RHEA: A Reactive, Heterogeneous, Extensible and Abstract Framework for Dataflow Programming

Anonymous Author(s)

Abstract

Robotics and IoT applications are perfect candidates that can benefit from the functional reactive programming paradigm. Moreover since the typical program can be represented as a dataflow graph the application can be conceptually separated and distributed in different machines and the several graph partitions can run in parallel and possibly in different execution stacks. In this paper we propose a general-purpose reactive framework that can express complex applications seamlessly and transparently integrating different sources and middlewares. The framework is abstract and extensible making it easy to integrate well-established technologies that rely on the PubSub model. We demonstrate the usability of the framework by providing applications in the domain of robotics and IoT.

Keywords dataflow programming, stream processing, functional reactive programming (FRP), declarative languages, implicit concurrency, node placement

1 Introduction

A typical application in Robotics or Internet of Things (IoT) needs to timely and continuously respond to time-varying external sensory data and as a result the reactivity of these applications is imperative. Typically, the programmer of such applications has to deal with asynchronous callbacks in conventional imperative programming languages in order to implement tedious and often error-prone behaviours that should comply with the reactive requirements.

A promising and relatively recent proposal for simplifying the implementation of reactive applications is the *functional reactive programming* (FRP) [7]. FRP makes heavy use of higher-order functional operators to define, essentially, a dataflow network of processing nodes. These high-level abstractions alleviates, as intended, the low-level implementation chores. FRP was originally proposed though as a framework for developing graphical user interfaces but fortunately the key high-level abstractions are generic enough that other domains can benefit from this approach. As a result of its generality and its popularity several general-purpose implementations emerge with different capabilities and prerequisites.

It is natural, therefore, to investigate whether robotics and IoT applications can fit into this new paradigm. Indeed most robotic applications follow the Robot Perception Architecture, where inputs to system are the robot's sensors, which are then processed by a dataflow graph, whose output is given as commands to the robot actuators. Moreover, robotics typically involve several different other robotic or IoT systems to enhance their sensing abilities. This combination naturally give rise to issues of distributing the dataflow graph to several robotic units and issues of heterogeneity and interoperability between different middlewares and protocols.

The first steps towards using FRP in robotics was identified in [9] and realized in Yampa[1], an FRP framework developed in Haskell. Yampa provides its functionality through an embedded DSL as it is customary for many of the FRP libraries. Although an interesting proposal, there is limited acceptance from the robotics community mainly because it does not integrate well with existing well-established robotics middlewares such as the Robot Operating System (ROS) [16]. Moreover, it does assume distributed execution and integration with other systems via the Reactive Streams Standard[2].

Motivated by the robotics and IoT community we propose RHEA, an abstract FRP general-purpose framework that aims to act as a unifying layer that can be mapped and executed simultaneously using different reactive libraries and existing middlewares such as ROS and MQTT. The programmer can transparently express a complex reactive applications within this framework that may use both sensing from several robots and IoT sensors. The framework places the dataflow nodes to computational resources and handles the serialization needed between different execution engines.

The rest of the paper is structured as follows. Section 2 provides some background context about dataflows and the state-of-art middlewares used in robotics and IoT. Section 3 presents the framework's architecture and capabilities. Section 4 discusses implementation details while Section 5 presents several optimizations that have been implemented in the framework. Section 6 demonstrates some use-cases of the framework mainly motivated by robotics and IoT. Section 7 discusses related work and finally Section 8 concludes with future directions.

REBLS'18, November 04–09, 2018, Boston, MA, USA
2018.

[1] https://wiki.haskell.org/Yampa
[2] http://www.reactive-streams.org

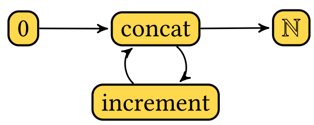


Figure 1. Natural numbers

2 Background

In this section we provide some background context needed. In particular, we briefly present the dataflow model and the notation of graphs we use in the remaining paper. We continue with a brief overview of the well-established middlewares used in robotics and IoT, namely the Robot Operating System (ROS) and the Message Queuing Telemetry Trasport (MQTT) protocol. Moreover, for the rest of the paper we assume familiarity with the functional reactive programming (FRP).

2.1 The Dataflow Computational Model

In the dataflow computational model, the program is represented as a dataflow graph, where nodes are independent computational units and edges are communication channels between these units. A node is fired immediately when its required inputs are available and therefore no explicit control commands are needed for execution. An immediate consequence is that the nodes of the graph can run independently and potentially in parallel as soon as their inputs are present.

Figure 1 shows a dataflow graph enumerating the set $\mathbb{N}$ of natural numbers. In the dataflow graph above, we can discern three types of nodes: sources, which do not have any incoming edge and act as value generators to initiate computation, sinks, which do not have any outgoing edges and inner nodes, which transform one or more incoming streams and redirect their output to other nodes. The *zero* node just produces a stream with a single value 0 and then terminates. *Concat* produces a single stream by concatenating the stream produced by *zero* and *increment*, while *increment* transforms its input stream by adding one to its values. Finally, the sink node displays the result, which is the stream of natural numbers.

Streams can be infinite, such as the stream produced by *concat* because it is the concatenation of a single-value stream and an infinite one. Moreover, the graph is cyclic as *concat* feeds input to *increment* and vice versa. The most interesting fact is that there nodes are independent and therefore can run in parallel. For instance, while *increment* is processing value 5 (i.e. to produce value 6), the previous result (i.e. value 5) passes through *concat* to reach the sink node, which can concurrently process it to display it.

Dataflow graphs can be executed both in a single-machine and in cluster of machines where each node can be placed in a different machine. A possible single-machine implementation could represent edges as in-memory queues, whereas a multi-machine one could realize them as channels between TCP sockets, allowing communication across the network.

2.2 Robotics and IoT Middlewares

In the following we briefly present the ROS and MQTT, the de-facto middlewares used in robotics and IoT. The architecture of both follow a topic-based publish-subscribe (PubSub) pattern to loosely couple different processes and at the same time maximize flexibility.

2.2.1 ROS

ROS is an open-source middleware for robot software, which emphasizes large-scale integrative robotics research [16]. It provides a *thin* communication layer between heterogeneous computers, from robots to mainframes and it has been widely adopted by the research community around the world, due to its flexibility and maximal support of reusability through packaging and composability. It provides a compact solution to the development complexity introduced by complex robot applications that consist of several modules and require different device drivers for each individual robot.

It follows a peer-to-peer network topology, implemented using a topic-based *PubSub* messaging protocol and its architecture reflects many sound design principles. Another great property of *ROS* is that it is language-agnostic, meaning that only a minimal specification language for message transaction has been defined, so contributors are free to implement small-size clients in different programming languages, with *roscpp* for C++ and *rospy* for Python being the most widely used ones.

A typical development scenario is to write several *nodes*, that subscribe to some topics and, after doing some computation, publish their results on other topics. The main architectural issue here is that subscribing is realized through asynchronous callback functions, so complicated schemes easily lead to unstructured code, which obviously lead to unreadable and hard-to-maintain code. Our approach gives a solution to the aforementioned problem.

2.2.2 MQTT

Internet of Things (IoT) conveys the concept of a multitude of heterogeneous devices, ranging from low-cost sensors to vehicles with embedded electronics, are connected and provide the ability to collect data and exchange it amongst themselves. The development of such systems though, due to their heterogeneity, is rather complex and costly. Recent development of a variety of middleware frameworks, showed that a standard protocol of communication is imperative along with supporting tools [14].

The most widely spread protocol is *MQTT*, which follows the *PubSub* messaging pattern and provides a very minimal and lightweight communication layer in order not to put a strain on the resource-bounded system [11]. For instance, an *IoT* application could connect to some sensors by subscribing to their corresponding topics, taking decisions that would result in some commands to some actuators, by publishing to their corresponding topics.

Fortunately, the dataflow model seems to be rather fitting for these scenarios [4], as every node in the graph is completely independent, and consequently can be any "*thing*". This useful property of the model makes it a good architectural choice for such applications. The only thing to consider is how these things will communicate in a standard way, so as to be able to add new types of *things* and integrate it in an effortless way to an existing dataflow network.

3 The RHEA Framework

3.1 Requirements and Objectives

The system should be *reactive*, as close as possible to the definition of the Reactive Manifesto[3]. In particular, the system should be

responsive meaning it should be able to handle successfully time-sensitive scenarios if at all possible. This is the cornerstone of usability and utility.

resilient meaning it is able to recover robustly and gracefully after a failure, due to the fact that nodes in the dataflow graph are completely independent and recovery of each one can be done in isolation.

elastic meaning it will adjust itself depending on the available resources and demanded workload. For instance, the granularity of the graph (i.e. number of nodes) is adjusted so as to match a heuristic-based value (e.g. total number of threads).

message-driven meaning it relies solely on asynchronous message-passing for inter-component communication leading to loose coupling, isolation, location transparency and the error propagation.

One of the major concerns while designing the framework was the ability to deploy it anywhere, from low-cost robots to mainframes. Apparently such attribute would require a very flexible runtime environment that have the ability to handle *heterogeneous* devices.

As the new technologies and frameworks arise the system should be able to adapt and be extended in order to remain useful and general-purpose. Therefore, careful consideration was taken to compose the system of different independent modules, which could *extensible* and easily modified. With that concept in mind, generality and abstraction were heavily emphasized during both the design and the implementation process.

The framework should be also *abstract* in terms of implementation details, as it is completely agnostic of any machine-specific requirements. It is designed as a unifying conceptual base for further extensions and careful consideration was taken not to restrict in any aspect.

[3]http://www.reactivemanifesto.org

3.2 Architecture

The RHEA framework consists of several clearly separated modules, whose interconnection is illustrated in Figure 2.

A typical workflow of the system has as follows. The user writes a program in the provided domain-specific language, which constructs an internal representation of the dataflow graph. Afterwards, using information about the available resources in the network, the constructed graph is optimized and partitions of the graph are assigned to physical computational resources. The optimized graph partitions are then distributed across the available machines for execution, maybe using a different evaluation strategy each time.

More specifically, a flexible evaluation strategy was employed where different executors can be considered when available. This design satisfies the requirements of extensibility, heterogeneity and abstraction of the system since each partial graph can be evaluated by a different *EvaluationStrategy* which could interpret it using a specific streams library or even compile into CUDA code for execution on a GPU. In Section 4 we provide three different evaluation strategies.

3.3 Dataflow Graphs

The kind of dataflow graph that can be expressed using the framework's stream language are directed cyclic graphs with possibly many inputs and outputs. The data channels (i.e. edges of the dataflow graph) are represented using the `Stream` data type, which is parametric, meaning that it can emit values of any data type, whether built-in or user-defined. The stream produced may terminate, successfully or erroneously, or even be infinite.

The construction of the internal dataflow graph is implicit, through a rich set of operators on the Stream data type. Each `Stream` object contains internally a dataflow graph of type `FlowGraph`, which is only to be accessed and manipulated by the internal module, evaluation strategies and optimizers. Therefore, an application developer only needs to work with the `Stream` type.

Source nodes are constructed using built-in functions of type `Stream`. For instance, `Stream.just(1,2,3)` produces the stream that emits just the values 1, 2 and 3. The return variable of these creation function is an object of type `Stream`.

Processing nodes can be divided into two classes: *single input* ones and *multiple input* ones. Single input nodes are inserted into an existing `Stream` object, by calling an operator on that object. Multiple input nodes are constructed by built-in function that take as argument already existing `Stream` objects.

Figure 3 shows an example of a single input node, namely that of `map`, which transforms the input stream (i.e. just the values 1, 2 and 3) by applying a user-defined function to every emitted value. Figure 4 shows an example of a multiple input node, namely that of `zip`, which transforms the input

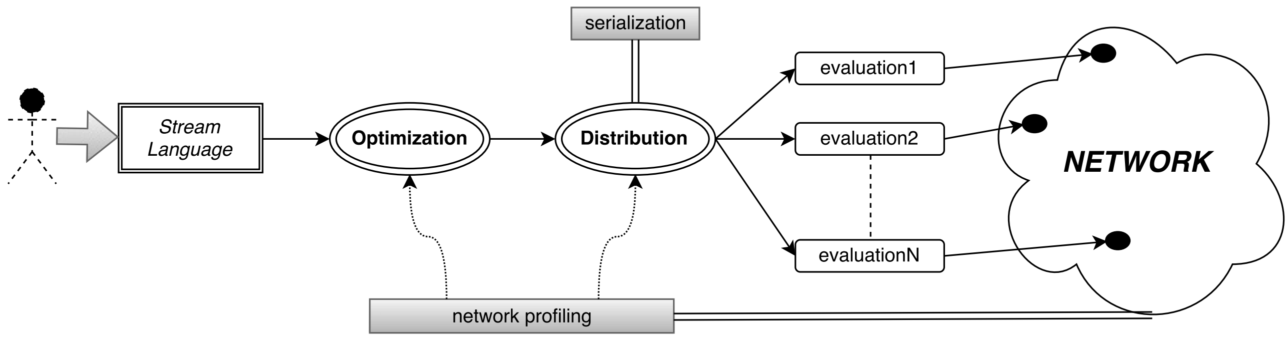
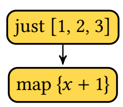
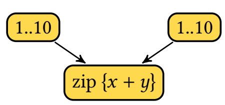


Figure 2. System architecture



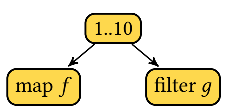

```
Stream.just(1, 2, 3)
        .map(x -> x + 1);
```

Figure 3. Single input processing node




```
Stream.zip(
    Stream.range(1, 10),
    Stream.range(1, 10),
    (x, y) -> x + y);
```

Figure 4. Multiple input processing node




```
Stream<Int> st = Stream.range(1, 10);
st.map(f).print();
st.filter(g).print();
```

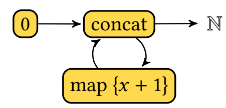
Figure 5. Split example

streams by applying a user-defined function to each emitted pairs of values. Here we also see the stream creation function `Stream.range`.

The variables returned by all processing nodes are `Stream` objects. These objects can be reused in different parts of the graph to enable splitting a node's output to different processing nodes or outputs. Figure 5 shows such an example, where the `filter` operator only emits values for which the given function returns true.

Cycles are constructed using the `loop` operator, which is a single input processing node. It requires a function that, given an input stream, constructs a subgraph that redirects its output to that input, therefore creating a feedback loop. Figure 6 shows an example of the `loop` operator to represent the natural numbers, just as the graph shown earlier in Figure 1. The `concat` operator is a multiple input node that concatenates its input streams.

Lastly, to evaluate a given dataflow graph and do something with its output values, we need to call the `subscribe`




```
Stream
    .just(0)
    .loop(s -> s.map(i -> i + 1));
```

Figure 6. Cyclic example

method of the `Stream` object and pass as argument a user-defined action (i.e. function with side-effects).

4 Implementation

Since extensibility is a major design priority, most individual critical components are defined using the Strategy design pattern, isolating the desired functionality in a separate interface and allowing the system to select the appropriate instantiating classes at runtime. The main pluggable components of the system, for which default implementations are already provided by the current implementation as separate libraries, are *Evaluation*, *Optimization*, *Distribution*, *Serialization* and *Network Profiling*.

Every value passed through the framework's streams is wrapped inside a `Notification` object, which discriminates stream values into three categories: `onNext` (when the stream provides a regular value), `onError` (when an error occurs) and `onComplete` (when the stream completes its output). This enables the system to handle gracefully error propagation.

In order to make the framework easy to integrate with other stream and dataflow technologies, every input and output node implements the interfaces that RSS defines, namely the Publisher and the Subscriber interface. This also enables users to define new types of sources or sinks, in order to integrate the framework with other general technologies (e.g. system events, HTTP requests, PubSub implementations, etc). In particular, a sink node (output) should implement the Subscriber interface, which essentially defines three methods corresponding to reactions to a `Notification`, one for each of the categories mentioned above. Moreover, a source node (input) should implement the Publisher interface, which defines a single method `subscribe(Subscriber)`, where a Subscriber requests the Publisher to start emitting values.

Many existing technologies provide these interfaces, or at least adapters from their internal representations, and therefore they are very easy to be integrated to the framework.

4.1 Execution

Every primitive operator corresponds to an expression implementing the `Transformer` interface and a complete dataflow is defined by a `Stream` variable and an object implementing the `Output` interface, which can be either an `Action`, a `Sink` or a list of these.

Roughly speaking, the `EvaluationStrategy` interface accepts the `Stream` variable and its corresponding `Output` and executes it, however desired. The strategies we have implemented so far follow:

RxJavaEvaluationStrategy which uses RxJava [4], an established and well-maintained library for asynchronous programming using the *Observable* type, which is very close, semantically, to our *Stream* type.

RosEvaluationStrategy which integrates the *ROS* middleware into the framework. This strategy's main objective is to set up a *ROS* client and configure every `RosTopic` used within the dataflow that needs to be evaluated to use this client. After that, evaluation is propagated to a generic strategy, for example, to RxJavaEvaluationStrategy.

MqttEvaluationStrategy which integrates the MQTT middleware into the framework, in the same way *ROS* is integrated.

4.2 Distributed Execution

An evaluation strategy executes the requested dataflow graph in a single machine, without concern about distribution and resource utilization. For distribution and cluster management, one needs to implement the `DistributionStrategy` interface by adjusting the granularity (i.e. size) of the graph to evaluate to fit the available resources (see, Section 5) and partition it across all computational resources, maybe using different evaluation strategies.

The default `DistributionStrategy` uses the *Hazelcast*[5] library to discover and manage multiple machines and used its internal decentralized PubSub model to communicate intermediate results across the network. Figure 7 illustrates the partitioning of a dataflow graph over several machines, where each machine – except the last one – outputs its result to a Hazelcast topic, from which another machine gets its input.

According to the distribution strategy being used, the available machines will require a certain initial configuration. For the Hazelcast case, a little piece of setup code needs to be executed on every member of the cluster, which is together with the main `EvaluationStrategy` class. Moreover, helpful information can also be added at this step, such as number of CPU cores. It is the distribution strategy's responsibility to ensure that this information is properly distributed and handled.

Apart from this initial configuration, the distribution strategy needs to enable members to declare certain capabilities that they possess, which are required by specialized nodes. For instance, a source node emitting values from a ROS topic must be executed on a machine having ROS installed, in order to set up a ROS client. The default implementation uses strings to represent capabilities and are declared in the initialization code of each machine separately.

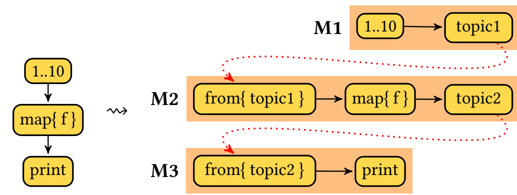


Figure 7. Partitioning

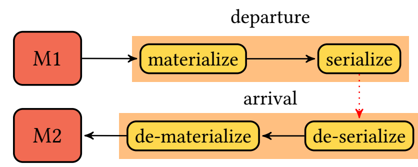


Figure 8. Serialization process

4.2.1 Serialization

As communication between machines across a network is mandatory, data types emitted through the streams must be serialized on departure and de-serialized on arrival at each machine. For this reason, each `DistributionStrategy` must be configured with a class implementing the `Serializer` interface, but we also provide a default one that covers most datatypes. Figure 8 depicts the serialization process in more detail.

5 Optimizations

This section describes three stages of optimization that the dataflow graph goes through before being evaluated:

- proactive filtering that places filtering operations as soon as possible;
- granularity adjustment that combines adjucent operation into a single more efficient operation;
- and node placement that places nodes that should be colocated to the same machine.

The optimization phases run sequentially, that is the output of one phase becomes input of the next. The main purpose

[4] http://github.com/ReactiveX/RxJava
[5] http://hazelcast.org/

Figure 9. Take/skip/distinct before map

Figure 10. Filter before map

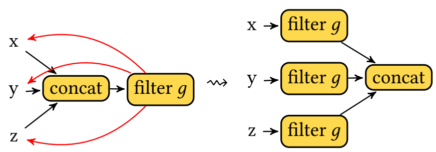


Figure 11. Filter/distinct before concat/merge

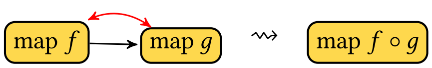


Figure 12. Merge maps

Figure 13. Combine map with filter

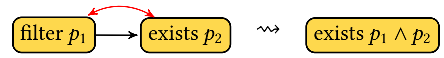


Figure 14. Combine filter with exists

Figure 15. Combine map with exists

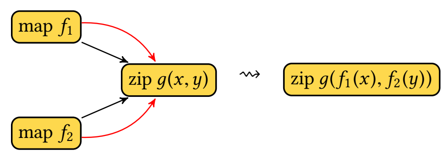


Figure 16. Combine map with zip

of the optimization graph is to achieve better performance and better utilization of the available resources.

5.1 Proactive filtering

The first optimization stage is a heuristic one, based on the fact that if a filter operation can be moved earlier (i.e. closer to source nodes) while preserving the original semantics, then there will be benefit concerning computational cost and cross-machine communication overhead. The figures below illustrate one representative example of each general class of graph transformation.

5.2 Granularity adjustment

Different nodes of the dataflow graph will be executed on a separate thread/process. The fact that graphs can grow very big, for instance when programming a swarm of robots, poses a problem when available computational resources are limited. For this reason, the second optimization stage tries to adjust the granularity of the dataflow graph to a desired value, which is normally the number of available threads amongst all machines. To reach the desired granularity, the optimizer applies some semantic-preserving transformation, as shown in the figures below.

In Figure 12 we merge two `map` operations into one `map` operation that uses the composition of the two initial functions, while in Figure 13 a `map` followed by a `filter` is substituted by a more complex equivalent operation, namely `filterMap`. In Figures 14 and 15 we apply some simple properties of the boolean functions involved to decrease the number of nodes. Lastly, in Figures 16 and 17 we utilize function composition to embed `map` operations into `zip` operations.

5.3 Node placement

After the first two passes, we have an optimized dataflow graph with fine-tuned granularity. At this stage, nodes are mapped to tasks and are deployed across the available machines, keeping resource utilization in mind. If the desired granularity has not been reached yet, the `DistributionStrategy` applies fusion to pairs of tasks until it reaches it, as shown in Figure 18.

The final decision to be made is where each of these newly constructed tasks will be executed, although some of them need to necessarily be placed on specific machines with certain skills.

Apart from these hard constraints, we need to minimize communication overhead. For this purpose, one must implement the `NetworkProfileStrategy` by providing a way to calculate network distance between available machines, which is then fed as input to the `NodePlacement` optimizer.

6 Applications

6.1 Hamming numbers

Consider the problem of enumerating the *Hamming numbers*, which are generated by the mathematical formula $\mathbb{H} = 2^i 3^j 5^k$, where $i, j, k \in \mathbb{N}$. There is an intuitive dataflow solution to the above problem, borrowed from the book of Lucid [17].

Figure 19 shows the dataflow graph with its corresponding RHEA code. The code is written in Scala and utilizes the "Pimp my library" design pattern [13]. In the example above, we define two new Stream operators, namely

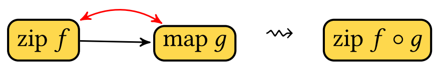


Figure 17. Combine zip with map

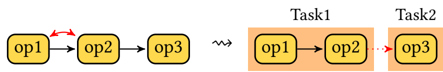
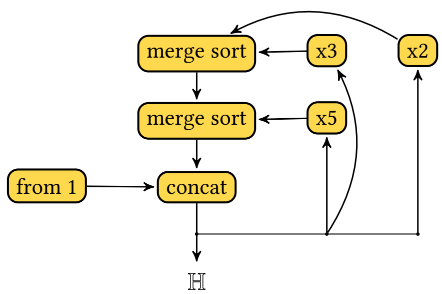


Figure 18. Task fusion




```
Stream.just(1)
  .loop(entry => (entry.multiply(2) mergeSort entry.multiply(3))
                   mergeSort entry.multiply(5))
  .distinct.print

class IntStream(stream: Stream[Int]) {
  def multiply(constant: Int): Stream[Int] = stream.map(i => i * constant)
  def mergeSort(other: Stream[Int]): Stream[Int] = ...
}
implicit def enrichStream(st: Stream[Int]): IntStream = new IntStream(st)
```

Figure 19. Hamming numbers

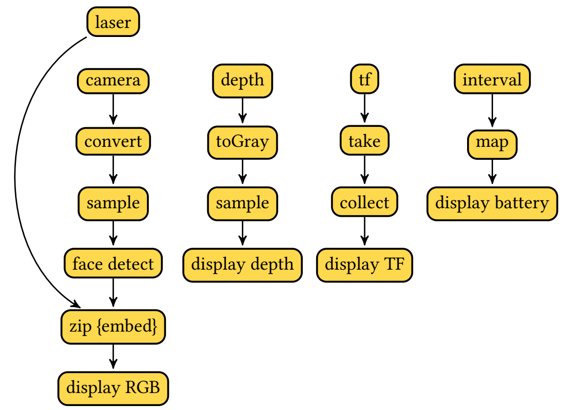
`multiply` (line 10), which just multiplies the stream with a constant, and `mergeSort` (line 13), which produces an ordered stream given two ordered streams as input. We also see the use of the `loop` operator (line 2), which allows us to define cycles.

6.2 Robot control panel

This application concerns real-time monitoring of a robot, that is publishing its information and sensor-data to ROS topics, through a graphical user interface (GUI).

The `/camera/rgb` topic provides the frames of the robot's camera as coloured images, while the `/camera/depth` provides frames that provide depth information. The `/tf` topic publishes parent-child relations of the internal topics of the robot's configuration, and finally the `/scan/` topic provides information from the robot's laser that gives horizontal depth information in polar coordinates.

The GUI displays the laser data embedded on the camera stream, while allowing for real-time face detection. Additionally, it displays the depth frames and the `tf` relations as a tree. Finally, a mock-up battery bar is displayed to show-case the framework's ability for simulation. Figure 20 illustrates the




```
Stream<LaserScan> laser = Stream.from(new RosTopic<>("/scan"));
Stream<Mat> image =
    Stream.<Image>from(new RosTopic<>("/camera/rgb"))
              .map(CvImage::toCvCopy)
              .sample(100, TimeUnit.MILLISECONDS)
              .map(this::faceDetect);
Stream.zip(laser, image, this::embedLaser)
          .subscribe(viz::displayRGB);
Stream.from(new RosTopic<>("/tf"))
          .take(50)
          .collect(HashMap::new, (m, msg) -> ...)
          .subscribe(viz::displayTF);
Stream.<Image>from(new RosTopic<>("/camera/depth"))
          .map(this::toGray)
          .sample(100, TimeUnit.MILLISECONDS)
          .subscribe(viz::displayDepth);
Stream.interval(2, TimeUnit.SECONDS)
          .map(v -> (100 - v) / 100.0)
          .subscribe(viz::displayBattery);
```

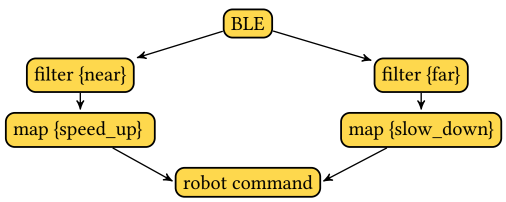
Figure 20. Robot control panel

dataflow solution to the above problem and its corresponding RHEA code.

The implementation details (i.e. the visualization class and methods `faceDetect` (line 7), `embedLaser` (line 8) and `toGray` (line 26)) are not shown for brevity's sake. It is worth noting that this model of programming encourages a clean separation of concerns between the individual components, namely between the sensor data manipulation and the actual visualization on the GUI.

6.3 Robot hospital guide

As a final example, we will examine a combined application that involve both robotics and IoT. Consider a robot that guides patients to different parts of a hospital, such as the gym or cafeteria. We assume also that the map localization, path finding and obstacle avoidance are already implemented and provided by default by ROS. The problem is to calibrate the robot's speed according to the patient's status. In order to keep track of the patient's distance from the robot, each patient carries a smartphone that acts as a bluetooth low-energy (BLE) beacon. The robot uses its bluetooth receiver to publish the distance from the signal source to an MQTT topic, which is then transformed by our stream application




```
Stream.configure(new HazelcastDistributionStrategy(
        RxjavaEvaluationStrategy::new,
        RosEvaluationStrategy::new,
        MqttEvaluationStrategy::new));

Topic<RobotCommand> vel = new RosTopic<>("/robot/cmd");

Stream<Proximity> ble = Stream.from(new MqttTopic<>("/ble"));

ble.filter(Proximity::isNear)
   .map(d -> Commands.SPEED_UP)
   .subscribe(vel);

ble.filter(Proximity::isFar)
   .map(d -> Commands.SLOW_DOWN)
   .subscribe(vel);
```

Figure 21. Robot hospital guide

to velocity commands for the robot, in the form of slowing down or speeding up.

The first module constitutes the main program logic, where a declared dataflow graph acts as a stream transformation from beacon information to velocity commands to the robot. Figure 21 shows the dataflow graph with its corresponding RHEA code.

The second module just uses the library *ReactiveBeacons*[6] to get a stream of beacon data via RxJava, and then publishes it to a MQTT topic, which is the input of the first module. The corresponding RHEA code follows:

```
Stream.configure(new HazelcastDistributionStrategy(
        RxjavaEvaluationStrategy::new,
        MqttEvaluationStrategy::new));

Stream.from(ReactiveBeacons.observe())
      .map(Beacon::getProximity)
      .subscribe(new MqttTopic<>("/ble"));
```

The aforementioned example showcases the framework's ability to combine different technologies and act as a high-level, declarative unified layer.

7 Related Work

7.1 Dataflow systems

The necessity for implicit parallelism and distribution of more and more applications, dealing with huge and/or complex data, has brought increasingly more attention to the dataflow programming model. Nowadays there are various implementation of dataflow systems focused on different applications such as Big Data batch and stream processing, Machine Learning and others. In the following we discuss the prominent dataflow systems and the similarities and differences with the proposed framework RHEA.

The two most prominent dataflow framework for scalable large-data processing is Apache Spark [18] and Apache Flink [3]. Both support a rich set of data-parallel operators and can be used for either batch or stream processing. They follow the same general approach as RHEA by implicitly creating a dataflow pipeline. It is also worth noting that Apache Flink optimizes the dataflow graph by applying semantically-equivalent graph rewriting [10]. Apart from these similarities there are also key differences. They focus on providing a full execution stack and therefore they do not provide the flexibility in using existing underlying system to perform execution. They accomplish distributed execution by partitioning on the data in contrast to RHEA that partitions the graph on the operations.

Similar to the flexibility that RHEA aims to provide is the project Apache Beam [2], formerly known as Cloud Dataflow and an evolution of FlumeJava [5]. Apache Beam shares the same key idea with RHEA namely they provide a unified abstract programming layer for batch and stream processing that can use multiple executors. They provide, for instance, an Apache Spark and an Apache Flink executor. However, the assume that the cluster is homogeneous and thus the whole program is executed using a single executor. Another framework that shares the same similarities with Apache Beam and RHEA is dispel4py [8], a Python framework that focuses on scientific workflows. It provides the ability to describe abstract workflows for distributed data-intensive applications. Similar to RHEA evaluation strategy concept, it allows different mappings to enactment systems, such as MPI and Apache Storm. However, it does not tackles heterogeneity issues and simultaneously executing graph partitions to different enactment systems.

Another dataflow framework from Google is TensorFlow [1], which is an open-source polyglot library for machine intelligence and especially construction of neural networks. Tensorflow differs from the aforementioned dataflow systems since it assumes that data are Tensors, namely multidimensional arrays, the nodes operate in Tensors and the program is a dataflow network of such operations. Tensorflow is similar to RHEA since it is inherently heterogeneous by considering different hardware devices such as CPU and GPUs. Moreover, it provides distributed execution by partitioning the dataflow graph similarly to what RHEA is proposing. The main difference compared to RHEA approach is that the system is not reactive, therefore it is not optimized for asynchronous stream of events.

[6] http://github.com/1083pwittchen/ReactiveBeacons

Regarding the reactive-aware systems, RxJava and AkkaStreams are the most mature frameworks that support programming languages that target the JVM. It is worth mentioning *Akka* [7] which is the foundation library where AkkaStreams relies on. In particular, Akka is a toolkit and runtime for highly concurrent, distributed and resilient message-driven applications and its execution model follows the Actor model. In this model one perceives abstract computational agents, called actors, that are distributed in space and communicate with point-to-point messages. In reaction to a message, an actor can create more actors, make local decisions, send more messages and determine how to respond to the next message received. Similar to the problem of *ROS* that our framework solved, using Actors can be daunting. AkkaStreams try to provide a higher-level abstraction from the Actor model and specifically it provides a convenient API for stream processing and also dataflow graph construction. The difference with RHEA is that AkkaStreams solely relies on Akka. On the other hand, RHEA offers the ability to choose between several evaluation strategies to match the application's needs.

7.2 Robotics and IoT

It is only natural that the dataflow model would make its way through the field of robotics, as many behaviours in control theory are expressed as dataflow diagrams.

Roshask [6] is a binding from the Haskell programming language to the basic *ROS* interfaces. Like RHEA, the approach is to overcome the shortcomings of *ROS* callbacks by viewing topics as streams. This allows for, and encourages, a higher level of abstraction in robot programming, while making the fusing, transforming and filtering of streams fully generic and compositional. RHEA and *roshask* were heavily influenced by the work of Hudak's group (Yale Haskell Group) on robot DSLs and FRP in general [7, 9, 15]. *Yampa* [9] is a DSL embedded in Haskell that realizes the FRP model, using arrows to minimize time and space leaks.

IoT applications often deal with much heterogeneity, due to the variety of sources that different devices introduce. Therefore, a component-based approach suits well to solve this problem and there are some dataflow frameworks that follow that approach. Another interesting *IoT* framework that follows a dataflow approach is *Node-RED* [4], which is a visual tool for wiring together hardware devices, APIs and online services in new and interesting ways. Applications called flows, are built immediately on a browser, and can be deployed on the Cloud with just a single click. The main advantage of this tool is that it encourages social development, due to the fact that flows are stored in JSON format, which can be easily imported and exported for sharing with others.

[7] http://akka.io

8 Conclusions and Future Work

The framework described in this paper offers a unified and extensible way for reactive applications to be developed. Primarily motivated by the well-established middlewares in robotics and IoT, the main focus of the framework is extensibility, heterogeneity. To that end, a constant effort to generalize and make components as abstract as possible was made.

The applications demonstrated the framework's ability to provide a higher level of abstraction, where the language only specifies how different components coordinate, without knowledge of the implementation details. The driving force for both frameworks is that some specific domains have fixated their methods on low-level programming, whereas more satisfactory paradigms can solve many shortcomings.

The set of operators aided expressibility, making it possible to specify any dataflow graph in a concise and readable manner. This disallowed optimizations suitable for less expressive models, but recent research suggest that general dataflow topologies have optimization opportunities that are yet to be found [10]. In the current reincarnation of the framework only a minimal optimization stage has been implemented, which nevertheless paves the path to more advanced optimization techniques, such as those used in Apache Flink [10].

Apart from a more sophisticated optimization phase that can be investigated as a future direction the are also other extensions that are equally interesting and challenging. Again motivated by the robotics domain, an interesting extension is to apply dynamic reconfiguration where the applications operate in environments that are constantly changing. For instance operating in an environment where battery powered robots participate or the connectivity of the cluster is unstable. Adaptive techniques for reconfiguring the dataflow graph distribution should be devised in such situations. Morever, these environments give rise to fault-tolerant execution and it is certainly challenging to propose methods for graceful recovery.

Lastly, another interesting future direction is alternative techniques regarding node placement. It is worth investinating techniques that use reinforcement learning to decide a reasonably efficient node placement. Similar techniques have been recently proposed in [12].

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 265–283.

[2] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael 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. *PVLDB* 8, 12 (2015), 1792–1803.

[3] 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 Rheinlander, Matthias J. Sax, Sebastian Schelter, Mareike Hoger, Kostas Tzoumas, and Daniel Warneke. 2014. The Stratosphere platform for big data analytics. *VLDB J.* 23 (2014), 939–964.

[4] Michael Blackstock and Rodger Lea. 2014. Toward a Distributed Data Flow Platform for the Web of Things. In *5th International Workshop on the Web of Things (WoT)*.

[5] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R Henry, Robert Bradshaw, and Nathan Weizenbaum. 2010. FlumeJava: easy, efficient data-parallel pipelines. In *ACM Sigplan Notices*, Vol. 45. ACM, 363–375.

[6] Anthony Cowley and Camillo J Taylor. 2011. Stream-oriented robotics programming: The design of roshask. In *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on.* IEEE, 1048–1054.

[7] Conal Elliott and Paul Hudak. 1997. Functional reactive animation. In *ACM SIGPLAN Notices*, Vol. 32. ACM, 263–273.

[8] Rosa Filguiera, Amrey Krause, Malcolm P. Atkinson, Iraklis A. Klampanos, and Alexander Moreno. 2017. dispel4py. *IJHPCA* 31, 4 (2017), 316–334.

[9] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. 2003. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming*. Springer, 159–187.

[10] Fabian Hueske, Mathias Peters, Matthias J Sax, Astrid Rheinländer, Rico Bergmann, Aljoscha Krettek, and Kostas Tzoumas. 2012. Opening the black boxes in data flow optimization. *Proceedings of the VLDB Endowment* 5, 11 (2012), 1256–1267.

[11] Dave Locke. 2010. Mq telemetry transport (mqtt) v3. 1 protocol specification. *IBM developerWorks Technical Library], available at http://www. ibm. com/developerworks/webservices/library/ws-mqtt/index. html* (2010).

[12] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. 2017. Device Placement Optimization with Reinforcement Learning. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017 (Proceedings of Machine Learning Research)*, Doina Precup and Yee Whye Teh (Eds.), Vol. 70. PMLR, 2430–2439.

[13] Martin Odersky. 2006. Pimp my library. *Artima Developer Blog, October 9* (2006).

[14] Koosha Paridel, Engineer Bainomugisha, Yves Vanrompay, Yolande Berbers, and Wolfgang De Meuter. 2010. Middleware for the internet of things, design goals and challenges. *Electronic Communications of the EASST* 28 (2010).

[15] John Peterson, Paul Hudak, and Conal Elliott. 1999. Lambda in motion: Controlling robots with Haskell. In *Practical Aspects of Declarative Languages*. Springer, 91–105.

[16] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. 2009. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, Vol. 3. 5.

[17] Edward A. Ashcroft William W. Wadge. 1985. *Lucid, the Dataflow Programming Language.* Academic Press.

[18] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, Steven D. Gribble and Dina Katabi (Eds.). USENIX Association, 15–28.