the new machine r_1, it initializes the persistent store by reading off the initial persistent variables map from the signature $S(C)$. Rule G-SEND sends an event from machine r to r_1. The auxiliary function send removes the event from the outbox of r, and adds it to the ghost trace, as well as to the inbox of r_1. The rule models the exact-once delivery network module.

Finally, a machine r can fail at any point in the execution. The rule G-RESET models the machine reset. As expected, upon reset, the local volatile state, including the event list E, `volatile` variables, environment L, are all lost. The fields map in the local state is re-initialized (using resetF) by reading off the `persistent` variables from $\Pi(r)$ and `volatile` variables from the signature $S(C)$. The bit b is also set to 0. We next present our main theorem of failure transparency.

3.2 Failure transparency

To state the theorem, we first define a notion of equivalence for local states $M(r)$. Below, r is an RSM id.

▶ **Definition 3.1** (Equivalence of local states). *Two local states, $M_1(r)$ and $M_2(r)$ are equivalent, written as $M_1(r) \cong M_2(r)$, if they are equal in all components, except for the volatile class fields in their field maps, i.e. $M_1(r).E = M_2(r).E$, $M_1(r).F_p = M_2(r).F_p$, $M_1(r).L = M_2(r).L$, $M_1(r).s = M_2(r).s$, and $M_1(r).b = M_2(r).b$.*

Our failure transparency theorm relies on non-interference of persistent state from volatile fields. We formally state the property below (we use $\longrightarrow_r$ to denote the machine r taking a step):

▶ **Property 3.2** (Non-interference). *Let $M; \Pi \longrightarrow_r^* M_1; \Pi$ be a run, s.t. each step in the run is a* G-LOCAL *step taken by machine r, and M_1 is terminal (i.e. $M_1(r).s =$ `skip`). Then, $\forall M'.\ M'(r) \cong M(r)$, there exists M'_1 s.t. $M'; \Pi \longrightarrow_r^* M'_1; \Pi$ where $M'_1(r) \cong M_1(r)$ and each step is a* G-LOCAL *step.*

In [29], we present an information-flow type system for R_{SM} that provides this non-interference property for well-typed programs. Note that non-determinism in our language does not raise any complications, since to get this property, we can essentially *replay* the non-deterministic choices from the run in the premise to the run in the conclusion.

Given Property 3.2, we are now ready to state the failure transparency theorem. We consider a run of a machine that processes an event end-to-end. We prove that, given any such run that includes failures (i.e. the rule G-RESET), we can construct a run without failures, but with same observable traces T.

▶ **Theorem 3.3** (Failure transparency). *Let $M; \Pi \longrightarrow_r^* M_p; \Pi_p \longrightarrow_r M_c; \Pi_c \longrightarrow_r^* M_1; \Pi_1$, where $M; \Pi$ is ready for a machine r (i.e. it satisfies the premises of the* G-START *rule), and*

1. *all steps in $M; \Pi \longrightarrow_r^* M_p; \Pi_p$ are either* G-START, G-LOCAL, *or* G-RESET,
2. *$M_p; \Pi_p \longrightarrow_r M_c; \Pi_c$ is a* G-COMMIT *step, and*
3. *all steps in $M_c; \Pi_c \longrightarrow_r^* M_1; \Pi_1$ are either* G-CREATE, G-SEND, *or* G-RESET

Then, $\forall M'.M'(r) \cong M(r)$, there exists M'_1 s.t.

(a) $M'; \Pi \longrightarrow_r^* M'_1; \Pi_1$,
(b) *none of the steps in (a) are* G-RESET, *and*
(c) $M'_1(r) \cong M_1(r)$

Crucially, Π_1, and hence the trace of machine r remains same in the conclusion of the theorem. Thus, we prove that the machine run with failures is a refinement of the machine run without failures with respect to its observable behavior.

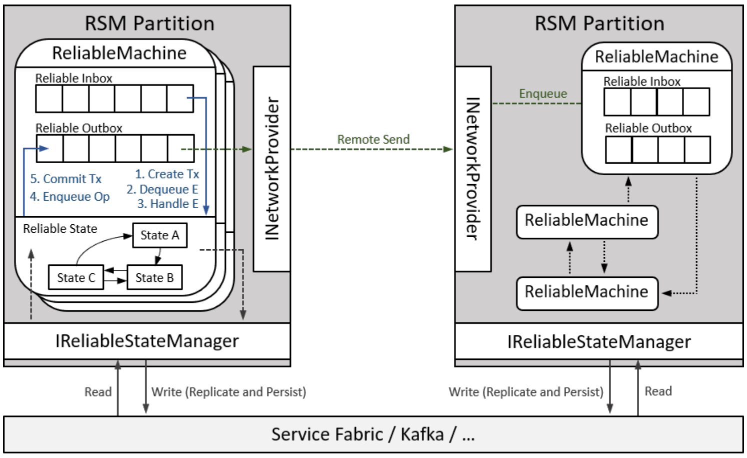


Figure 6 The Reliable State Machines implementation.

4 Implementation

This section describes an instantiation of RSMs as a .NET object-oriented programming framework. The framework is split into two logical parts: the *frontend* and the *backend*. The frontend implements the programmer-facing APIs while the backend is responsible for the distributed-system aspects, including state persistence and inter-machine communication. An illustration of the RSM architecture is shown in Figure 6.

The frontend exposes an `RSM.ReliableMachine` base class. An RSM is programmed as a class that derives from `ReliableMachine`. An RSM instance is an object of such a class and event handlers are implemented as methods of the class. The base class implements the functionality to drive a state machine. The state machine structure is based on `P#`, similar to the word-count code shown in Section 2. We focus the discussion here on the reliability aspects of RSMs. The frontend also provides a runtime, `RSM.ReliableMachineRuntime`, that implements the APIs for creating RSMs and sending messages between them. Each RSM carries a reference to the runtime in order to invoke these APIs. The runtime is also responsible for `rsmId` management, ensuring that each RSM is associated with a unique id throughout its lifetime.

The frontend provides two generic types for declaring local persistent state of an RSM: `RSM.PersistentRegister<T>` and `RSM.PersistentDictionary<TKey,TValue>`. The former implements a `Get-Put` interface for getting access to the underlying `T` object, similar to the `load` and `store` semantics of our formal language. The object is automatically serialized (on `Put`) and deserialized (on `Get`) in the background.[2] The `PersistentDictionary` type is similar, although it additionally allows access to individual keys. This has the advantage that if an RSM handler only accesses a few keys, then only those keys (and their corresponding values) are serialized and stored, without having to serialize the entire dictionary.

The programmer can declare fields inside an RSM class with these `Persistent` types to get access to the persistent local state. Any other fields in the class are treated with volatile semantics. The current state of the state machine is maintained in a `PersistentRegister` so that the RSM resumes operation from the correct state on failover.

[2] We use the `protobuf-net` serializer in RSMs, although other mechanisms are possible.

RSMs, once created, continually listen to incoming messages, until they are explicitly halted. `ReliableMachine` exposes an option of halting the RSM. The runtime reclaims all resources associated with an RSM when it halts.

The runtime works against `RSM.IReliableStateManager` and `RSM.INetworkProvider` interfaces, each of which are implemented by the backend. The `IReliableStateManager` interface is responsible for creating the inbox and outbox queues, as well as to back the persistent fields of an RSM. The `INetworkProvider` interface allows communication between RSMs. We provide two backend implementations: one using Azure Service Fabric (Sections 4.1 and 4.2) and the other one using Apache Kafka (Section 4.3). We additionally provide a `P#`-based backend for the purpose of high-coverage systematic testing (Section 4.4).

4.1 Azure Service Fabric backend

Background. Azure Service Fabric (SF) [4] provides infrastructure for designing and deploying distributed services on Azure. A user begins by setting up an SF cluster on a required number of Azure VMs. SF sets up a replicated on-disk storage system on the cluster. An application deployed to an SF cluster benefits from having access to co-located storage, instead of having to access a remote storage system. The store uses primary-secondary-based replication. The user can choose a replication factor (say R) in which case each update to the store is applied to R replicas, with each replica located on a different machine. Updates are only allowed on the primary, after which they are propagated to the secondaries.

SF provides various means of programming a service for deployment to an SF cluster. The most relevant to our discussion is a stateful application called *reliable services* [8]. Such an application consists of multiple *partitions* [6]; each partition roughly resembles an individual process constituting the failure domain for the application. Each partition is associated with its own primary and $R - 1$ secondaries. The partition's process is co-located with the primary. (Thus, an application with N partitions will have a total of N primaries and $RN - N$ secondaries, distributed evenly across the SF cluster.) From the programmer's perspective, each partition gets its own `StateManager` [9] object that provides access to its store. When a machine carrying a primary fails, one of its secondaries is promoted to become a primary and the corresponding partition is re-started on the new primary. A new secondary is elected and brought up to date in the background. Thus, a machine failure results in restarting of any partition located on it, but all data written to their `StateManager` is still available on restart.

The SF `StateManager` provides APIs for transactional access to storage [7]. A user can create a transaction, use it to perform reads and writes to the store, and then commit it. SF transactions have the database ACID semantics [21], i.e., they are atomic, consistent, isolated, and durable with respect to other transactions. As a form of convenience, the user can access the store via a dictionary interface (`IReliableDictionary`) and a queue interface (`IReliableQueue`). These interfaces are shown in Listing 4. (We qualify the SF interfaces with `SF` and the RSM types with `RSM` to avoid any confusion.) The `SF.IReliableQueue` interface, for example, supports enqueue and dequeue operations, each of which require the associated transaction. (These are awaitable `C#` methods [3], hence the return type `Task`.) These operations appear to take place (with respect to other transactions) only when their associated transaction is committed. A transaction can span multiple of these reliable collections. The method `DictionaryToQueueAtomicTransfer` in Listing 4 illustrates an atomic transfer of a value from a dictionary to a queue: it reads from a dictionary and writes to the queue in the same transaction.

```
interface SF.IReliableDictionary<TKey, TValue> {
  Task SetAsync(SF.ITransaction, TKey, TValue);
  Task<ConditionalValue<TValue>> TryGetValueAsync(SF.ITransaction, TKey);
}

interface SF.IReliableQueue<T> {
  Task EnqueueAsync(SF.ITransaction, T);
  Task<ConditionalValue<T>> TryDequeueAsync(SF.ITransaction);
}

async void DictionaryToQueueAtomicTransfer(SF.IReliableDictionary<int, int> D,
SF.IReliableQueue<int> Q)
{
   int key = ...
   using (var tx = StateManager.CreateTransaction())
   {
     var v = await D.TryGetValueAsync(tx, key);
     if(v.HasValue) {
       await Q.EnqueueAsync(tx, v.Value);
     }
     await tx.CommitAsync();
   }
}
```

Listing 4 Reliable collection interfaces of service fabric (shown partially) with sample usage.

RSM backend. We can now describe a vanilla implementation of RSMs using SF. Various optimizations are described in Section 4.2. An RSM program deploys as a stateful service on an SF cluster. A single partition contains exactly one instance of `RSM.ReliableMachineRuntime` that may host any number of RSM instances. `RSM.IReliableStateManager` is implemented as a wrapper of the SF `StateManager` and `RSM.INetworkProvider` is implemented using the SF remoting library for RPC communication [5].

The runtime remembers all hosted RSM instances in a persistent dictionary of the type `SF.IReliableDictionary<rsmId, bool>`. When a partition comes up (or fails over), it creates a new runtime, which then immediately reads this dictionary to identify the set of RSMs that it had hosted before failure (if any). It then re-creates the RSMs with the same ids. All persistent state associated with an RSM is attached to its id so that an RSM can rehydrate its state on failover as long as it retains its id.

The types `RSM.PersistentDictionary` and `RSM.PersistentRegister` are implemented as wrappers of `SF.IReliableDictionary`. The RSM types hide SF transactions from the programmer. The inbox and outbox are just SF reliable queues (`SF.IReliableQueue`). An RSM executes as an event-handling loop. Each iteration of the loop constructs an SF transaction (say, `Tx`) and performs a dequeue on the inbox using the transaction. If it finds that the queue is empty, the loop terminates and is woken up later only when a message arrives to the RSM. (This ensures that the RSM takes no compute resources when it has no work to perform.) If a message is found in the inbox, then the RSM goes on to execute the corresponding handler. Any access made by the handler to a persistent field gets attached with the same transaction `Tx`. Sending a message m to an RSM r is performed as an enqueue of the pair (m, r) to the outbox queue, also on the same transaction `Tx`. When the handler finishes execution, the RSM commits `Tx` and repeats the loop to process other messages in the inbox. Using the same transaction throughout the lifetime of a handler ensures that all effects of processing a message happen atomically with the dequeue of that message.

Networking and exact-once delivery. RSMs have two additional background tasks: the first one is responsible for emptying the outbox, and the other one listens on the network for incoming messages to add them to the inbox. These tasks are spawned on-demand as work arrives in order to avoid unnecessary polling. These tasks co-operate to ensure exact-once delivery between RSMs, even under network failures or delays (as long as the connection is eventually established).

```
do:
  create transaction tx1
  (m, r2) = Outbox.Dequeue(tx1);
  c = SendCounter[r2].Get(tx1);
  SendCounter[r2].Put(c + 1, tx1);
  do:
     send (m, c, r1) to r2
  repeat until an ack is received within
       timeout
  commit tx1
repeat forever
```

Listing 5 Outbox draining task for RSM $r1$.

```
On receiving (m, c, r1):
  create transaction tx2
  d = ReceiveCounter[r1].Get(tx2);
  if d == c then:
     ReceiveCounter[r1].Put(d+1, tx2);
     inbox.Enqueue(m, tx2);
  send ack back to r1;
  commit tx2
```

Listing 6 Input ingestion procedure for RSM $r2$.

The runtime maintains two reliable dictionaries called `SendCounter` and `ReceiveCounter` that map `rsmId` to `int`. Pseudo-code for the outbox-draining task of an RSM with id `r1` is shown in Listing 5. It creates a transaction `tx1` and performs a dequeue on the outbox to obtain the pair $(m, \texttt{r2})$ of message and destination, respectively. It then sends the tuple $(\texttt{r1}, \texttt{SendCounter[r2]}, m)$ over the network to `r2` and waits for an acknowledgement. If it gets the acknowledgement within a certain timeout period, it increments `SendCounter[r2]` and commits `tx1` to complete the message transfer. If it times-out waiting for an acknowledgement from `r2`, it retries by sending the message again.

The automatic retry implies that the receiver might get duplicate messages; however, each such duplicate will be attached with the same counter value, which the receiver can use for de-duplication. This is achieved in the input-ingestion procedure shown in Listing 6. The receiver `r2`, when it gets the tuple $(m, c, \texttt{r1})$, first checks if c equals `ReceiveCounter[r1]`. If so, it increments `ReceiveCounter[r1]` and enqueues m to its inbox. If not, it drops the message because its a duplicate. Regardless, it always sends an acknowledgement back to `r1`.

Note that each of the tasks including input-ingestion, outbox-draining, and the event-handling, use their own transactions that are different from each other. This enables the RSM to run these tasks completely independently and in parallel to each other. SF transactions provide ACID semantics, so concurrent enqueue and dequeue operations on queues are safe.

RSM creation. When an RSM `r1` wishes to instantiate a new RSM of class C, it first creates a globally unique `rsmId` r. This creation can be done in several ways. Our implementation uses inter-partition communication to first decide the partition that will host the newly created RSM. It then grabs a unique counter value from that partition. The pair of partition name and unique counter value on that partition makes the `rsmId` globally unique. Once this value r is obtained, `r1` enqueues the pair (r, C) to its outbox. No RSM is actually created until the pair is committed to the outbox. If `r1` fails before committing, then the value r is lost forever. When `r1` is restarted, it will construct a new (but still globally unique) id.

The outbox-draining task of `r1`, when it picks up a tuple (r, C), will send a message to the partition on which r is located. Like before, this message is sent repeatedly until acknowledged. On the receipt of this message, the RSM runtime instantiates a new RSM of type C *only if* it does not already have an RSM associated with r. If it does have such an RSM, then it drops the message because it must be a duplicate request, one that it has carried out already. The recipient sends back an acknowledgement to the sender regardless.

4.2 Optimizing the SF backend

The following lists some of the most important performance optimizations that we found useful for the SF backend.

Shared inbox and outbox. Creating a separate reliable queue for the inbox and outbox of each RSM does not scale well unfortunately, especially when the application creates a large number of RSMs. Each creation incurs an I/O operation. To optimize the RSM creation time, we instead use a single data structure that is shared across all RSMs in the same partition: one for all inboxes and one for all outboxes. These shared structures are implemented as an `SF.IReliableDictionary` whose key is a tuple of `rsmId` and an index (`long`). Each RSM maintains its own head and tail indices, denoting the contiguous index range that contains its inbox or outbox contents. An RSM `r1`, for instance, can enqueue m to its outbox by writing it to the key (`r1`, tail) and incrementing tail. For efficiency, the head and tail values are only kept in-memory. On failover, the RSM runtime reads through the shared dictionary to identify the per-RSM head and tail values, before it instantiates the RSMs with these values. Additional care is required to ensure proper synchronized access to head and tail values by the various tasks associated with an RSM. Using these shared structures allowed us to significantly reduce machine creation time (Section 6.1).

Batching. We use batching in various forms to optimize overall throughput (Section 6.1). First, the event-handling loop of an RSM can dequeue multiple messages from its inbox in the same transaction and process all of them (sequentially, one after the other) before committing all of their effects together. The commit is a high-latency operation because SF must replicate all updates to the secondaries and wait for a quorum to acknowledge. This form of inbox-batching helps hide some of this latency. Second, the outbox-draining task can dequeue multiple messages from the outbox in the same transaction, and as long as they are intended for the same destination partition, send them over the network as a batch.

Non-persistent inbox. Sending a message m from RSM `r1` to `r2` requires several I/O operations: `r1` first commits m to its outbox, next it sends m over the network to `r2`, and finally `r2` commits m to its inbox. Interestingly, we can do away with a persistent inbox and only keep it in memory without sacrificing any of the RSM framework guarantees. Our optimization works as follows. The input-ingestion task of `r2` simply enqueues m to an in-memory inbox but it does not immediately send an acknowledgement back to `r1`. Instead, `r2` waits until it is done processing m. After `r2` commits the effects of processing m to its own outbox, it sends the acknowledgement back to `r1`, after which `r1` will remove m from its outbox. This is safe since the message sits in the (persistent) outbox of `r1` until `r2` is done processing it.

4.3 Kafka backend

Apache Kafka [2, 25] is a popular distributed messaging platform that has been used in large production systems by companies such as Netflix and Spotify [23]. Kafka supports fault-tolerant named sequence of messages called *topics*. A *producer* appends messages to the tail of a topic. A message is retained in the topic for a predefined period of time, after which it is deleted automatically. In order to read a message, a *consumer* subscribes to the topic and maintains a per-topic index, referred to as the consumer's *offset*. The read cycle involves the consumer reading the message at its offset, incrementing the offset, and then storing the new offset value in a topic of its own called the *offset-topic*. Kafka supports different consumers to read from different offsets of a topic concurrently. Starting in version v0.11.0, Kafka introduced the notion of cross-topic *transactions*. These allow a producer to write to multiple topics transactionally. Consumers cannot observe the writes made in a transaction until the transaction commits. A Kafka *stream* is a combination of a Kafka producer and consumer: it consumes messages from an input topic and publishes messages to one or more output topics. Kafka supports building stateful applications on top of streams via a key-value state store and convenient Java/Scala APIs. Exact-once processing of messages can be achieved by transactionally writing the offset, state and published messages to their respective topics.

Kafka-based RSMs. A Kafka RSM (K-RSM) has an associated *inbox topic*, and a *state topic* for its persistent local state. The RSM also maintains its read offset into the inbox as part of its persistent local state. An RSM executes as follows: it reads a message m from the inbox at its read offset and starts a Kafka transaction `Tx`. It then runs the handler code for m. Any changes to the persistent local state are written to the state topic under `Tx`.

Any message sends are written directly to the inbox topics of the receiver K-RSMs, also under `Tx`. Finally, the incremented offset is written to the state topic and the transaction `Tx` is committed. Note that there was no need to have an *outbox*: Kafka transactions ensure that the effects of processing a message by one RSM are not observed by other RSMs until its transaction commits. (SF transactions, on the other hand, cannot span across reliable collections in different partitions, which is why we needed an outbox for the SF backend.) Restart of an RSM simply involves recovering its state from the state topic that additionally provides it the read offset of the last un-processed message.

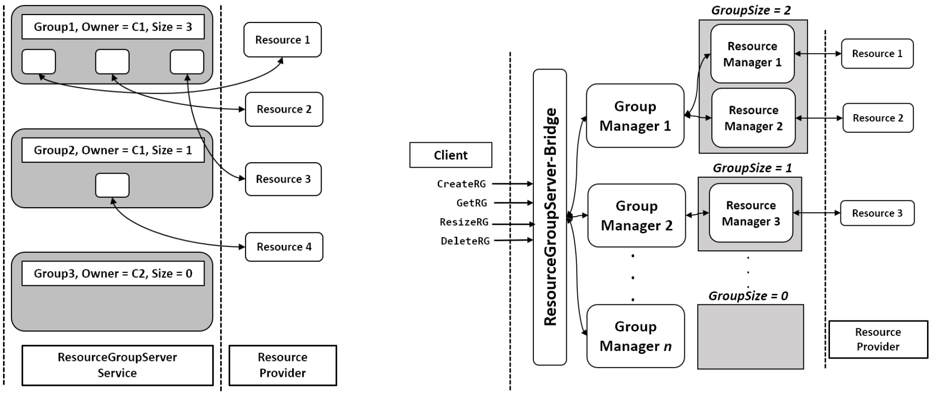
A user begins by starting a Kafka cluster, configured to their own requirements. The K-RSM backend then attaches to the cluster to execute the RSM program. Unlike SF reliable collections, Kafka topics must be preallocated to a fixed number, which would typically be much smaller than the number of RSM instances that a program may create. The K-RSM backend shares a single topic across multiple RSM instances, which works because each RSM maintains its own offset value. The assignment of RSMs to topics is currently done in a simple round-robin fashion but more sophisticated policies are possible as well. Similar to the SF backend, messaging in Kafka benefits greatly from batching: both when writing to a destination topic and when reading from the inbox topic.

4.4 `P#` backend

We additionally designed a backend for the purpose of testing RSM programs. The backend does not support distribution; it simulates the entire program execution in a single process. The backend essentially translates an RSM program to a `P#` program for systematic testing against a specification. We first briefly summarize `P#` capabilities [15].

`P#` provides an in-memory framework for implementing concurrent programs; it does not provide any support for distribution or persistence. A `P#` program consists of multiple state machines that communicate via messages. The `PSharpTester` tool takes a `P#` program as input and repeatedly executes it multiple times. It takes over the scheduling of the program so that it can search over the space of all possible interleavings. `PSharpTester` employs a state-of-the-art portfolio of search strategies that has proven to be effective in finding bugs quickly [18, 16]. A user can write a specification in the form of a monitor that is checked by the `PSharpTester` in each execution of the program. Both safety and liveness specifications [28] are supported.

The `P#` backend for RSMs allows one to write specification monitors in the same way as `P#` and test their correctness using `PSharpTester`. It is worth noting that the backend is designed with the intention of testing the user logic as opposed to the RSM runtime itself. For this, the backend ensures that only the concurrency (and complexity) in the user program is exposed to the `PSharpTester`; the concurrency inside the runtime (which is useful for gaining performance) is disabled.



(a) Overview of the microservice. **(b)** Architecture of the RSM based implementation.

Figure 7 Achitecture of the ResourceGroupServer service.

An RSM translates almost directly to a `P#` machine, with the following modifications. First, the backend provides mock implementations for all persistent types (simulated in-memory for efficiency). Second, the three tasks associated with an RSM (i.e., input-ingestion, event-handling and outbox-draining) are run sequentially, one after the other. Third, the exact-once network delivery algorithm is assumed correct, so the outbox-to-inbox transfer is done atomically (and in-memory).

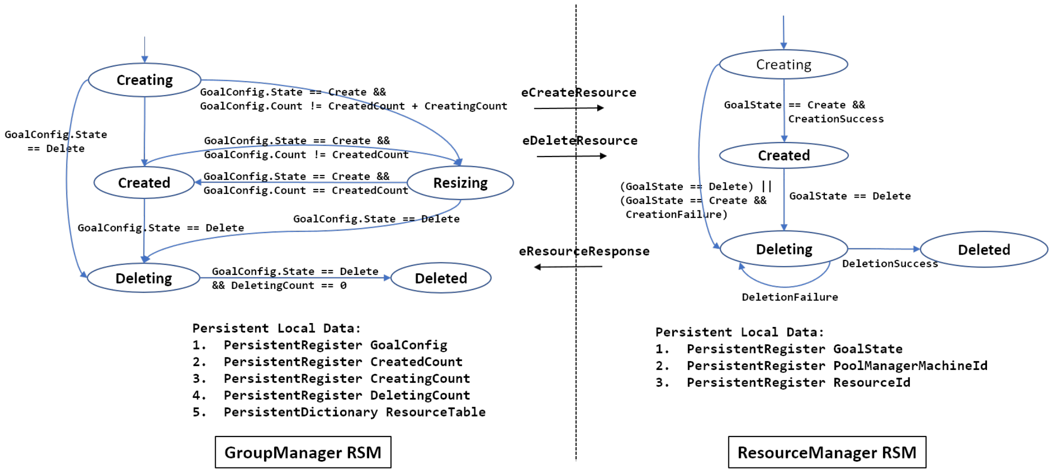
An important aspect of the backend is simulating failures in the RSM program. The failure-transparency property of RSMs from Theorem 3.3 crucially helps here: as long as the programmer correctly uses the volatile state as per Property 3.2, failures have no effect at all on the observable behavior of an RSM. Thus, the backend only needs to test for Property 3.2 on the program. This is done as follows. The backend, at the time it is about to commit a transaction in the event-handling loop of an RSM, non-deterministically chooses to carry out the following steps: (1) record the persistent state of the RSM (both local state and outbox), (2) reset the volatile state of the RSM, (3) abort the transaction, thus requiring the RSM to re-process the input message, and (4) when the RSM reaches the commit point again, assert that the persistent state equals the recorded state. If a failure of this assertion is reported by the `PSharpTester`, the programmer is informed of the incorrect usage of volatile state.

5 Case-Study: ResourceGroupServer

We used the RSM framework to redesign the core functionality of an in-production service on Microsoft Azure, which we refer to as *ResourceGroupServer*. This section describes the operations supported by the service (Section 5.1) and its implementation using RSMs (Section 5.2), highlighting the gains in programmability and testing of the service. We demonstrate scalability of the RSM code in Section 6.2.

5.1 Service description

The ResourceGroupServer (RGS) is a generic resource management service. A cloud platform will typically provide various kinds of compute and storage resources, for instance, virtual machines, that can be used in conjunction by a user to implement certain functionality. RGS is designed to offer a convenient abstraction over a low-level resource provider to maintain a collection of resources. A user can request the RGS for a set of n resources (called a *group*). The RGS then calls into the resource provider to allocate these resources.



```
Persistent Local Data:
1.   PersistentRegister GoalConfig
2.   PersistentRegister CreatedCount
3.   PersistentRegister CreatingCount
4.   PersistentRegister DeletingCount
5.   PersistentDictionary ResourceTable
```

```
Persistent Local Data:
1.   PersistentRegister GoalState
2.   PersistentRegister PoolManagerMachineId
3.   PersistentRegister ResourceId
```

Figure 8 The group manager and resource manager RSM state machines.

Fig. 7a shows a high-level view of RGS. Each group has a designated *owner* and supervises a number of resources. Individual resources can turn *unhealthy* (e.g., a VM becomes unresponsive), in which case, it is the responsibility of RGS to explicitly delete that resource and allocate a new one to ensure that each group eventually reaches its desired size. Also, there should be no *garbage* resources: one that is allocated by the resource provider but is not associated with any group.

A client C can fire a group creation request to RGS, with the desired number of resources n as a parameter. In response, RGS creates a fresh group, owned by C, with n resources in it. The client can query the health, resize or delete any existing group that it owns.

RGS must be responsive and scalable. It must be able to handle creation requests from multiple clients at the same time. Further, the creation of a group itself should not add much overhead over the actual allocation of the resources. RGS should also tolerate failures: if it crashes, it should not lose information about the groups that it had already created, or was in the middle of creating. For instance, if a requested group of size 10 had reached size 3 when the RGS crashed, it must resume and allocate only the remaining 7.

5.2 RSM-based ResourceGroupServer

We implemented RGS using RSMs. We denote this implementation as `RsmRgs`. It supports the core functionality that was described in the previous section. In comparison, the real production service (denoted `ProdRgs`) offers a richer API to its clients, but the additional features are unrelated to matters of reliability or concurrency. Fig. 7b shows the high-level architecture of `RsmRgs`. There are two RSM types: one called the *resource manager* (RM) that is responsible for the lifetime of a single resource, and another called *group manager* (GM) that is responsible for the lifetime of a single group. This division ensures that the complexities of dealing with the external resource provider are limited to the RM. Future changes to the resource provider APIs will likely not impact the GM.

A client can issue requests such as `CreateRG`, `GetRG`, `ResizeRG` or `DeleteRG` to `RsmRgs`. These requests are translated to messages that are directed to the GM that owns the corresponding group. The state machine structures of RM and GM are shown pictorially in Figure 8. We explain the functioning of these RSMs by tracing through the `CreateRG` operation.

```
void ScaleUp(RsmId gmId, int toCreate)
{
  for (int i = 0; i < toCreate; i++) {
    // Start off an RM to allocate a fresh resource.
    var id = create(ResourceManager);
    send (id, eCreateResource(gmId, ResourceGoalState.Create));
    // Record the creation in the resource table, and we're done.
    store (ResourceTable[id], ResourceState.Creating);
    store (CreatingCount, (load CreatingCount) + 1);
  }
}
```

Listing 7 `ScaleUp` operation to create resources in a group.

In response to a client's creation request, `RsmRgs` creates a new GM instance. Each such instance maintains three counters: `CreatingCount`, `CreatedCount` and `DeletingCount` which are, respectively, the number of resources that are under creation, already created, and under deletion. The GM additionally maintains a `GoalConfig` that specifies the desired number of resources in the group (`Count`), and the intended `State` of the group (either `Create` or `Delete`). Finally, GM also maintains a dictionary `ResourceTable` containing the `rsmId`s of all the RM instances that it owns.

A GM instance starts off in the `Creating` state with an empty `ResourceTable` and each counter set to 0. Its `GoalConfig` will get initialized to the group size that was requested by the client (on receiving the creation request) and the RSM will transition to its `Resizing` state realizing that it does not have enough resources created. In the resizing state, the RSM looks at the difference between `GoalState.Count` and `CreatingCount` + `CreatedCount`, say m, and fires off the operation `ScaleUp(gmId, m)` whose code is shown in Listing 7, where `gmId` is the `rsmId` of the current GM instance. We note that this entire operation is devoid of any failover or retry logic: the GM does not have to worry about failures of the machine hosting it, or about the failures of the RM instances that it creates. The runtime ensures that the exact number of instances requested will be created eventually (and no more).

An RM instance reliably persists the handle (`GroupManagerMachineId`) to the GM instance that created it, the goal state (`GoalState`) that is either `Create` or `Delete`, and the resource identifier (`ResourceId`) returned by the resource provider. An RM starts off in the `Creating` state, fires off a request to the resource provider, which if successful (`CreationSuccess`) causes a transition to the `Created` state. It then informs the GM about successful creation of the resource. The GM waits in its `Resizing` state until it gets enough success responses from its RM instances, i.e., until `GoalConfig.Count == CreatedCount`.

If a resource ever goes unhealthy, the corresponding RM instance transitions to the `Deleting` state and asks the resource provider to de-allocate the resource. On successful deallocation, the RM transitions to the `Deleted` state, and informs the GM, upon which the GM will issue the `ScaleUp` operation to allocate a new resource. Pool deletion is similar and implemented via a corresponding `ScaleDown` operation. Both RMs and GMs halt themselves after transitioning to the `Deleted` state.

Correctness. We use the `P#`-testing backend to check the conformance of `RsmRgs` to the following specifications. The testing helped weed out several bugs while implementing the RSM program. These properties were tested against a model of the resource provider where the allocation of a resource can non-deterministically fail (but eventually allocation is successful on repeated attempts) and the resource can go unhealthy at any time.

▶ **Property 5.1.** *Immediately following a* `ScaleUp` *or* `ScaleDown` *operation, the number of resources under creation, or already created, equals the desired number of resources.*

▶ **Property 5.2.** *If a client issues the sequence of requests* `CreateRG`(n_1), `ResizeRG`(n_2), *…,* `ResizeRG`(n_k)*, then* `RsmRgs` *will eventually create a group with exactly* n_k *resources.*

▶ **Property 5.3.** *On issuing a* `DeleteRG`*, eventually all resources of the group are disposed.*

A comparison of `RsmRgs` with `ProdRgs`. The resource and group managers lend themselves naturally to a state machine encoding. The state machines manage the life-cycle of a resource or a group, respectively. `ProdRgs` had a similar design, however, communication was not through message passing but rather via shared tables, maintained as SF reliable collections. One agent would update a table and other agents would continuously poll these tables to get the updates. Polling increased CPU utilization: `RsmRgs` uses roughly 10× less CPU than `ProdRgs`. Implicit communication also made the code harder to reason for correctness.

A direct comparison between the code size of `ProdRgs` and `RsmRgs` is not possible because the former implements more features. However, `RsmRgs` implements all of the core functionality in approximately 2000 lines of code, several times smaller than the corresponding functionality in `ProdRgs`. The designers of `ProdRgs` attest to the benefits listed here.

To contain code complexity, `ProdRgs` was not designed to be responsive during resize operations: it would wait to finish one resize operation before looking at subsequent resize requests. `RsmRgs`, on the other hand, is fully responsive in such scenarios. The GM state machine can handle new resize requests while it is in the `Resize` state: it simply updates its `GoalConfig` and issues either `ScaleUp` and `ScaleDown` until the group reaches its goal state. Importantly, the `P#`-based testing infrastructure of RSMs provides strong confidence in exploring a more responsive (and more complex) state-machine design.

6 Evaluation

This section reports on a performance evaluation of our RSM implementation. Section 6.1 measures common performance metrics on micro-benchmarks. Section 6.2 evaluates the performance of our ResourceGroupServer implementation. We draw comparisons with the Reliable Actors programming model of Service Fabric [35] (denoted `sfActor`). Reliable actors are an implementation of the "virtual actors" paradigm [12]. It serves as a useful baseline for experimentation because it builds on SF much like our SF backend implementation. Further, reliable actors do not provide failure transparency guarantees, although the programmer is given access to a persistent key-value store. This allows us to measure the relative overheads with providing a by-construction fault-tolerant runtime. In the rest of this section, we use the generic term *agents* to denote both `sfActor`s and RSMs.

6.1 Microbenchmarks

Our microbenchmarks evaluate the following three scenarios: (*i*) *creation*: where we measure the creation time for agents, (*ii*) *messaging latency* between two agents and (*iii*) *processing throughput*, where we measure the time taken to process a sequence of messages by an agent. In the subsequent discussion, we use `sfRSM` and `bRSM` to denote the SF-based RSM implementation, with and without optimizations mentioned in Section 4.2, respectively. We use `kRSM` to denote the Kafka-based RSM implementation.

Cluster Setup. The `sfActor`, `bRSM` and `sfRSM` services were deployed on a 5-node Service Fabric cluster on Microsoft Azure, where each node had a `D4_v2` configuration (8 CPU cores, 28GB RAM, and a 400GB local solid-state drive). The Kafka experiments were run on an

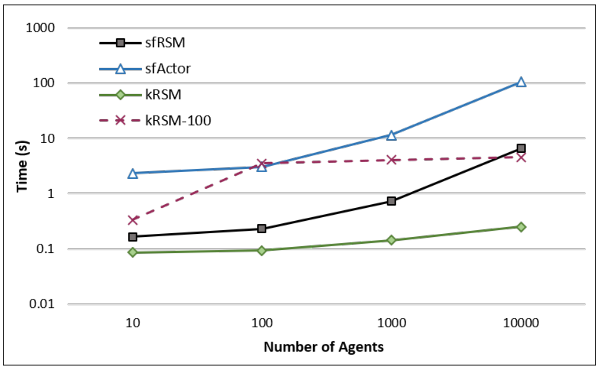


Figure 9 `sfActor` and `sfRSM` creation times.

Table 1 Messaging latencies.

Framework	0.5	0.9	0.99	Mean
sfActor	4.5	8	9.8	4.5
sfActor-Persist	12.5	23	23.5	11.9
sfRSM	23	31.5	70.6	22.8
kRSM	8.8	10	13.5	9.1

Azure HDInsight cluster with the following configuration: (*i*) 2 *head* nodes of type `D4_v2` executing the RSM runtime and application (*ii*) 3 *worker* nodes hosting the Kafka topics, with a total of 24 cores and 84GB RAM, and a total of 6 premium disks of size 1TB each (*iii*) 3 nodes for running Apache Zookeeper, with a total of 12 cores and 21GB RAM. (Zookeeper serves as a coordinator for Kafka nodes and manages cluster metadata.) Because of the different cluster setup, `sfRSM` and `kRSM` are not directly comparable.

Creation. It is important to keep overheads with creation low in order to provide most flexibility in programming RSM applications. In this experiment, we measure the time taken by a client to sequentially create n agents. Both the client and the created agents reside on the same partition, which allows us to eliminate any networking overheads from the creation times. Fig. 9 summarizes the results. The average creation time for `sfRSM` is 5.1ms, nearly 14X faster than `sfActor` (71.4ms), whereas the average creation time of `bRSM` (22.2ms) is 4.4X that of `sfRSM`. The speedup in creation time for `sfRSM` primarily stems from the shared inbox-outbox optimization.

The creation times for both `sfActor` and `sfRSM` scale linearly with the number n of agents created. For `sfRSM`, the bulk of the creation time is expended in committing a single SF transaction, which persists the initial local state of the machine and its `rsmId` to the runtime. Creations in `kRSM` are measured differently. We create the Kafka topics ahead of time because (*i*) creating topics on-the-fly is much slower than pre-creating them in bulk, and more importantly (*ii*) there is a limit to the number of topics that can be supported on each worker node. A `kRSM` creation now simply involves assigning two existing topics from the pool, along with persisting the id and initial state. We run two experiments, `kRSM`-1 and `kRSM`-100, where we multiplex the n RSMs onto a single topic and 100 topics, respectively. Note that all the writes during creation for `kRSM`-1 are batched into a single transaction, while the writes for `kRSM`-100 involve 100 transactions. As Fig. 9 shows, both `kRSM`-1 and `kRSM`-100 creations are fairly lightweight, with average creation times of 2.4ms and 2.6ms respectively (but discounting the topic pre-creation time).

In a separate experiment, we measured the creation throughput, by firing 1000 creation requests in parallel. `sfActor` and `bRSM` could achieve a maximum throughput of 287 and 67 creations per second, respectively, while `sfRSM` could hit a maximum of 1189 creations per second. The faster creations for `sfRSM` stems from its optimizations, which result in frugal CPU and IO requirements. `kRSM` creation throughput was 6661 creations per second.

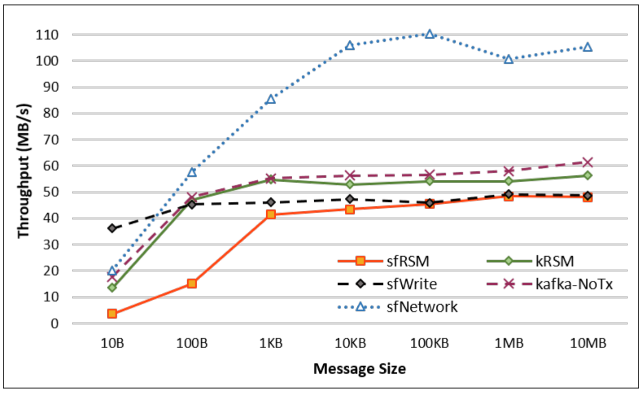


Figure 10 Throughput measurements with RSMs, SF and Kafka.

Messaging. This experiment measures the cost of exact-once messaging. The experiment comprises of two agents that repeatedly send a single message (50 bytes) back-and-forth and we measure messaging latencies. Messaging in sfActor is unreliable (best-effort, and lost on failures). We optionally make the agents in sfActor persist their incoming message.

Table 1 shows the latency measurements at different quantiles. Unsurprisingly, sfActor exhibits the lowest latencies. When we persist the messages in sfActor, which introduces one write per message transfer, it increases the latency significantly. sfRSM requires two write operations per message, making it nearly twice as expensive as sfActor with message persistence. Kafka, being a messaging system, is optimized for low-latency operations, even with exact-once guarantees. kRSM has better latency than sfActor with message persistence.

Throughput. In this experiment, a producer and a consumer agent are located on different partitions. The producer keeps sending messages (with a varying payload size) to the consumer. The consumer simply keeps a running count of the number of bytes received. We measure the time taken to process all the requests, and report the throughput in MB/s.

Fig. 10 summarizes the results. sfRSM automatically batches messages to increase throughput. sfActor has no default batching mechanism, although increasing message sizes decreases benefits to be gained from batching. At large message sizes, sfActor was able to achieve a maximum throughput of 86.6MB/s. In comparison, the maximum throughput for sfRSM (across all message sizes) was 48.4MB/s. To account for this difference, we precisely timed all micro-operations involved in the sfRSM runtime.

Sending a message from the producer to the consumer involves writing to the outbox and then sending the message over the network. We separately measured the best throughput of writing to an SF reliable collection (sfWrite) and sending data over the network as fast as possible via (unreliable) RPC (sfNetwork). Clearly, the throughput of sfRSM will be bounded by the smaller of these two values. As Fig. 10 shows, the writes constitute the limiting factor, and sfRSM incurs very little overhead over the sfWrite throughput, especially for large message sizes. Smaller message sizes implies a larger number of messages per batch, which increases the serialization overhead, and the number of times the consumer executes its handler. This effect, consequently, widens the gap between the sfRSM and sfWrite throughputs for smaller message sizes. This result shows that any improvements in the write throughput of reliable collections will directly speed up RSMs. The gap between sfRSM and sfNetwork is the cost of reliable messaging. Nonetheless, even at the small message size of 100bytes, sfRSM are able to do roughly $150K$ message transfers per second; enough for many

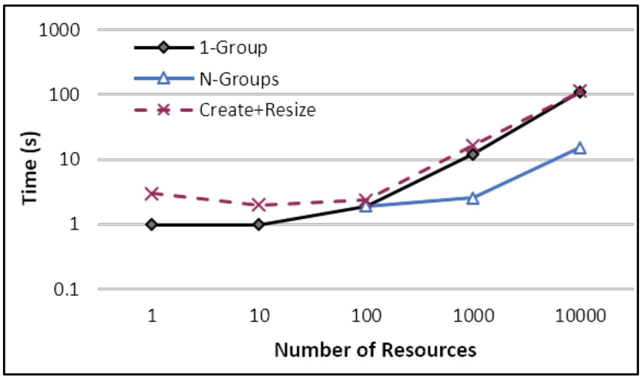


Figure 11 `RsmRgs` resource creation.

realistic applications. `kRSM` throughput peaks at 56.2MB/s. With Kafka, the persistence and message transfer happen together as a topic write. The upper bound for `kRSM` is to use non-transactional writes (`kafka-NoTx`). Fig. 10 shows that `kRSM` have little overhead compared to the throughput of `kafka-NoTx`.

6.2 `RsmRgs` Case Study

Performance. We measure the time taken to create a given number of resources in a single partition, assuming that the resource provider calls are instantaneous. Fig. 11 summarizes the results.

In the first experiment, denoted as 1-Group, we create a *single* group with progressively increasing number of resources. The more realistic scenario, which arises in production, is to have *multiple* groups of small sizes in a single partition. In the N-Groups experiment, we create multiple groups (each of size 100) in parallel such that the total number of resources matches the X-axis. We make two observations: (i) the creation times for both 1-Group and N-Groups increase linearly with the number of resources (ii) for the same number of resources, the increased parallelism in N-Groups results in the creation times being an order of magnitude faster than 1-Group. We would like to emphasize that the workloads here are realistic, and are based on requirements provided by the developers of the in-production `ProdRgs` service. The aforementioned results were reviewed by the developers, who confirmed that `RsmRgs` comfortably scales to production workloads.

To evaluate the responsiveness of `RsmRgs`, we issue `CreateRG`(y), followed immediately by `ResizeRG`(x), where $x = y/100$. The requirement is to ensure that the total time stays close to `CreateRG`(x). The Create+Resize line in Fig. 11 summarizes the result (with the value x on the X-axis). We see that as we increase x, the Create+Resize curve lies very close to 1-Group, which is testament to the service's responsiveness. For small x, the gap is wider because almost all of the y allocations kick-in by the time the resize request is processed.

Testing. For testing, we create mocks of both the client and the Resource Provider services, since they are external to `RsmRgs`. Our mocks are vanilla `P#` machines. The testing exercise was done on a laptop with a dual-core i7 processor, with 8GB RAM. The tester performed 100 iterations, with a scheduling strategy chosen from a predefined portfolio, with each exploration having a depth of $10,000$ steps. Note that the test for Property 5.1 is a safety-check, while the tests for Properties 5.2 and 5.3 are liveness-checks. The client issued a `CreateRG`(100) request. We deliberately injected a bug in the `ScaleUp` operation by removing the updates to CreatingCount. The resulting violation of Property 5.1 was detecting in 0.75s, generating an error witness of around 64 steps. We fixed the error, and issued CreateRG(100), followed by ResizeRG(5), and Property 5.2 was verified in 147.9s. To verify Property 5.3, we issue CreateRG(50) followed by a deletion and the tester verified the property in 119.1s. We further injected a bug by converting the CreatedCount to be volatile. (This means that if the machine was in the middle of a creation operation when it failed, it would lose track of all the resources it had created, and therefore the group would never reach the Created state.) The tester is able to quickly find a violation of property 5.2, in 5.5s.

Other applications. We have evaluated the applicability of the RSM language and runtime by encoding several other real-world applications. One example is a Banking application, comprising *account* and *broker* RSMs, with the latter being tasked with transferring money from one account to the other, without incurring any financial losses on failures. This specification can be encoded as a liveness property. Another example is a Survey application [32, 37], where *subscribers* can create surveys, which users can respond to. Each survey is managed by an RSM, and an overall coordinator RSM creates surveys, reports survey status, deletes surveys, etc. From a user perspective, responsiveness is a key metric. The application also needs to ensure specifications like a user vote is counted exactly once. The RSM framework allowed us to design these responsive applications, with all the specifications thoroughly tested.

7 Related Work

Actor frameworks. In actor-based programming [24], a natural fit for cloud services, an application comprises concurrent entities (called *actors*), each maintaining its local state, which is not shared among other actors. Communication and co-ordination between actors happens via message passing. Some popular instances of actor-based frameworks and languages include Akka [1], Erlang [20], and Orleans [12]. Fault-tolerance in these frameworks is achieved by checkpointing state to a persistent store, which is automatically restored upon actor rehydration. The responsibilities of checkpointing the state, ensuring consistency with the rest of the system, managing messaging retries and de-duplication, all rests with the programmer. More specifically, unlike RSMs, these frameworks do not provide failure transparency by-construction. Orleans introduced the concept of “virtual actors”: these actors need not be explicitly created. They are instantiated on demand when they receive a message. Further, they are location independent, allowing the Orleans runtime to dynamically load-balance the placement of actors across a cluster, even putting frequently-communicating actors together [30]. RSM instances must be explicitly created, but they are location independent. Our implementation, however, currently does not attempt to move an RSM after it has been created. The initial placement of a fresh RSM can be controlled by the programmer, after which the instance is permanently tied to that location. Service Fabric Reliable Actors [35] are also an implementation of the virtual actors paradigm. We provide an empirical comparison of RSMs with Reliable Actors in Section 6.

Reactive programming. Reactive frameworks [10] are used in the development of event-driven and interactive applications. These frameworks provide a programmatic way of setting up a *dataflow graph* that marks functional dependencies between variables. As the value of certain variables change over time, the rest of the dependent variables are updated automatically. Recent work [27] describes an extension to REScala [34] in order to provide fault-tolerance support in distributed reactive programming. The framework relies on taking snapshots of critical data and then uses replay to construct the entire program state on failure. This requires deterministic execution. Further, the input signals are not captured as part of the snapshots, causing them to differ on re-execution or even get duplicated. These issues require programmer support. On the other hand, RSMs can support non-deterministic handlers and guarantees exact-once processing because input (i.e. inbox) is part of the reliable state that RSMs maintain. The REScala extension provides eventual consistency for updates to shared data, making use of state-based conflict-free replicated data types (CRDTs) [36]. RSMs do not have shared state; maintaining common state between two RSMs can be done by creating (and communicating with) another RSM that owns the state. RSM messaging is reliable: this provides strong consistency between RSMs, however, it is less resilient to network outages than CRDTs because the latter allows for progress even in a disconnected state.

Big-data analytics. Big-data processing systems such as SPARK [38] and SCOPE [13] are popular frameworks for data analytics. They provide a SQL-like programming interface that gets compiled to map-reduce stages for distributed execution on a fault-tolerant runtime. These systems, however, are meant for data-parallel batch processing. They execute on immutable input that is known ahead of time.

Other frameworks. Ramalingam et al. [32] provide a monadic framework that makes functional computation idempotent. Their transformation records the sequence of steps that have already been executed. On re-execution, such steps are skipped. Idempotent computation enables fault-tolerance: simply keep re-executing until completion. Their work focuses on state updates made by a single sequential agent. They assume determinism of the computation and do not handle communication. RSM programs, on the other hand, support multiple concurrent agents with possible non-deterministic execution. RSMs ensure idempotence by atomically committing the effects of processing of a message along with the dequeue of the message from the inbox. Another class of languages for distributed systems, including Orca [11] and X10 [14], rely on distributed shared memory. They enable applications that span multiple machines while allowing the freedom to access memory across machine boundaries. They mostly focus on in-memory computation, without support for state persistence or fault tolerance.

The setting of Replicated State Machines [26] concerns a single *deterministic* state machine, replicated for fault tolerance. All replicas agree on a global ordering of submitted operations. This is the foundational concept in the domain of distributed consensus. In contrast, RSMs are at a higher level of abstraction, allowing a programmer to string together concurrent *interacting* state machines to encode fail-free business logic. RSMs delegate consensus to the storage layer.

References

1 Akka. `https://akka.io/`. [Online; accessed 10-January-2019].

2 Apache Kafka. `https://kafka.apache.org/`. [Online; accessed 1-January-2019].

3 Asynchronous programming with async and await in C#. `https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/`.

4 Azure Service Fabric. `https://azure.microsoft.com/services/service-fabric/`.

5 Azure Service Fabric Communication. `https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-reliable-services-communication-remoting`.

6 Azure Service Fabric Partitioning. https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-concepts-partitioning.

7 Azure Service Fabric Reliable Collections. https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-reliable-services-reliable-collections.

8 Azure Service Fabric Reliable Services. https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-reliable-services-introduction.

9 Azure Service Fabric Reliable State Manager. https://docs.microsoft.com/en-us/dotnet/api/microsoft.servicefabric.data.ireliablestatemanager?view=azure-dotnet.

10 Engineer Bainomugisha, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. A survey on reactive programming. *ACM Comput. Surv.*, 45(4):52:1–52:34, 2013. doi:10.1145/2501654.2501666.

11 Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: A Language For Parallel Programming of Distributed Systems. *IEEE Trans. Software Eng.*, 18(3):190–205, 1992. doi:10.1109/32.126768.

12 Philip A Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. Orleans: Distributed virtual actors for programmability and scalability. *MSR-TR-2014–41*, 2014.

13 Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In Jason Flinn and Hank Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, pages 285–300. USENIX Association, 2014. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/boutin.

14 Philippe Charles, Christian Grothoff, Vijay A. Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In Ralph E. Johnson and Richard P. Gabriel, editors, *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 519–538. ACM, 2005. doi:10.1145/1094811.1094852.

15 Pantazis Deligiannis, Alastair F. Donaldson, Jeroen Ketema, Akash Lal, and Paul Thomson. Asynchronous programming, analysis and testing with state machines. In David Grove and Steve Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 154–164. ACM, 2015. doi:10.1145/2737924.2737996.

16 Pantazis Deligiannis, Matt McCutchen, Paul Thomson, Shuo Chen, Alastair F. Donaldson, John Erickson, Cheng Huang, Akash Lal, Rashmi Mudduluru, Shaz Qadeer, and Wolfram Schulte. Uncovering Bugs in Distributed Storage Systems during Testing (Not in Production!). In Angela Demke Brown and Florentina I. Popovici, editors, *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016.*, pages 249–262. USENIX Association, 2016. URL: https://www.usenix.org/conference/fast16/technical-sessions/presentation/deligiannis.

17 Ankush Desai, Vivek Gupta, Ethan K. Jackson, Shaz Qadeer, Sriram K. Rajamani, and Damien Zufferey. P: safe asynchronous event-driven programming. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 321–332. ACM, 2013. doi:10.1145/2491956.2462184.

18 Ankush Desai, Shaz Qadeer, and Sanjit A. Seshia. Systematic testing of asynchronous reactive systems. In Elisabetta Di Nitto, Mark Harman, and Patrick Heymans, editors, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 73–83. ACM, 2015. doi:10.1145/2786805.2786861.

19 Enterprise workloads in the cloud. https://www.forbes.com/sites/louiscolumbus/2018/01/07/83-of-enterprise-workloads-will-be-in-the-cloud-by-2020/#636ee7856261.

20 Erlang. https://www.erlang.org/. [Online; accessed 10-January-2019].

21 Jim Gray. The Transaction Concept: Virtues and Limitations (Invited Paper). In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 144–154. IEEE Computer Society, 1981.

22 Maurice Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. doi:10.1145/78969.78972.

23 Kafka Powered By. https://kafka.apache.org/powered-by. [Online; accessed 1-January-2019].

24 Rajesh K. Karmani and Gul Agha. Actors. In David A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1–11. Springer, 2011. doi:10.1007/978-0-387-09766-4_125.

25 Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: a Distributed Messaging System for Log Processing. In *6th International Workshop on Networking Meets Databases (NetDB)*, 2011.

26 Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, 1978. doi:10.1145/359545.359563.

27 Ragnar Mogk, Lars Baumgärtner, Guido Salvaneschi, Bernd Freisleben, and Mira Mezini. Fault-tolerant Distributed Reactive Programming. In Todd D. Millstein, editor, *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*, volume 109 of *LIPIcs*, pages 1:1–1:26. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018. doi:10.4230/LIPIcs.ECOOP.2018.1.

28 Rashmi Mudduluru, Pantazis Deligiannis, Ankush Desai, Akash Lal, and Shaz Qadeer. Lasso detection using partial-state caching. In Daryl Stewart and Georg Weissenbacher, editors, *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, pages 84–91. IEEE, 2017. doi:10.23919/FMCAD.2017.8102245.

29 Suvam Mukherjee, Nitin John Raj, Krishnan Govindraj, Pantazis Deligiannis, Chandramouleswaran Ravichandran, Akash Lal, Aseem Rastogi, and Raja Krishnaswamy. Reliable State Machines: A Framework for Programming Reliable Cloud Services. *CoRR*, abs/1902.09502, 2019. arXiv:1902.09502.

30 Andrew Newell, Gabriel Kliot, Ishai Menache, Aditya Gopalan, Soramichi Akiyama, and Mark Silberstein. Optimizing distributed actor systems for dynamic interactive services. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, pages 38:1–38:15, 2016. doi:10.1145/2901318.2901343.

31 P#. P#: Safe Asynchronous Event-Driven Programming. https://github.com/p-org/PSharp. [Online; accessed 1-January-2019].

32 Ganesan Ramalingam and Kapil Vaswani. Fault tolerance via idempotence. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 249–262. ACM, 2013. doi:10.1145/2429069.2429100.

33 Amr Sabry and Matthias Felleisen. Reasoning about Programs in Continuation-Passing Style. *Lisp and Symbolic Computation*, 6(3-4):289–360, 1993.

34 Guido Salvaneschi, Gerold Hintz, and Mira Mezini. REScala: bridging between object-oriented and functional style in reactive applications. In Walter Binder, Erik Ernst, Achille Peternier, and Robert Hirschfeld, editors, *13th International Conference on Modularity, MODULARITY '14, Lugano, Switzerland, April 22-26, 2014*, pages 25–36. ACM, 2014. doi:10.1145/2577080.2577083.

35 Service Fabric Reliable Actors. https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-reliable-actors-introduction.

36 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. *JHAL-Inria*, page 50, 2011. URL: https://hal.inria.fr/inria-00555588.

37 The TailSpin Scenario. https://docs.microsoft.com/en-us/azure/architecture/multitenant-identity/tailspin. Accessed: 2019-1-10.

38 Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In Steven D. Gribble and Dina Katabi, editors, *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 15–28. USENIX Association, 2012. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.