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

Anonymous Author(s)

Abstract

The dataflow computational model enables writing highly parallel programs, to be deployed on a heterogeneous network, in a concise and readable way. The main advantage is the fact that the system can be conceptually separated into several independent components that can be run in parallel and deployed on different machines. Therefore, concurrency and distribution is implicit and little or no responsibility is given to the programmer. The framework proposed in this thesis constitutes the underlying system that make this style of programming possible in JVM-based languages (e.g. Java, Scala, Closure), while at the same time making it easy to integrate other technologies that rely on the PubSub model, in order to move away from imperative languages and enter a higher level of abstraction. Particular emphasis was put on three domains, namely *Big Data*, *Robotics* and *IoT*.

CCS Concepts • **Software and its engineering → Data flow architectures**; **Data flow languages**;

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

1 Introduction

1.1 Main concept

Our main contribution is the design and implementation of a framework for dataflow programming to be deployed anywhere, ranging from low-performance robots and sensors to clusters of computer and even the Cloud.

The main idea is to provide the programmer with a different execution model, the dataflow model, which allows for a more abstract way of thinking and has the advantage of exposing opportunities for parallelism (amongst CPU cores) and distribution (amongst computational machines), which can then be automatically realised by the ”intelligent” underlying system.

Therefore, the programmer will be able to utilize available computational resources without any effort, while at the same time reducing development time/cost and maintaining a much cleaner and easier-to-refactor software system. Resource utilization may appear in the form of faster execution (i.e. by concurrently doing computations on multiple machines) or more robust error-handling (i.e. by using backup machines to rerun nodes that were hosted on a faulty machine).

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

1.2 Motivation

1.2.1 Declarative languages

Software is becoming increasingly more complex, as computing capabilities are strengthened and user needs become more demanding. Thus the need for higher abstraction becomes imperative, as it provides a more structured, easier to debug and maintainable way of developing software. In other words, abstraction in computer science acts as a mean to overcome complexity.

In programming languages, the level of the aforementioned abstraction is measured regarding the amount of low-level details a programmer has to specify. Therefore, languages can be divided in two categories: the imperative ones, in which the programmer specifies what needs to be done and how to do it, and the declarative ones, where the programmer only specifies what needs to be done and rely on the underlying compiler/interpreter to produce the exact commands that will realize the desired behaviour. The most well-known declarative programming paradigms are functional and logic programming, each providing higher abstraction in different aspects. Our approach was greatly influenced by the functional paradigm.

1.2.2 Data versus Computation

A common problem in heterogeneous systems is that different representations of the same entities/data-types coexist in the same software and, as a consequence, pure computational tasks are intermingled with data-converting tasks. This makes the code less readable and harder to maintain and understand. In the dataflow execution model, where the program is modelled as directed graph of data flowing between operations, there is a clear separation of these two aspects as data (edges) are completely decoupled from computation (nodes). This motivation is strengthened even more, when cross-machine communication is included, and apart from converting data from one representation to another, serialization(i.e. conversion to bytes) is also mandatory.

1.2.3 Dataflows in Robotics

In control theory, which is the main background theory used in robotics, most architectures and/or algorithms are represented as dataflow diagrams for the sake of clarity and

intuition. Translating these diagrams into common "imperative" software is not an easy task and is usually the source of bugs. Thus, having a dataflow execution model will nullify the need for such a translation.

Specifically, most robotic applications follow the *Robot Perception Architecture (RPA)*, 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 robotic systems, whose combination is even more challenging. If each individual system is represented as a dataflow graph, composing them together is as trivial as connecting inputs with outputs, which is not the case in a traditional architecture, which is not component-based.

1.2.4 Dataflows in Big Data

Another reason for following a dataflow approach is the attention that it recently has drawn in the *Big Data* field. As data size is growing exponentially and distribution is not a luxury but a necessity, a more scalable and decentralized architecture is destined to be examined in more depth. As we will discuss in Section 8, there are many recent frameworks that became famous for their scalability due to the fact that they rely on a dataflow approach.

2 Background

2.1 The dataflow computational model

The increased interest in parallelism during the 70's gave rise to the dataflow execution model, which is an alternative to the classical "von-Neumann" model. In the dataflow model, everything is represented in a dataflow graph, where nodes are independent computational units and edges are communication channels between these units. A node/unit is fired immediately when its required inputs are available and therefore no explicit control commands are needed for execution. Figure 1 shows a dataflow graph enumerating the set $\mathbb{N}$ of natural numbers.

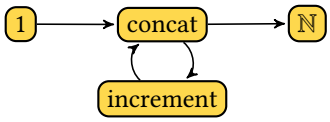


Figure 1. Natural numbers

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.

The main advantage of the dataflow model is its implicit parallelism, deriving from the fact that the computational units are totally independent and therefore can be executed in parallel. 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 Functional reactive programming

A relatively recent programming paradigm is *Functional Reactive Programming (FRP)*, which provides a conceptual framework for implementing reactive (i.e. time-varying and responding to external stimuli) behaviour in *hybrid systems* (i.e. containing both continuous and discrete components), such as robots, in functional programming languages.

To implement such systems in conventional imperative languages, one must use asynchronous *callbacks* (i.e. each change is handled by a registered *callback* function). Although this solution is satisfactory for simple schemes, more complex scenarios eventually lead to highly incoherent code structure, often called *spaghetti code*, in the sense that control rapidly moves between disconnected parts of the system, similar to the notorious *GOTO* command. This phenomenon stems from the unary nature of *callback* functions, which requires some kind of "internal plumbing" in order to achieve mechanisms for handling combination of changes (e.g. when multiple changes occur simultaneously). *FRP* provides a solution to this shortcoming of *callback* functions, because changes are represented as variables (*signals*), which can be passed as parameters to arbitrary functions, called *signal functions*.

FRP first appeared as a composable library for graphic animations [10], but quickly evolved into a generic paradigm [3, 11, 31]. Moreover, extensive research has investigated *FRP* as a framework for robotics [16, 28].

Although appealing at first, *FRP* was not appropriate for systems with real-time constraints, due to uncontrollable

time- and space- leaks [32]. The solution was a generalization of monads called *arrows* [19], which provided the necessary guarantees that the aforementioned common errors do not occur. Let's see the example of calculating a robot's x-coordinate. Here is the mathematical formula drawn from control theory:

$$x = 1/2 \int (vr + vl) \cos\theta$$

Below is the corresponding *FRP* code:

```
x = let
  v = (vrSF &&& vlSF) >>> lift (+)
  t = thetaSF >>> arr cos
  in (v &&& t) >>> lift (*) >>> integral >>> lift (/2)
```

As the above may seem counter-intuitive and difficult to understand, the *arrow notation*[27] was introduced:

```
x = proc inp -> do
    vr <- vrSF -< inp
    vl <- vlSF -< inp
    theta <- thetaSF -< inp
    i <- integral-< (vr+vl) * cos(theta)
returnA -< (i/2)
```

The main advantages of *FRP* are its close correspondence to mathematics[3], which make it an ideal framework for modelling real-time systems, and its concise representation of time-varying values via *signals*.

2.3 Publish-Subscribe model

Publish/Subscribe (PubSub) is a messaging pattern that became popular due to the loose coupling of its components, suited for the most recent large-scale distributed applications.

There is no point-to-point communication and no synchronization. *Publishers* advertise messages of a given type to a specific message class or *topic* that is identified by a *keyword*, whereas *subscribers* listen on a specific *topic* without any knowledge of who the publishers are. The component responsible for relaying the messages between machines and/or processes and finding the cheaper dissemination method is called the *message broker*. Figure 2 illustrates an abstract representation of the *PubSub* model.

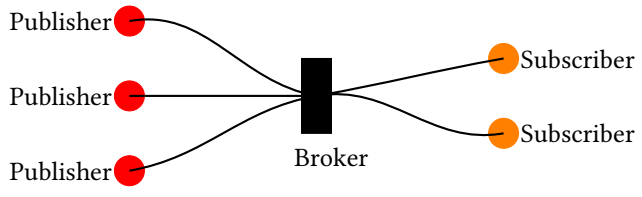


Figure 2. PubSub typical layout

2.4 ROS: Robot Operating System

ROS is an open-source middleware for robot software, which emphasizes large-scale integrative robotics research citeROS. 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.

2.5 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[26]. 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[21].

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[5], 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.

2.6 The Reactive Streams Standard

PubSub is widely used by different frameworks but still lacks standardization. The *Reactive Streams Standard (RSS)* is an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure. This encompasses efforts aimed at runtime environments (JVM and JavaScript) as well as network protocols[1].

RSS defines two minimal interfaces for the roles of *Subscriber* and *Publisher*[1]. A *Subscriber* implementation should define reactions to observed values, including normal and erroneous termination, whereas a *Publisher* implementation should accept requests from *Subscribers* and start emitting values to them.

Below we see a minimal example of using *RSS* to define a publisher that emits values 1..10 and a subscriber that prints all observed values and finally connect them together.

```
Publisher<Integer> pub = new Publisher {
        void subscribe(Subscriber<Integer> sub) {
                for (int i = 1; i <= 10; i++)
                        sub.onNext(i);
                sub.onComplete();
        }
};

Subscriber<Integer> sub = new Subscriber {
        void onNext(Integer i) { println(i); }
        void onComplete() { println("Complete"); }
        void onError(Throwable t) { t.printStackTrace(); }
};

pub.subscribe(sub);
```

3 Requirements

The design was heavily influenced by principles set out by the FRP and dataflow models.

3.1 Reactive

The system should be *reactive*, as close as possible to the definition of the Reactive Manifesto[20].

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.

[1] http://www.reactive-streams.org/reactive-streams-1.0.0-javadoc/

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.

3.2 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.

Figure 3 illustrates a simple example of a robot application pipeline, where input to the dataflow graph is what the robot's camera senses and, after some image processing and some computation-heavy decision making, a command to an actuator of the robot is executed. Orange nodes are deployed on the robot's on-board computer, the green node is deployed on an off-board GPU and the red node is deployed on the main server.

Figure 3. Heterogeneity pipeline

3.3 Extensible

As the work described in this thesis 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.

3.4 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 Approach

This section presents the framework's main characteristics and capabilities.

4.1 System architecture

The RHEA ecosystem consists of several clearly separated modules, whose interconnection is illustrated in figure 4.

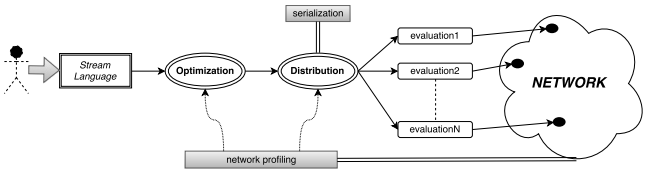


Figure 4. System architecture

The user writes a program in the provided stream language, 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.

The following subsections will present the aforementioned stream language, while the other components, namely *optimization, distribution, evaluation, serialization and network profiling*, will be discussed in Sections 5 and 6.

4.2 Supported 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. Figure 5 depicts such a graph, where inputs and outputs are colored red and green respectively.

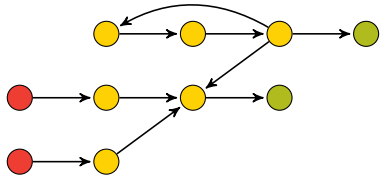


Figure 5. Example of supported graph

4.3 Stream language

This section presents the framework's language for defining streams and executing them. Due to space constraints, only a very small subset of the provided operators will be presented.

4.3.1 The Stream data type

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.

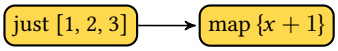
4.3.2 Graph construction syntax

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 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.

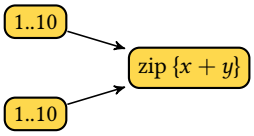
Single input nodes are inserted into an existing Stream object, by calling an operator on that object. Figure 6 shows an example of a single input node, namely that of *map*, which transforms the input stream (i.e. just the values 1, 2 and 3) by applying a user-defined function to every emitted value (i.e. $f(x) = x + 1$).




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

Figure 6. Single input processing node

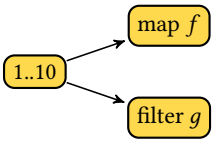
Multiple input nodes are constructed by built-in function that take as argument already existing Stream objects. Figure 7 shows an example of a multiple input node, namely that of *zip*, which transforms the input streams (i.e. two stream that emit the values 1..10) by applying a user-defined function to each emitted pairs of values (i.e. $f(x, y) = x+y$). Here we also see another stream creation function, namely *Stream.range*.




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

Figure 7. Multiple input processing node

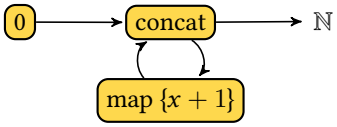
The variables returns by all processing nodes are *Stream* objects. These objects can be reused in different parts of the graph to enable splitting a node's output to different processing nodes or outputs. Figure 8 shows such an example, where the *filter* operator only emits values for which the given function returns true.




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

Figure 8. Split example

Cycle construction is a bit trickier, as no direct manipulation of the internal graph is permitted. 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 9 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.




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

Figure 9. Cyclic example

4.3.3 Evaluation syntax

To evaluate a given dataflow graph and do something with its output values, we need to call the *subscribe* method of 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. Figure 10 shows an example of printing the output of the single-input example in figure 6.


```
Stream.just(1, 2, 3)
      .map(x -> x + 1)
      .subscribe(System.out::println);
```

Figure 10. Evaluation example

4.3.4 Execution semantics

As the execution is completely asynchronous, the order of stream declaration and/or evaluation does not matter at all. If there is absolute necessity for control flow management, the programmer can use the altered *BlockingStream* type that blocks program execution to next *subscribes* until the currently subscribed streams terminate successfully.

Nonetheless, this utility is helpful for rapid prototyping and testing purposes. The source code below shows an example of accumulating the 10 first natural numbers in a list and printing that list.

```
Stream<Int> s1 = Stream.nat();
BlockingStream<Int> s2 = s1.toBlocking();
List<Int> list = s2.toList();
System.out.print(list);
```

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. Figure 11 illustrates all the pluggable components of the framework around the core, which are normally deployed in separate libraries and for which default implementations are already provided by the current implementation.

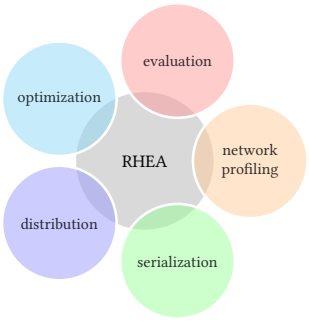


Figure 11. The RHEA ecosystem

Internal representation For representing the internal structure of the dataflow graph, the *JGrapht* open-source Java library was used, which provides many graph data structures and common graph-theoretic algorithms[2].

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.

[2]http://jgrapht.org/

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.

5.1 Evaluation

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* just takes the *Stream* variable and its corresponding *Output* and executes it, however desired.

The strategies we have implemented so far follow:

RxJavaEvaluationStrategy
: Uses rxjava[3], which is a famous 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 intergrated.

5.2 Distribution

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.

Hazelcast The default *DistributionStrategy* shipped with the framework uses the *Hazelcast*[4] library to discover and manage multiple machines and used its internal decentralized *PubSub* model to communicate intermediate results across the network. Figure 12 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.

Machine configuration According to the distribution strategy being used, the available machines will require a certain initial configuration. For the *Hazelcast* case, a little

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

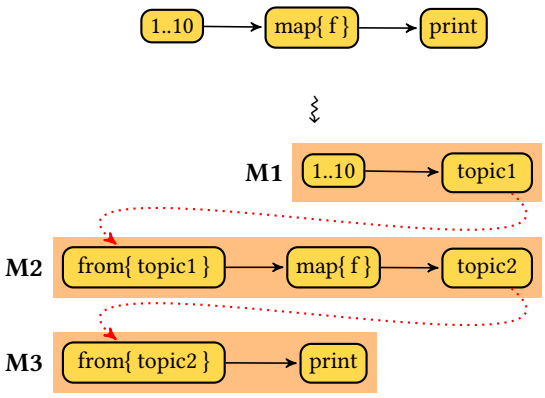


Figure 12. Partitioning

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.

Serialization As communication between machines across a network is mandatory, data types emitted through the streams must be serialized on departure and deserialized 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 13 depicts the serialization process in more detail.

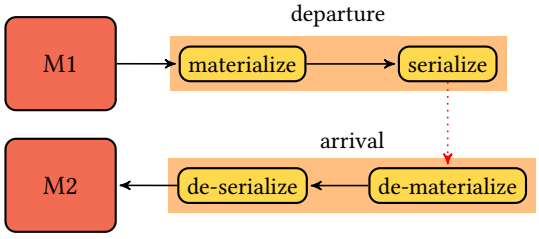


Figure 13. Serialization process

6 Optimization

This section describes three stages of optimization the dataflow graph goes through before being evaluated, which are illustrated in Figure 14. The purpose of the optimization graph is to achieve better performance and utilization of the available resources.

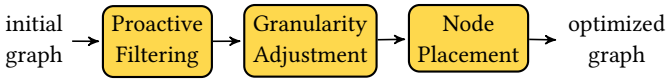


Figure 14. Optimization stages

6.1 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 show the corresponding graph transformations.

Transformations The figures below illustrate one representative example of each general class of graph transformation

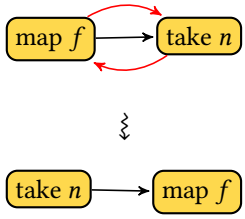


Figure 15. Take/skip/distinct before map

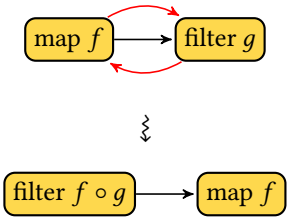


Figure 16. Filter before map

6.2 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.

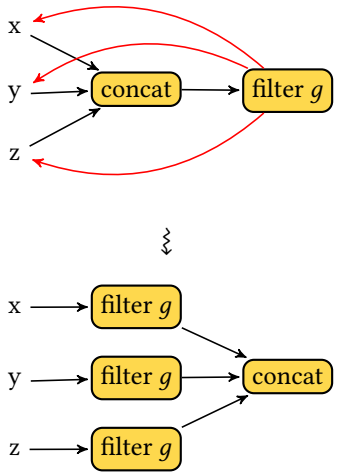


Figure 17. Filter/distinct before concat/merge

Transformations To reach the desired granularity, the optimizer applies some semantic-preserving transformation, as shown in the figures below (for simplicity, only a single example of each general case is demonstrated).

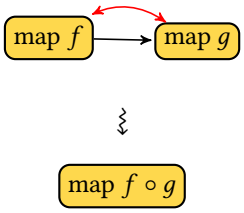


Figure 18. Merge maps

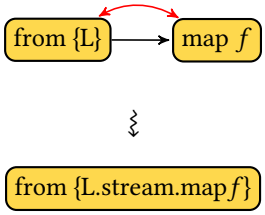


Figure 19. Embed map in creation

In figure 18 we merge two *map* operations into one *map* operation that uses the composition of the two initial functions, while in figures 19 and 20 we utilize Java's built-in stream operators on its collections, namely *map* and *repeat*. In figure 21 a *map* followed by a *filter* is substituted by a more complex equivalent operation, namely *filterMap*, while in figures 22 and 23 we apply some simple properties of the boolean functions involved to decrease the number of nodes. In figures 24 and 25 we utilize function composition

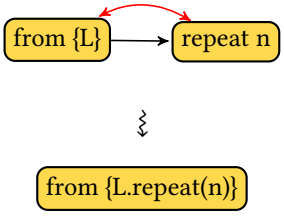


Figure 20. Embed repeat in creation

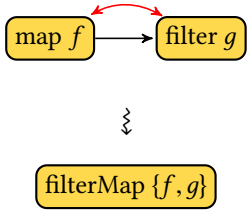


Figure 21. Combine map with filter

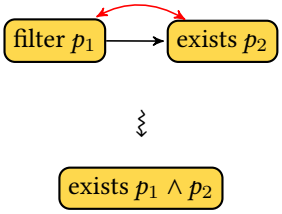


Figure 22. Combine filter with exists

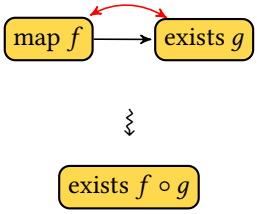


Figure 23. Combine map with exists

to embed *map* operations into *zip* operations. Lastly, in figure 26 we remove sub-graphs that are not reachable, due to *never* operations.

6.3 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 27.

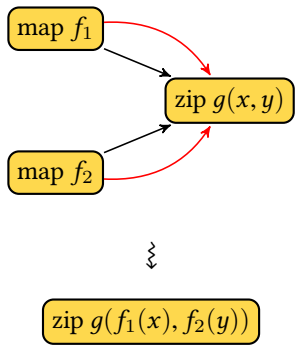


Figure 24. Combine map with zip

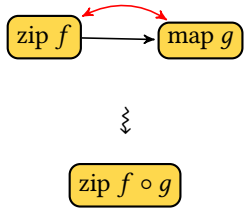


Figure 25. Combine zip with map

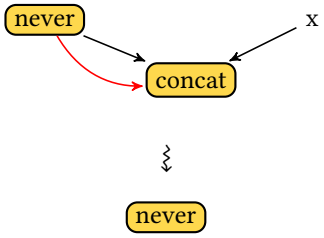


Figure 26. Meaningless nevers

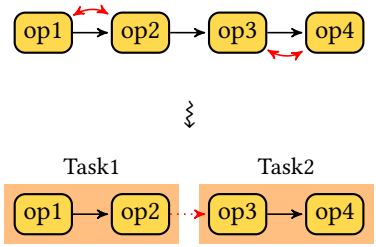


Figure 27. Task fusion

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.

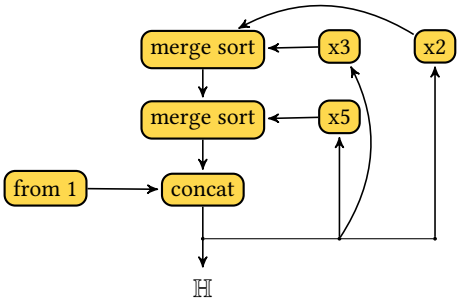
At this stage, we identify the aforementioned network distance as cost and apply brute-force to find the optimal placement of the (groups of) tasks that minimize that cost.

7 Applications

This section will demonstrate some use-cases of the framework.

7.1 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[33]. Figure 28 shows the dataflow graph with its corresponding RHEA code.




```
Stream.just(1)
  .loop((entry: Stream[Int]) =>
    (entry.multiply(2) mergeSort entry.multiply(3))
      mergeSort
    entry.multiply(5) : Stream[Int])
  .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] = {
    val queue = new PriorityQueue[Int]()
    Stream.zip(stream, other, (x, y) => {
      val min: Int = Math.min(x, y)
      val max: Int = Math.max(x, y)
      queue.add(max)
      if (min < queue.peek())
        min
      else {
        queue.add(min)
        queue.poll() }})
    .concatWith(Stream.from(queue))
  }
}
implicit def enrichStream(st: Stream[Int]): IntStream =
  new IntStream(st)
```

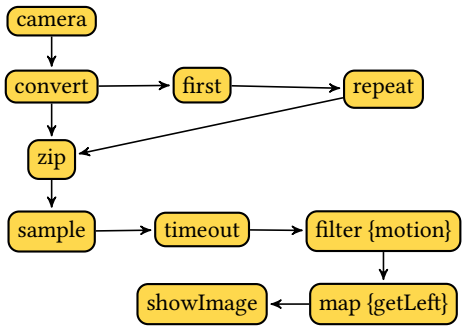
Figure 28. Hamming numbers

The code is written in Scala to utilize the *Pimp my library* design pattern[25], 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.

7.2 Camera surveilance

Moving to a more realistic, but still minimal, use-case, this example demonstrates how cleanly we can express a surveillance application from a robot's camera.

The camera should send frames to be displayed on the screen, only when motion is detected. Figure 29 shows the dataflow graph with its corresponding RHEA code.




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

Stream<Mat> image =
  Stream.from(new RosTopic<>("/camera"))
        .map(CvImage::toCvCopy);

Stream<Mat> initial = image.first().repeat();

Stream.zip(image, initial, Pair::new)
      .sample(100, TimeUnit.MILLISECONDS)
      .timeout(1, TimeUnit.MINUTES)
      .filter(Surveillance::motionDetected)
      .map(Pair::snd)
      .subscribe(window::showImage);

boolean motionDetected(Pair<Mat,Mat> pair) {
  Mat m1 = pair.getLeft(), m2 = pair.getRight(), m = new Mat();
  Core.absdiff(m1, m2, m);
  Imgproc.threshold(m, m, 80, 255, Imgproc.THRESH_BINARY);
  Imgproc.erode(m, m, Imgproc.MORPH_RECT(3,3));
  for (int i = 0; i < m.rows(); i++)
       for (int j = 0; j < m.cols(); j++) {
         double[] pixel = m.get(i, j);
          double sum = pixel[0]  + pixel[1] + pixel[2];
         if (sum > 50) return true;
  }
  return false;
}
```

Figure 29. Camera surveillance

In the code above, we can see how easy it is to view a *ROS* topic as a stream, using the *RosEvaluationStrategy* (line 3). Moreover, there is a nice separation between program logic (stream declaration in lines 6-17) and implementation details (*motionDetected* function in line 19).

7.3 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 showcase the framework's ability for simulation. Figure 30 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 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.

7.4 Robot hospital guide

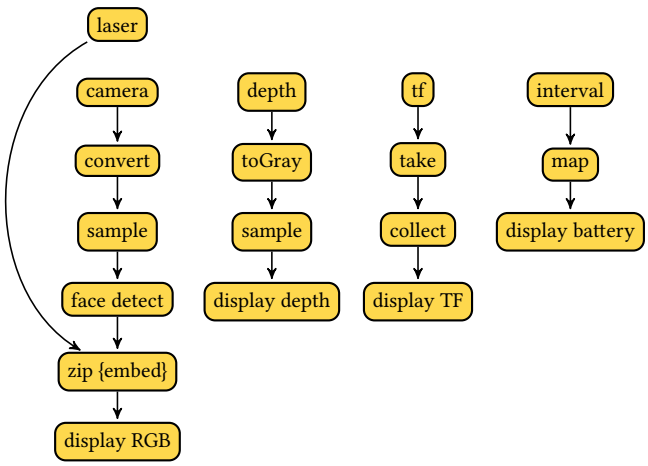
As a final example, we will examine a more IoT-based application. Consider a robot that guides patients to different parts of a hospital, such as the gym or cafeteria. Assuming map localization, path finding and obstacle avoidance are already implemented, there still remains a problem with calibrating 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 31 shows the dataflow graph with its corresponding RHEA code.

The second module just uses the *ReactiveBeacons* library[5] to get a stream of beacon data via *rxjava*, and then publishes

[5]https://github.com/pwittchen/ReactiveBeacons




```
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) -> {
      List<Transform> transforms = msg.getTransforms();
      for (Transform transform : transforms) {
        String parent = transform.getHeader().getFrameId();
        String child = transform.getChildFrameId();
        if (!m.containsKey(parent)) {
          Set<String> init = new HashSet<>();
          init.add(child);
          m.put(parent, init); }
        else m.get(parent).add(child); }})
    .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 30. Robot control panel

it to a *MQTT* topic, which is the input of the first module. The corresponding RHEA code follows:

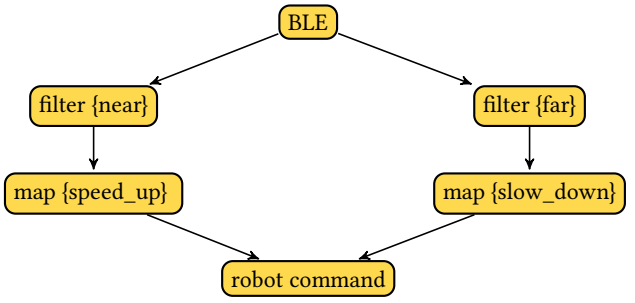
```
Stream.configure(new HazelcastDistributionStrategy(
          RxjavaEvaluationStrategy::new,
          MqttEvaluationStrategy::new));

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.

8 Related Work

This section discusses related work in the fields of *Big Data*, *Robotics* and *IoT*.




```
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 31. Robot hospital guide

8.1 Big Data

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.

8.1.1 MapReduce

MapReduce was developed by Google and provides a very simple model, and corresponding runtime, that allows automatic concurrency/distribution on a cluster by allowing only a very minimal program structure. First, the user specifies a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Although it was widely adopted at first, quickly many problems that could not be expressed with the above formalism were found and therefore a more expressive model was required.

8.1.2 Flumejava

A generalization of the *MapReduce* framework is *FlumeJava*, again developed by Google, which tries to overcome its *MapReduce's* shortcomings, by allowing more expressive

pipelines consisting of multiple *MapReduce* stages. Additionally, *FlumeJava* provides some more abstract operations that, when evaluated, are compiled into primitive *MapReduce* steps.

Although *FlumeJava* was more attractive due to its expressibility, still the pipeline constructed could not formulate all problems that are needed in many big-data applications. For instance, the constructed dataflow could not contain cycles, which is an integral part of *incremental computation*, used extensively nowadays for machine learning and data analysis.

8.1.3 Spark

A very well-known and well-adapted framework for scalable large-data processing is Apache's *Spark*[6]. 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.

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*.

It offers a rich set of data-parallel operators ($\simeq 80$) that can be used interactively from Scala, Python, Java or R. The code below shows the classic word-counting example in Spark's Scala API.

```
Spark.textFileStream("hdfs://...") /* Get file stream */
     .flatMap(_.split(" ")) /* Split into words */
     .map(x => (x, 1)).reduceByKey(_ + _) /* Count words */
```

8.1.4 Cloud Dataflow

Continuing the search for more expressive models, Google recently released the *Cloud Dataflow* framework[7], which is an evolution of *FlumeJava*[7], 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.

8.1.5 Stratosphere

The frameworks discussed so far follow, more or less, an imperative approach, which enables automatic distribution/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)[12]. 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.

8.1.6 Naiad

Another high-level dataflow system that follows a declarative approach similar to RHEA is *Naiad*, which unifies incrementally iterative computations with continuous data ingestion into a new technique called differential computation.

Offering the high throughput of batch processors, the low latency of stream processors and the ability to perform iterative and incremental computations at the same time is extremely challenging and none of the aforementioned frameworks manage to provide it. Applications that need all these features need to rely on multiple platforms, at the expense of efficiency, maintainability and simplicity.

*Naiad*citenaiad combines all of these features in a unifying framework, that provides a generic low-level platform, that a wide variety of high-level programming models can be built upon, enabling such diverse tasks as streaming data analysis, iterative machine learning, and interactive graph mining.

Its main contribution is the definition of a new computational model, namely the *Timely Dataflow* model, which is an extension to the dataflow model we introduced in the first section, by allowing a more efficient and lightweight coordination mechanism for capturing opportunities for parallelism. This is achieved by enriching the dataflow model with timestamps that represent logical points in the computation.

8.1.7 Akka

Definitely one the most mature frameworks for distribution targeting the JVM, *Akka*[8] is a toolkit and runtime for

[6] http://spark.apache.org/

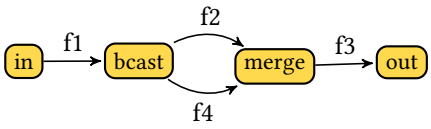
[7] https://cloud.google.com/dataflow/

[8] http://akka.io/

highly concurrent, distributed and resilient message-driven applications. It is also one of the founders of the *Reactive Streams*[1] initiative.

Its approach follows the *Actor* model[15], 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. Figure 32 demonstrates a dataflow graph generated by the DSL code below it.




```
val g = FlowGraph { implicit b =>
  import FlowGraphImplicits._
  val in = Source(1 to 10)
  val out = Sink.ignore
  val bcast = Broadcast[Int]
  val merge = Merge[Int]
  val f1, f2, f3, f4 = Flow[Int].map(_ + 10)
  in ~> f1 ~> bcast ~> f2 ~> merge ~> f3 ~> out
                bcast ~> f4 ~> merge
}
```

Figure 32. Akka DSL

The main reason RHEA offers a more flexible solution to *ROS* shortcomings than *Akka*, is that *Akka* is a pretty heavyweight 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*.

8.1.8 dispel4py

A less-known framework for Python is *dispel4py*[9]. It provides the ability to describe abstract workflows for distributed data-intensive applications.

[9] https://github.com/dispel4py

Similar to our *EvaluationStrategy* concept, it allows different mappings to enactment systems, such as MPI[10] and Apache Storm[11].

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.

8.2 Robotics

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.

8.2.1 roshask

Roshask[8] 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.

Below is the classic Talker-Listener *ROS* example, where one node publishes messages to a topic and another one listens for them.

```
sayHello :: Topic IO S .String
sayHello = repeatM (fmap mkMsg getTime)
where mkMsg = S .String . ("Hi "++) . show

tn :: Node ()
tn = advertise "chatter" (topicRate 1 sayHello)

main :: IO ()
main = runNode "talker" tn

showMsg :: S .String -> IO ()
showMsg = putStrLn . ("Msg: "++) . S.data

main = runNode "listener" $
runHandler showMsg = << subscribe "chatter"
```

8.2.2 Yampa

RHEA and *roshask* were heavily influenced by the work of Hudak's group (Yale Haskell Group) on robot DSLs and FRP in general[10, 11, 16, 28, 31, 32]. *Yampa*[12] is a DSL embedded in Haskell that realizes the FRP model, using arrows to minimize time-/space- leaks.

[10] http://www.mpich.org/
[11] http://storm.apache.org/
[12] https://wiki.haskell.org/Yampa

8.2.3 Flowstone

Flowstone[13] is a programming environment that mixes graphical and text-based programming in Ruby that can be used for robotics, image/signal processing and interconnecting heterogeneous sources. It follows a variant of the dataflow model, where applications are built by linking together functional blocks called components. Its main advantage is that it is stand-alone, so no compiling is necessary, which allows for rapid prototyping.

8.3 Internet of Things

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.

8.3.1 NoFlo

NoFlo[14] is a JavaScript implementation of *Flow-based Programming*[22], which is a particular form of dataflow programming, based on bounded buffers, information packets with defined lifetimes, named ports and separate definition of connections.

NoFlo applications consist of components that are connected together in a graph. This allows for clear separation of control flow from the actual software logic, helping you organize large applications easier than traditional OOP paradigms. You can design *NoFlo* applications using a web-based graph editor[15].

8.3.2 Node-RED

Another interesting *IoT* tool for JavaScript following a dataflow approach is *Node-RED*[6], 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. The online flow library[16] has had a huge number of contributions so far.

8.4 Miscellaneous

8.4.1 TensorFlow

Another dataflow framework from Google is *TensorFlow*[17], 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, etc...) 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.

8.4.2 Ziria

The dataflow computational model also found applications in the field of wireless systems programming, particularly in the domain of *software-defined radio (SDR)*.

Ziria[29] is a DSL that offers programming abstractions suitable for wireless physical (PHY) layer tasks while emphasizing the pipeline reconfiguration aspects of PHY programming. *Ziria* also implements many specialized optimization steps that enable it to be on par and in many cases outperforms a hand-tuned state-of-the-art C++ implementations on commodity CPUs.

9 Conclusions & Future Work

The framework described in this thesis 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[18]. A minimal optimization stage has been implemented, which paves the path to more advanced optimization techniques, such as those used in *Naiad*[23] and *Stratosphere*[17].

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.

[13] http://www.dsprobotics.com/flowstone.html
[14] http://noflojs.org/
[15] https://flowhub.io/
[16] http://flows.nodered.org/
[17] https://www.tensorflow.org/

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*[30], 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[9], 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[4, 24], 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[29]. 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.

This is a general notion in computer science, owning its existence to the fact that the problems we are facing are getting increasingly more complex, while resources meet certain realistic bounds, and therefore a higher abstraction layer is mandatory for maintaining readability, efficiency and expressibility.

References

[1] 2015. Reactive Streams Standard. http://www.reactive-streams.org/.

[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] Edward Amsden. 2011. A survey of functional reactive programming. *Unpublished* (2011).

[4] 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.

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

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

[7] 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.

[8] 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.

[9] 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.

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

[11] Conal M Elliott. 2009. Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*. ACM, 25–36.

[12] 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.

[15] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*. Morgan Kaufmann Publishers Inc., 235–245.

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

[17] Fabian Hueske, Aljoscha Krettek, and Kostas Tzoumas. 2013. Enabling operator reordering in data flow programs through static code analysis. *arXiv preprint arXiv:1301.4200* (2013).

[18] 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.

[19] John Hughes. 2000. Generalising monads to arrows. *Science of Computer Programming* 37, 1–3 (2000), 67 – 111. https://doi.org/10.1016/S0167-6423(99)00023-4

[20] Bonér Jonas, Dave Farley, Roland Kuhn, and Martin Thompson. 2014. Reactive Manifesto. http://www.reactivemanifesto.org/.

[21] 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).

[22] J Paul Morrison. 1994. Flow-based programming. In *Proc. 1st International Workshop on Software Engineering for Parallel and Distributed Systems*. 25–29.

[23] 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.

[24] Anh Nguyen-Tuong, Andrew S Grimshaw, and Mark Hyett. 1996. Exploiting data-flow for fault-tolerance in a wide-area parallel system. In *Reliable Distributed Systems, 1996. Proceedings., 15th Symposium on*. IEEE, 2–11.

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

[26] 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).

[27] Ross Paterson. 2001. A New Notation for Arrows. *SIGPLAN Not.* 36, 10 (Oct. 2001), 229–240. https://doi.org/10.1145/507546.507664

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

[29] 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.

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

[31] Zhanyong Wan, Walid Taha, and Paul Hudak. 2001. Real-time FRP. In *ACM SIGPLAN Notices*, Vol. 36. ACM, 146–156.

[32] Zhanyong Wan, Walid Taha, and Paul Hudak. 2002. Event-driven FRP. In *Practical Aspects of Declarative Languages*. Springer, 155–172.

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