Fault-tolerant Distributed Reactive Programming

Ragnar Mogk
Technische Universität Darmstadt, Germany

Lars Baumgärtner
Philipps-Universität Marburg, Germany

Guido Salvaneschi
Technische Universität Darmstadt, Germany

Bernd Freisleben
Philipps-Universität Marburg, Germany

Mira Mezini
Technische Universität Darmstadt, Germany

Abstract

In this paper, we present a holistic approach to provide fault tolerance for distributed reactive programming. Our solution automatically stores and recovers program state to handle crashes, automatically updates and shares distributed parts of the state to provide eventual consistency, and handles errors in a fine-grained manner to allow precise manual control when necessary. By making use of the reactive programming paradigm, we provide these mechanisms without changing the behavior of existing programs and with reasonable performance, as indicated by our experimental evaluation.

2012 ACM Subject Classification Software and its engineering → Software fault tolerance Software and its engineering → Data flow languages

Keywords and phrases reactive programming, distributed systems, CRDTs, snapshots, restoration, error handling, fault tolerance

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.1

Funding This work is supported by the European Research Council (ERC, Advanced Grant No. 321217), by the German Research Foundation (DFG, SFB 1053 and SA 2918/2-1), and by the LOEWE initiative in Hessen, Germany (HMWK, NICER).

Acknowledgements We thank all contributers of REScala and related projects, Julian Haas for his contributions on CRDTs, and all reviewers of this paper for their comments and suggestions.

1 Introduction

Ubiquitous connectivity together with web, mobile, and Internet of Things (IoT) computing platforms require software developers to consider distributed execution as an integral part of reactive applications. In a distributed reactive application, multiple connected devices update their state and behavior in response to the flow of events and data. Examples include notifications and messaging (instant messengers, chats), activity streams (social networks), data visualization applications (e.g., Jupyter), multi-user collaborative applications (e.g., Google Docs, Microsoft Office), and multi-player online games.

Developing such applications is challenging. Their inverted control flow is typically modeled in some form of continuation-passing style, resulting in the so-called callback hell [17].

© Ragnar Mogk, Lars Baumgärtner, Guido Salvaneschi, Bernd Freisleben, Mira Mezini;
licensed under Creative Commons License CC-BY
32nd European Conference on Object-Oriented Programming (ECOOP 2018).
Editor: Todd Millstein; Article No. 1; pp. 1:1–1:26

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Designs using continuation-passing style are fragile, hard to maintain, and hard to reason about. In particular, callback-based communication makes handling of exceptional conditions during the execution of an application challenging [39]. These issues become even more apparent in a distributed setting, where the control flow is spread across several networked nodes and faults occur due to lost connections or shutdowns of remote devices. For example, a mobile device may get disconnected during a network request (e.g., due to a weak cellular link or mobility) and may have to eventually shut down due to excessive battery consumption while trying to reconnect.

State of the art frameworks offering automated fault tolerance (e.g., Spark [49], Flink [6]) are designed for applications that process data without user interaction and that are deployed on cluster architectures, which are easier to control than mobile wireless systems with intermittent connectivity. Approaches for building reactive distributed applications (e.g., actors [24]) cannot provide restoration or synchronization automatically, because they do not have knowledge of the overall dataflow in the application. Finally, reactive programming languages [13, 14, 18, 8], for designing reactive applications in a declarative, modular, and composable manner [43, 42], do not support (automated) handling of networking or application faults.

In this paper, we extend REScala [44] to create a fault-tolerant reactive programming language for developing distributed reactive applications. Our extensions retain the syntax and functionality of REScala for local devices. REScala has first-class abstractions for *events* and *signals*, collectively called *reactives*. Events produce distinct occurrences of values, e.g., an event corresponding to an input field produces the text a user submits. Events can be derived from each other using operations such as filters or transformations, and they can be aggregated into signals. Signals represent time-changing values, such as the latest text a user submitted. Signals resemble spreadsheet cells where the value of a cell is derived from the values of other cells and a change causes updates of all derived values. These abstractions enable developers to program reactive applications without inversion of control. Reactives and their derivations form a dynamic *dataflow graph* with nodes corresponding to reactives and edges corresponding to the dataflow between reactives. The dataflow graph has been used to automate coordination of message propagation between multiple devices [15, 28, 47]. However, none of the existing approaches provide fault tolerance.

REScala enhances the traditional dataflow graph to support recovery after crashes and adds a distribution mechanism to cope with unreliable network connections, thus simplifying the development of fault-tolerant distributed reactive applications. By combining the declarative dataflow style of reactive programming with structured techniques for eventually consistent replication [10, 45, 22] and snapshots [6], REScala provides application-wide fault tolerance with little overhead in terms of both performance and syntactic clutter. Furthermore, REScala provides language abstractions for propagating and handling errors at the application level to enable developers to handle faults when the default behavior of REScala is undesirable and to seamlessly integrate application-level fault handling into the dataflow graph.

Contributions. We make two high-level contributions. First, we use features of reactive programming to generalize existing techniques for automated fault handling to work for distributed reactive applications. Second, we extend distributed reactive programming to enable declarative fault handling. In detail, we make the following contributions:

- We extend the update propagation mechanism of reactive programming to support recovery of managed application state by consistently restarting an application after a

crash as if the crash never occurred (Section 3). The extension is transparent to the application and produces little overhead in the common case (without crashes).

- We integrate eventually consistent data types with reactive programming to cope with distributed state management across devices in the presence of faults (Section 4). Our approach maintains strict consistency on a single device, but uses eventual consistency for every distributed path in the dataflow graph to guarantee availability [21].
- We design language abstractions for error propagation and adapt the runtime semantics correspondingly (Section 5), thus enabling developers to programmatically handle faults when the default behavior of REScala is undesirable, and to seamlessly integrate application-level fault handling into the dataflow graph.
- We provide an implementation of the fault-tolerant runtime and its error propagation abstractions (Section 6).
- We provide empirical evidence that REScala guarantees eventual and crash consistency in an efficient and transparent way (Section 7). To this end, we evaluate REScala using case studies to analyze the programming interface, and using microbenchmarks to evaluate the performance behavior.

The core sections mentioned above are complemented by a high-level presentation of REScala in Section 2, including an overview of the addressed kinds of faults, and a discussion of related work in Section 8. Section 9 concludes the paper and outlines areas for future work.

2 REScala from the User Perspective

In this section, we introduce REScala from the point of view of a programmer developing a simplified shared calendar application. The application tolerates disconnects and crashes, and users can update their calendar even when they are disconnected. We first discuss the fault model in detail, and then we introduce REScala by implementing the calendar application.

2.1 Faults

We use the term *fault* to refer to the origin of a failure and *error* to refer to the representation of a fault in the language [25]. REScala tolerates crashes and disconnects. REScala does not address data corruption (malicious or accidental).

Crashes happen when a device hosting part of the application runs out of battery, reboots after a crash or update, or runs out of memory, resulting in the OS to terminate the application. In these cases, the state of the application must be restored – on the same device – after a crash. Permanent faults are not addressed, because there are no spare devices or connections in the scenario we consider, i.e., we cannot equip users with new mobile phones.

Disconnects between devices are due to crashes of remote devices or due to broken network links. Disconnects cause messages to get lost, resulting in an inconsistent state across devices. REScala addresses the case where faulty devices recover after a crash and broken links are eventually restored – otherwise, state on the disconnected device is lost.

Current reactive programming approaches [15, 28, 47, 27, 31, 14] do not provide mechanisms for any kind of fault tolerance and delegate the responsibility for handling errors to the language into which the reactive framework is embedded – sidestepping the issue. In contrast, REScala provides tolerance of the above faults in the reactive programming paradigm in an automated manner.

```
1  val newEntry = Evt[Entry]()
2  val automaticEntries: Event[Entry] = App.nationalHolidays()
3  val allEntries = newEntry || automaticEntries

5  val selectedDay: Var[Date] = Var(Date.today)
6  val selectedWeek = Signal { Week.of(selectedDay.value) }

8  val entrySet: Signal[Set[Entry]] =
9    if (distribute) ReplicatedSet("SharedEntries").collect(allEntries)
10   else allEntries.fold(Set.empty) { (entries, entry) => entries + entry}

12 case class Entry(title: Signal[String], date: Signal[Date])

14 val selectedEntries = Signal {
15   entrySet.value.filter { entry =>
16     try selectedWeek.value == Week.of(entry.date.value)
17     catch { case DisconnectedSignal => false }
18   }
19 }

21 allEntries.observe(Log.appendEntry)
22 selectedEntries.observe(
23   onValue = Ui.displayEntryList,
24   onError = Ui.displayError)
```

Figure 1 Excerpt of REScala source code for the shared calendar application.

2.2 Shared Calendar Application in REScala

A user of the shared calendar application can create new calendar entries and select the displayed week. The calendar will be synchronized with other users when a connection is available.

Figure 1 shows our implementation. We refer to it as we introduce REScala's events, signals, conversions between events and signals, and how they are relevant for fault tolerance. The dataflow graph of the application is depicted in Figure 2, where the node labels correspond to identifiers used for reactives in the code example and edges represent the dataflow between those. The highlighted part shows all reactives reachable from `automaticEntries` and to which changes are automatically propagated. The rest of this section describes how this graph is created and how it behaves.

Events. Distinct occurrences of values are produced by events in REScala. There are input events and derived events. Input events are denoted by the keyword `Evt` (cf. Line 1) and allow to emit values using `evt.fire(value)`. The imperative firing of events typically happens as part of an integration with some external event source, such as an imperative UI library where a button press triggers a callback that fires the event, and is thus not shown in the code example. Derived events are defined by filtering or aggregating other events on which they depend. For example, the `||` operator combines two events into a new event that emits all values emitted by either of its dependencies, e.g., `allEntries` (Line 3) emits entries whenever generated by the user or from some external data source, such as a stream of national holidays.

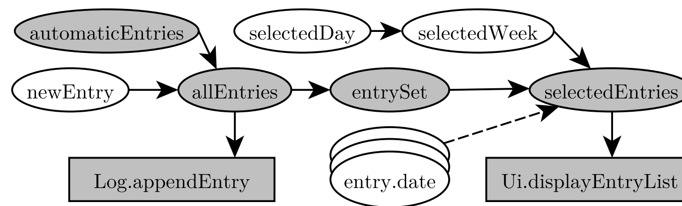

Figure 2 Dataflow graph for the calendar application. Nodes reachable from `automaticEntries` are highlighted.

Signals. Values that change over time are represented as signals in REScala. There are input signals and derived signals. Input signals are denoted by `Var`. In Line 5, an input signal is used to represent the currently selected day. The value of a `Var` can be changed by imperative code, e.g., `selectedDay.set(Date.tomorrow)`. The `Signal` keyword uses a user-defined computation – the *signal expression* – to express a derived signal. The signal expression can access other signals, called its *dependencies*. Accessing dependencies is explicit – using the `value` method to syntactically mark accessed dependencies. In Line 6 of our example, the `selectedWeek` signal is derived from `selectedDay.value`. The current value of a derived signal is updated automatically by executing the given computation whenever its dependencies change, similar to a formula in spreadsheets. Changes of a signal are automatically propagated to *derived* signals that use the signal in their definitions.

Crash-tolerant signals. Signals hold state and are restored after a crash (c.f., Section 3), either by loading the value from a snapshot, or by recomputation. For example, a user may have selected a different day than today, thus the selected day has to be stored in persistent storage. On the other hand, the selected week is recomputed from the selected day. REScala uses the Scala type system to statically ensure that the values of signals that are included in snapshots are serializable. For example, the `Var[Date]` in Line 5 requires that `Date` is serializable. Explicit annotations by the programmer are not required due to type inference and implicit parameters.

Folds and replication. Signals are convertible to events by aggregating individual event occurrences into an updating signal value – similar to folding over (infinite) lists. We refer to such signals as *fold signals* or simply *folds*. A fold, such as in Line 10, creates a signal with an initial value and updates it according to the parameter function every time the event fires. Lines 8 to 10 define a list of all calendar entries as the signal `entrySet` by folding over emitted calendar entry events. When the flag `distribute` is false, in Line 10, the `fold` operator aggregates the calendar entries emitted by the `allEntries` event into a list of all calendar entries. Line 9 demonstrates a distributed aggregation that has the same behavior as the local `fold`, aggregating all entries into a set. However, a `ReplicatedSet` has a name, `SharedEntries` in this case, and elements of the replicated set are shared with other devices that also use a `ReplicatedSet` with the same name.

Fault-tolerant folds. Fold signals are particularly interesting for fault tolerance, since they aggregate state that (a) must be included in a snapshot for restoration after a crash, and (b) is reliably replicated to other devices. Additionally, a distributed aggregation such as a `ReplicatedSet` is also synchronized after a crash to ensure that all devices eventually see the same set of entries. REScala uses built-in data types based on CRDTs [45] to ensure that

changes to all replicas eventually become consistent in the presence of crashes and message losses. The order in which entries are added, however, may be different on every device, and the intermediate values are visible.

Dynamic dependencies. Until now, all dependencies have been static, i.e., the dataflow does not change during runtime. However, signals may have a *dynamic* set of dependencies to support *higher-order* signals, i.e., signals whose values contain *inner* signals. For example, the entries (Line 12) we have been using in the calendar have a title and a date, both of which are signals and may change their value. The `selectedEntries` (Line 14) is derived by filtering the `entrySet` (Line 15). The filter function dynamically accesses the inner `date` signal of each entry (Line 16).

Interactions with the environment. In addition to firing and setting inputs, events and signals can be *observed* to produce side effects. Observing an event executes the side-effecting handler function every time the event emits a value, e.g., each entry is appended to a log file in Line 21. Observing a signal bridges between time-changing values and ordinary imperative state. For example, in (Line 22) the current state of the UI is overwritten when it becomes inconsistent with the signal after a signal change. REScala guarantees that the handler function of a signal observer is always called with the most recent value of the signal after an update, which allows the application to extend its invariants from the dataflow graph to external imperative libraries (in this case, the invariant is that the UI always displays the values held by the `selectedEntries` signal). Both event and signal observers can take an additional parameter to observe errors (Line 24), as explained next.

Explicit error handling. Reactives in REScala propagate errors along the dataflow graph. Errors can be handled as exceptions in signal expressions. For example, in Line 16, if the network connection fails before an inner date signal is transferred for the first time, then the access to the unavailable entry date signal throws a `DisconnectedSignal` exception. The default error handler postpones further evaluation of the `selectedEntries` until the signal is available. Instead of the default behavior, Line 17 explicitly catches the exception and returns false, causing the filter to drop the entry. Explicit error handling enables the use of application specific knowledge for more precise control of application behavior.

3 Fault-tolerant Application State

As illustrated in Section 2, crashes of individual devices during the execution of a distributed reactive application may result in a loss of the state of the reactive subgraph hosted on these devices. Loss of local device data is problematic since such data often contain important private or unsynchronized information of the current user. To address this issue, REScala provides automatic snapshots and recovery.

Snapshot anatomy. Conceptually, a snapshot of an application is a function that maps unique keys, denoting reactives, to their current values. The REScala runtime performs an analysis of the dataflow graph to minimize the number of key-value pairs that need to be stored. Applications often store redundant derived state in memory for efficiency. For example, a histogram displayed to the user can be recomputed from database entries, but it would be expensive to repeat this process for every frame the application displays. Local REScala applications typically consist of many small derived parts of the state (i.e., single

reactives) to take advantage of incremental updates. In such a setting, REScala detects derived state and excludes it from snapshots. Precisely, in REScala, the only reactives that cannot be derived are *vars* and *fold signals* (signals that aggregate event occurrences), since their state depends on past user interactions. All other reactives are either stateless events or derived signals that can re-execute their user-defined expression to recompute their state. We say that vars and fold signals constitute the *essential state*, and REScala recovers the state of the dataflow graph from the essential state.

Creating snapshots. Snapshots are created between semantically related changes, such that ongoing control flow does not interfere. REScala can detect related changes, because the dataflow graph makes semantic relations in applications explicit as (transitive) edges between reactives. For example, in Section 2, creating a new entry (a) updates the list of entries, (b) updates the list of selected entries, and (c) causes the UI to refresh. Explicit relations are used to determine when the transitive changes caused by an input event are fully applied to all reachable reactives and a snapshot is created after one such *update* to the dataflow graph. As a result, each snapshot corresponds to a single user perceived update, e.g., all changes of the state triggered by a new calendar entry belong to the same snapshot.

Incremental snapshots. Storing a full snapshot of all the essential state after each update is wasteful, since an update only affects parts of the application. REScala knows all updated reactives – they constitute the transitive closure of changed inputs, e.g., the highlighted reactives in Figure 2, starting with `automaticEntries`. REScala stores snapshots incrementally, by only changing the values in the snapshot that correspond to updated reactives. As a result, the cost of creating snapshots only grows linearly with the size of updates, instead of linearly with the size of the application. This scaling behavior supports efficiently composing large applications out of multiple parts, such as network, UI, and background services, in case the added parts do not increase the size of updates.

Recovering state. For recovery, REScala re-executes the application to restore the dataflow graph. During this recovery process, the value of each reactive is restored to the state before the crash. Events do not have state, so no value is restored. Fold signals and vars recover their values directly from the snapshot. Derived signals recompute their values from their inputs. The acyclic dataflow graph ensures that inputs are restored before derived signals, hence they can be used to recompute the derived signals. Like snapshot creation, the recovery process is incremental as reactives are restored as soon as they are created during the re-execution of the application. Thus, REScala allows the restored parts of the application to already handle new interactions, while other parts are still recovering.

Observers. REScala only restores state that is part of the dataflow graph. To ease integration with external libraries, REScala executes observers on signals during restoration. For example, when the list of selected entries of the calendar is restored, the observer that informs the UI about updates is executed. Observers allow the application to implement an invariant between the state in the dataflow graph and external state. Executing the observers during recovery allows the application to uphold its invariants, i.e., that the imperative state that is modified by the handler corresponds to the latest value of the signal. However, it is ultimately the responsibility of the application to use correct handlers. Events have no state to be restored (because snapshots are only stored between updates), so the handlers on event observers are not executed during restoration.

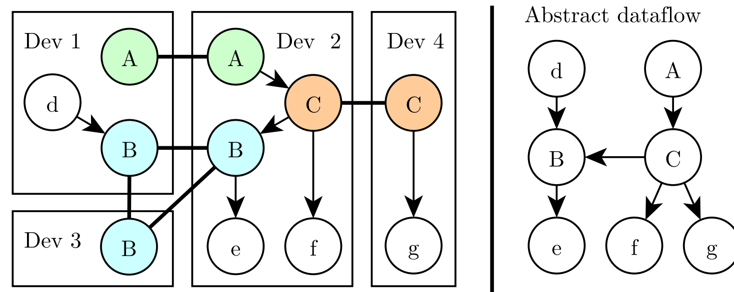

Figure 3 Full dataflow graph of a distributed application (left) and abstract dataflow (right).

Recomputation versus full snapshots. We make two arguments why recomputation is preferable over storing more values in the snapshot. First, a snapshot is created every time an update occurs, while restoration only happens when a device fails. Hence, storing only necessary state has performance benefits if the latter is only a small portion of the overall state (cf. Section 7.2 for an empirical evaluation). In a previous study [44], we reported that in a typical reactive application only 14% of the dataflow graph contains essential state. Second, only the essential parts of the state need to be serializable, thus allowing the use of data types that cannot be (efficiently) serialized for the rest of the application. REScala uses the type system together with type inference and implicit parameters to ensure that the static type of each reactive containing essential state is serializable.

4 Managing Distributed State

This section presents how REScala keeps the application responsive when network connections are not reliable. The key idea is to make the dataflow graph fault-tolerant and eventually consistent, instead of handling fault tolerance at the level of individual messages.

Replicated signals. Fault-tolerant dataflow graphs use *replicated signals* to model shared state among multiple devices. For illustration, consider Figure 3, which shows a dataflow graph spanning four devices (left), and the dataflow without distribution that is represented by this graph (right). Reactives A, B, C are replicated signals representing data shared across the devices (C is replicated in Dev 2 and Dev 4, A in Dev 1 and Dev 2, and B in Dev 1, Dev 2, Dev 3).

Replicated signals behave as normal signals with regard to device-local dataflow, e.g., their state is stored in the local snapshot. However, unlike directed dataflow connections between reactives on the same device, connections between replicas work in any direction, i.e., each device can change the state of its replica independently, even while being disconnected from the rest. As a result, the state of the replicas of a replicated signal can diverge in the presence of disconnects. Replicated signals use eventually consistent synchronization to support fault tolerance, i.e., the state of replicas diverges when disconnected to remain responsive and eventually converges when connections are available. Replicated signals are implemented using state-based conflict-free replicated data types (CRDTs) [45]. CRDTs provide automatic conflict-free merging of diverged state for a wide range of common data types [45]. For instance, the example in Figure 1 illustrates the usage of a replicated reactive of type `ReplicatedSet`:

```
9 ReplicatedSet("SharedEntries").collect(allEntries)
```

The underlying CRDT of `ReplicatedSet` is a set with a single commutative, associative, and idempotent operation, which adds to the set each value associated with an occurrence of the `collect`ed event. The set of entries is synchronized between all devices that use a replicated signal with the same name and type. Due to the properties above, the state of replicas can always be synchronized, and eventually all devices will converge to the same set containing all added elements. In addition to `ReplicatedSet`, REScala currently supports replicated counters, last-writer-wins registers, ordered lists, and replicated data types that allow adding and removing elements from sets and lists. By using conflict-free data types – an existing technique already known to programmers – we provide simple and intuitive semantics for sharing state across devices.

Integration with the overall dataflow graph. The well-defined set of operations on CRDTs enables their integration into the update propagation and enables state restoration, because operations of each CRDT, and how it changes based on local and remote events, are visible to REScala. As a result, changes to the state of a replicated signal are immediately propagated to local derived signals in the same manner as local changes update the whole reachable part of the dataflow graph at the same time, i.e., each update either is visible by all reactives of the local device or not visible at all. In contrast, the state of reactives on different devices can temporarily diverge. For example, consider the graph in Figure 3. An update that affects reactive A on Dev 2 will immediately affect reactive B on Dev 2 because they are connected by the local dataflow graph. However, if A is updated on Dev 1, B is only indirectly affected and synchronization with Dev 2 is required to complete the update. Such an inconsistency, where an update is applied to A but not yet to a connected reactive B, is called a glitch. To prohibit distributed glitches, Dev 1 would have to wait for the update on B to arrive, whenever A is changed. REScala allows distributed glitches in favor of availability.

Replicated signals are stored and restored when a device fails. Replicated signals have unique names shared between devices, which are used as keys in the snapshot, allowing multiple devices to include their replica in their snapshot and to synchronize changes after restoration. Thus, snapshots combined with conflict-free replication allow a device to disconnect, store and restore local modifications that survive crashes, and merge the local snapshot with the current state in the replicas on reconnect.

Publishing signals. Using CRDTs to implement replicated signals allows bidirectional communication, but the changes that one can perform on a signal are limited to the operations implemented by the CRDT. Alternatively, REScala allows to *publish* any signal – not only those based on CRDTs – but the published signal may only be changed by the publishing device. To prevent conflicting changes, other devices can only read the published signal. For example, each individual calendar entry (title and date) in the shared calendar is published by the device that puts the entry into the calendar, thus only the creator of an entry can change it. Publishing is a special case of eventually consistent replication. To publish a signal, REScala creates a replicated signal with a last-writer-wins CRDT, a data type where the merge function always selects the most recent value. Since only one device is allowed to write, there are no races between writes.

Distributed event propagation. Events are not distributed directly, but have to be converted to signals. However, we leave the responsibility to decide about the concrete conversion to the programmer, because of the trade-off between reliability and communication

overhead involved in the conversion. For example, the `latest(n)` operator can be used to create a signal containing the latest `n` occurrences of the event. As a result, the connection can be lost for the duration of `n` event occurrences without loss of data. If more than `n` events occur when the device is disconnected, the oldest events will be lost. Similar operators can be used to define time or priority based policies, allowing the application developer to tune the software behavior as necessary.

5 Error Propagation

REScala uses CRDTs to achieve fault-tolerant replicated state in an automatic way. However, by default, a disconnect of a CRDT does not trigger any action. The application eventually receives new updates after the reconnection and the inconsistency is resolved. In some scenarios, the application cannot simply wait for an eventual update, but has to act sooner. For this reason, developers should be able to program application-specific behavior in case of faults. For example, if a connection fails, a different replica may be selected manually, or the missing values are approximated, e.g., using a default value, if the application can continue the execution tolerating the inaccuracy. To support custom fault handling, we introduce errors as a programming abstraction in REScala. Errors are pushed into the dataflow graph when a device becomes disconnected (such a condition is established using timeouts). Errors are propagated along the same path as values in the dataflow graph, similar to how exceptions propagate along the path of the values returned by function calls.

In the following, we describe the API of our error propagation and handling mechanism from a user's point of view, and show how to handle errors that occur due to faults in the underlying distributed system as well as local errors due to faults like missing files or exceptions in external libraries. The new error-aware semantics of the reactives is a superset of their original semantics, thus existing code carries over unchanged.

Injecting errors into the dataflow graph. We extend the API of `Evt` and `Var` with operations for firing errors. `Evt.admit(error)` behaves similar to the existing `Evt.fire(value)` (similar for `Var.set`), but it starts the propagation with an error instead of a value. The main use of this API is to support the integration of existing frameworks, e.g., converting an error of a networking library to an error in the dataflow graph. Consider an existing networking library with a callback-based API. When a timeout occurs in the network, the imperative library callback is converted into a reactive propagation:

```
val fromNetwork = Evt[NetworkMessages]()
Network.onTimeout { error => fromNetwork.admit(error) }
```

Observe and recover. We extend the observer's API to accept an additional handler parameter called `onError`, which is used to observe propagated errors. This handler has the same purpose as `catch` blocks, and, similar to the standard `observe` call, has the goal of producing a side effect, e.g., displaying an error message. The error handler on observers can be missing: any unhandled error terminates the program in the same way as traditional uncaught exceptions. In the calendar example in Figure 1, any error is displayed to the user by the error handler defined on the signal in Line 22 using the extended observer API:

```
22 selectedEntries.observe(
23   onValue = Ui.displayEntryList,
24   onError = Ui.displayError)
```

Instead of simply observing errors, the application developer can recover the error inside the dataflow graph using the `recover` operator for signals and events, which is parameterized with a recovery function that converts an error to a normal value. The value is then propagated as the output of the `recover` operator. Any normal value that flows through the `recover` operator in the dataflow graph is propagated without change. The `recover` operator handles errors while they are propagated through the dataflow graph and before they reach an observer. Recovering from an error is most useful, when errors can be locally converted back into normal values. This case is relevant in several applications. For example, values can have local fallbacks, such as an unavailable location service that can be replaced by using more expensive or inaccurate local data. Another example is a signal holding an UI widget, where an error can be handled by displaying it to the user.

Signal expressions. A user-defined computation in a signal expression may access any number of dependencies. When any of the dependencies propagates an error, the error is raised as a Scala exception by the `.value` call performing the access. The exception may be handled by the application using the default Scala exception handling mechanisms. Unhandled exceptions in a user-defined computation are propagated along the dataflow graph. The use of Scala exceptions enables our error handling scheme to integrate well with most libraries in the JVM. For example, the shared calendar in Figure 1 filters the list of all calendar entries to only include entries of the current week in Line 14, and all entries containing an error are removed using a Scala `try`/`catch` block:

```
14 val selectedEntries = Signal {
15   entrySet.value.filter { entry =>
16     try selectedWeek.value == Week.of(entry.date.value)
17     catch { case e: NetworkError => false }
18   }
19 }
```

When the `entry.date()` in Line 17 contains an error, the error is thrown as a Scala exception and handled in the `catch` by returning false, causing the filter to drop the entry.

Folds. Recall that the `fold` operation supports converting events into signals. Given an event `e`, an initial value `init` and a function `f`, which are passed to it as parameters, `fold` returns a signal that is initialized with `init` und gets updated every time `e` fires by applying `f` to the current value of the signal. Thus, unlike other derived reactives, a `fold` signal accesses its own current value, i.e., the fold (indirectly) depends on the complete history of the event. In our example, the signal `allEntries` (Line 10) is constructed by reading the list of all entries and appending each read entry to create a value that accumulates all event occurrences:

```
10 allEntries.fold(Set.empty) { (entries, entry) => entries + entry}
```

The current accumulated value of the fold is treated like any ordinary dependency. If it is accessed and it holds an error, the error is thrown as a Scala exception. If the exception is not handled inside the user-defined computation (i.e., the function body of `fold`), then an error is propagated by the `fold` reactive to other reactives that depend on it. On the other hand, by handling the exception a developer can resume the computation of the `fold` reactive after an error. We present an example for fold with custom error handling next.

Example: Fold with custom error handling. To illustrate the use of the error handling API in fault-tolerant REScala, in the following we present and discuss the implementation of a user-defined operator on events. The operator, called `count`, is defined in terms of `fold`. It counts the number of non-error event occurrences and forwards error event occurrences without increasing the count. The implementation of `count` is shown below:

```
1 def count() = fold(0) { (state, occurrence) =>
2   occurrence // access the (unused) value to propagate potential errors
3   try state + 1 // increase count in non-error case
4   catch { case (value, error) => value + 1 } // continue counting after errors
5 }
```

The `count` signal starts with its initial state initialized to zero (Line 1). The folding function takes the current `state` of the fold and the incoming event, called `occurrence`, as parameters. When `occurrence` is accessed in Line 2, there are two possibilities: the access raises an error or a normal value. (a) in case of an error, the execution of the user-defined computation is aborted (because the access is not enclosed in a `try`/`catch` block), the state of the fold is not increased and the error is stored in the `fold` for future processing. (b) If the access of `occurrence` returns a normal value, the latter is ignored (we only count the number of non-error occurrences) and the execution attempts to access `state` in Line 3. If the current `state` is a normal value, it is incremented, and the increased count is returned (Line 3). If the current `state` is an error, the latter is thrown when `state` is accessed and immediately caught in Line 4. The pattern match in the `catch` block binds the last non-error value stored in the fold and the current error. Our example handler ignores the error and continues by incrementing the last non-error state, thus implementing a `counter` that resumes counting when a new occurrence arrives after an error.

6 Implementation

Our fault tolerance mechanisms are an extension of REScala, which in turn is implemented as a Scala library. We added efficient support for fault tolerance while preserving compatibility with existing applications.

Distribution. REScala uses a custom message passing mechanism for distribution based on TCP and Websockets (for Web clients); it does not provide any specific mechanism for peer discovery – the latter has to be implemented by the application. The synchronization mechanism of REScala supports any topology, e.g., client-server or peer to peer.

The mechanism for detecting changes of replicated signals is the same as the one used for local propagation of updates, i.e., a change is detected when the value of a signal is replaced by a new one. Change detection relies on immutability of values in signals, i.e., changes via side effects are not detected. When a change is detected, the new value of the replicated signal is serialized and sent over the network. Values are serialized using Circe[1], which supports type-safe serialization for most built in immutable Scala data types. Custom serializers can be provided using typeclasses. The serializer for signals is special and causes the signal to be published as described in Section 4.

[1] `http://circe.io/`

Snapshots. Our snapshot and restoration mechanism supports storing snapshots in arbitrary key-value stores. We have two implementations for in-memory stores: one that writes directly to disk (for JVM and Android) and another one that uses HTML5 localStorage [3] for web browsers.

Using fault-tolerant reactives is thread-safe, but care must be taken when reactives are created in a multi-threaded environment. Snapshots in REScala require unique IDs to identify reactives, and our current implementation uses thread-local counters to generate IDs, e.g., the IDs `UI-0` and `UI-1` are associated to the first and to the seconds reactive created in the context of the UI thread. This ID generation strategy is well suited for applications with a fixed set of threads, but it cannot cope with the case in which threads are created and used dynamically in a thread pool, because the IDs generated for those threads do not remain the same between restarts of the application. REScala requires the developer to explicitly handle the assignment of IDs to reactives if the automatic mechanism is insufficient. In practice, it is in most cases sufficient to ensure that dynamically scheduled tasks (e.g., those in thread pools) are assigned deterministic names to enable correct automatic generation of unique IDs.

Errors. The error propagation mechanism in integrated into the implementation of the dataflow graph through the following extensions. First, we extend the types of the values held by reactives. The REScala implementation (without support for errors) distinguishes between `Changed[T]` and `Unchanged[T]` for the data type of the values held by reactives – these two different types of values are propagated differently. To support error propagation, we introduce a third type, `Error`, and update the case distinctions in the propagation logic, whenever any of the types is accessed. Second, we modify the reevaluation function such that any exception thrown during the execution of user-defined computations, is propagated as an `Error`. Overall, our implementation strategy for errors induces little performance overhead when no faults are present, as shown in the empirical evaluation in Section 7.2.

7 Evaluation

Our integration of fault tolerance mechanisms into the reactive language runtime comes with synergetic effects between the two. On the one hand, snapshots and restoration maintain the consistency guarantees of reactive programming on individual devices in the presence of faults, distributed signals bridge dataflow graph across devices, and our error propagation mechanism enables principled handling of exceptional cases. On the other hand, the dataflow graph is instrumental in enabling fault tolerance in distributed applications at little cost in terms of both the burden on the programmer and the performance overhead. Specifically, the language runtime ensures that (a) the graph is consistent between updates, providing a point in the execution where snapshots can be taken efficiently, (b) derived values are automatically and consistently recomputed during restoration and remote updates, and (c) the application cannot change the state arbitrarily, so snapshots always remain consistent with the current state and changes are detected and distributed.

In the following, we empirically evaluate the claim that our fault tolerance features come at little cost in terms of both the burden on the programmer and the performance overhead.

7.1 Non-invasive Fault Tolerance

Our extensions for fault tolerance are non-invasive, meaning that existing applications implemented with REScala are made fault tolerant with minimal effort. To validate this claim,

Case study	observe		fire		change		Total		Description
	ok	nok	ok	nok	ok	nok	ok	nok	
CRDTs			9				9		CRDT integration
Datastructures			5				5		Reactive collections
Dividi			1		3		4		P2P distributed ledger
Editor			42		10	1	52	1	Swing text editor
Examples			39	2	9	19	48	21	Swing/console examples
Mill game			14		7	1	21	1	Turn based swing UI
Pong game	3		15	5	4	5	22	10	Multiplayer swing game
Reactive streams			1				1		Interface integration
Scalafx	3		1				4		JavaFX integration
Scalatags	2				1		3		HTML DOM integration
Swing			2			2	2	2	Swing integration
RSS			15		4		19		Swing RSS reader
Shapes	1		17		4	1	22	1	Swing drawing app
Todolist			11				11		TodoMVC app
Universe		1	8		2	9	10	10	Console simulation
Total	9	1	180	7	44	38	233	46	

Figure 4 Possibly problematic operators in case studies and extensions.

we answer the following research questions:

- (*RQ1*) To what extent do snapshots and restoration affect the application semantics?
- (*RQ2*) To what extent does the integration of replicated signals into the dataflow graph affect the application semantics?
- (*RQ3*) How many changes to a reactive application are necessary to support error propagation and handling?

To answer these questions, we analyze a set of case studies, consisting of ten applications (including games, simulations, and GUI applications) and five integrations with external libraries (e.g., an API to access the HTML DOM, bindings for JavaFX and for Java Flow), comprising a total of 13.000 LoC[2]. The case studies are listed in Figure 4 and their code is publicly available[3].

(RQ1) Effects of state snapshotting/restoration on application semantics. Snapshotting is invisible to an application, since snapshots are automatically created at the end of an update propagation. Restoration, on the other hand, is visible to the application, since restoration re-executes the application to restore the dataflow graph (cf. Section 3). The value of signals may differ between its first (normal) start and a restoration, causing different application behavior. Furthermore, certain inputs to the dataflow graph may be duplicated while restoring. For example, if a new calendar entry is added to the shared calendar via `newEntry.fire(startedEntry)` during startup of the application, then the `startedEntry` would be added every time the application is restored, resulting in multiple such entries in the list of `allEntries`, which is obviously not the desired behavior. We refer to problems with different behavior during restoration as *restoration inconsistency*.

[2] Lines are counted with CLOC (`cloc.sourceforge.net`) excluding comments and blank lines
[3] Repository available at `www.rescala-lang.com`.

To quantify the extent of restoration inconsistencies, we inspect all input and output interactions of imperative code with the dataflow graph in our case studies. These interactions are easy to localize, since they occur via a well-defined interface of the dataflow graph, consisting of the operations `fire`, `set`, and `observe`. The columns for `fire` (input interactions, also containing `set`) and `observe` (output interactions) of Figure 4 summarize our findings.

Firing events on some occurrence in the external world via the `fire` and `set` operations serves the purpose of entering new values into the dataflow graph, e.g., a user clicking a button, time passing, or receiving a network message. In our case studies, 180 out of 187 `fire` calls serve such a purpose and are not affected by state restoration. The 7 remaining calls that do exhibit the restoration inconsistency problem are instances of the same event usage anti-pattern: they incrementally build state during application startup. For example, the Pong game initializes the UI elements, and adds them one by one to a list of all UI elements, as shown below. As a result, this list would grow after each restoration.

```
val addElement = Evt[UIElement]
val allUIelements: Signal[List[UIElement]] = addElement.list()
addElement.fire(ball); addElement.fire(player1); addElement.fire ...
```

Firing of events must not be misused for initializing reactives. Manual inspection of usages of the `fire` method is required to find such misuses.

We also analyzed if `observe` calls on signals cause inconsistencies during restoration. We found a total of 10 usages of signal observers in the case studies (event observers are more common with 150 usages). Out of those 10 signal observers, 9 are not affected by restoration inconsistencies. 7 of them are in bindings for external libraries and are used to set properties of UI toolkits, e.g., the window title as in `titleText.observe(UI.window.setTitle)`. Triggering these observers during restoration correctly causes the UI to display the restored state. Two observers execute cleanup code, which is not affected by restoration either. The only observer that is affected by restoration inconsistency is in a simulation application (`Universe` row in Figure 4). The simulation uses mutable state outside of REScala, and if a fault occurs during a simulation step, this state is not restored.

We conclude that the state snapshotting/restoration feature of our approach operates mostly transparently. This means: (a) most of the potentially problematic interactions (181 out of 189, roughly 96%) are unproblematic in our fault-tolerant runtime, (b) the few problematic cases can be avoided, if application developers use the correct APIs of the dataflow graph, and ensure that mutable state outside of REScala is also able to tolerate faults.

(RQ2) The effect of introducing eventually consistent updates. Eventually consistent updates may affect the behavior of existing applications in two ways. First, they break the invariant that each occurrence of an input `Evt` is handled individually. Instead, after devices were disconnected for a while, all changes are replicated as a single large change to other devices. These combined changes cause problems when the application expects each change individually, e.g., if our shared calendar were to display a notification each time an entry is added, the notification may be triggered for a group of entries, instead of each individual entry, and as a result, the notification system has to be able to handle multiple entries at once.

Second, they break assumptions that usages of the `change` operator on signals may make about its behavior. The `change` operator is used to reify and handle each individual change

of a signal, and usages of `change` may assume that every intermediate change of the signal will occur individually. However, with eventual consistency intermediate changes may be grouped as described above, hence assumption changes become invalid. For illustration, consider a simple clock implemented as below. The computation of `minutes` relies on `seconds` change to 0. However, with eventually consistent propagation `seconds` could change from 59 to 2 skipping the intermediate step, because an aggregated update is received over the network, resulting in a missed minute.

```
val tick: Event[Unit] = ... // fires once per second
val seconds = Signal { tick.count() % 60 }
val minutes = Signal { seconds.change.filter(_ == 0).count() % 60 }
```

To quantify to which extent the introduction of replicated signals affects the application semantics due to the existence of `change` operations on signals, we investigate whether the semantics of our case studies relies on each individual signal change being visible, as opposed to relying on a notification about its latest change. The results of this analysis are shown in the change column of Figure 4. Roughly 46% of `change` operators (38 out of 82 in 7 out of 15 case studies) have different behavior when individual changes are grouped or skipped due to eventual consistency. The results indicate that replicated signals with eventual consistent semantics cannot be introduced transparently, which, in fact, is not surprising. One way to mitigate the problem is to keep computations that require strong consistency on a single device, and only distribute their results via replicated signals. As discussed in Section 9, manual handling of network errors has the potential to enforce consistency at the cost of availability, but this is not currently supported.

(RQ3) Changes to application code needed to propagate and handle errors. The integration of error propagation into the normal change propagation allows to propagate errors mostly transparently – additional code is required only at specific places where the developer wants to handle errors. The key point is that intermediate reactives do not have to be updated to propagate the error, minimizing the total amount of application code that requires modification. To demonstrate that error propagation does not "pollute" application code, in the following, we discuss how we refactored one of the existing case studies – a simple two player Pong game – to add support for handling application-level errors.

The case study consists of two application windows, one for each player. Without handling faults, if one player dropped, the game would get stuck or simply terminate. Figure 5 shows an abstract representation of the dataflow graph of the case study. Altogether, we update the game at three locations out of the 250 total LoCs.

To evaluate error handling in REScala, we added functionality to allow players to leave and join the game. When a player disconnects, an error gets inserted into the position signal of the racket of that player:

```
UI.onClose{ Racket.pos.admit(PlayerDisconnected) } // set position to error
```

Following the dataflow of `Racket.pos` through the dataflow graph of the application, one can identify the places where the error needs to be handled. There are two such locations: when displaying the players on the screen and inside the game logic handling cleanup of data structures for disconnected players.

For handling the error when displaying the players, we reused an existing try/catch block that handled missing game objects and added a handler for the `PlayerDisconnected` exception.

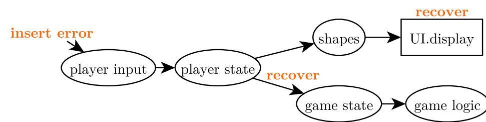

Figure 5 Recovering from errors in Pong.

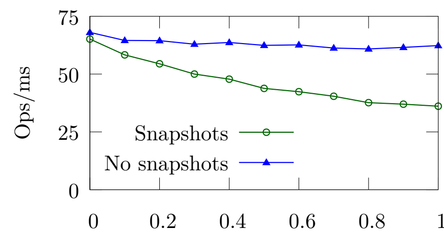

Figure 6 The cost of snapshots.

```
case _: NoSuchElementException | _ : PlayerDisconnected =>
// remaining handler unchanged
```

As a final modification to the code, failed connections are observed and the corresponding player is removed from the game. To remove the player, a list of disconnected players is derived from the the list of players, by filtering on the player connection:

```
val disconnectedPlayers = Signal{ players.value.filter { p =>
    Try(p.connection.value).isFailure} }
disconnectedPlayers.observe(Game.removePlayers)
```

If accessing the connection raises an error (checked with `Try(...).isFailure`), then the player is considered disconnected. The resulting list of failed players is observed and these players are removed from the game (closing the connection and updating the list of players).

7.2 Performance Evaluation

We use microbenchmarks to evaluate the performance of different features of REScala. We evaluate each feature individually, since they do not influence each other and can be disabled by applications as required. Specifically, we answer the following questions:[4]

- (*RQ4*) What is the performance overhead introduced by our snapshotting mechanism?
- (*RQ5*) What is the performance tradeoff between restoring state from the snapshot versus recomputing the state?
- (*RQ6*) How does the performance of our recovery mechanism compare to the performance of the recovery mechanism of an industrial-strength data streaming system?
- (*RQ7*) How does language-integrated error propagation affect application performance?

Experimental setup. We use existing microbenchmarks of the base reactive language, which are available from the Github repository[5] in version `v0.21.1`. The benchmarks are implemented using the OpenJDK benchmarking framework Java Microbenchmark Harness [4] version 1.19. We perform 25 iterations of each benchmark and report the average. To reduce the influence of non-deterministic optimizations, we fork the JVM 5 times, each doing 5 iterations with proper warm-up. Each iteration runs for about 1 second. We run the benchmarks on an Intel Xeon CPU E5-2670 @ 2.60GHz, using one core only, since the benchmarks are not multi-threaded, and we use the OpenJDK 1.8.0_141 Server VM with default parameters on CentOS Linux (Kernel 3.10).

[4] We do not evaluate the efficiency of our CRDT as we do not contribute performance improvements over existing work [5, 45].
[5] See `www.rescala-lang.com`.

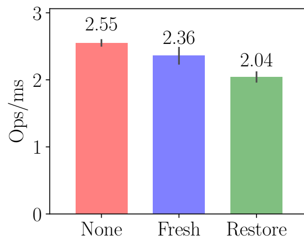

Figure 7 Cost of restoration.

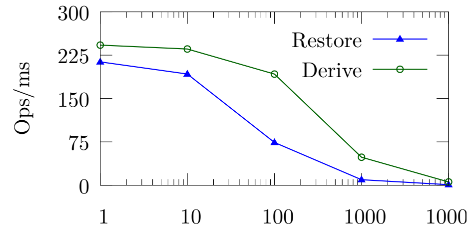

Figure 8 Restoring vs. recomputing lists of various sizes.

(RQ4) Overhead of snapshots. Snapshots happen after every update to the dataflow graph and affect the overall application performance. Snapshot overhead consists of the internal overhead for determining all the updated state and of the overhead for serializing that state. The snapshot is stored in an in-memory database, because we do not want to measure time spent writing to disk, since this overhead is not specific to our solution. We quantify the snapshot overhead as a function of the number of folds in an application, since only the state of fold signals is included in a snapshot. For this purpose, we parameterize our benchmarks with the number of fold signals in the graph.

Figure 6 shows the throughput for a dataflow graph consisting of a single input event with 100 reactives derived from it, on the x-axis is the percentage of folds out of these derived reactives, the other reactives are stateless. We selected this topology since it allows us to create a full snapshot of all fold reactives with a single input change. To factor out the influence of computations not involved in snapshotting, user-defined computations of both folds and stateless derived reactives only do simple integer arithmetics with negligible overhead. We executed the benchmark twice, with and without snapshots enabled. The relative throughput is on the y-axis of Figure 6 (higher is better).

We observe that the throughput of the benchmark with snapshotting is overall lower than without and further decreases when the number of fold signals is higher. In the best case, i.e., there are no fold signals, the overhead is minimal; our solution incurs performance overhead only when state is actually stored, i.e., there is no overhead for an active but unused feature. In the worst case, i.e., when every reactive is a fold, the throughput of the run with snapshots is still about 58% of that with no snapshot. For typical reactive programs, however, which contain roughly 14% fold signals [44], the relative throughput is 85%. Moreover, the numbers reported so far are rather conservative and the real average throughput is higher, because typically only part of the graph, i.e., only a subset of the folds in each benchmark configuration, changes its state during an update. To recap, we consider the overhead of our snapshot mechanism reasonable.

(RQ5) Restoring from snapshots versus recomputing. We first quantify the overall cost that recovery adds when restarting the application (1). We also quantify the tradeoff between taking minimal snapshots versus taking bigger snapshots (2).

Regarding (1), Figure 7 shows the results of measuring the cost of recovery for the graph from (RQ4). Each bar on the x-axis shows the throughput of creating a graph (a) without any support for fault tolerance, (b) with support for fault tolerance but when restoring from an empty (fresh) snapshot, as is the case when an application is started for the first time, and (c) when restoring the graph from an existing fully populated snapshot. The overhead we observe in the last case is the result of creating the initial snapshot and restoring the

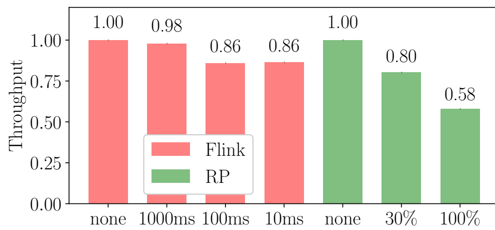

Figure 9 Flink vs. REScala snapshot performance.

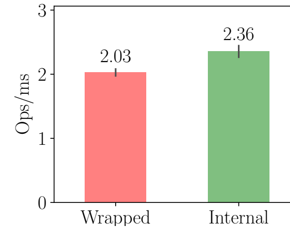

Figure 10 Integrated error propagation versus `Try`-based solution.

(serialized) values from the snapshot. We conclude from Figure 7 that while restoration has a certain overhead, the cost is comparable to normal application startup times, since REScala restores the graph of 100 reactives twice per millisecond, compared to starting the application, which is performed 2.5 times per millisecond.

Regarding (2), as already mentioned, our approach minimizes the amount of state that is stored in snapshots, hence we tradeoff restoring derived state against recomputing it. Intuitively, one would assume that our restoration has higher overhead compared to one that starts from a maximal snapshot, as it has to recompute more. However, a small experiment indicates that this does not necessarily have to be the case. In the experiment, we run two versions (labeled Restore and Derive) of a benchmark with a dataflow graph that stores a list containing integers 1 to N. In the Restore version the list is part of the snapshot, while in the Derive version the snapshot only contains the size of the list and the list itself is recomputed during restoration. The graphs in Figure 8 show the results, with N in the x-axis and throughput in the y-axis. We observe that (a) both restoring and recomputing derived state get linearly more expensive with the size of N and (b) recomputing the list given its size is faster than restoring from a complete snapshot of it. This indicates that our approach of deriving as much state as possible from minimal snapshots during recovery does not only make snapshotting efficient, but can also be beneficial to restoration performance.

(RQ6) Comparison to an industrial-strength data streaming system. Our objective in this experiment is to compare the performance of our prototype implementation for snapshots and recovery to a functionally similar industrial-strength system. The objective is to measure an upper bound for the performance of our system. We chose Flink [6], a state-of-the-art, industrial strength, big data processing engine for real-time analytics used, among the others, in the Alibaba real-time search ranking, in Zalando's business process monitoring and in Netflix's complex event processing system [2]. Flink is suitable as a reference due to the following reasons: (a) it is functionally similar to reactive applications in that it also manages state inside of a dataflow graph (a property it shares with other streaming systems), (b) it is implemented in Scala, hence the runtime environment is similar to ours, (c) it is well known for its focus on fault tolerance, (d) it is also possible to enable/disable snapshots, and (e) both Flink and REScala serialize snapshots to memory.

We implemented a similar graph structure as in (RQ4) for Flink. However, Flink and REScala target different usage scenarios, where REScala immediately reacts to individual occurrences of input events, such as button clicks, Flink processes and aggregates complete input streams of data. Hence, we do not compare the absolute performance of Flink and REScala, but only measure the relative overhead of creating snapshots.

In Figure 9, we show the throughput relative to execution without snapshots (checkpoints in Flink terminology). Snapshots in Flink are created periodically instead of after each

update (we have created them every 10 ms, 100 ms, and 1000 ms, respectively), and always include the complete state of the system. While the overhead of REScala is higher when a full snapshot is created, in the case when only 30% of the dataflow graph is stored in the snapshot – which is the realistic case – the relative overheads of both systems are similar.

We conclude that the performance of our snapshot algorithm is comparable to Flink. Yet our prototype is a proof-of-concept, and has not been extensively optimized. This observation is an indication of the benefits of exploiting features of the reactive programming paradigm, specifically automatically managed state, in the design of REScala.

(RQ7) Performance effects of language-integrated error propagation. In Section 5, we motivated the need for language-integrated error propagation for the quality of application design. By answering RQ3, we empirically provided evidence that our approach to error propagation indeed barely pollutes the application code. The experiments presented below analyze the potential performance effects of this non-invasive error handling. Specifically, we analyze (a) the potential performance tradeoffs of the language-integrated error propagation compared to programmatic error handling, and (b) the overhead of the error propagation system in the absence of errors.

These experiments show that there is no additional cost. As discussed in Section 6, this is due to the tight integration of errors into the existing runtime. Moreover, language-integrated error propagation exhibits better application performance compared to programmatic error handling.

For (a), we implemented a reactive program with programmatic error handling by using Scala's `Try` to propagate errors, so every `Signal[A]` becomes a `Signal[Try[A]]`. Using `Try` is the idiomatic way to represent errors as values in Scala, similar to the `Maybe` data type in Haskell. As shown in Figure 10, our solution outperforms the solution that uses `Try`-wrappers. This improvement is due to the fact that language integration merges error propagation into the internal data structures of the language runtime, while `Try`-wrappers require an additional layer of indirection; in addition, the solution that uses wrappers requires unwrapping code at every signal expression[6]. The first line of code below just adds two values, compared to the second line of code where adding two values becomes unwieldy when nested `Try` expressions need to be unwrapped, even when using Scala's special `for` syntax:

```
Signal { a.value + b.value } // without wrapper
Signal { for (av <- a.value; bv <- b.value) yield av + bv } // wrapped in Try
```

For (b), we use the REScala benchmark *natural graph*, a graph with 25 reactives that are connected in a way to mimic real applications [44], to show how the performance of an average application is affected. All user-defined computations only perform arithmetic additions to minimize the amount of work that is spent on actual computation and maximize the relative overhead of the error propagation. We did not measure any performance degradation when error propagation is enabled but the application does not use it, thus developers only have to pay for what they use.

[6] Other representations of errors are possible, but they all have to share the same pattern of `Try`, both values and errors need to be represented in a single data type, and the application developer has to manually differentiate the two cases in signal expressions.

7.3 Threats to Validity

There are both internal and external threats to the validity of our results. Internal threats are due to the inspection of case studies for analyzing the non-invasiveness of our approach, as what is non-invasive is subjective and depends on our experience in developing REScala applications. Also, the results are not confirmed by subjects without experience with REScala.

An external threat is that the benchmarks may be too small and not sufficiently diverse to be representative of reactive programming applications for the results to be generalizable. Unfortunately, at the time being there are no standard benchmarks for reactive programming languages. Given the lack of any widely accepted benchmark suite (to the best of our knowledge), our selection of the benchmarks is strongly based on our experience with REScala applications. Extending the benchmark suite with more, diverse and larger-scale case studies will be addressed in future work.

8 Related Work

Languages for reactive applications. Non-distributed languages for traditional desktop applications are usually concerned with I/O errors during execution, but typically do not provide facilities to snapshot or restore program state. In the object-oriented paradigm, reactive software is often developed using the Observer design pattern. This approach, extensively discussed in the literature [17, 27, 31, 13], leads to the inversion of the control flow, which complicates code analysis and induces highly error-prone broadly scoped side-effecting operations: since observers do not return a value, computational results need to be passed through imperative state changes, prohibiting all of the techniques for fault tolerance discussed in this paper.

Functional reactive programming (FRP) [18] models time-changing values, whose denotations are functions focusing on the problem of formally modelling continuous time. FRP has been used in a number of areas, including robotics [23], network switch programming [20, 48], wireless sensor networks [36], and reliable software for spacecraft [37]. In general, FRP seems to be a natural fit for distributed applications [29, 40, 41, 15, 11], with events representing messages from the network or user input. However, many functional reactive languages and frameworks do not provide support for unreliable networks. Typically, reactive languages [27, 31, 14] simply delegate the responsibility for error handling to the host language, and ultimately to the programmer. In distributed reactive programming [15, 28, 47], reactives on different devices are connected to each other and update messages are sent over the network whenever a remote dependency changes. In the presence of faulty devices and unreliable connections, such update messages may get lost causing several problems, such as (a) glitches, (b) changes that are visible on one host but not on another host, or (c) application unresponsiveness when new changes cannot be processed while messages are being resent to a device that failed and is restored.

Unreliability has been partially investigated in the context of some FRP derivates. Timeouts have been introduced to a distributed runtime and dataflow [35]. ReactiveExtensions (Rx) [26] integrate and propagate errors into the dataflow. However, to the best of our knowledge, no solution exists to automatically restore and reconnect a dataflow graph after a crash. DREAM [28, 29] is a middleware for distributed reactive programming, which lets the programmer choose among different levels of consistency guarantees in distributed reactive systems, including FIFO consistency, causal consistency, glitch freedom and atomic consistency. However, none of these approaches provides the consistency guarantees of REScala automatically. Ur/Web [12] is a multitier programming language that uses reac-

tive programming to update the client UI. However, to the best of our knowledge, there is no integration of RPC errors and the reactive part, hindering application-wide reasoning and lacking common abstractions for distribution and reactivity.

Actor and cloud languages. Actors [1, 7, 9, 33, 46, 24] are well-known abstractions to model concurrent and distributed systems. Actors do not share mutable state and communicate only via message passing. The result is a loose application structure that makes automatic reasoning about overall system consistency very hard. Furthermore, we consider message-passing to be rather an implementation mechanism for enabling communication, which is by no means a proper substitute for providing first-class composable and programmable abstractions in the language, as it is the case with REScala.

Languages such as Erlang and Akka support restarts of crashed actors, possibly on a different device, but it is the responsibility of the application logic to be robust against such crashes. Otherwise, state on restarted actors is lost, and application properties are violated. Orleans [9] and extensions to Akka [1] can automatically restore the state of single actors after a crash. However, state is stored without consistency guarantees between multiple actors, and it is still difficult to reason about application properties. Akka additionally requires manual changes to each actor that requires fault-tolerant state, making it impossible to reuse existing actors developed without support for fault tolerance.

AmbientTalk [46] is an actor language specifically designed for mobile ad hoc networks, and Direst [34] builds on top of AmbientTalk and adds reactive abstractions and automatic eventually consistent state distribution. However, Direst uses a centralized replica to provide eventual consistency, hindering any communication between devices when the centralized replica is unavailable. Furthermore, applications in Direst cannot dynamically reconfigure their dependencies – a necessary concept for existing dynamic applications. Hence, Direst cannot support common reactive patterns, such as dynamically selecting the current view of an application at runtime, thus limiting reusability of components.

MBrace [16] extends F# with expressions for cloud computations. The use of immutable global references allows the distributed runtime to automatically re-execute tasks on failed devices without causing inconsistencies. Errors that are raised during the evaluation of cloud expressions, e.g., because a remote resource is unavailable, are transparently propagated along the dataflow path of the expression, even across the distribution boundaries, allowing non-localized error handling. However, since the distributed state is immutable, the abstractions are not well suited to reactive applications where the program state changes dynamically in response to input from the user or the execution environment/context.

Batch and stream processing languages. Frameworks for big data processing, such as Spark [49] and Flink [6], handle crashes of worker machines to minimize lost work when machines fail. They have recently also adapted syntax similar to FRP, but are not suited for reactive applications. Running in cluster environments with full control of communication and distribution of work among machines, Spark and Flink can offer abstractions for distribution and fault tolerance with suitable correctness guarantees. However, to provide these guarantees, applications are written in specific DSLs, and the execution runtime is not connected to the embedding application. The use of a DSL limits the capability to integrate with other libraries, and the DSL is not designed for reactive applications.

Building blocks for distributed applications. Several approaches provide building blocks to develop applications in distributed systems.

The counterpart of observers in non-distributed software are pub-sub systems in distributed software, with similar problems [19, 38, 30].

Eventual consistent data types such as CRDTs [45] or CloudTypes [10] are important building blocks providing well-understood tradeoffs between consistency and responsiveness. In case of connection failures, eventual consistent data types become out of sync with other replicas, but when the failures are only temporary, a consistent state can be automatically restored. However, on their own these data types cannot provide any application-wide correctness guarantees.

Function passing [32] is a style of distributed programming that defines a graph of immutable values and operations over these values. The result is a graph similar to Spark RDDs, but using arbitrary Scala functions instead RDD transformations, combining an abstraction for distributed systems with reusability of most Scala functions. However, since fault tolerance and reactivity are not part of the language, the language cannot enforce or check any properties.

9 Conclusion

In this paper, we presented REScala, a reactive programming language to support the development of fault-tolerant distributed reactive applications. REScala automatically handles crashes and disconnects between devices, supporting application specific recovery strategies. The fault tolerance mechanism provided by REScala is mostly transparent to the programmer, it preserves strong consistency on local devices in the presence of faults, and it ensures eventual consistency across distributed devices. It has no performance overhead when no faults occur and acceptable overhead otherwise. Our evaluation shows that creating snapshots and recovering from them has comparable overhead to similar existing solutions.

There are several areas for future work. We have discussed distributed glitch freedom in Section 4. In future work, we plan to adapt the propagation algorithm of Drechsler et al. [15] to detect such glitches and to use the error propagation mechanism to enable developers to compromise between availability and correctness. Finally, we plan to formalize our programming model to provide rigorous guarantees about application correctness in the presence of crashes and disconnects.

References

1 Akka documentation. `http://akka.io/docs`, 2017.

2 Flink success stories. `https://cwiki.apache.org/confluence/display/FLINK/Powered+by+Flink`, 2017.

3 HTML5 localStorage. `https://www.w3schools.com/html/html5_webstorage.asp`, 2017.

4 Java microbenchmark harness. `http://openjdk.java.net/projects/code-tools/jmh/`, 2017.

5 Mehdi Ahmed-Nacer, Claudia-Lavinia Ignat, Gérald Oster, Hyun-Gul Roh, and Pascal Urso. Evaluating CRDTs for Real-time Document Editing. In *Proceedings of the 11th ACM Symposium on Document Engineering*, DocEng '11, 2011. `doi:10.1145/2034691.2034717`.

6 Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, Felix Naumann, Mathias Peters, Astrid Rheinländer, Matthias J. Sax, Sebastian Schelter, Mareike Höger, Kostas Tzoumas, and Daniel Warneke. The Stratosphere Platform for Big Data Analytics. *The VLDB Journal*, 2014. `doi:10.1007/s00778-014-0357-y`.

7 Joe Armstrong. Erlang. *Communications of the ACM*, September 2010. doi:10.1145/1810891.1810910.

8 Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. A Survey on Reactive Programming. *ACM Computing Survey*, 45(4), August 2013. doi:10.1145/2501654.2501666.

9 P. Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. Orleans: Distributed Virtual Actors for Programmability and Scalability. Technical report, (MSR-TR-2014-41, 24), 2014. URL: http://aka.ms/Ykyqft.

10 Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. Cloud Types for Eventual Consistency. In *European Conference on Object-Oriented Programming (ECOOP)*, 2012.

11 Andoni Lombide Carreton, Stijn Mostinckx, Tom Van Cutsem, and Wolfgang De Meuter. Loosely-coupled Distributed Reactive Programming in Mobile Ad Hoc Networks. In *Proceedings of the 48th International Conference on Objects, Models, Components, Patterns*, TOOLS'10, 2010. URL: http://dl.acm.org/citation.cfm?id=1894386.1894389.

12 Adam Chlipala. Ur/Web: A Simple Model for Programming the Web. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, 2015. doi:10.1145/2676726.2677004.

13 Gregory H. Cooper and Shriram Krishnamurthi. Embedding Dynamic Dataflow in a Call-by-value Language. In *Proceedings of the 15th European Conference on Programming Languages and Systems*, ESOP, 2006. doi:10.1007/11693024_20.

14 Evan Czaplicki and Stephen Chong. Asynchronous Functional Reactive Programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2013. doi:10.1145/2491956.2462161.

15 Joscha Drechsler, Guido Salvaneschi, Ragnar Mogk, and Mira Mezini. Distributed REScala: An Update Algorithm for Distributed Reactive Programming. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA, 2014. doi:10.1145/2660193.2660240.

16 Jan Dzik, Nick Palladinos, Konstantinos Rontogiannis, Eirik Tsarpalis, and Nikolaos Vathis. MBrace: Cloud Computing with Monads. *Proceedings of the Seventh Workshop on Programming Languages and Operating Systems*, 2013. doi:10.1145/2525528.2525531.

17 Jonathan Edwards. Coherent Reaction. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA, 2009. doi:10.1145/1639950.1640058.

18 Conal Elliott and Paul Hudak. Functional Reactive Animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, ICFP, 1997. doi:10.1145/258948.258973.

19 Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2), 2003. doi:10.1145/857076.857078.

20 Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A Network Programming Language. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP, 2011. doi:10.1145/2034773.2034812.

21 Seth Gilbert and Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *ACM SIGACT News*, 33(2), June 2002. doi:10.1145/564585.564601.

22 Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. Incremental Consistency Guarantees for Replicated Objects. *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016. arXiv:1609.02434.

23 Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, Robots, and Functional Reactive Programming. In *Summer School on Advanced Functional Programming 2002, Oxford University*. 2003. doi:10.1007/978-3-540-44833-4_6.

24 Rajesh K. Karmani and Gul Agha. Actors. In *Encyclopedia of Parallel Computing*. 2011. doi:10.1007/978-0-387-09766-4_125.

25 Jean-Claude Laprie. Dependable Computing: Concepts, Limits, Challenges. In *Proceedings of the Twenty-Fifth International Conference on Fault-tolerant Computing*, FTCS, 1995. URL: http://dl.acm.org/citation.cfm?id=1899254.1899261.

26 Jesse Liberty and Paul Betts. *Programming Reactive Extensions and LINQ*. 2011.

27 Ingo Maier and Martin Odersky. Deprecating the Observer Pattern with Scala.react. Technical report, 2012.

28 Alessandro Margara and Guido Salvaneschi. We Have a DREAM: Distributed Reactive Programming with Consistency Guarantees. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS, 2014. doi:10.1145/2611286.2611290.

29 Alessandro Margara and Guido Salvaneschi. On the Semantics of Distributed Reactive Programming: The Cost of Consistency. *IEEE Transactions on Software Engineering*, 2018.

30 R. Meier and V. Cahill. Taxonomy of Distributed Event-based Programming Systems. In *22nd International Conference on Distributed Computing Systems Workshops*, 2002. doi:10.1109/ICDCSW.2002.1030833.

31 Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: A Programming Language for Ajax Applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA, 2009. doi:10.1145/1640089.1640091.

32 Heather Miller, Philipp Haller, Normen Müller, and Jocelyn Boullier. Function Passing: A Model for Typed, Distributed Functional Programming. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward!, 2016. doi:10.1145/2986012.2986014.

33 Mark S. Miller, E. Dean Tribble, and Jonathan Shapiro. Concurrency Among Strangers. In *Proc. Int. Symp. on Trustworthy Global Computing*. 2005. doi:10.1007/11580850_12.

34 Florian Myter, Tim Coppieters, Christophe Scholliers, and Wolfgang De Meuter. I Now Pronounce You Reactive and Consistent: Handling Distributed and Replicated State in Reactive Programming. In *Proceedings of the 3rd International Workshop on Reactive and Event-Based Languages and Systems*, REBLS, 2016. doi:10.1145/3001929.3001930.

35 Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. Handling Partial Failures in Distributed Reactive Programming. *4th Workshop on Reactive and Event-based Languages & Systems*, 2017.

36 Ryan Newton, Greg Morrisett, and Matt Welsh. The Regiment Macroprogramming System. In *2007 6th International Symposium on Information Processing in Sensor Networks*, 2007. doi:10.1109/IPSN.2007.4379709.

37 Ivan Perez. Fault Tolerant Functional Reactive Programming. *International Conference on Functional Programming (ICFP)*, 2018.

38 Peter R. Pietzuch and Jean M. Bacon. Hermes: A Distributed Event-based Middleware Architecture. In *Proceedings. 22nd International Conference on Distributed Computing Systems Workshops*, 2002. doi:10.1109/ICDCSW.2002.1030837.

39 J. Ploski and W. Hasselbring. Exception Handling in an Event-Driven System. In *Availability, Reliability and Security. ARES.*, 2007. doi:10.1109/ARES.2007.85.

40 José Proença and Carlos Baquero. Quality-Aware Reactive Programming for the Internet of Things. In *Fundamentals of Software Engineering - 7th International Conference, FSEN*, 2017.

41 Bob Reynders, Dominique Devriese, and Frank Piessens. Multi-Tier Functional Reactive Programming for the Web. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward!, 2014. doi:10.1145/2661136.2661140.

42 G. Salvaneschi, S. Proksch, S. Amann, S. Nadi, and M. Mezini. On the Positive Effect of Reactive Programming on Software Comprehension: An Empirical Study. *IEEE Transactions on Software Engineering*, 43(12), Dec 2017. doi:10.1109/TSE.2017.2655524.

43 Guido Salvaneschi, Sven Amann, Sebastian Proksch, and Mira Mezini. An Empirical Study on Program Comprehension with Reactive Programming. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE, 2014. doi:10.1145/2635868.2635895.

44 Guido Salvaneschi, Gerold Hintz, and Mira Mezini. REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY, 2014. doi:10.1145/2577080.2577083.

45 Marc Shapiro, Nuno Pregui, Carlos Baquero, and Marek Zawirski. A Comprehensive Study of Convergent and Commutative Replicated Data Types. 2011.

46 Tom Van Cutsem, Elisa Gonzalez Boix, Christophe Scholliers, Andoni Lombide Carreton, Dries Harnie, Kevin Pinte, and Wolfgang De Meuter. AmbientTalk: Programming Responsive Mobile Peer-to-peer Applications with Actors. *Computer Languages, Systems & Structures*, 40(3-4), October 2014. doi:10.1016/j.cl.2014.05.002.

47 Tom Van Cutsem, Stijn Mostinckx, Elisa Gonzalez Boix, Jessie Dedecker, and Wolfgang De Meuter. AmbientTalk: Object-oriented Event-driven Programming in Mobile Ad Hoc Networks. *Proceedings - International Conference of the Chilean Computer Science Society, SCCC*, 2007. doi:10.1109/SCCC.2007.4396972.

48 Andreas Voellmy, Hyojoon Kim, and Nick Feamster. Procera: A Language for High-level Reactive Network Control. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN, 2012. doi:10.1145/2342441.2342451.

49 Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI, 2012. URL: https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf.