Skitter: A DSL for Distributed Reactive Workflows

Mathijs Saey
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
mathsaey@vub.ac.be

Joeri De Koster
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
jdekoste@vub.ac.be

Wolfgang De Meuter
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
wdmeuter@vub.ac.be

Abstract

Writing real-time applications that react to vast amounts of incoming data is a hard problem, as the volume of incoming data implies the need for distributed execution on a cluster architecture. We envision such an application can be created as a data processing pipeline which consists of a set of generic, reactive components, which may be reused in other applications. However, there is currently no programming model or framework that enables the reactive, scalable execution of such a pipeline on a cluster architecture. Our work introduces the notion of *reactive workflows*, a technique that combines concepts from scientific workflows and reactive programming. Reactive workflows enable the integration of these generic components into a single workflow that can be executed on a cluster architecture in a reactive, scalable way. To deploy these reactive workflows, we introduce a domain specific language, called *Skitter*. Skitter enables developers to write *reactive components* and compose these into reactive workflows, which can be distributed over a cluster by Skitter's runtime system.

CCS Concepts • **Software and its engineering → Domain specific languages**; *Distributed programming languages*; Data flow languages;

Keywords Scientific Workflows, Distributed Programming, Reactive Programming

ACM Reference Format:
Mathijs Saey, Joeri De Koster, and Wolfgang De Meuter. 2018. Skitter: A DSL for Distributed Reactive Workflows. In *Proceedings of the 5th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS '18), November 4, 2018, Boston, MA, USA*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3281278.3281281

REBLS '18, November 4, 2018, Boston, MA, USA

ACM ISBN 978-1-4503-6070-8/18/11…$15.00
https://doi.org/10.1145/3281278.3281281

1 Introduction

The ubiquity of smartphones and the advent of the "Internet of things" made it possible for companies to gather enormous amounts of real-time data. In order to create applications that provide useful information in response to this data, developers need to have the ability to *react* to these incoming data streams in real-time. Building these server-side **reactive big data** applications is far from trivial, as the scale of these data streams typically implies that a single computer cannot process the incoming data fast enough to react in a timely manner. Instead, such an application needs to run on a *cluster* to remain reactive.

Currently, two main approaches are used to create such applications: deploying a set of loosely coupled software components (i.e., microservices) over a set of machines, or using a big data framework. Leveraging microservices makes it easy to add new software components to a data processing pipeline, but forces developers to manually distribute the various components over the nodes in the cluster. Similarly, developers need to provide an ad hoc solution to recover the component in the case of *partial failure* — the failure of one or multiple nodes of the cluster. Big data frameworks, on the other hand, are explicitly designed to run on a cluster. Therefore, these systems automatically distribute their computations over a cluster while remaining resilient to partial failure. However, these frameworks are often *batch-based*, which prohibits reactivity. Stream processing systems [9], are big data frameworks that enable the creation of reactive big data applications. However, these frameworks limit expressivity to achieve scaling [2], and do not allow easy reuse of existing data processing software. This leads to difficult to reuse, tightly coupled, monolithic applications.

To facilitate the creation of reactive big data applications, we aim to design a system which has three important properties: **reactivity**, **scalability**, and **composability**. We touch on each of these properties in the following paragraphs.

Reactive The real-time nature of the applications we target implies that the external world can continuously push data into the system. In order to deal with this, our approach should be reactive: data processing components should be able to accept and process data as soon as it enters the system.

Horizontally Scalable In order to remain reactive when an ever increasing amount of data enters the system, we envision that this system should be horizontally scalable — it

should be able to scale when provided with additional computational resources (i.e., nodes in a cluster). Distributing a program over a cluster poses extra issues to be dealt with: first of all, one of the nodes of the cluster may fail in isolation, leading to a so-called partial failure. Thus, the system should be *fault tolerant*: it should be able to recover from the failure of one or multiple nodes in the system. Second, the components in an application need to be distributed over the cluster; this is often done by *replicating* these components. However, replicating various components introduces a new set of problems as any shared state between these components needs to be synchronized over the network.

Composable In order to enable the reuse of data processing components between various applications, applications should be built in a composable, component-oriented way. To reduce the engineering effort required to create reactive big data applications, we explicitly target the reuse of existing data processing software as components in this system.

To enable the creation of scalable, reactive big data applications from a set of (existing) components, we introduce an approach inspired by by reactive programming [4], where the workflow and its components are automatically activated based on data from the outside world. We call such a workflow a *reactive workflow*. Such a workflow system, which is reactive at its core, has not been explored yet by the state of the art.

In this paper, we present *Skitter*[1], a novel, domain specific language. Skitter is centered around reactive workflows, which consist of a set of connected *reactive components*. These components represent a single data processing step that is automatically executed when data enters the reactive workflow, or when a connected component produces new data. Reactive workflows can be executed on a cluster by Skitter's runtime system, which automatically handles distribution, replication and partial failure concerns.

To enable the reuse of existing data processing software, Skitter defines a protocol to write *wrappers* around existing applications, which may be written in a 3rd party language. Existing software may generate side-effects (e.g., I/O) which cannot be tracked by Skitter, but which may influence how this software can be distributed over a cluster. To track these concerns, Skitter introduces the notion of *effects*. These effects enable a component developer to provide additional information about the side-effects a component may generate when it reacts to incoming data. In turn, Skitter's runtime system uses this data to handle the aforementioned partial failure and replication concerns.

This work provides the following contributions: first, we introduce the notion of reactive workflows. Second, we provide a domain specific language, called Skitter, which makes it possible to express and execute such reactive workflows on a cluster. This is done in such a manner that this workflow is resilient to partial failure. Third, we introduce the notion of component effects, which enable the reuse of existing data processing software inside a (reactive) workflow. Fourth and finally, we provide an implementation of effects inside Skitter. The introduction of Skitter and reactive workflows (Section 2), is accompanied by an evaluation of Skitter (Section 3), and a discussion of related and future work (Sections 4 and 5, respectively).

2 Skitter

In this section, we define *reactive workflows* and their related concepts. Reactive workflows form the basis of Skitter, which is also discussed in this section. Specifically, we discuss how Skitter makes it possible to create and execute reactive workflows, and how it enables the reuse of existing data processing software as reactive components.

Reactive workflows are data processing applications which consist of a set of connected *reactive component instances*. Each of these instances automatically react every time data arrives from the external world, or when a connected component instance produces data. Reactive components define a data processing step that is considered atomic for the workflow program and which is reusable across various reactive workflows. A component can occur multiple times in the same workflow; a single occurrence of a component in a workflow is called a component *instance*. Depending on the functionality of the component, an instance stores a potentially mutable state, and may be *replicated* by the underlying runtime system. We provide additional information about each of these concepts in the following sections.

Skitter programs consist of two parts: the textual definition of reactive components and the visual composition of these components into a reactive workflow. We envision that the reactive components are written by developers with experience in reactive or distributed programming. The workflows can be designed visually by domain experts, who may not have any programming experience. Skitter's underlying runtime can then distribute the execution of a reactive workflow over a cluster. An example of such a visual workflow can be found in Figure 1.

We implemented Skitter on top of the Elixir[2] programming language. We decided on Elixir for its focus on distributed systems and because it runs on top of the Erlang VM, which has a proven track record of scaling to large systems (used by Amazon, Facebook, Ericsson, ...). Furthermore, Elixir implements the actor model [1] which provides a natural way to treat a component instance as an isolated execution unit.

[1] Skitter can be found online at https://github.com/mathsaey/skitter

[2] https://elixir-lang.org/

2.1 Component Definition

A reactive component is a collection of functions and meta-information. The functions define how the component *reacts* to incoming data, and (optionally) how its state is created, destroyed, and persisted. The meta-information defines how the component can be embedded inside a workflow and the effects it may generate when reacting.

```
1 component FahrenheitToCelsius, in: [fahrenheit], out: [celsius] do
2   react fahrenheit do
3     ((fahrenheit - 32) * (5 / 9)) ~> celsius
4   end
5 end
```

Listing 1. A trivial reactive component which converts any temperature it receives from Fahrenheit to Celsius.

Before we discuss how Skitter handles state and I/O, we will discuss the basic meta-information and function implementations that every component needs to provide. Listing 1 contains the definition of a simple component that converts each number it receives from Fahrenheit to Celsius. As this component is stateless, it only foresees the `react` function, which defines the action that is triggered by Skitter every time the component receives new data. Since a reactive component is designed to process incoming data, every component has to define such a `react` function.

Besides this, a component definition must specify how other components can be connected to itself before it can be embedded inside a workflow. Therefore, each component specifies a set of in and out ports. The in ports of a component specify which data an instance of this component can receive and react to. Out ports specify the data a component can publish. While reacting, a component may publish data on an out port through the use of the *spit* operator (i.e., the ~> shown on line 3). This data is automatically sent to any component that is connected to the out port. Since a component needs to receive data to react, it must always specify at least one in port. Specifying an out port, on the other hand, is not required.

It is often useful to associate some state with a component instance. For instance, a component instance may need to be initialized with some parameters in order to remain sufficiently generic. An example of such a component is shown in Listing 2. This component filters data points based on whether or not they are located within a certain area. In order to keep this component generic, the target area — which is provided by the user in the workflow definition — is provided to the component when it is initialized by the runtime system. The `init` function will receive this initialization argument and use the update operator (<~ on line 6) to store this state in a *field*; when the component is reacting to incoming data, it can read the state associated with the instance from these fields (e.g., on Line 12). Note that the fields of a component instance need to be statically defined in the

```
1  component GeoFilter, in: [geo_json], out: [inside, outside] do
2    fields area
3
4    init area_string do
5      area_struct = ... # Convert area_string into native format
6      area <~ area_struct
7    end
8
9    react geo_json do
10     coord = ... # Extract geo data and convert into native format
11
12     if Topo.within?(coord, area) do
13       geo_json ~> inside
14     else
15       geo_json ~> outside
16     end
17   end
18 end
```

Listing 2. A stateful reactive component which filters out data points based on whether or not they are located within a given area.

component definition (Line 2). In order to achieve horizontal scalability, Skitter will automatically replicate component instances over a cluster. To make this possible, the state of a component instance is immutable by default.

```
1  component Average, in: [number], out: [current_average] do
2    fields total, counter
3    effect state_change
4
5    init _ do
6      total <~ 0
7      counter <~ 0
8    end
9
10   react number do
11     total <~ total + number
12     counter <~ counter + 1
13
14     total / counter ~> current_average
15   end
16 end
```

Listing 3. A component with a mutable state, that calculates the average of all values it received. Note the effect declaration on line 3.

The state of a Skitter component is immutable by default. However, some components do need to access a mutable state. For instance, any component that performs some form of aggregation over its inputs needs some way to preserve the aggregated value between invocations. In order to support mutable state while remaining scalable by default, Skitter components may only modify their state while reacting if they explicitly specify this behavior. An example of a component with a mutable state is shown in Listing 3. This component stores an average of all the numbers it receives; the current average is spit to the current out port every time the component instance reacts to data. A component can specify that it might modify its state while reacting by specifying the `state_change` *effect* (shown on Line 3). By specifying this effect, a component developer signals the Skitter runtime

that a component instance may modify its state while it is reacting. In turn, Skitter's runtime engine will ensure that this state is recoverable in the case of partial failure. It will also adjust its replication strategy to ensure that the instance state remains consistent.

Instead of using an effect system, Skitter could alternatively analyze the source code of a component to infer if it updates its state while reacting. We decided on the use of an effect system over a tracking system based on three reasons: existing data processing software, consistency, and explicitness. First and foremost, Skitter allows the reuse of existing data processing software as a component. This reuse is typically achieved by writing a wrapper around this existing data processing software. We aim to support existing data processing software regardless of the technology that was used to create this software. Since it is infeasible to create a tracking system that can verify whether an arbitrary piece of software has a form of internal state, our approach uses an effect system. Second, the `state_change` effect is not the only effect that is supported by Skitter, for instance, later in this section, we discuss an effect which specifies that a component may perform I/O. Since arbitrary Elixir code may be embedded inside `react`, it is infeasible to track whether or not this code performs any I/O. Therefore, any I/O effect should be explicitly declared. In order for the language to remain consistent, we decided that every effect should be specified explicitly. Third, when an automatic tracking system is used, the `state_change` effect could be activated by an accidental use of the update operator. Since the activation of this effect has major implications for the performance of the overall reactive workflow, we prefer the use of an explicit effect system.

Reusing Existing Software As mentioned, Skitter is designed to allow the reuse of existing data processing software. This is generally done by writing a *wrapper* around an existing piece of software that may be written in another programming language; an example of such a wrapper can be seen in Listing 4. A component developer that writes a wrapper around existing software may not always be able to pass the internal state of this software to Skitter. This may be practically infeasible, especially when the software is written by another developer, or when its source code cannot be accessed. Therefore, component developers can avoid passing data to Skitter's runtime by specifying that the mutable state of their component instance is hidden from Skitter. This is done by adding the `hidden` property to the `state_change` effect (shown on Line 2). Specifying this property allows a component to avoid passing its state to Skitter. Instead, this component instance manages its own state. However, in order to guarantee fault tolerance, these components are required to persist their state when this is requested by Skitter's runtime system. This is done through Skitter's *checkpoint* mechanism, which allows the runtime to request the creation of a new checkpoint (through `create_checkpoint`), or the removal of an old one (`clean_checkpoint`). Should failure occur, Skitter can restore the state of the component instance by recovering the latest checkpoint (`restore_checkpoint`). This mechanism is discussed in additional detail in Section 2.4.

```
1  component NearbyUsers, in: [user, location], out: [nearby] do
2    effect state_change hidden
3    fields exec
4
5    init _ do
6      exec <~ Executable.start("user_tracker")
7    end
8
9    terminate do
10     Executable.send(exec, "quit")
11   end
12
13   react usr, loc do
14     Executable.send(exec, "update_location #{usr} #{loc}")
15     Executable.read(exec, "nearby_users #{loc}") ~> nearby
16   end
17
18   create_checkpoint do
19     Executable.send(exec, "checkpoint")
20   end
21
22   clean_checkpoint checkpoint do
23     Executable.send(exec, "clean_checkpoint #{checkpoint}")
24   end
25
26   restore_checkpoint checkpoint do
27     Executable.start("user_tracker --from-checkpoint #{checkpoint}")
28   end
29 end
```

Listing 4. A component that reuses an existing application to find users near a given location. Note the checkpoint functions and the use of `terminate`.

```
1  component Archive, in: [data] do
2    effect external_effect
3    fields conn, table
4
5    init {url, username, password, table_name} do
6      conn <~ Database.open_connection(url, username, password)
7      table <~ table_name
8    end
9
10   terminate do
11     Database.close_connection(conn)
12   end
13
14   react data do
15     after_failure do
16       res = Database.get_by(table, data.id)
17       if res != nil do
18         skip
19       end
20     end
21     Database.write(table, data)
22   end
23 end
```

Listing 5. A component that performs I/O. Note the use of `after_failure` and `skip`. `skip` aborts the current call to *react*, but does not undo any spits or state updates that already occurred.

Skitter provides one effect besides the `state_change` effect: `external_effect`. This effect specifies that a component may cause some external effect — i.e., I/O — while it reacts to incoming data; an example of such a component can be found in Listing 5. Skitter defines this effect due to the influence that the occurrence of I/O has on the recovery from a partial failure. Consider what happens when a cluster node crashes while some component is reacting. When the component does not generate any external effects, the runtime can re-execute the call to react, and proceed with the execution of the workflow. However, if this component might cause an external effect, re-executing react may cause the same external effect to be activated twice. Not re-executing react would lead to the workflow ending up in an inconsistent state, as any connected components will not be activated. To deal with this issue, Skitter offers the `after_failure` block. This block can be used inside `react`, and will only be executed if the current invocation of `react` occurs after a previous call to `react` with the same data did not complete due to partial failure. Note that the same `after_failure` block can be activated multiple times when a call to `react` fails multiple times. Shortly put, the `after_failure` block enables developers to verify if an external effect already occurred and to deal with this accordingly.

Skitter's component definition language allows developers to write reactive components by specifying a set of meta-information about the component and by implementing a small set of key functions, which are summarized in Table 1. The meta-information defines how a workflow can be embedded inside a workflow and specifies which effects a component may generate. The functions allow a component to react to incoming data, and manage state throughout the lifetime of the component instance.

2.2 Workflow Definition

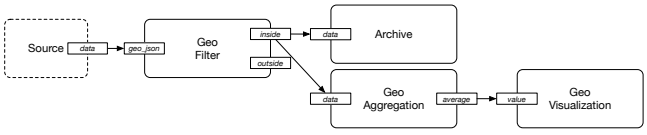

Figure 1. A Skitter workflow that calculates and visualizes the average noise level within a given geographical area.

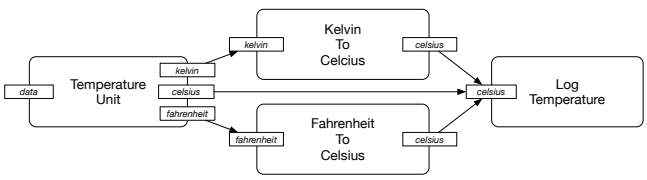

Figure 2. A legal use of multiple incoming links for a single in port.

A reactive workflow consists of three entities: a set of reactive component instances, a *source*, and a set of *links*; an example of a reactive workflow in Skitter can be seen in Figure 1. Reactive component instances are defined based on their component type and the data which is used to initialize them. In Skitter, this data is provided through the use of a contextual menu (not shown here). The source of a reactive workflow is an entity with an out port that connects a reactive workflow to the external world. Links represent connections between in and out ports in a reactive workflow. For reasons which we explain in Section 2.3, a reactive workflow must process each incoming data record in isolation and may not contain cycles. A workflow has to satisfy certain conditions in order to be valid:

- A component instance can only process one input per workflow invocation. Therefore, a reactive workflow must be built in such a way that an in port receives at most one data record per invocation. It is worth noting that this does not imply that an in port cannot have more than one incoming link. This is the case because components may spit values based on conditions. An example of a legal use of multiple incoming links for a single in port can be found in Figure 2.
- A component instance can only react when it has received an input for each in port; if a component instance only receives inputs for a subset of its in ports at a given invocation, it will store these inputs forever without reacting, thus leaking memory. Therefore, the following two conditions must hold: first, every in port must be connected to at least one out port. Second, a reactive workflow must be built in such a way that an in port can not receive a partial input for an invocation. These conditions are not identical, as an in port that is connected to an out port does not necessarily receive data from its connected out port; this can happen when a component in the workflow spits a value based on a condition.
- An out port does not have to be connected to an in port, and may be connected to multiple in ports. Any value spit to an out port with no connections will be discarded. A value that is spit to an out port that has multiple outgoing connections is copied and sent to each connected in port.

In order to enable non-programmers to create reactive workflows, we plan to enable the composition of existing components in a visual programming language. An example of a workflow designed in such a language can be seen in Figure 1. In this example, we aim to visualize a metric (e.g., the noise level) of a specific area in a city. All the noise and coordination data enters the `GeoFilter` defined in Listing 2, which filters the data within the desired area. All the measurements inside the area are persisted by the `Archive` component,

Table 1. Summary of the functions a reactive component written in Skitter can implement.

Name	Arguments	State Access	Description
`react`	One for each in port	read†	Process data
`init`	Single initialization argument	read / write	Initialize instance state on creation
`terminate`	*None*	read	Clean up resources before instance removal
`create_checkpoint`‡	*None*	read	Create checkpoint of current state
`restore_checkpoint`‡	Checkpoint to restore	read / write	Recover instance from existing checkpoint
`clean_checkpoint`‡	Checkpoint to remove	read	Remove old checkpoint

† read / write when the `state_change` effect is present.
‡ Only available when `state_change hidden` is present.

which is defined in Listing 5. The `GeoAggregation` component computes various metrics of our target area, which are visualized in a live view by the `GeoVisualization` component.

```
workflow do
  _ = {Source, _, data ~> filter.geo_json}
  filter = {
    GeoFilter, "...", # Target area represented as a geojson string
    inside ~> datastore.value,
    inside ~> aggregator.value
  }
  data_store = {DataStore, "noisetube raw data"}
  aggregator = {GeoAggregation, _, average ~> visualizer.value}
  visualizer = {GeoVisualisation, _}
end
```

Listing 6. Textual representation of the workflow shown in Figure 1.

As a first step towards such a visual language, Skitter offers a textual workflow definition language shown in Listing 6. This example maps directly to the visual representation shown in Figure 1. In this language, a workflow is represented by a list of named component instances. Each instance consists of the name of the component, the initialization argument and a set of links. Finally, Skitter provides a primitive component, `Source`, with a single out port. This primitive component represents the source of a reactive workflow.

2.3 Workflow Execution

In order to scale with the amount of computational resources at its disposal, Skitter attempts to exploit the parallelism which is present in a given workflow. There are two sources of parallelism which can be exploited in a reactive workflow: the parallelism inherent in the workflow, and the parallelism which can be obtained from processing incoming data records in parallel.

In order to extract the parallelism which is present in a given workflow, Skitter uses the *dataflow model* [7]. In this model, an operation (i.e., an invocation of `react`) can be executed when all of its inputs are present. We decided to use this model due to its ability to extract the latent parallelism present in a given program, as multiple operations are allowed to execute in parallel, if this is permitted by the data dependencies. For instance, in the example workflow shown in Figure 1 the `Archive` and `GeoAggregation` component instances can react to incoming data in parallel.

The data-driven nature of reactive workflow make it possible to process the various data records which enter the system in isolation. To achieve this, our execution strategy relies on a variant of the dataflow model, called *tagged-token* dataflow [3]. Processing the various data records in parallel implies that various invocations of `react` may be active concurrently for a single component instance. This enables components with an immutable state to scale with the amount of incoming data. However, when components do need to change their state, the Skitter runtime needs to ensure that this state remains consistent.

2.4 Component Orchestration

Since every invocation of `react` effectively occurs in isolation, Skitter needs to ensure that the states of the component instances in a reactive workflow remains consistent. This is done by the component orchestration algorithm. This algorithm manages the creation of component replicas and synchronizes changes between them. Furthermore, this algorithm ensures that changes to this state are not lost if partial failure occurs.

Table 2. Amount of replicas that can be created for a component instance depending on the effects it has on its state. External effects are omitted as they do not influence the amount of replicas. *n* implies that an arbitrary amount of replicas can be created.

Immutable State	*n*
Mutable State	1 or *n* when synchronized
Hidden Mutable State	1

Table 2 summarizes the various replication strategies Skitter can employ. When the state of a component instance is immutable, or when it has no state at all, a runtime can create an arbitrary amount of replicas of this instance. Skitter creates a new replica of a stateless instance whenever it needs to react to a new data record; when a replica has finished reacting, it is eliminated. This is possible due to the lightweight nature of Elixir processes (i.e., actors), which are cheap to create and destroy. When a component has a mutable state, Skitter can spawn a single replica and ensure that all requests get served by this replica. Alternatively, Skitter can spawn an arbitrary amount of replicas and use a synchronization mechanism to ensure that the state of these replicas remains consistent. Since it is expensive to synchronize an arbitrary mutable state in a distributed system, Skitter currently uses the former approach; in future work, we aim to investigate language-level mechanisms which enable the efficient synchronization of the state of a component instance. Finally, Skitter's runtime cannot control the state of a component instance which hides its mutable state. Therefore, only a single replica of such an instance is created.

An instance replica can be created at two points: when the workflow is loaded by Skitter, and when a data record arrives. In both cases, the component orchestration algorithm must decide where to create this replica. When Skitter loads a workflow, it creates a single replica for each component with a mutable state. Currently, these replicas are distributed over the cluster according to a round-robin strategy. When a data record enters the system, Skitter must decide where a record will be processed. As each data record is processed in isolation (as discussed in Section 2.3), a data record can be processed on any node. The chosen node will process the received data record, creating instance replicas or forwarding records to the appropriate replica as needed. Once again, Skitter selects a node according to a round-robin strategy. The round-robin strategy that is used to determine where a record is processed, and where a replica with a mutable state is created does not take the current load of the nodes in the cluster into account. In the future, we will dynamically select the appropriate node to process data records and move replicas between nodes based on the current load.

Table 3. Steps to recover a component instance after a failure depending on its effects.

	No Effects	External Effects
Immutable State	replay	replay[†]
Mutable State	restore, replay	restore, replay[†]
Hidden State	restore, replay[‡]	restore, replay[†‡]

[†] Replay should use `after_failure`.
[‡] Replay all inputs since last checkpoint.

Table 3 summarizes the approach Skitter takes to deal with partial failure. When a component instance is reacting on a cluster node that crashes, Skitter automatically replays the react invocation on a replica on a different machine. Any spits that already occurred during the crashed invocation of react may safely be replayed, as spit data is only forwarded to connected component instances after an invocation of react has finished. When a component has external effects, Skitter will use the `after_failure` mechanism while replaying. If the component has a mutable state, Skitter will fetch the latest copy of the instance state before reacting. If the mutable state is hidden, any input that was sent to the instance after the latest checkpoint will be replayed, the `after_failure` mechanism will be used for this if needed; while recovering, data that is spit during the replay of a react is ignored. Due to the way Skitter replays inputs in the case of partial failure, the react function of components with a mutable state is expected to be deterministic: when provided with the same inputs and the same state, it should return the same state.

3 Evaluation

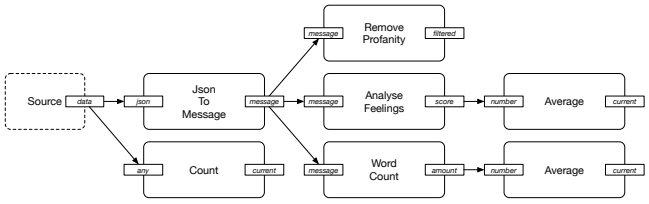

Figure 3. Tweet processing workflow.

In this section, we evaluate Skitter. Based on a tweet processing workflow shown in Figure 3, we provide a quantitative evaluation of the scaling behavior of a preliminary version of the Skitter runtime system, and briefly discuss to what extent a component developer needs to reason about distribution.

The tweet processing workflow receives a set of tweets, which are represented as JSON strings, and uses various data processing software to perform operations on these tweets. First, the workflow counts the total amount of tweets it has received over its lifetime. Second, the workflow decodes the json and extracts the tweet message from this decoded json. This message is sent to various other steps, which remove profanity from the tweet, rate the sentiment of the tweet, and count the amount of words contained within the tweet. The results of the latter two operations are sent to two `Average` components, which track the average word count and sentiment score of all tweets. It is worth noting that most of these components are built using existing data processing software. Specifically, the `RemoveProfanity`[3],

[3] https://github.com/xavier/expletive

`AnalyseFeelings`[4], and `JsonToMessage`[5] components are built using existing Elixir code. We discuss our experiments and their results in the next few paragraphs.

Experimental Setup In order to measure the throughput of Skitter and its scaling behavior, we set up the following experiment[6]. We used the workflow shown in Figure 3, and let it process a set of tweets which were scraped from twitter at an earlier point in time. The aforementioned tweets were read from a file and pushed into Skitter one by one; once the workflow finished processing all the data, Skitter shut itself down. Using the `time` command, we measured the time that elapsed between the first data record entering the system and the successful termination of Skitter. Afterwards, we calculated the average amount of tweets Skitter processed every second based on the elapsed time and the provided amount of data. We repeated this process multiple times, and report the average throughput in this paper. To measure the scaling behavior of our system, we repeated this experiment multiple times with a varying amount of cluster nodes at Skitter's disposal. Furthermore, we repeated the experiment with a different amount of tweets. Our benchmarks were executed on a cluster which consists of 11 nodes. Each of these nodes is equipped with a 4-core Intel® E5-1620 Xeon® CPU with 8 hardware threads running at 3.50GHz and with 32GB of RAM. Communication between the nodes is done using a 10 Gigabit Ethernet connection. Each node ran `Ubuntu 16.04.05` and used `Elixir 1.7.1/OTP 21`. Every node ran a single Elixir instance, which was set up to use 8 threads (i.e., one for every hardware thread). One of the nodes was used to push data into the system, while the ten other nodes were used as workers for Skitter.

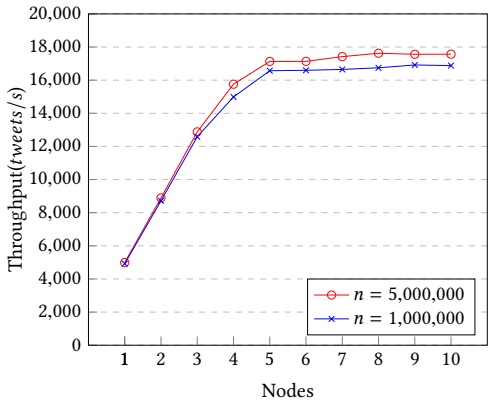

Figure 4. Throughput of the tweet processing workflow for n tweets.

[4]https://github.com/dantame/sentient
[5]https://github.com/michalmuskala/jason
[6]The code of our benchmark can be found online at: https://soft.vub.ac.be/~mathsaey/artefacts/rebls-2018-evaluation.zip

Results The results of our experiments can be seen in Figure 4. These results indicate Skitter can scale along with the amount of computational resources at its disposal for this benchmark. In both benchmarks, Skitter's throughput initially increases with the amount of nodes at its disposal. This trends slows down after the addition of the fourth node, and ends after the addition of the fifth node. Every node that is added to the system after this point has a negligible impact on Skitter's throughput. After this point, the CPU's of the workers are no longer fully saturated, showing that the throughput is effectively dictated by the speed at which data enters the system. As this speed is bounded by the speed of the network, Skitter's throughput is effectively capped after this point. This initial benchmark seems to indicate Skitter can scale effectively along with the amount of computational resources at its disposal. However, as we only provide a single experiment, we cannot draw any permanent conclusions. In future work, we aim to examine Skitter's throughput and scaling behavior in a multitude of scenarios.

Distribution Concerns A developer who writes code for a distributed system must reason about 4 concerns: distribution (i.e., deploying the computations over the various nodes in the cluster), communication, fault tolerance and replication. Frameworks or languages that are designed to run on a cluster will often hide some of these concerns from a developer. In this paragraph, we discuss these concerns, and how they are handled by Skitter. We do this based on our experience developing the tweet processing workflow. Specifically, we investigate the amount of code we had to write that explicitly deals with distribution. Overall, the entire tweet processing application only contains two lines of code that deal with distribution. Both of these lines are `effect` declarations. Besides these two lines the remaining application code deals with data processing or workflow logic. In contrast, creating an ad hoc implementation of this application in Elixir would require one to manually distribute and replicate data processing steps over the cluster machine. Furthermore, any recovery from partial failure would have to be provided by the developer. Overall, Skitter is designed in such a way that one needs to write very little code that deals directly with distribution: the effect system, checkpoints and the `after_failure` block are the only elements present in Skitter which force the developer to deal directly with distribution. Out of these features, the checkpoint system is only required in the case that the state of a component is not managed through Skitter.

4 Related Work

Our work aims to create reactive, scalable workflows built from existing data processing software. Current approaches for large-scale or data driven processing, such as stream

processing [9], scientific workflows [8] or reactive programming [4] address some of the issues we aim to tackle. In this section, we briefly compare these approaches to our work.

Stream processing Throughout the last decade, there has been a great deal of work that focuses on large-scale data processing (commonly known as *Big Data* processing). More recently, some of this work [10, 13] started to focus on processing real-time data with *stream processing* frameworks. These frameworks are highly relevant to our problem area, and are designed with distributed execution in mind. Unfortunately, to achieve scaling, these frameworks only allow a programmer to use a limited set of primitives. While some of these primitives enable the invocation of external data processing software, they don't provide the abstractions which are required to replicate these programs over a cluster.

Scientific workflows Previous work makes it possible to combine existing data processing software into *scientific workflows* [5, 12]. However, none of these systems are reactive, i.e. they don't automatically start to process data when it it arrives from some external data source (such as a cellphone or a sensor). Instead, these tools are almost entirely *query-driven*. A consequence of this query-driven nature is that scientific workflow systems are all inherently *batch-based*: they work on a complete data set, which is not compatible with the real-time nature of the applications we target.

Reactive programming Work in this area lead to the design of programming languages that automatically respond to data as soon as it arrives from an external data source. While distributed reactive programming languages exist (e.g.,[6]), they are not designed to be used on a cluster. Therefore, these languages do not automatically distribute a program over a cluster machine, instead, programmers have to manually specify where each computation is executed. Furthermore, reactive languages have issues dealing with long lasting computations and effectful statements [11], which are required by contemporary data processing applications.

5 Future Work

So far, our work on Skitter mainly focused on the design of language abstractions that allow the creation of reactive components which can be distributed over a cluster while remaining resilient to partial failure. In future work, we would like to investigate how we can improve the scaling behavior of Skitter. Before we do so, we aim to provide a comprehensive benchmark set that we can use to evaluate the runtime behavior of Skitter in more detail. We mainly aim to improve Skitter's scaling behavior in two ways: stateful component replication and scheduling. We discuss these in the next paragraphs.

Comprehensive Benchmarks In order to gain a better understanding of Skitter's behavior when it is executing different workflows under varying amount of loads, we aim to create a benchmark suite for Skitter. Such a suite would contain various workflows, which contain a different combination of components of varying computational intensity with varying effects. Such a benchmark suite would allow us to gain a better understanding of Skitter's limitations and would aid us in identifying areas where Skitter can be improved. Furthermore, we would like to implement common big data applications in order to compare Skitter with existing tools.

Stateful component replication Components with a mutable state are currently not replicated by Skitter. This prevents them from scaling together with the amount of incoming data they receive, which can slow down an entire reactive workflow. In the future, we would like to investigate mechanisms that allow us to replicate component instances with a mutable state. Particularly, we aim to enhance Skitter with mechanisms that can merge the state and spits of the various replicas of a component instance, which each receive a part of the data the component instance receives.

Load Balancing Skitter's current runtime system uses a simple approach to distribute the various component replicas and data records over a cluster, which does not take the current load of the cluster nodes into account. In the future, we will dynamically select the appropriate node to process an incoming data record. Furthermore, we will move replicas to different nodes when needed. To do this, we will investigate the impact various load balancing techniques may have on Skitter's absolute performance and scaling behavior.

6 Conclusion

We facilitated the creation of reactive big data applications by building a system that enables the creation of reactive, scalable applications out of composable components. We achieved this by introducing Skitter and *reactive workflows*.

Reactive workflows combine notions from the fields of reactive programming and scientific workflows. Reactive workflows consist of a set of connected *reactive components*, each of which represents a single data processing step that is automatically executed when data enters the reactive workflow, or when a connected component produces data.

Skitter is a domain specific language which makes it possible to write reactive components which can be composed into reactive workflows. These reactive workflows can be executed on a cluster by Skitter's runtime system, which automatically handles replication and fault tolerance concerns.

Skitter's runtime system aggressively replicates each component instance over a cluster. However, components may modify their state, or generate I/O, which can cause issues when various replicas of this instance exist. Therefore, we

introduce effects. Effects allow a component developer to specify additional information about the side-effects a component may generate when it reacts to data. Skitter's runtime system uses this information to ensure the reactive workflow can be executed efficiently while remaining resilient to partial failures.

Acknowledgments

This work is funded by the *FLAMENCO* project of the Flemish agency for Innovation by Science and Technology.

References

[1] Gul Agha. 1986. Actors: A Model of Concurrent Computation in Distributed Systems. (1986).

[2] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. 2015. The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1792–1803. https://doi.org/10.14778/2824032.2824076

[3] Arvind and R. S. Nikhil. 1990. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Trans. Comput.* 39, 3 (March 1990), 300–318. https://doi.org/10.1109/12.48862

[4] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. 2013. A Survey on Reactive Programming. *Comput. Surveys* 45, 4 (Aug. 2013), 52:1–52:34. https://doi.org/10.1145/2501654.2501666

[5] Paolo Di Tommaso, Maria Chatzou, Evan W Floden, Pablo Prieto Barja, Emilio Palumbo, and Cedric Notredame. 2017. Nextflow Enables Reproducible Computational Workflows. *Nature Biotechnology* 35 (April 2017), 316. https://doi.org/10.1038/nbt.3820

[6] Joscha Drechsler, Guido Salvaneschi, Ragnar Mogk, and Mira Mezini. 2014. Distributed REScala: An Update Algorithm for Distributed Reactive Programming. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 361–376. https://doi.org/10.1145/2660193.2660240

[7] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. 2004. Advances in Dataflow Programming Languages. *Comput. Surveys* 36, 1 (March 2004), 1–34. https://doi.org/10.1145/1013208.1013209

[8] Ji Liu, Esther Pacitti, Patrick Valduriez, and Marta Mattoso. 2015. A Survey of Data-Intensive Scientific Workflow Management. *Journal of Grid Computing* 13, 4 (Dec. 2015), 457–493. https://doi.org/10.1007/s10723-015-9329-8

[9] Saeed Shahrivari. 2014. Beyond Batch Processing: Towards Real-Time and Streaming Big Data. *Computers* 3, 4 (Oct. 2014), 117–129. https://doi.org/10.3390/computers3040117

[10] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. 2014. Storm@Twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14)*. ACM, New York, NY, USA, 147–156. https://doi.org/10.1145/2588555.2595641

[11] Sam Van den Vonder, Joeri De Koster, Florian Myter, and Wolfgang De Meuter. 2017. Tackling the Awkward Squad for Reactive Programming: The Actor-Reactor Model. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS 2017)*. ACM, New York, NY, USA, 27–33. https://doi.org/10.1145/3141858.3141863

[12] Michael Wilde, Mihael Hategan, Justin M Wozniak, Ben Clifford, Daniel S Katz, and Ian Foster. 2011. Swift: A Language for Distributed Parallel Scripting. *Parallel Comput.* 37, 9 (2011), 633–652. https://doi.org/10.1016/j.parco.2011.05.005

[13] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. 2012. Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters. *HotCloud* 12 (2012), 10–10.