NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

FACULTY OF EXACT SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BACHELOR THESIS

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

Orestis Melkonian

Supervisors: **Panos Rondogiannis,** Professor EKPA
Angelos Charalambidis, Researcher NCSR

ATHENS

APRIL 2016

BACHELOR THESIS

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

Orestis Melkonian
A.M.: 1115201000128

SUPERVISORS: **Panos Rondogiannis,** Professor EKPA
Angelos Charalambidis, Researcher NCSR

ABSTRACT

Summary here

SUBJECT AREA: Programming Languages

KEYWORDS: dataflow, stream, frp, distributed systems, model-driven architecture program synthesis declarative languages node placement

”τὰ όντα ιέναι τε πάντα καὶ μένειν ουδέν”
(all entities move and nothing remains still)
- Heraclitus

ACKNOWLEDGEMENTS

I would like to thank Angelos Charalambidis for his immensely helpful supervision and guidance throughout the whole period of 6 months that I was present in NCSR.

I would also like to thank Professor Panos Randogiannis for being a major influence in my current research interests through the undergraduate courses ”Theory of Computation” and ”Principles of Programming Languages”, where I was introduced to a much more declarative way of programming and started appreciating mathematical sensibility in computing.

CONTENTS

LIST OF FIGURES

PROLOGUE

This bachelor thesis is a continuation of my internship at the National Centre for Scientific Research ”Demokritos”, particularly in the Software and Knowledge Engineering Laboratory (SKEL).

The main task I was assigned was the implementation of a framework for robot programming using a dataflow approach. During that internship, I came to realize that my work could be easily generalized to cover a much broader application area than just robot software.

The name of the framework stems from the ancient Greek Titaness Rhea(*Ρέα*), daughter of earth goddess Gaia(*Γαία*) and sky god Uranus(*Ουρανός*) and etymologically derives from the verb *ρέω*(to flow).

1. INTRODUCTION

1.1 Main concept

My 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) that the ”intelligent” underlying system can automatically realize. Therefore, the programmer will be able to gain good performance and utilization of the available computational resources, while at the same time reducing development time and cost and maintaining a much cleaner and easier-to-refactor software system.

1.2 Motivation

1.2.1 Declarative languages

Software is becoming increasingly more complex each year, as computing capabilities are strengthened and user needs become more and more demanding. Thus the need for higher abstraction becomes mandatory, 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, which specify what needs to be done and how to do it, and the declarative ones, which only specify 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. My 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, 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. convertion 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 tasks and is usually the source of bugs. Thus, having a dataflow execution model will nullify the need for such a translation.

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 was destined to be examined in more depth. As we will discuss in the *Related Work* chapter, there are many recent frameworks that became popular for the scalability due to the fact that they rely on a dataflow approach.

1.3 Outline

There are nine chapters which compose this thesis: *Introduction, Background, Approach, Implementation, Applications, Related Work, Future Work, Conclusions*.

Background introduces the reader to basic background knowledge, necessary for complete understanding.

Approach presents the main characteristics of my approach.

Implementation gives a more detailed specification of the framework.

Applications present some use-cases, ranging from general mathematical problems to real-life robot scenarios.

Related Work discusses relevant concepts and technologies, which influenced major decisions concerning the design and implementation of the framework.

Future Work suggests some interesting topics for future research, whose embedding in the framework is meaningful.

Conclusions sums up.

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.

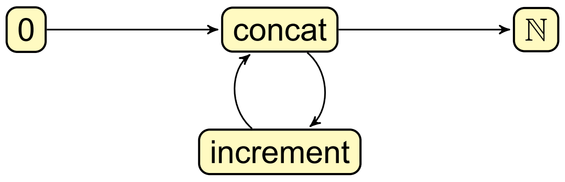


Figure 1: Natural numbers

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. The communication can either be an in-memory data storage or even a TCP connection across the network. Its great flexibility and composability make 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 model of programming 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. It first appeared as a composable library for graphic animations [1], but quickly evolved into a generic paradigm [2, 3, 4]. Moreover, extensive research has investigated FRP as a framework for robotics [5, 6].
Although appealing at first, FRP was not appropriate for systems with real-time constraints, due to uncontrollable time- and space- leaks [7]. The solution was a generalization of monads called *arrows* [8], 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 FRP code corresponding to the formula above:

```
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, a new notation was conceived, notably the *arrow notation* [9]:

```
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, which make it an ideal framework for modelling real-time systems, and its concise representation of time-varying values via *signals*.
For a more detailed view of the FRP idiom and all of its variants please see the cited literature.

2.3 Publish-Subscribe model

Publish/Subscribe(PubSub) is a communication paradigm 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 *message broker*. Figure 2 illustrates an abstract representation of the PubSub model.

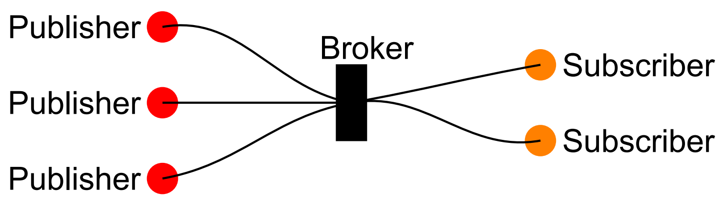


Figure 2: PubSub typical layout

2.4 ROS: Robot Operating System

ROS is an open-source middleware for robot software, which emphasizes large-scale integrative robotics research [10]. It provides a *thin* communication layer between heterogeneous computers, from robots to mainframes and tt 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 nice 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. My approach gives a solution to the aforementioned problem.

2.5 Internet of things

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.

Fortunately, the dataflow model seems to be rather fitting for these scenarios, 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 of *things*.

2.6 The Strategy design pattern

In software engineering, and especially in object-oriented programming(OOP), a *design pattern* is a general repeatable solution to a commonly occurring problem in software design [11].

One such solution, the *Strategy* design pattern is used when a particular algorithm can be implemented by a variety of behaviours/classes [11]. In such a case, a good idea is to isolate the algorithm in a separate *interface* and allow the system to select the appropriate instantiating classes at runtime.

Figure 3 illustrates the basic UML diagram of the strategy design pattern.

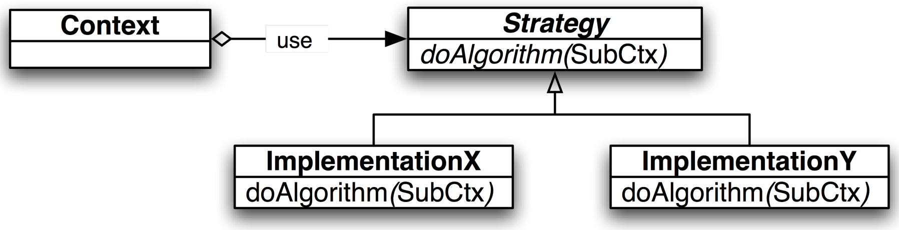


Figure 3: Strategy design pattern

3. APPROACH

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

3.1 Reactive

The system is *reactive*, as close as possible to the definition of the Reactive Manifesto [12].

The system is *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 is *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 textitisolation. 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 is *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 is *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.

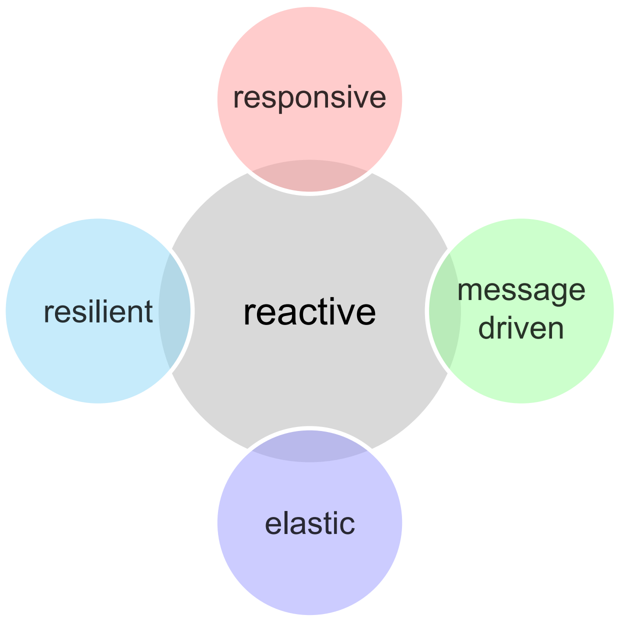


Figure 4: Reactive properties

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 Implementation chapter), which could interpret it using the Java 8 Streams library or even compile into CUDA code for execution on a GPU.

Figure 5 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 lab's main cluster.

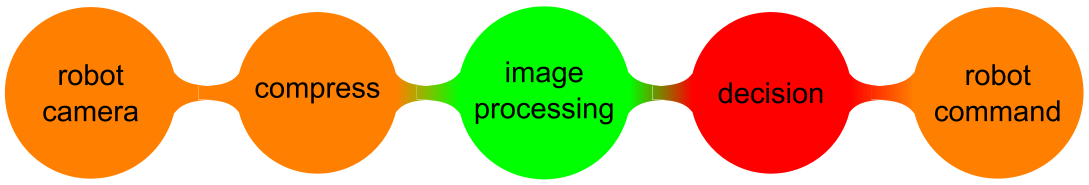


Figure 5: Heterogeneity pipeline

3.3 Extensible

While doing my internship at NCSR ”Demokritos”, I realized that the goal I was pursuing was impossible to reach closure in a 6-month period by a humble undergraduate student. Therefore, I decided that everything should be implemented in a way that will allow future contribution by me or other researchers/developers.

With that concept in mind, I started generalizing and abstracting away everything I had done so far with the hope that the framework will raise attention later on. I can now say I am satisfied with the level of abstraction the core system has reached and I 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 or runtime-specific requirements. It is designed as a unifying conceptual base for further refined extensions and careful consideration was taken not to restrict in any aspect, architectural or not. The above was achieved by making many parts of the core system pluggable, which allows 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 what are its semantics.

4. IMPLEMENTATION

This section examines the major characteristics of the framework's implementation.

4.1 Notifications

Every value passed through the framework's *streams* is wrapped inside a *Notification* object, which discriminated 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).

4.2 The Reactive Streams Standard

In order to make the framework easy to integrate with other stream and/or dataflow technologies, I decided that every input/output node (i.e. publisher/subscriber in the PubSub terminology or source/sink in the Dataflow terminology) should implement the interfaces that the Reactive Streams Standard (RSS) define. 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 [13].

A sink node (output) should implement the Subscriber interface [1], 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[2], which defines a single method *subscribe(Subscriber s)*, 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.

[1]http://www.reactive-streams.org/reactive-streams-1.0.0-javadoc/org/reactivestreams/Subscriber.html
[2]http://www.reactive-streams.org/reactive-streams-1.0.0-javadoc/org/reactivestreams/Publisher.html

4.3 Software structure

The core system[3] is organized in the following top-level packages:

org.rhea_core.internal	all internal functionalities such as graphs, expressions and notifications
org.rhea_core.evaluation	everything associated with the evaluation of the constructed dataflow graph
org.rhea_core.distribution	everything associated with distributing the evaluation across the available computational resources
org.rhea_core.optimization	includes some built-in optimizers for adjusting the granularity of the graph
org.rhea_core.io	defines the interfaces that sources/sinks should implement
org.rhea_core.util	helpful utilities needed throughout the project

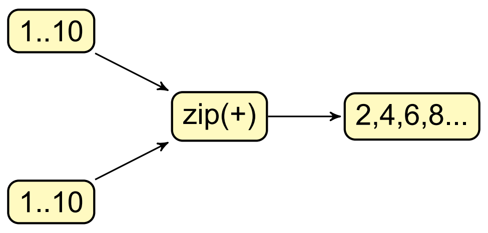
4.4 Internal representation

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-theory algorithms[4]. The main class representing the internal dataflow graph is *FlowGraph*, which is located in the *org.rhea_core.internal* package.

4.5 Stream variables

The construction of the aforementioned internal graph is always implicit, through a rich set of stream operators. The only data type handled by the programmer is *org.rhea_core.Stream*, which contains the parametric type of its single output. Each *Stream* object contains internally a *FlowGraph*, which is only to be accessed and manipulated by the internal module, evaluation strategies and optimizers.

Figure 6 illustrates the dataflow graph(left) produced by the framework code(right) using the *Stream* data type.



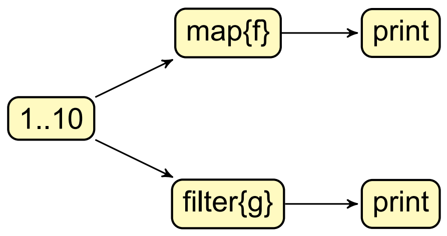
```
Stream.zip(
      Stream.range(1, 10),
      Stream.range(1, 10),
      (x, y) -> x + y);
```

Figure 6: Simple stream example

These variables can be used, preferably together with their parametric type, and reused in different parts of the graph. This is necessary, for instance, when you wish to split a

[3]https://github.com/rhea-flow/rhea-core
[4]http://jgrapht.org/

node's output to different inputs. Figure 7 shows such an example.



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

Figure 7: Split example

4.6 Stream operators

This section displays the set of primitive operators, from which all the available stream operators are derived. Type information is not shown for the sake of readability, but all operations are type-safe.

In the marble diagrams on the right, circles represent *onNext* notifications, green bars *onComplete* and red x signals an *onError*.

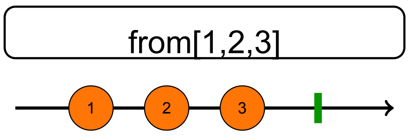
4.6.1 Creation

The following operators act as source nodes in the dataflow graph.

from (Iterable **i**)

inputs: none

output: **i** as a stream



fromSource (Source **s**)

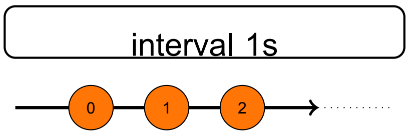
inputs: none

output: the values emitted by **s**

interval (TimeInterval **t**)

inputs: none

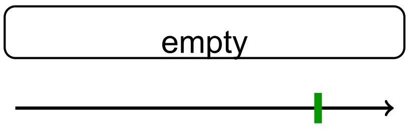
output: the natural numbers emitted every **t**



empty

inputs: none

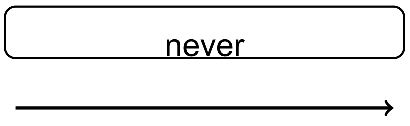
output: an empty stream



never

inputs: none

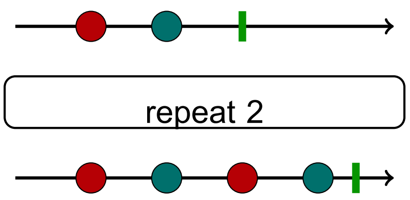
output: a stream that emits no notification



repeat (int **n**)

inputs: single

output: repeats the values of the input stream **n** times, or infinitely if **n** < 0



defer (**f** : $() \to$ *Stream*)

inputs: none

output: the stream generated by the given stream factory **f**

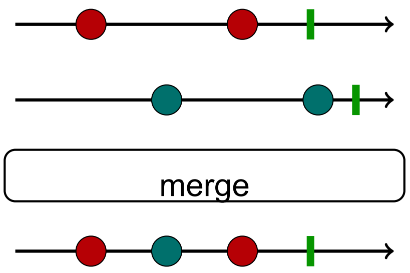
4.6.2 Combining

The following operators combine multiple streams to produce another one.

merge

inputs: many

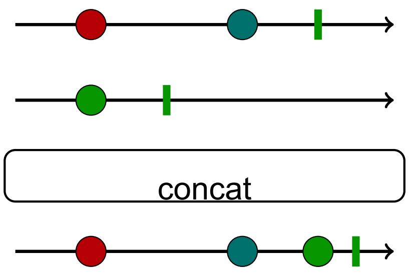
output: merges the input streams and completes as soon as any of them completes



concat

inputs: many

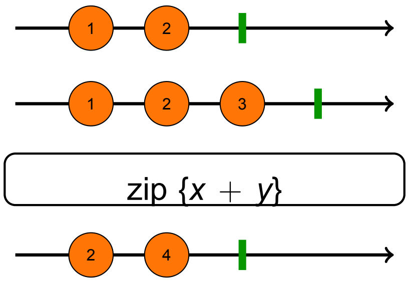
output: concatenates the input streams in the given order



zip (**f** : $A1...An \to B$)

inputs: many

output: zips the input streams with the given function **f**



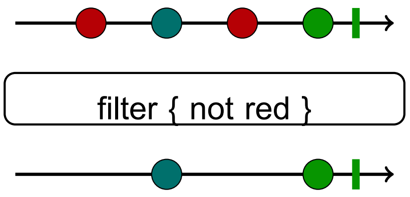
4.6.3 Filtering

The following operators filter the values emitted by another stream.

filter ($\mathbf{f} : A \rightarrow Boolean$)

inputs: single

output: emits only values i of the input stream, where $f(i) = true$



filterMap (**f**: A $\rightarrow$ B, **g**: B $\rightarrow$ Boolean)

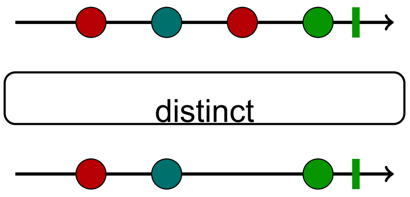
inputs: single

output: emits only values $f(i)$ of the input stream the, where $(f \circ g)(i) = true$

distinct

inputs: single

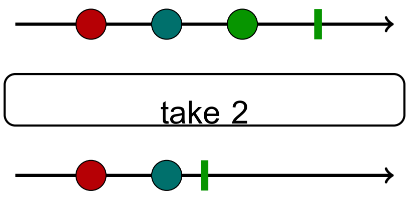
output: removes all duplicate values of the input stream



take (int **n**)

inputs: single

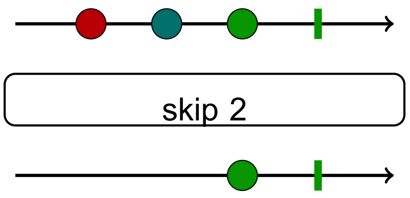
output: extracts the first (last) **n** values of the input stream, if $\mathbf{n} > 0 (< 0)$



skip (int **n**)

inputs: single

output: skips the first (last) **n** values of the input stream, if $\mathbf{n} > 0 (< 0)$



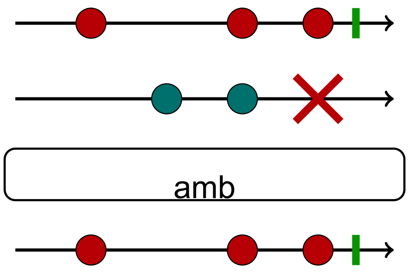
4.6.4 Conditional

The following operators behave depending on some conditions on their input stream

amb

inputs: many

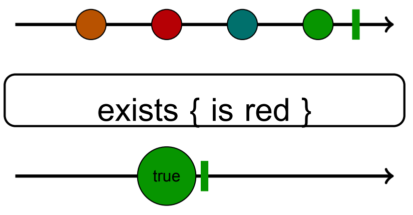
output: emits the values of one its input streams, whichever emits a value or terminates first



exists (**f** : $A \rightarrow Boolean$)

inputs: single

output: returns True, if the input stream contains a value i, where $f(i) = true$ and False otherwise



takeUntil (Stream **s**)

inputs: single

output: extracts values of the input stream, until stream **s** emits a value

skipUntil (Stream **s**)

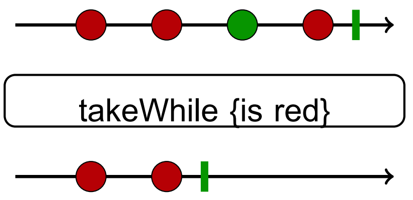
inputs: single

output: skips values of the input stream, until stream **s** emits a value

takeWhile (**f** : $A \rightarrow Boolean$)

inputs: single

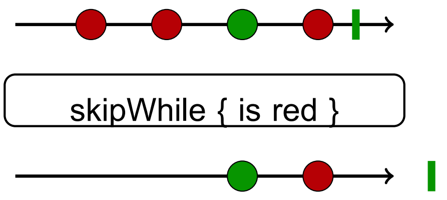
output: extracts values of the input stream until a value i is emitted, where $f(i) = false$



skipWhile (**f** : $A \rightarrow Boolean$)

inputs: single

output: skips values of the input stream until a value i is emitted, where $f(i) = false$



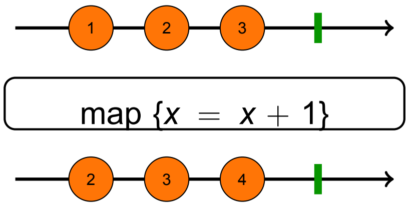
4.6.5 Transformational

The following operators transform their input stream

map (**f** : $A \rightarrow B$)

inputs: single

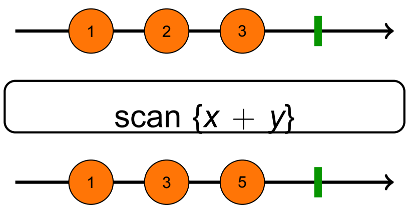
output: transforms the input stream by applying function **f** to every value emitted



scan (B seed, **f** : $B \times A \to B$)

inputs: single

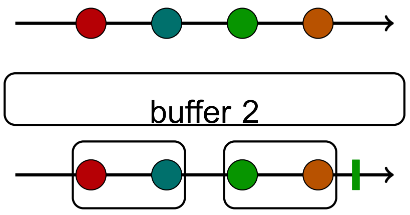
output: transforms the input stream by sequentially applying function **f** to every value emitted and emitting each result along the way



buffer (int **n**)

inputs: single

output: packs together every **n** values of the input stream into a single List item



buffer (TimeInterval **t**)

inputs: single

output: packs together values of the input stream emitted every **t** into a single List item

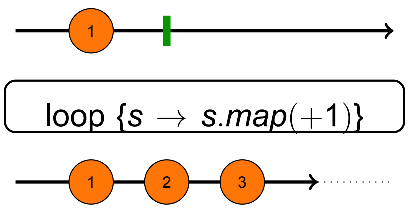
4.6.6 Feedback

This operator enables cycles in the dataflow graph.

loop (**f** : *Stream* $\to$ *Stream*)

inputs: single

output: attaches a sub-graph to the input stream's output, whose result act as feedback to the attachment point



4.6.7 Error-handling

The following operators are a mean to handle errors.

onErrorResume (Stream **s**)

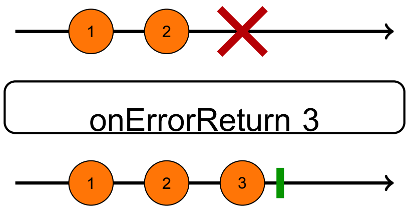
inputs: single

output: mirrors the input stream, but instead of emitting an *onError* Notification when an error occurs, continues emitting values of the given stream **s**

onErrorReturn (**f** : *Throwable* $\rightarrow$ *A*)

inputs: single

output: mirrors the input stream, but instead of emitting an *onError* Notification when an error e occurs, emits the value $f(e)$ followed by a *onComplete* Notification



retry (int **n**)

inputs: single

output: mirrors the input stream, but instead of emitting an *onError* Notification when an error occurs, resubscribes to it **n** times if $n > 0$, infinitely otherwise

4.6.8 Backpressure

The following operators specify how a node behaves when the requests are too intense to handle, computationally or memory-wise. *Backpressure* is the mechanism that handles fast publishers that interact with slow subscribers.

onBackpressureBuffer

inputs: single

output: buffers values that cannot be handled by the subscriber to emit them later on

onBackpressureDrop

inputs: single

output: drops values that cannot be handled by the subscriber, instead of emitting them

onBackpressureLatest

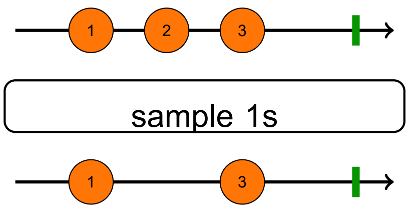
inputs: single

output: drops values that cannot be handled by the subscriber and always buffers the last one, instead of emitting them

sample (TimeInterval **t**)

inputs: single

output: emits only the most-recent emitted value from the input stream within intervals of **t**



timeout (TimeInteval **t**)

inputs: single

output: mirros the input stream, but emits *onError* if there is no emission within time windows of **t**

4.6.9 Utility

The following operators provide some helpful utilities.

doOnNext (Action **a**)

inputs: single

output: execute Action **a** whenever onNext(Complete/Error) is called

cache

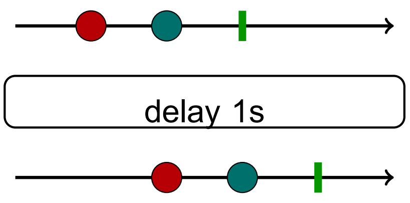
inputs: single

output: caches values emitted by the input stream for future subscribers

delay (TimeInterval **t**)

inputs: single

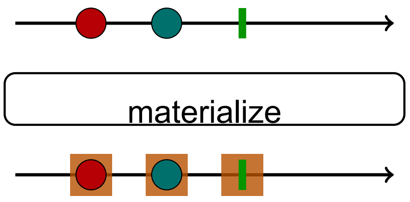
output: emits the values of the input stream shifted forward in time by **t**



materialize

inputs: single

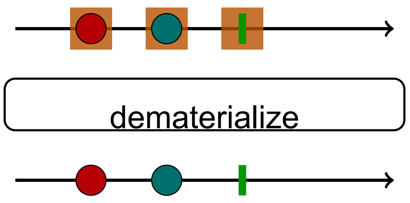
output: wraps all values of input stream as Notifications



dematerialize

inputs: single

output: reverses the effect of **materialize**



All other operators can be produced by combining the above primitive ones (e.g. *flatMap* $\equiv$ *map* $\circ$ *merge*).

4.7 Evaluation

Every primitive operator corresponds to an expression implementing the *Transformer* interface, defined in the *org.rhea_core.internal.expressions* package.

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.

In order to evaluate a constructed dataflow graph the strategy design pattern is used, therefore a class implementing the *EvaluationStrategy* interface, found in the *org.rhea_core.evaluatio* package, needs to be provided. An *EvaluationStrategy* just takes the *Stream* variable and its corresponding *Output* and executes it, however desired.

The strategies I have implemented so far follow:

RxJavaEvaluationStrategy[5]

Uses rxjava[6], which is a famous and well-maintained library for asynchronous programming using the *Observable* type, which is very close, semantically, to my *Stream* variable.

RosEvaluationStrategy[7]

Integrates the *ROS* middleware into the framework, by providing the *RosTopic* class, which implements the *AbstractTopic* interface defined in the *org.rhea_core.io* package. 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).

[5]https://github.com/rhea-flow/rx-eval
[6]https://github.com/ReactiveX/RxJava
[7]https://github.com/rhea-flow/ros-eval

MqttEvaluationStrategy[8]

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

4.8 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, the strategy design pattern is used again, specifically the *DistributionStrategy* interface, which is defined in the *org.rhea_core.internal.distribution* package. Its responsibility is to take the whole initial graph that we need to evaluate and, after adjusting its granularity (i.e. size) to fit the available resources (see *Optimization* section), partition it across all computational resources, maybe using different evaluation strategies.

4.8.1 Hazelcast

Due to the RSS being in its infant stage, no working implementation exists for superimposing a network protocol onto it (e.g. RSS over TCP). For this reason, I relied upon the open-source *Hazelcast* library[9] to discover and manage multiple machines and used its internal decentralized PubSub model to communicate intermediate results across the network. Figure 8 illustrates a dataflow graph on the left and the same graph partitioned over several machines on the right.

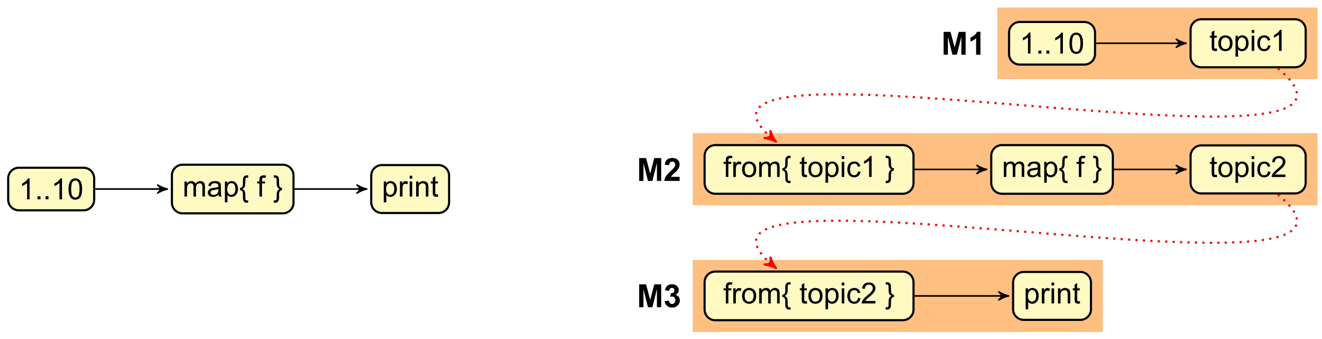


Figure 8: Partitioning

4.8.2 Machine configuration

According to the distribution strategy being used, the available machines will require a certain initial configuration. For the *Hazelcast* case, a little piece of setup code needs to be executed on every member of the cluster, which is together with the main *Strategy* class. Moreover, helpful information can also be added at this step, such as number of

[8]https://github.com/rhea-flow/mqtt-eval
[9]http://hazelcast.org/

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.

4.9 Optimization

This section describes three stages of optimization the dataflow graph goes through before being evaluated. To aid extensibility the strategy design pattern is again used, whose corresponding interface *OptimizationStrategy* resides in the *org.rhea_core.optimization* package. Figure 9 illustrates the optimization stages.

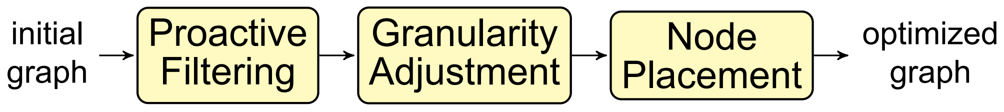


Figure 9: Optimization stages

4.9.1 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 across-machine communication overhead. The figures below show the corresponding graph transformations.

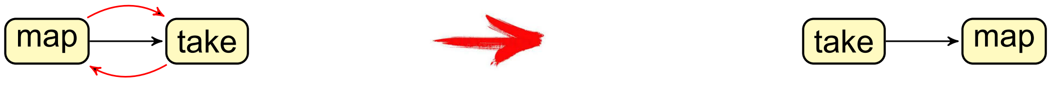


Figure 10: Take/skip/distinct before map

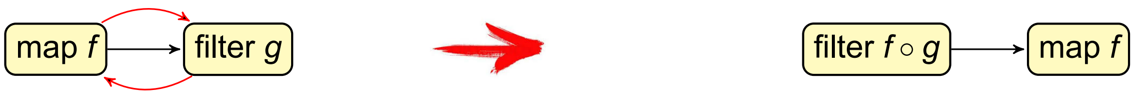


Figure 11: Filter before map

4.9.2 Granularity adjustment

Different nodes of the dataflow graph will be executed on a separate thread/process. The fact that graphs can grow very big 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 the number of available threads amongst all machines.

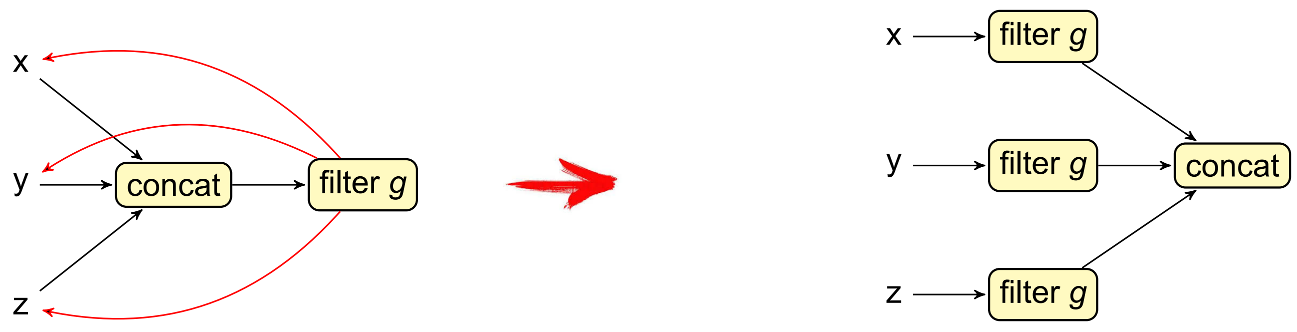


Figure 12: Filter/distinct before concat/merge

To achieve this, 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).

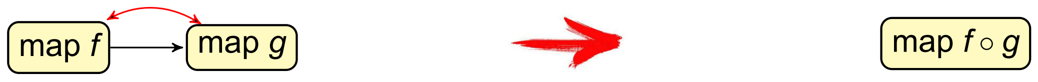


Figure 13: Merge maps

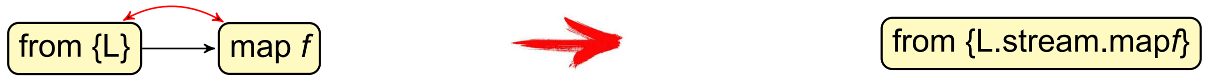


Figure 14: Embed map in creation

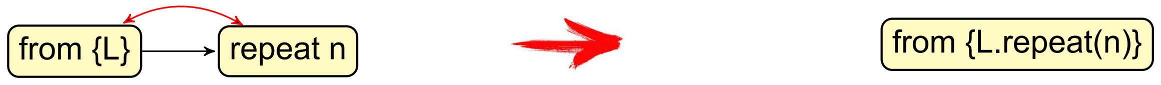


Figure 15: Embed repeat in creation

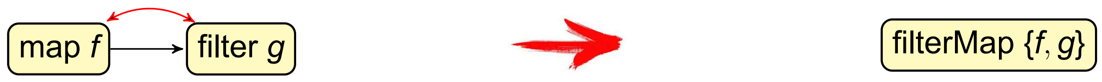


Figure 16: Combine map with filter

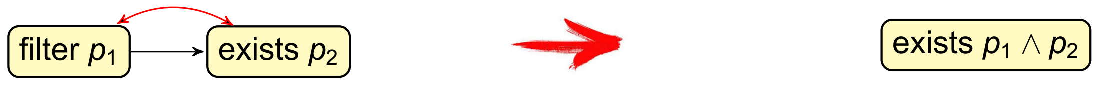


Figure 17: Combine filter with exists

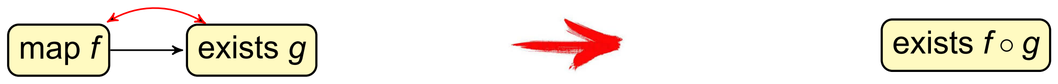


Figure 18: Combine map with exists

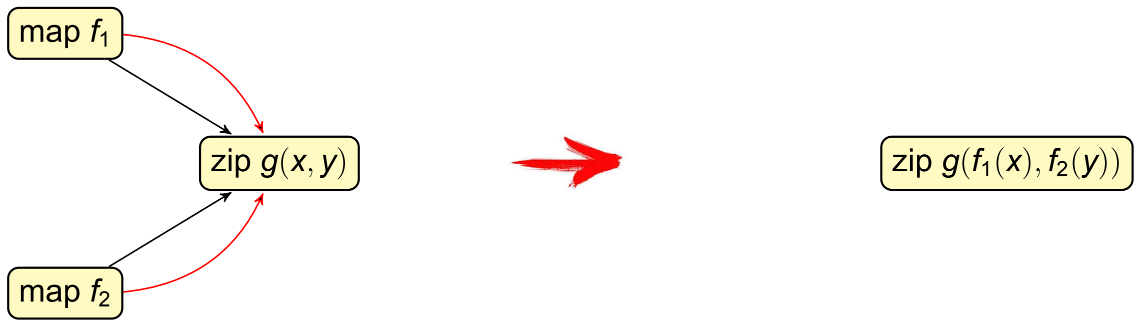


Figure 19: Combine map with zip

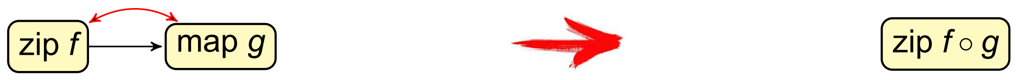


Figure 20: Combine zip with map

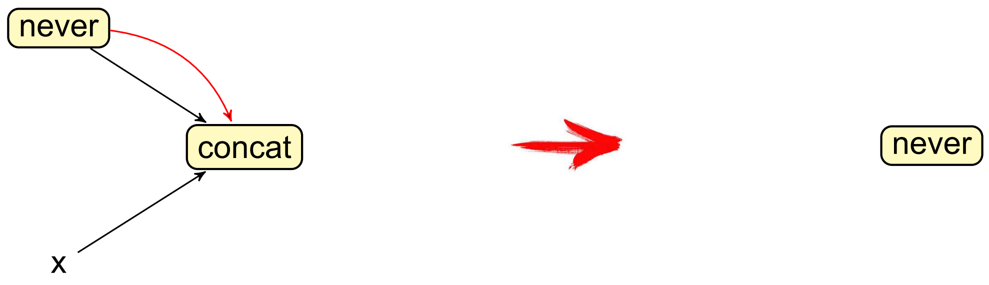


Figure 21: Meaningless nevers

4.9.3 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.

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 22.

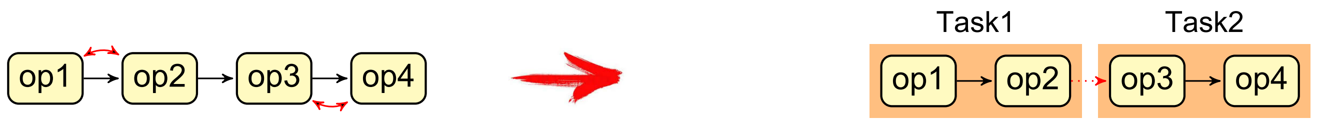


Figure 22: Task fusion

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, the strategy design pattern is again used, namely the *NetworkProfileStrategy* that is defined in the *org.rhea_core.distribution* package. Its responsibility is to calculate a network distance between each pair of available machines, which is 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.

4.10 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, define in the *org.rhea_core.serialization* package. The byte representation of the objects is parametric for maximum flexibility.

A default *Serializer* is provided with the core system, which can serialize every class implementing the *Serializable* interface. In addition to that, the JsonIO library[10] is used which allows serialization of many types of classes, but still does not cover every possible one. Figure 10 depicts the serialization process in more detail.

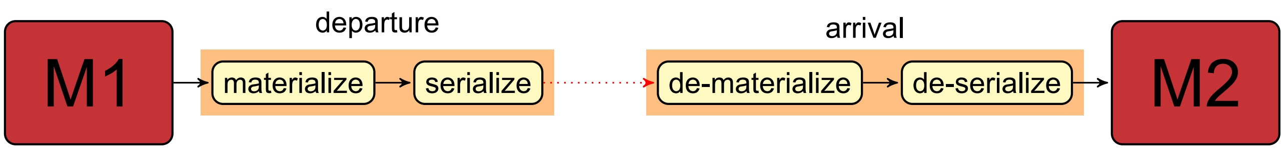


Figure 23: Serialization process

[10]https://github.com/jdereg/json-io

5. APPLICATIONS

5.1 Hamming numbers

5.2 Camera surveilance

5.3 Robot control panel

5.4 Robot hospital guide

6. RELATED WORK

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

6.1 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. Its easy to understand and maintain structure and declarative approach to programming, while not losing expressibility, has attracted many frameworks to utilize it.

6.1.1 GoogleDataflow

Google recently released the *GoogleDataflow* framework[11], which is an evolution of *FlumeJava*[**?**], which in turn is a successor of the famous *MapReduce* framework[**?**].

MapReduce was a very simple model that allowed automatic concrrency/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.

This gave birth to *FlumeJava*, which is a generalization of the *MapReduce* framework that allows more expressive pipelines of more primitive operations like *MapReduce*.

Although *FlumeJava* was more attractive due to its expressibility, still the pipeline constructed could not formulate all problems that are needed in some 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.

And this is how *GoogleDataflow* came to exist, offering a fully generic dataflow framework integrated with many other closely-related technologies from Google, like Cloud Storage[12], Cloud PubSub[13], Cloud Datastore[14], Cloud Bigtable[15] and BigQuery[16].

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-

[11]https://cloud.google.com/dataflow/
[12]https://cloud.google.com/storage/
[13]https://cloud.google.com/pubsub/
[14]https://cloud.google.com/datastore/
[15]https://cloud.google.com/bigtable/
[16]https://cloud.google.com/bigquery/

driven execution.

6.1.2 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 my 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).

Though 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. Figure ? illustrates a neural network as a dataflow graph.

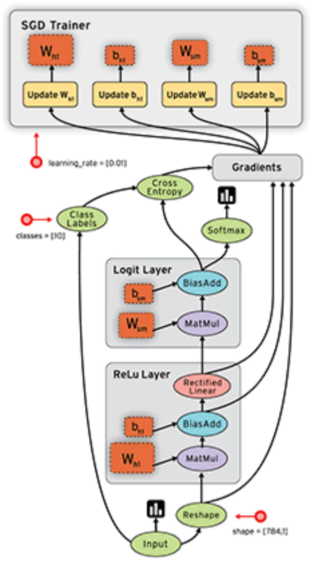


Figure 24: TensorFlow graph

6.1.3 Akka

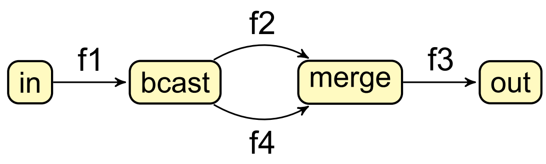
Definitely one the most mature frameworks for distribution targeting the JVM, *Akka*[18] is a toolkit and runtime for highly concurrent, distributed and resilient message-driven applications. It is also one of the founders of the Reactive Streams[13] initiative.

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

[17]https://www.tensorflow.org/
[18]http://akka.io/

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 my framework solved, which is the inappropriate nature of callbacks for complex scenarios, *Akka* developers also felt the same necessity for a more flexible and composable programming model, so they started the AkkaStreams library[19] which provides a convenient API for stream processing and also dataflow graph construction with an interesting DSL. Figure ? demonstrates a dataflow graph on the left, generated by the DSL code on the right.



```
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 25: Akka DSL

6.1.4 Spark

A very well-known and well-adapted framework for scalable large-data processing is Apache's *Spark*[20]. It was developed to overcome the shortcomings of the *MapReduce* framework mentioned above, providing a much more efficient and flexible runtime.

It follows the same general approach as *RHEA*, meaning that it is completely generic and encourages domain-specific libraries to be built upon it. For instance, *MLib*[21] is a library for machine learning and *GraphX*[22] is a library for iterative graph algorithms, both sitting onto *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 */
```

[19]http://doc.akka.io/docs/akka-stream-and-http-experimental/1.0-M2/scala.html
[20]http://spark.apache.org/
[21]http://spark.apache.org/mllib/
[22]http://spark.apache.org/graphx/

6.1.5 Naiad

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[**?**] 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 treaming 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 I introduced in the first chapter, 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.

Figure ? shows a Naiad application that support real-time queries on continually updated data.

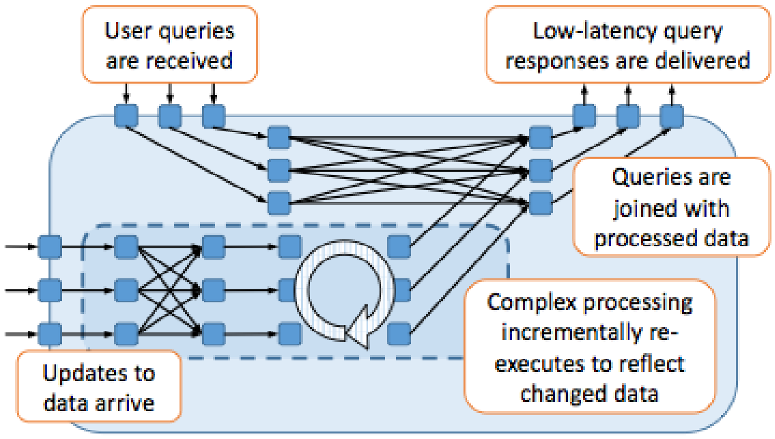

User queries are received
Low-latency query responses are delivered
Queries are joined with processed data
Complex processing incrementally re-executes to reflect changed data
Updates to data arrive

Figure 26: Naiad application

6.1.6 dispel4py

A less-known framework for Python is *dispel4py*[23]. It provides the ability to describe abstract workflows for distributed data-intensive applications.

Similar to my *EvaluationStrategy* concept, it allows different mappings to enactment sys-

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

tems, such as MPI[24] and Apache Storm[25].

Its main disadvantages are that only has a Python API and that it only allows low-level specification of the graph's nodes, through the definition of *Processing Elements*. Therefore, it is impossible to compose larger graphs from simpler ones easily and the source code becomes chaotic and difficult to maintain.

6.2 Robotics

6.2.1 Flowstone

6.2.2 Yampa

6.2.3 roshask

6.3 Internet of Things

6.3.1 NodeRed

6.3.2 NoFlo

[24] http://www.mcs.anl.gov/project/mpich-high-performance-portable-implementation-mpi
[25] http://storm.apache.org/

7. FUTURE WORK

7.1 More evaluation strategies

7.2 Dynamic reconfiguration

7.3 Advanced network profiling

7.4 DSLs for framework extension

7.5 Advanced fault-tolerance

7.6 Integration with other technologies

7.7 Visual language

7.8 Stream reasoning

8. CONCLUSIONS

ABBREVIATIONS, INITIALS AND ACRONYMS

A table of all abbrevations used throughout the thesis follows.

FRP	Functional Reactive Programming
JVM	Java Virtual Machine
NCSR	National Centre for Scientific Research
ROS	Robot Operating System
IoT	Internet of Things
CPU	Central Processing Unit
TCP	Transmission Control Protocol
PubSub	Publish/Subscribe
OOP	Object-oriented Programming
UML	Unified Modelling Language
GPU	Graphics Processing Unit
DSL	Domain-specific Language
RSS	Reactive Streams Standard
API	Application Programming Interface
MPI	Message Passsing Inteface

REFERENCES

[1] C. Elliott and P. Hudak, “Functional reactive animation,” in *ACM SIGPLAN Notices*, vol. 32, pp. 263–273, ACM, 1997.

[2] E. Amsden, “A survey of functional reactive programming,” *Unpublished*, 2011.

[3] Z. Wan, W. Taha, and P. Hudak, “Real-time frp,” in *ACM SIGPLAN Notices*, vol. 36, pp. 146–156, ACM, 2001.

[4] C. M. Elliott, “Push-pull functional reactive programming,” in *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pp. 25–36, ACM, 2009.

[5] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson, “Arrows, robots, and functional reactive programming,” in *Advanced Functional Programming*, pp. 159–187, Springer, 2003.

[6] J. Peterson, P. Hudak, and C. Elliott, “Lambda in motion: Controlling robots with haskell,” in *Practical Aspects of Declarative Languages*, pp. 91–105, Springer, 1999.

[7] Z. Wan, W. Taha, and P. Hudak, “Event-driven frp,” in *Practical Aspects of Declarative Languages*, pp. 155–172, Springer, 2002.

[8] J. Hughes, “Generalising monads to arrows,” *Science of Computer Programming*, vol. 37, no. 1–3, pp. 67 – 111, 2000.

[9] R. Paterson, “A new notation for arrows,” *SIGPLAN Not.*, vol. 36, pp. 229–240, Oct. 2001.

[10] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, “Ros: an open-source robot operating system,” in *ICRA workshop on open source software*, vol. 3, p. 5, 2009.

[11] E. Gamma, *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[12] B. Jonas, D. Farley, R. Kuhn, and M. Thompson, “Reactive manifesto.” http://www.reactivemanifesto.org/, 2014.

[13] “Reactive streams standard.” http://www.reactive-streams.org/, 2015.