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) [8]. 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 [11] 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). 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 discusses the main requirements and objectives that drove the design of the framework. Section 4 presents the framework's characteristics and capabilities. Section 5 discusses implementation details and Section 6 presents the techniques used for distributed execution. Section 7 presents several optimizations that have been implemented in the framework. Section 8 demonstrates some use-cases of the framework mainly motivated by robotics and IoT. Section 9 discusses related work and finally Section 10 concludes with future directions.

PL'18, January 01–03, 2018, New York, NY, USA
2018.

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

2 Background

2.1 The dataflow computational model

In the dataflow 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

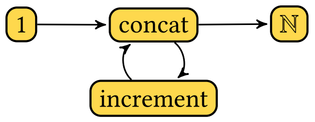


Figure 1. Natural numbers

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.

A possible single-machine implementation could represent edges as in-memory data storage, whereas a multi-machine one could represent them as channels between TCP sockets, allowing communication across the network. Its great flexibility and composability makes it a good candidate for the underlying architecture of a framework with a high level of abstraction.

2.2 Robotics and IoT Middlewares

ROS *ROS* is an open-source middleware for robot software, which emphasizes large-scale integrative robotics research [18]. 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* and *rospy* 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.

Internet of things - MQTT The birth of the Internet gave rise to a concept called *Internet of Things (IoT)*, which is essentially the ability of many heterogeneous devices, ranging from low-cost sensors to vehicles with embedded electronics, to collect data and exchange it amongst themselves using the Internet. This gave rise to smart grids, smart homes and eventually smart cities.

The development of such systems though, due to their heterogeneity, is rather complex and costly. Typical software architectures were not meant to be used in such environments and therefore new tools and concepts needed to be invented. Recent development of a variety of middleware frameworks, showed that a standard protocol of communication is imperative along with supporting tools [16]. The most widely spread protocol is *MQTT*, which follows the *PubSub* messaging pattern and provides a very minimal communication layer in order not to put a strain on the resource-bounded system [13].

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 Requirements

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

The system should be *responsive*, meaning it should be able to handle time-sensitive scenarios if at all possible. This is the cornerstone of usability and utility, but more than that, it enables quick error-detection and error-handling.

The system should be *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. Another thing to note here is that special error messages are built-in and make it very easy to propagate errors between *components*, in case the error- handling part of a component is decoupled from the computational logic. This leads to much more robust architectures for large-scale systems, where fault- tolerance is mission-critical.

The system should be *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).

The system should be *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 mentioned above. Location transparency is critical to preserve the semantics whether on a single host or a machine cluster.

Heterogeneous One of the major concerns while designing the framework was the ability to deploy it anywhere, from low-cost robots to mainframes. Obviously, such attribute would require a very flexible runtime environment. To satisfy this requirement, the strategy design pattern was used for evaluation, meaning that the core system only builds the internal representation of the dataflow graph and partitions it across the available computational resources. From there onwards, each partial graph can be evaluated by a different *EvaluationStrategy* (see, Section 5), which could interpret it using a specific streams library or even compile into CUDA code for execution on a GPU.

Extensible As the work described in this paper is quite fundamental and ambitious, it seemed highly unlikely that it would reach closure. Therefore, careful consideration was taken to compose the system of different independent modules, which could easily be extended/modified, allowing many future contributions.

With that concept in mind, generality and abstraction were heavily emphasized during both the design and the implementation process. We can say now we are satisfied with the level of abstraction the core system has reached and hope the stressful refactoring that the framework went through will blossom in the form of future contributions.

Abstract The framework is *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, architectural or not. This was achieved by making many parts of the core system pluggable, allowing for easy refactoring on most of its internal functionality. Moreover, the internal graph representation does not include information on how a node is executed, but only on its semantics.

4 The RHEA Framework

4.1 System Architecture

The RHEA ecosystem consists of several clearly separated modules, whose interconnection is illustrated in Figure 2. The user writes a program in the provided stream language,

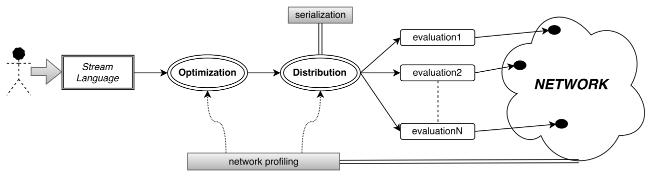


Figure 2. System architecture

which constructs a dataflow graph internally. Afterwards, using information about the available resources in the network, the constructed graph is optimized (i.e. in terms of performance, communication cost and node placement). The optimized graph is then distributed across the available machines for evaluation, maybe using a different technique each time.

4.2 Constructing 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.

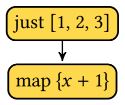
The construction of the internal dataflow graph is always implicit, through a rich set of operators on the Stream data type. Each *Stream* object contains internally a dataflow graph of type *FlowGraph*, which is only to be accessed and manipulated by the internal module, evaluation strategies and optimizers. 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

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

stream that emits just the values 1, 2 and 3. The return variable of these creation function is an object of type *Stream*.

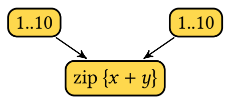
Processing nodes can be divided into two classes: *single input* ones and *multiple input* ones. Single input nodes are inserted into an existing Stream object, by calling an operator on that object. Figure 3 shows an example of a single input node, namely that of *map*, which transforms the input stream (i.e. just the values 1, 2 and 3) by applying a user-defined function to every emitted value. Multiple input nodes are



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

Figure 3. Single input processing node

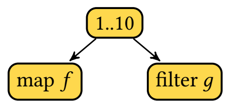
constructed by built-in function that take as argument already existing Stream objects. Figure 4 shows an example of a multiple input node, namely that of *zip*, which transforms the input streams by applying a user-defined function to each emitted pairs of values. Here we also see the stream creation function *Stream.range*.



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

Figure 4. Multiple input processing node

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.

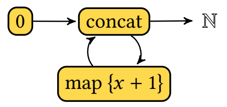


```
Stream<Int> st = Stream.range(1, 10);
st.map(f).print();
st.filter(g).print();
```

Figure 5. Split example

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.

To evaluate a given dataflow graph and do something with its output values, we need to call the *subscribe* method of



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

Figure 6. Cyclic example

the *Stream* object. The argument passed to *subscribe* is either a user-defined action (i.e. function with side-effects) or an object implementing the *Subscriber* interface.

5 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*.

Internal representation For representing the internal structure of the dataflow graph, the *JGrapht*[4] open-source Java library was used, which provides many graph data structures and common graph-theoretic algorithms.

Notifications 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).

External Input-Output In order to make the framework easy to integrate with other stream and/or dataflow technologies, every input/output node should implement the interfaces that *RSS* defines. 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).

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. 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]http://jgrapht.org/

5.1 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. An *EvaluationStrategy* takes the *Stream* variable and its corresponding *Output* and executes it, however desired. The strategies we have implemented so far follow:

RxJavaEvaluationStrategy Uses RxJava [5], an established and well-maintained library for asynchronous programming using the *Observable* type, which is very close, semantically, to our *Stream* type.

RosEvaluationStrategy Integrates the *ROS* middleware into the framework. This strategy's job 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 (e.g. RxJava).

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

6 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 *Optimization* section) and partition it across all computational resources, maybe using different evaluation strategies.

The default *DistributionStrategy* shipped with the framework uses the *Hazelcast*[6] library to discover and manage multiple machines and used its internal decentralized *Pub-Sub* model to communicate intermediate results across the network. Figure 7 illustrates the partitioning of a dataflow graph over several machines, where each machin – except the last one – outputs its result to a *Hazelcast* topic, from which another machine gets its input.

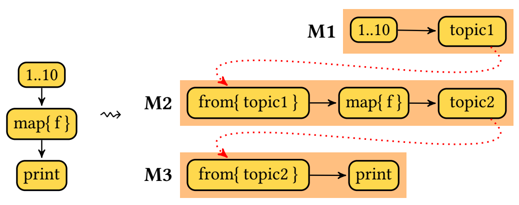


Figure 7. Partitioning

According to the distribution strategy being used, the available machines will require a certain initial configuration.

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

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 *Strategy* 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 skills 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. In the *Hazelcast* case, these skills are just *strings* and are declared in the initialization code of each machine separately.

6.1 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.

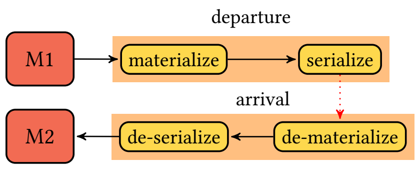


Figure 8. Serialization process

7 Optimizations

This section describes three stages of optimization the dataflow graph goes through before being evaluated, which are illustrated in Figure 9. The purpose of the optimization graph is to achieve better performance and utilization of the available resources.

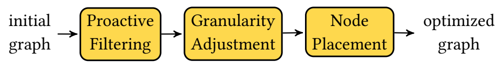


Figure 9. Optimization stages

7.1 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.

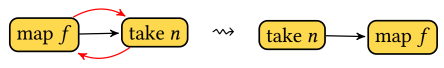


Figure 10. Take/skip/distinct before map

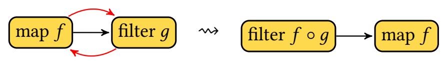


Figure 11. Filter before map

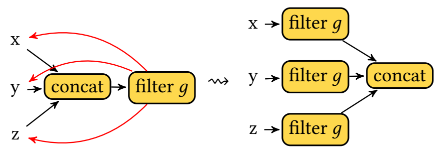


Figure 12. Filter/distinct before concat/merge

7.2 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.

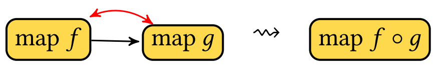


Figure 13. Merge maps

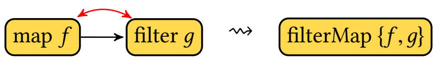


Figure 14. Combine map with filter

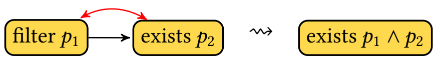


Figure 15. Combine filter with exists

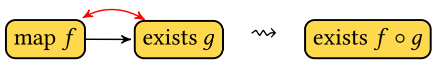


Figure 16. Combine map with exists

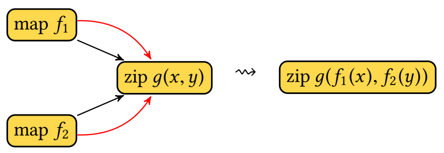


Figure 17. Combine map with zip

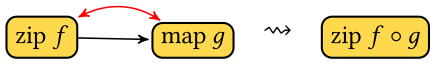


Figure 18. Combine zip with map

In Figure 13 we merge two *map* operations into one *map* operation that uses the composition of the two initial functions, while in Figure 14 a *map* followed by a *filter* is substituted by a more complex equivalent operation, namely *filterMap*. In figures 15 and 16 we apply some simple properties of the boolean functions involved to decrease the number of nodes. Lastly, in figures 17 and 18 we utilize function composition to embed *map* operations into *zip* operations.

7.3 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 19.

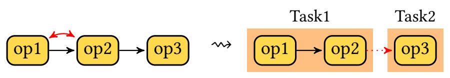


Figure 19. Task fusion

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.

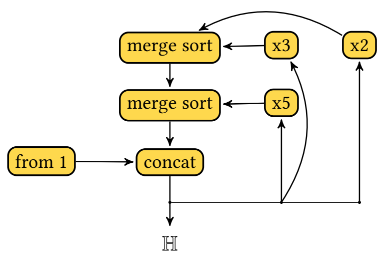
Apart from these hard constraints, we need to minimize communication overhead. For this purpose, one must implement the *NetworkProfileStrategy* by providing a way to calculate network distance between available machines, which is then fed as input to the *NodePlacement* optimizer.

8 Applications

8.1 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, taken from the book of *Lucid*, which is the first functional dataflow language [21]. Figure

20 shows the dataflow graph with its corresponding RHEA code.



```
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 20. Hamming numbers

The code is written in Scala to utilize the *Pimp my library* design pattern [15], which is used to easily add new functions to already existing libraries, using Scala's *implicit conversions* (line 28). In the example above, we define two new Stream operators, namely *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 power of the *loop* operator (line 2), which allows us to define cycles in an effortless manner.

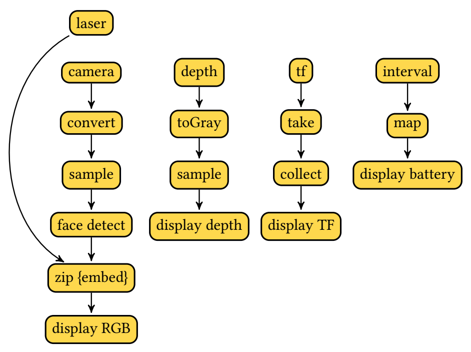
8.2 Robot control panel

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 21 illustrates the 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 evident that this



```
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 21. Robot control panel

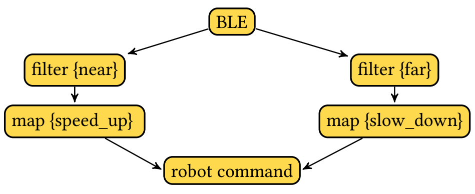
model of programming encourages a clean separation of concerns between the individual components, namely between the sensor data manipulation and the actual visualization on the GUI.

8.3 Robot hospital guide

As a final example, we will examine a more IoT-based application. Consider a robot that guides patients to different parts of a hospital, such as the gym or cafeteria. Assuming map localization, path finding and obstacle avoidance are already implemented, there still remains a problem with calibrating the robot's speed according to the patient's status.

To keep tract of the patient's distance from the robot, each patient carries a smart-phone 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 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 22 shows the dataflow graph with its corresponding RHEA code.



```
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 22. Robot hospital guide

The second module just uses the library *ReactiveBeacons*[7] 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"));
```

This example clearly show-cases the framework's ability to combine different technologies and act as a high-level, declarative coordination language.

9 Related Work

9.1 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. The main reason many frameworks have adopted it, is the high level of abstraction it provides with its declarative approach, making it simple to structure and maintain a complex system, while at the same time not losing its expressive power.

A very well-known and well-adapted framework for scalable large-data processing is Apache's *Spark* [22]. Although not a dataflow framework, it was developed to overcome the shortcomings of the *MapReduce*, similar to *FlumeJava*, by providing a much more efficient and flexible runtime, offering a rich set of data-parallel operators ($\simeq 80$) that can be used interactively from Scala, Python, Java or R.

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

It follows the same general approach as RHEA, in the sense that it is completely generic and encourages domain-specific libraries to be built upon it. For instance, *MLib* is a library for machine learning and *GraphX* is a library for iterative graph algorithms, both stacked upon *Spark*.

Definitely one the most mature frameworks for distribution targeting the JVM, *Akka* [8] is a toolkit and runtime for highly concurrent, distributed and resilient message-driven applications. Its approach follows the *Actor* model, where one perceives abstract computational agents, called actors, that are distributed in space and communicate with point-to-point messages that are buffered in a queue. 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, which is the inappropriate nature of callbacks for complex scenarios, *Akka* developers also felt the necessity for a more flexible and composable programming model, so they developed the *AkkaStreams* library, which provides a convenient API for stream processing and also dataflow graph construction with an interesting DSL.

The main reason RHEA offers a more flexible solution to *ROS* shortcomings than *Akka*, is that *Akka* is a pretty heavy-weight library, and consequently may prove over-abundant for simple use-cases. On the other hand, RHEA offers the ability to choose between several *EvaluationStrategies* to match your application's needs, therefore a simple application would just use a lightweight library like *rxjava*.

9.2 Unified dataflow languages

Continuing the search for more expressive models, Google recently released the *Cloud Dataflow* framework [9], which is an evolution of *FlumeJava* [5], allowing cycles and therefore incremental computation. It is a completely domain-agnostic dataflow framework integrated with many other closely-related technologies from Google, like Cloud Storage, Cloud PubSub, Cloud Datastore, Cloud Bigtable and BigQuery. It is open-source, offers fully automatic resource management that auto-scales for optimal throughput and provides increased reliability and data consistency. Moreover, it provides a unified programming model through its API, while allowing data monitoring and demand-driven execution.

In contrast to RHEA, graphs constructed by *Cloud Dataflow* are designed to be deployed only on cloud infrastructures, and therefore no support for complete heterogeneity is provided. In terms of network optimization, namely node placement, *Cloud Dataflow* lets the cloud system targeted to make all decisions, while RHEA profiles the network and decides autonomously.

[8] http://akka.io
[9] http://cloud.google.com/dataflow/

A less-known framework for Python is *dispel4py* [10]. It provides the ability to describe abstract workflows for distributed data-intensive applications. Similar to our *EvaluationStrategy* concept, it allows different mappings to enactment systems, such as MPI and Apache Storm. Its main disadvantages are that it has only an API for Python and only allows low-level specification of the graph's nodes, through the definition of *Processing Elements*. Therefore, it is inconvenient to compose larger graphs from simpler ones and the source code becomes chaotic and difficult to maintain.

9.3 Dataflow optimization

The frameworks discussed so far follow, more or less, an imperative approach, which enables automatic distribution and concurrency by using immutable data structures. *Stratosphere* [2], on the other hand, follows a declarative programming approach similar to RHEA, which enables writing highly parallel code directly from the language's semantics.

Apart from offering a language of a much higher abstraction level, *Stratosphere* has internalised several interesting and novel approaches to optimization of dataflow graphs, especially concerning cyclic graphs (i.e. incremental computation) [9]. These optimizations are generic, in the sense that most frameworks can adopt them without much effort. Integrating these optimization into RHEA, as future work, would certainly be of great benefit to the performance of the system.

9.4 Heterogeneous data processing

Another dataflow framework from Google is *TensorFlow* [1], which is an open-source polyglot library for machine learning and especially construction of neural networks. The interesting fact is that, although it started out as a rigid neural network library, it quickly generalized to a dataflow construction library, much similar to our own project, which started out as a robotics library. Its main features are its portability to multiple computational architetures (e.g. CPU, GPU) and multiple language APIs (e.g. C++, Python), although its main advantage are its domain-specific operators for neural nets (i.e. common subgraphs, auto-differentiation). Through the edges/streams connecting the nodes, only a single but flexible data type is allowed, namely the *Tensor* type, which essentially is a multi- dimensional array that usually represents features or weights. In contrast to RHEA's Streams, *Tensors* cannot be infinite, mainly due to the fact that their size is determined by the dimensionality of the problem being solved, which is, in most cases, a fixed constant.

9.5 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 [8, 11, 17]. *Yampa* [11] is a DSL embedded in Haskell that realizes the FRP model, using arrows to minimize time-/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* tool for JavaScript following 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.

10 Conclusions and Future Work

The framework described in this paper was designed with extensibility in mind, aiming to act as a fundamental basis, onto which various domain-specific libraries or DSLs will rely in the future. To that end, a constant effort to generalize and make components as abstract as possible was made.

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 (e.g. *Map-Reduce*), but recent research has shown that general dataflow topologies have optimization opportunities that are yet to be found [12]. A minimal optimization stage has been implemented, which paves the path to more advanced optimization techniques, such as those used in *Naiad* [14] and *Stratosphere* [12].

The extensible nature of RHEA allows for many meaningful extensions, such as more evaluation/distribution strategies to support integration with other software ecosystems, Moreover, the design of a block-based visual language interface would certainly make the framework even more accessible to novice programmers and smooth the learning curve associated with dataflow programming.

There are also significant shortcomings to the framework design and implementation that could be addressed in future work. For instance, *dynamic reconfiguration* is needed to handle environments that are constantly changing. A concrete contribution to the RHEA would be to integrate *HotWave* [20], which is an *aspect-oriented programming (AOP)* framework that supports dynamic (re)weaving of previously loaded classes. This would allow the user to specify the desired adaptive behaviour for reconfiguring where nodes are executed, what operation they perform, and so forth.

Another major drawback is that network profiling - an integral part of node placement - is achieved via calculation of *round-trip time (RTT)*, which is naively expensive and may outweigh the benefits of exploiting network proximity. A possible decentralized approach that we could employ would be the *Vivaldi* coordinate system [7], which assigns synthetic coordinates to hosts such that the distance between the coordinates of two hosts accurately predicts the communication latency between them.

A last major drawback that ultimately needs to be addressed is fault-tolerance, since there are no advanced methods for specifying behaviour for graceful error-recovery. This is essential for large machine clusters, in which systems it is certain that host failures and other faults will be a common occurrence. The functional nature of the dataflow model enables fault-tolerance, in addition to parallelism, due to the fact that a node can be moved to another machine for execution, while preserving the original semantics. This contribution path can draw heavy influence from recent research on fault-tolerance for stream processing engines [3], which provide efficient models for availability and data recovery/consistency, by using data replication and parallel recovery of lost state.

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. This is exactly what *Ziria* accomplishes in the domain of wireless systems programming [19]. 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.

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] 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.

[3] Magdalena Balazinska, Hari Balakrishnan, Samuel R Madden, and Michael Stonebraker. 2008. Fault-tolerance in the Borealis distributed stream processing system. *ACM Transactions on Database Systems (TODS)* 33, 1 (2008), 3.

[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] Frank Dabek, Russ Cox, Frans Kaashoek, and Robert Morris. 2004. Vivaldi: A decentralized network coordinate system. In *ACM SIGCOMM Computer Communication Review*, Vol. 34. ACM, 15–26.

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

[9] Stephan Ewen, Kostas Tzoumas, Moritz Kaufmann, and Volker Markl. 2012. Spinning fast iterative data flows. *Proceedings of the VLDB Endowment* 5, 11 (2012), 1268–1279.

[10] Rosa Filguiera, Amrey Krause, Malcolm P. Atkinson, Iraklis A. Klampanos, and Alexander Moreno. 2017. dispel4py. *IJHPCA* 31, 4 (2017), 316–334. https://doi.org/10.1177/1094342016649766

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

[12] 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.

[13] 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).

[14] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 439–455.

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

[16] 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).

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

[18] 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.

[19] Gordon Stewart, Mahanth Gowda, Geoffrey Mainland, Bozidar Radunovic, Dimitrios Vytiniotis, and Cristina Luengo Agulló. 2015. Ziria: A DSL for wireless systems programming. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 415–428.

[20] Alex Villazón, Walter Binder, Danilo Ansaloni, and Philippe Moret. 2010. Advanced runtime adaptation for Java. *ACM Sigplan Notices* 45, 2 (2010), 85–94.

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

[22] 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.