VRIJE
UNIVERSITEIT
BRUSSEL

Graduation thesis submitted in partial fulfilment of the requirements for the degree of de Ingenieurswetenschappen: Applied informatics

DISTRIBUTING FIRST-CLASS REACTORS IN A CLUSTER ENVIRONMENT

Hans Van der Ougstraete

June 2024

Promotors: prof. dr. Wolfgang De Meuter, prof. dr. Joeri De Koster
Supervisor: Bjarno Oeyen

sciences and bioengineering sciences

VRIJE
UNIVERSITEIT
BRUSSEL

Proefschrift ingediend met het oog op het behalen van de graad van Master of Science in de Ingenieurswetenschappen: Toegepaste informatica

DISTRIBUTING FIRST-CLASS REACTORS IN A CLUSTER ENVIRONMENT

Hans Van der Ougstraete

Juni 2024

Promotors: prof. dr. Wolfgang De Meuter, prof. dr. Joeri De Koster
Supervisor: Bjarno Oeyen

wetenschappen en bio-ingenieurswetenschappen

Abstract

Reactive Programming (RP) is a powerful paradigm that models data flow as streams of events, enabling systems to react to changes and events efficiently. This document explores the principles of reactive programming, its applications, and the development of a framework for distributing first-class reactive programs in a cluster environment. We use the novel reactive language Haai, that due to its independent form (not relying on a base language) addresses key challenges in reactive programming by preventing the base language to intervene with the reactivity of the program. In this work we introduce the Haai virtual machine, implemented in Elixir, leveraging the Erlang Virtual Machine (BEAM) to enable the distribution of reactive programs across a cluster.

The evaluation of the cluster environment demonstrates its flexibility, scalability, and fault tolerance, allowing for the simulation of large cluster environments. A practical use case, the distributed reactive melody generator, showcases the framework's application in transforming streams of data into synthesized sound by sending network messages to a Supercollider sound server. This use case highlights the potential of reactive programming in creating innovative and dynamic systems that respond to real-time events. Overall, this work contributes to the advancement of reactive programming by providing a robust framework for distributed reactive programming and demonstrating its practical applications in a musical context.

Contents

Chapter 1

Introduction

Reactive Programming (RP) represents a broad programming paradigm centered around responding to changes and events within a system. This paradigm involves modeling data flow as streams of events and utilizing declarative and composable abstractions to manage asynchronous and event-driven scenarios. The canonical reactive program model, often exemplified by spreadsheets, is based on time-varying values and the propagation of change.

The choice of reactive programming is motivated by several key advantages. Firstly, RP enables systems to respond quickly to changes in data and events, making it ideal for real-time applications. Secondly, reactive systems can scale up or down based on demand, allowing for efficient resource utilization and adaptability to varying loads. Thirdly, the event-driven nature of reactive programming facilitates communication between components through messages, promoting modular design with a minimal dependency between modules (loose coupling). Finally, reactive programming encourages the use of declarative constructs, which can lead to more readable and maintainable code. Composable abstractions allow for the creation of complex behaviors from simple building blocks.

This document delves into the intricacies of reactive programming, its applications, and the development of a framework to experiment with the distribution of first-class reactors in a cluster environment. We also present a practical use case involving a sound-generating instrument that reacts to numerical input and delivers values to a sound-generating server.

The canonical reactive program model (Bainomugisha et al., 2013) is based on time-varying values and propagation of change. Spreadsheets offer a good example to illustrate this. When a cell containing a formula that includes other cells it is dependent on the cells that are used in that formula. As an example lets take a cell A3 that contains the formula "=A1*A2" whenever the value changes in the cells A1 or A2, A3 will automatically update. We call each spreadsheet cell a time-varying value or signal because there concrete value changes over time due to events. In this example, an event could be when a user changes the value A1 or A2 using the gui of the spreadsheet program.

The two fundamental abstractions defined by RP to represent time-varying values are behaviors and events. Behaviors have a value at any moment (continuous) and events have a value at specific moments in time (discrete). Listing 1.1 shows the spreadsheet example represented in the RP language Haai (Oeyen et al., 2024). In this example a3 is defined as a reactor that continuously contains the result of a1 * a2. For each time that an event occurs updating a1 or a2, a3 will be automatically updated.

```
(defr a3 (* a1 a2))
```

Listing 1.1: Haai code

The updates to a3 occur in such a way that the value for a3, which could be a dependency to another cell in the spreadsheet, is always correct or updated such that there are no invalid or intermediary values. The rhythm of the propagation of change for reactor a3 depends on the speed that events occur. Figure 1.1 shows the value for reactor a3 (* a1 a2) over time presenting three updates t=0, t=1 and t=2.

Reactor a3 is an example of a small reactive program. Multiple reactors can be combined in a bigger reactive program resulting in a reactor composed out of reactors. If our system is constructed solely from reactors we call this a purely reactive program since reactors serve as the only building blocks.

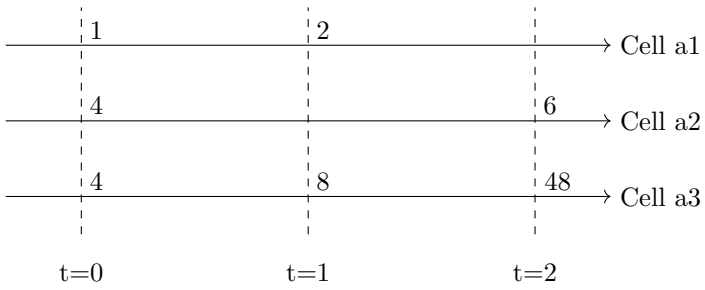

Figure 1.1: Timeline of cells a1, a2 and a3 over time.

1.1 Problem Statement

To our best knowledge we did not find a framework to experiment with the distribution of first-class reactors. We developed a framework to study the distribution of first-class reactors in a cluster environment and create a declarative specification to deploy a distributed reactive program into different cluster environments. To show the working state of the framework we deployed a sound generating instrument that reacts on input and delivers values to be send to a sound generating server.

1.2 Structure of the Thesis

We first look at non-reactive programming and distributed reactive programming in section 2. Chapter 3 explains the distribution of the runtime and the tools used to build it. After the evaluation of the distribution framework and the musical use case of the distributed reactive melody generator in section 4 we conclude.

Chapter 2

Reactive Programming

Reactive programming (RP) and reactive streams (RS) are terms that are often used interchangeably. Both adhere to the same canonical model. For as much as they have in common there are two fundamental differences, the way dependencies between signals are established and how updates to signals are orchestrated.

Signal Dependencies

The way signal dependencies are established in the source code differs in the following way. In RP languages these dependencies are automatically taken care of by the language. Programmers implement there desired functionality in RP similar as they would in a non-reactive language. In RS the dependencies are explicitly programmed by the programmer by making use of operators provided by the RS library. The programmer creates a linear application of those operators often called a pipe as shown in listing 2.1, where an observable existing of the numbers 1 to 10 will be the source of the pipe. The second or last operator in the pipe is Sum, who will produce one value to be printed on screen.

Signal Orchestration

The operators used in RS define independently for each signal how these will be updated. In contrary to RP where it is again the system that updates signals in a global way. As can be seen in listing 1.1, no special code defines how or in what order the values a1 and a2 need to be updated. Both approaches can fail and result in inconsistencies in the values, if for example the RP language updates signals in the wrong order, dependent signals end up having the wrong value, this is called a glitch. On the other hand in RS, if the programmer does not exactly know what an operator does and uses it in a pipe or collection of operators, inconsistencies might arise and also produce a glitch.

The responsibility to handle signal dependencies and orchestration correct is at the system side for RP and in the developers hand for RS.

```
rx.of(1,2,3,4,5,6,7,8,9,10).pipe(
        operators.filter(lambda i: i %2 == 0),
        operators.sum()
).subscribe(lambda x: print("Value␣is␣{0}".format(x)))
```

Listing 2.1: Python, RxPy library

2.1 State-of-the-Art

2.1.1 non-Distributed Reactive Programming

Reactive programming has many applications, here we will show three selected examples. Fran (Elliott and Hudak, 1997) the first reactive programming language, originated to compose interactive multimedia animations based on behaviors and events. Listing 2.2 show a small fran code example that animates a bitmap file with the moveXY behavior given the argument wiggle (a value that changes over time form 0 to +1 to -1 back to 0 and so on) for x and 0 for y.

```
leftRightCharlotte = moveXY wiggle 0 charlotte
charlotte = importBitmap "../Media/charlotte.bmp"
```

Listing 2.2: Fran, animation (Elliott, 1998)

ReactiFi (Sterz et al., 2021) a high-level reactive programming language to program Wi-Fi chips on mobile consumer devices without expert knowledge of Wi-Fi chips. In listing 2.3 we show part of the code for adaptive file sharing programmed on the Wi-Fi chip. A ReactiFi program exists out of reactives, like monitor, frames en count in the example. Reactives are reactive definitions of individual processing steps that are triggered by incoming events. The source are wifi frames gathered in 'monitor' mode. Those frames are filtered for a certain address ADDR and summed in the last step (count) of the example.

```
 val monitor = Source(Monitor)
 val frames  = monitor.filter(frame -> { frame.dst == ADDR })
 val count   = frames.fold({ 0 })((count, frame) -> { count + 1 })
```

Listing 2.3: ReactiFi, Wi-Fi file sharing (Sterz et al., 2021)

The domain-specific language Yampa (Courtney et al., 2003) embedded in Haskell for programming hybrid (mixed discrete-time and continuous-time) systems. n Yampa, a signaling function (SF) is a fundamental concept that represents a continuous-time signal transformer. An SF takes an input signal and produces an output signal over time. Signaling functions are the basic building blocks of Yampa programs and are used to model and manipulate continuous-time behaviors. Listening 2.4 shows such a signal function named fallingBall. This SF describes the behavior of a bouncing ball with a position (p) and velocity (v). When the discrete event 'hitGround' occurs the velocity of the ball reverses to simulate the bounce of the ball.

```
-- The bouncing ball signal function
fallingBall :: SF () Ball
fallingBall = proc () -> do
 rec
   -- Velocity integrates to position
   v <- integral -< gravity
   p <- integral -< v

   -- Event that occurs when the ball hits the ground
   let hitGround = if p <= 0 then Event () else NoEvent

   -- Velocity changes direction on hit
   v <- (arr (\v -> if p <= 0 then -v else v) <<< identity) -< v

 returnA -< (p, v)
```

Listing 2.4: Yampa, bouncing ball (Meisinger, 2021)

Most, if not all, of the reactive programming languages are build on top of a sequential host language. Fran and Yampa are build on Haskell, ReactiFi is build on Scala and C. This allows the reactive programming languages to leverage the existing ecosystems of the host language. For example a host languages can provide functions to be called form the reactive programming language, this is called lifting, the host-functionality is lifted form the host language into the RP language adapted to work with continuous and discrete notions of time. On the other hand, when a system is composed of those two components, the host part and the reactive part, they are prone to the *Reactive Thread Hijacking Problem* and the *Reactive/Imperative Impedance Mismatch* problems (Vonder et al., 2020). Both problems might turn the reactive program nonreactive. In the first case the event-handling logic could block the main thread of execution, preventing other events from being processed on time. Secondly reactive and imperative paradigms have fundamentally different ways of handling state and flow, which can lead to error-prone code. In section 2.2 we will introduce a novel reactive language Haai that by design excludes both problems mentioned above. This will help us in distributing the reactors since both problems will not arise by design.

2.1.2 Distributed Reactive Programming

Many distributed systems can be seen as some kind of reactive system. For example an internet of things application with many connected sensors that feed information into a reactive system. Or a distributed reactive system with strong computational differences for different behaviors that should finish in a bounded time. The distribution can optimize the placement of code with respect to different computational capacities related to the application.

Once working in a distributed setting, typical problems arise like data inconsistencies and node disconnects and crashes. n the RP literature, inconsistencies in the data are called glitches. A glitch is a momentary inconsistency in a time-varying value. For instance, consider a distributed reactive system that monitors the temperature of a network of sensors. Each sensor reports its temperature reading to a central processing unit, which then calculates the average temperature across all sensors. If one sensor temporarily malfunctions and reports an incorrect temperature, this erroneous value can introduce a glitch into the system. The central processing unit may briefly calculate an inaccurate average temperature based on the faulty data, leading to a momentary inconsistency. However, once the sensor corrects its reading or is identified as faulty, the system can recover and resume providing accurate temperature information. Being free of glitches is called glitch-freedom (Margara and Salvaneschi, 2018).

As an example we show REScala. Built on top of the Scala programming language. The framework introduces several core concepts, including signals, events, and signal functions. Signals represent time-varying values that model the state of a system over time, while events are discrete occurrences that trigger changes in signals or other events. Signal functions, also known as signal transformers, take one or more signals as input and produce one or more signals as output, enabling the transformation, combination, or filtering of signals. ReScala provides a flexible and efficient way to build reactive systems, making it well-suited for applications ranging from real-time data processing to interactive user interfaces. Listing 2.5 shows a small example where the program prints the actual time + 1000ms for 5 seconds. The observe method is used to receive and print the value in side the offsetSignal.

```
object ReScalaExample extends App
{
        // Define a signal that represents the current time
        val timeSignal = Signal { System.currentTimeMillis() }

        // Adds a constant value to the time signal
        val offsetSignal = timeSignal.map(_ + 1000)

        // Print the current value of the offset signal
        offsetSignal.observe(println)

        // Keep the application running
        Thread.sleep(5000)
}
```

Listing 2.5: REScala, time + 1000

Regarding to distribution ReScala provides a solution to node disconnects and crashes in distributed reactive programs. REScala does not delegate the responsibility to handle errors to the host language (Scala). REScala has been extended to support recovery after node crashes and cope with unreliable network connections by enhancing the data flow graph for a particular program (Mogk et al., 2018). Listing 2.5 shows a shared calendar application written in REScala. This is a distributed application where one calendar is shared among many users. In order to provide recovery after crashes, signals such as `Var[Date]` are stored in snapshots. This allows for the signal to be reloaded after a crash. In cases where a signal is dependent on dynamic signals in the network of distribution, the REScala program handles errors explicitly. For example the selectedEntries in code listing 2.6 expects a value from another node (selectedWeek.value). The `try catch` block will by default wait for the value but by returning false in the catch block the filter will drop this signal and continue filtering when a 'disconnectedSignal' error is received. By doing so the program does not block and keeps on running reactively.

```
val newEntry = Evt[Entry]()
val automaticEntries: Event[Entry] = App.nationalHolidays()
val allEntries = newEntry || automaticEntries

val selectedDay: Var[Date] = Var(Date.today)
val selectedWeek = Signal { Week.of(selectedDay.value) }

val entrySet: Signal[Set[Entry]] =
 if (distribute) ReplicatedSet("SharedEntries").collect(allEntries)
 else allEntries.fold(Set.empty) { (entries, entry) => entries + entry}

case class Entry(title: Signal[String], date: Signal[Date])

val selectedEntries = Signal {
 entrySet.value.filter { entry =>
        try selectedWeek.value == Week.of(entry.date.value)
        catch { case DisconnectedSignal => false }
 }
}

allEntries.observe(Log.appendEntry)
selectedEntries.observe(
 onValue = Ui.displayEntryList,
 onError = Ui.displayError)
```

Listing 2.6: REScala, shared calendar application, (Mogk et al., 2018)

2.2 Purely reactive programming

In all previous examples the reactive programming language is build on top of a host language that lend some functionality to the reactive program. What we will discuss next is the idea of a purely reactive program. A reactive program constructed by combining only reactive programs that we call reactors.

2.2.1 Reactors

In Haai, a purely reactive computer language, the only construct available for expressing computations is a reactor. A reactor is a sort of blue print (as shown in figure 2.1) that will be used to deploy any number of deployments based on that reactor.

This blue print holds the following information: a name, number of sources, number of sinks and two types of instructions. Deployment-time instructions to be run at deployment time and reaction-time instructions that will loop over and over once the reactor is deployed.

We can write a single reactor or combine multiple reactors in one composed reactor or reactive program. A composed reactor is build solely out of reactors, these can be native or user defined. Every computation in a pure RP program is done by a reactor, some are native to the runtime others are user defined. For example the runtime could provide native reactors for basic arithmetic. All functionality in the purely reactive programming language Haai comes from the language itself, it is not build on top of a general-purpose host language.

The deployment of a reactor reacts on a data stream, that data altered by the reaction is found in the sink of the reactor. The deployment of a reactor is further explained in section

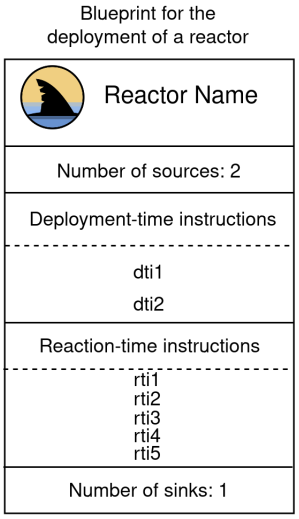

Figure 2.1: Graphical representation of a reactor

2.3.1.

2.2.2 Haai

We make use of the novel purely reactive language Haai. Haai is reactive all-the-way-trough. As such the only construct available for expressing computations is a reactor. A reactor has sources or inputs and sinks or outputs. For each time the sources are updated the reactor will react and produce a new sink value. The number of sources and sinks are variable per reactor definition. In Haai there are no functions, only reactors. This allows us to build reactive programs without functions (Oeyen et al., 2024). Haai is an independent language, it does not work on top of a host-language.

Listing 2.7 represents three user defined reactors: consonance, duration and main. The main reactor is build out of the two user defined reactors consonance and duration and makes no use of native reactors (except for the ones in consonance and duration). Consonance and duration make use of multiple native reactors to perform arithmetic. The arguments to the reactor are the sources and the keyword 'out' defines the sink of the reactor.

```
(defr (consonance f)
        (out (* ci f)))

(defr (duration bpm)
        (def q (/ 60000 bpm))
        (out (* lm q)))

(defr (main f bpm)
        (out (consonance f))
        (out (note_length bpm)))
```

Listing 2.7: Haai code

The reactor in listing 2.7 has two sources (f and bpm) and two sinks defined with the keyword out. To port the Haai code (listing 2.7) into the cluster environment we compile the Haai code into a byte code representation (listing 2.8) that was developed for the Remus virtual machine. "a virtual machine that has been carefully designed for resource constrained computing environments."(Oeyen et al., 2022)

Remus Instruction Set

The Remus instruction set in table 2.1 is the set of low-level operations that the virtual machine can execute. A Haai program or Reactor will after compilation exist out of a combination of instructions. Instructions form the Remus instruction set as listed here under. The Haai virtual machine is an implementation of this instruction set in Elixir. The following example program is build up out of a combination of the Remus instruction set. It is the compilation of the main reactor shown in the code snippet above.

Instruction	**Arguments**	**Description**
I-ALLOCMONO	[reactor]	Allocate memory for the given reactor.
I-LOOKUP	[signal]	Lookup the value for signal
I-SUPPLY	[from, destination, index]	Move a value.
I-REACT	[memory_location]	Apply the reaction found in memory.
I-CONSUME	[memory_location, index]	Position the result of react in run-time-memory.
I-SINK	[rti_location, sink_index]	Position a value from run-time-memory into sink.

Table 2.1: Remus instruction Set (Oeyen et al., 2022)

Bytecode example

In listing 2.8 we show compiled Haai code from listing 2.7. In this bytecode example based on the Remus instruction set we see two user defined reactors named consonance and note_length followed by the main reactor with 2 sources and 2 sinks that represents the actual reactive program that will be deployed in the cluster environment.

```
[
 [:consonance, 1, 1,
        [
                ["I-ALLOCMONO", :multiply]
        ],
        [
                ["I-LOOKUP", :ci],
                ["I-SUPPLY",["%RREF",1],["%DREF",1],1],
                ["I-SUPPLY",["%SRC",1],["%DREF",1],2],
                ["I-REACT",["%DREF",1]],
                ["I-CONSUME",["%DREF",1],1],
                ["I-SINK",["%RREF",5],1]]
        ],

 [:duration, 1, 1,
        [
                ["I-ALLOCMONO", :divide],
                ["I-ALLOCMONO", :multiply]
        ],
        [
                ["I-SUPPLY",60000,["%DREF",1],1],
                ["I-SUPPLY",["%SRC",1],["%DREF",1],2],
                ["I-REACT",["%DREF",1]],
                ["I-LOOKUP",:1m],
                ["I-SUPPLY",["%RREF",4],["%DREF",2],1],
                ["I-CONSUME",["%DREF",1],1],
                ["I-SUPPLY",["%RREF",6],["%DREF",2],2],
                ["I-REACT",["%DREF",2]],
                ["I-CONSUME",["%DREF",2],1],
                ["I-SINK",["%RREF",9],1]]
        ],

 [:main, 2, 2,
        [
                ["I-ALLOCMONO", :consonance],
                ["I-ALLOCMONO", :note_length]
        ],
        [
                ["I-SUPPLY",["%SRC",1],["%DREF",1],1],
                ["I-REACT",["%DREF",1]],
                ["I-SUPPLY",["%SRC",2],["%DREF",2],1],
                ["I-REACT",["%DREF",2]],
                ["I-CONSUME",["%DREF",1],1],
                ["I-SINK",["%RREF",5],1],
                ["I-CONSUME",["%DREF",2],1],
                ["I-SINK",["%RREF",7],2]
        ]
 ]
]
```

Listing 2.8: Remus bytecode as Elixir nested lists

2.3 Haai virtual machine

To run the Haai code (compiled into bytecode) we implement in this thesis a virtual machine for Haai. Implementing a virtual machine (VM) to run a new programming language offers several compelling advantages. A VM provides a layer of abstraction that insulates the language from the underlying hardware, ensuring portability across different platforms. The VM manages the memory and can be used to facilitate the implementation of advanced language features.

The purpose of the Haai virtual machine (Hvm) is to enable the distribution of first-class reactor Haai programs in a cluster environment. The Hvm is developed in Elixir, a programming language know for its powerful features in terms of distribution and concurrency. It leverages the Erlang Virtual Machine (BEAM), which is renowned for its ability to build scalable, fault-tolerant, and distributed systems (Logan et al., 2010).

2.3.1 Deploying a Reactor

A reactor can be seen as a blueprint for its deployment, similar to how an object created from a class in object-oriented programming is an instantiation of the blueprint that the class represents. The deployment of the reactor will request a certain size of memory. That total block of memory (shown in figure 2.2) on the right side of the reactor exists out of four main parts: sources, deployment-time memory (dtm), reaction-time memory (rtm) and the sinks.

When we go from a reactor to a deployment we can find the following information in the reactor or blueprint for the deployment: the number of sources and sinks, the deployment time instructions (dti) and the reaction time instructions (rti). For each source and each sink a position is reserved in the memory. Based on the deployment-time instructions defined in the reactor we can prepare the deployment-time memory (dtm). This deployment-time memory part of the full memory block will reserve space for the deployment of the reactors found in the dti. The dtm size is defined upfront by matching the deployment time instructions that will be executed once, when the reactor is deployed.

The full memory block is almost allocated, the space for the sources, the sinks and dtm has been reserved, the last step is to reserve space for the reaction-time memory (rtm). The space required for rtm is equal to the number of instructions found in the reaction time instructions (rti). For each instruction in rti there is one block reserved in rtm. A deployed reactor uses the four memory parts: sources, dtm, rtm and sinks to react according to the reaction-time instructions found in rti. Knowing the index of the rti instruction that is executed is primordial, since we based the implementation around this index. The index of an instruction relates directly to a location in memory.

2.3.2 Starting the Deployment

The deployment of a reactor connects to the runtime in order to communicate with the system. The purely reactive programming language is connected to the 'outside world' with regular Elixir functions that express how the RP receives and sends information for the given domain or use case. Sources are connected to a function that call's the next value for the sources after each iteration of the deployed reactor. Similarly the sinks are connected to a function that given the values of the sink handles them in the way required by the use case. The runtime will, after the deployed reactor reacted, call the function that handles the sinks. When the 'sinks function' terminates the runtime calls the 'sources function', on termination new values are provided to the sources of the deployed reactor and everything iterates on and on for as long the runtime is active. Section 4 presents a musical use case where the values found in the sink serve as values used in OSC messages that are transmitted over the network.

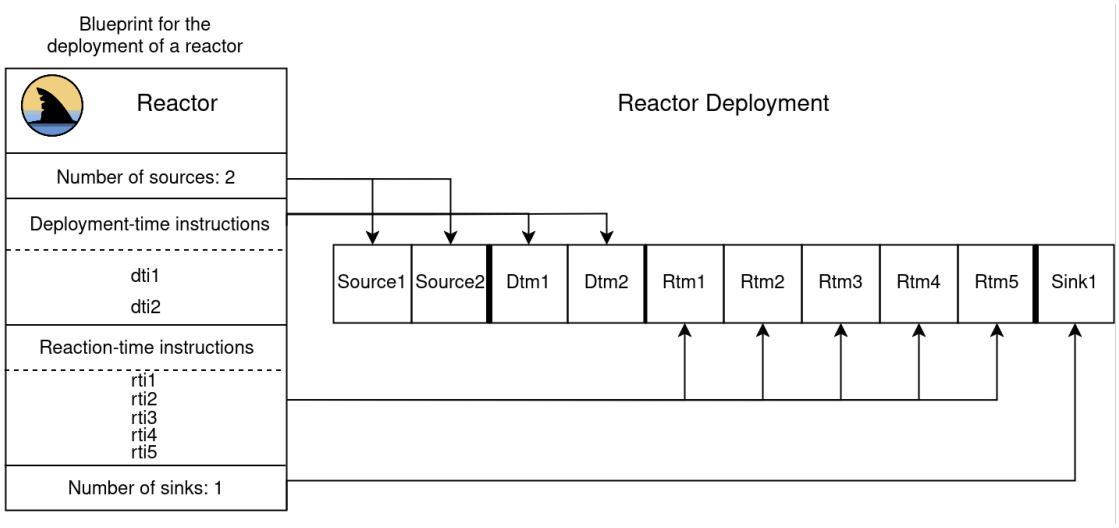

Figure 2.2: Reactor deployment, from blueprint to memory block

2.3.3 Managing the Reactors State

Managing the state of a reactor is handled by a single actor per reactive program or reactor deployment. This actor is built around the GenServer module in Elixir, a behavior module that provides a generic server implementation. We call this module Memory. Memory serves as a single actor that encapsulates multiple functions designed to respond to various messages or calls. These functions collectively define the behavior of the Memory. One instance of Memory represents the memory block (2.2)of a deployed reactor. All calls to memory are synchronous this ensures that the values within memory are accessed by only one operation at a time, thereby maintaining the integrity of the memory block.

Chapter 3

Distribution

The cluster environment over which we distribute the first-class reactors is an environment where no communication between reactors (on different nodes) exists. In this thesis we focus on starting different configurations on different cluster nodes. A configuration exists out of three things. The bytecode for a (composed) reactor, connecting functions for the source and sink and the node(s) on which that configuration should be deployed.

Technically the cluster is setup on a shared nothing architecture (Stonebraker, 1986). This allows us to build the cluster environment with independent nodes. Since nothing is shared we can easily add and remove nodes form the cluster environment. This flexibility and modularity allows us to organize different sized cluster environments. A cluster contains a master node and worker nodes. The master node will distribute a configuration to worker nodes. All the nodes in one cluster environment have the same code base (hvm) but do not necessarily run the same composed reactor. The master node can distribute different reactors (represented in byte code) and different connecting functions (represented in Elixir code) based on the specifications in the configuration. The master node contains two libraries. One with all (composed) reactors and a second one with all connecting functions.

To organize the deployment configurations for a cluster environment we developed a small domain specific language. This allows us to describe the configuration of the deployed cluster environment in a declarative way. Describing per worker node the composed reactor and connecting functions that have to be send.

3.1 Multi-Node Deployment Configuration

The declarative configuration of the distribution is presented as a yaml file. YAML is a human-readable data serialization language that is often used for writing configuration files. The structure of a YAML file is a map or list that follows a hierarchy based on the indentation. This allows us to define with a declarative syntax and purpose-specific abstractions, how the deployment should take place. By decoupling the configuration from the logic we have a cleaner and more modular codebase. This separation allows non-developers to easily modify configuration settings without delving into the code. For example the use case in this work is a musical instrument. The user input is restricted to three items. What data to connect to the input, how to react on the data and what to do with the values leaving the reactors. For each domain documentation is required for the user to know what reactors, connectors and sinks are available for the application, and how they work.

Listing 3.1 shows a small example, where two deployments are defined with purpose-specific abstractions like task, reactor, node, connector and sinks. This deployment configuration file will start the same reactor p1 with different connector functions in the same cluster environment.

Abstraction 'task' and 'node' are linked to the system and not to the application domain. Reactor, connector and sinks are the abstractions that come with the domain and will differ according to the use case. In this work all composed reactors have two inputs or connectors and one output or sink.

```
deployments:
  - task: start
    reactor: p1
    node: node2@0.0.0.0
    connector_1: f1
    connector_2: t2
    sinks: s1
  - task: start
    reactor: p1
    node: node3@0.0.0.0
    connector_1: f3
    connector_2: t1
    sinks: s1
```

Listing 3.1: YAML code

The YAML file is parsed and validated in the translation process. After validation the declarative configuration in the yaml file is stored in a Json file. Json files are commonly used for data serialization, making it a good format to exchange data between systems. The distribution module in Elixir will read and handle all configuration information found in the Json file.

For each abstraction in the YAML file a list with valid elements exist and all input will be validated against that list in the translation tool. When the deployment configuration succeeds validation, the translated configuration information will be transmitted to the distribution module on the master node in the elixir cluster environment.

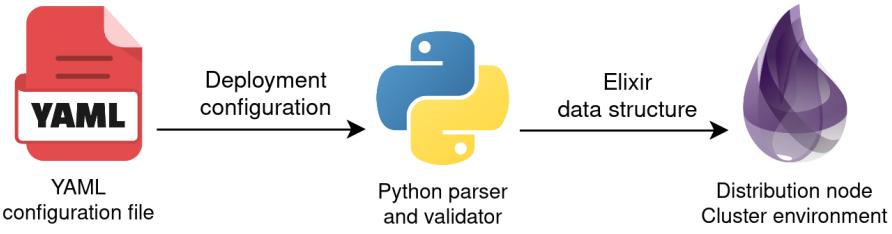

Figure 3.1: The DSL in YAML is parsed with Python into Elixir's list data structure

Figure 3.1 visualizes the process of sending the configuration file through the Python translation and validation tool. Once validated a Json file (containing the configuration) is prepared for the distribution module on the master node in the cluster environment.

3.2 Distributed Hvm

The distribution module is considered the master of the cluster environment. The cluster is controlled in a centralized way with a master node and worker nodes. The master node controls the deployment of a worker node. The worker node in this setting of reactive programming will not stop working until the master node requires it to stop. Each worker node can be seen as a (composed) reactor.

Deploying a reactor on the cluster involves sending the bytecode for the requested reactor and the Elixir code for the connecting functions to the Hvm module on the selected node. The ability to send the actual code to deploy a reactor allows us to dynamically alter the cluster environment. At this point no failure detection has been build into the reactors. For now any management of the cluster environment is not part of the reactive program.

Once the cluster is established, with one master connected to all worker nodes, the reactor or our reactive program is started and will keep on reacting, that is producing sink values based on the sources each iteration of the reactor receives. This setup ensures that the reactive program can respond to changes and maintain operations effectively across the distributed system. Figure 3.2 shows the concept of deploying the reactors from the distribution module into a chosen number of nodes. One reactor can be deployed to an 'unlimited' number of nodes. Each deployment counts as an individual deployment of the configured reactor.

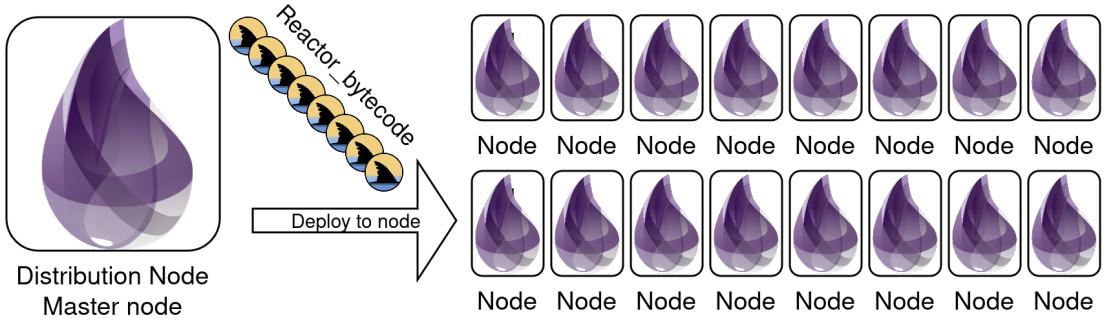

Figure 3.2: Distributing reactor deployment on Elixir cluster

3.3 Cluster Environment

The cluster environment of our system is built upon the actor model of computation (Hewitt, 2015). In this context, actors serve as the fundamental units of the reactive programming environment. The actor model is a mathematical model where actors are universal primitives of computation. Each actor possesses its own state and behavior and communicates with other actors through message passing. This model provides a robust framework for concurrent and distributed systems, ensuring scalability and fault tolerance.

In our cluster environment, actors are the primary entities that facilitate communication and computation. Currently, the communication between actors is confined to individual nodes within the cluster. This design choice simplifies the initial implementation and ensures that each node operates independently, reducing the complexity of inter-node communication.

A worker node in the cluster consists of two primary actors. The first actor is responsible for the process of the hvm, which handles the runtime for deployed reactors. The second actor

manages the state, or memory, of the hvm. To ensure the consistency of the hvm's memory, communication between these actors is synchronous. This synchronous communication guarantees that each call to the memory finishes before the next call is accepted, thereby maintaining data integrity and preventing race conditions.

The actor model of Erlang, which our system is inspired by, represents each process as an actor with a mailbox. This model allows for both synchronous and asynchronous communication. Asynchronous communication is particularly advantageous in distributed systems as it enables actors to continue their operations without waiting for responses, thereby improving overall system performance and responsiveness. Future enhancements, such as inter-node communication and leveraging asynchronous messaging, could further improve the system's capabilities.

Chapter 4

Evaluation

4.1 Evaluating the Cluster Environment

The single master node with a virtually unlimited amount of worker nodes that are all individual from each other gives us a flexible setup to work with. For example, the isolation of failure makes sure that when one node fails for some reason the cluster will not be affected by it. Similarly having the same code base on all nodes helps pointing out eventual problems with worker nodes. From a performance viewpoint the cluster environment is not resource hungry. The number of nodes that simulate the cluster can easily be a tenfold on one machine . The reactors itself have a very light computational cost in the musical use case. The reactors only use some basic arithmetic operations. One Elixir node consumes about 90MB of memory and virtually no processor time when idle and about 3% when running the use case. This allows to simulate a cluster on one laptop machine with over a hundred nodes. Potentially allowing for big cluster environments on physically spaced machines. In the scope of this work only cluster environments running on one laptop where tested. The size of the cluster was never a problem until the hardware resources of the test machine became the limiting factor around 100 nodes.

In the (next) section 4.2 we present the Distributed reactive melody generator as a use case for the cluster environment. The idea that an unlimited amount of devices (internet of things) generate information that can be reacted on in a way that the information is transformed into synthesized sound. The 'instrument' generates sound on a continuous basis. The sound never stops for as long as the system exists. The artistic idea is somewhat similar to Longplayer, a one-thousand year long composition located at Trinity Buoy Wharf in London (Trust, 1999).

4.2 Distributed reactive melody generator

For a melody to be generated we need at least one source of information. This source can be manipulated and transferred to the sound server. The maximum number of sources will be defined by the actual system and will depend mainly on the systems hardware constraints (that are expandable). The distributed first-order reactors in the cluster environment provide the sink values to be send to the sound server with the OSC protocol (Schmeder et al., 2010). The OSC messages are send to a sound server to trigger generation of sound. It is the reactors that, by reacting on incoming streams of data, produce the values that result in a rhythmical flow of OSC messages to the sound server. We have a virtually unlimited amount of information that is reactively transformed in a never ending audible piece of sound.

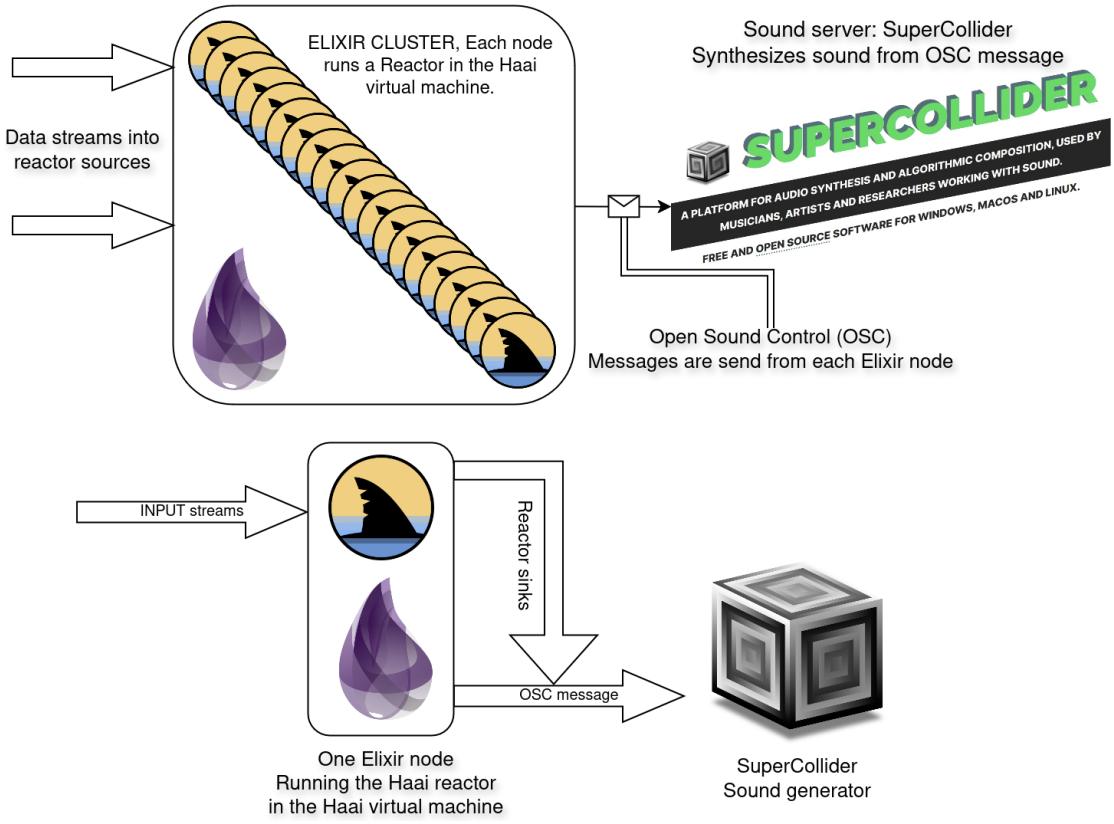

Figure 4.1: Distriuted Reactive Melody Generator

In the current configuration the flow of rhythmical messages is related to the duration of each sound. A reactor will iterate at a varying speed related to the duration of the sound that will be generated form the sink values of that reactor. Each reactor iterates over and over but each iteration has a defined duration, this duration is virtually equal to the duration for a sound calculated by the reactor. A reactor will calculate the new values, sleep for the calculated duration and then provide the new values in the sink.

4.2.1 Sound Server

The synthesizer that produces the sound is a Synthdef inside the sound server named Supercollider. Supercollider is an open-source platform for audio synthesis and algorithmic composition. Developed by James McCartney in the late 1990s (Wilson et al., 2011), it has since become a powerful tool in the fields of music technology, computer music, and sound art. Supercollider provides a flexible and expressive environment for creating and manipulating sound in real-time, making it an invaluable resource for both research and artistic exploration. Some main features and usage examples of Supercollider include: Audio Synthesis, Real-time Processing and Algorithmic Composition.

4.2.2 Sound Server Communication

The communication between the nodes in the Elixir cluster and the sound server is done over the network using the UDP network protocol. The sound server can handle a virtual unlimited amount of messages, but is practically bounded to the udp buffer size on the actual machine that runs the sound server. As such the Elixir cluster of reactors can be of any feasible size and the sound server will respond on any number of messages by synthesizing the requested sound.

4.2.3 Musical Values

The reactor or program that runs on all nodes in the cluster does for each note the following. It calculates two values given two input streams of numbers. The first source is a stream of base frequencies, represented as a float number, the second source is again a stream of numbers representing a tempo in beats per minute (bpm)

Consonant notes

Consonant notes sound harmonious when played together. consonance can be expressed as a ratio between two notes or frequencies. Ratios like the perfect fifth (frequency ratio of 3:2) and the major third (frequency ratio of 5:4) are very commonly used ratios in a musical setting. One of the two reactors in the main program used for this use case calculates such a note for a given base note, both are expressed in there actual frequency represented as a float number.

Tempo

We can express the speed of a musical note progression in beats per minute (bpm). From that number we can then calculate the duration of individual notes in the musical performance.

We calculate the time in milliseconds. Since one minute is 60000 milliseconds one can calculate the length of a quarter note with $q = \frac{60000}{bpm}$ with the assumption that we use a 4/4 time signature. From the length of the quarter note it is easy to find other durations. For example, the duration of a halve note is double that of a quarter note or the duration of a eight note is half the time of a quarter note.

One of the two reactors in the main program used for this use case calculates exactly that, the length for a quarter note and improvises a deviser or multiplier to produce some duration.

Chapter 5

Conclusion

Reactive Programming (RP) has emerged as a powerful paradigm that models data flow as streams of events, enabling systems to react efficiently to changes and events. This document has explored the principles of reactive programming, its applications, and the development of a framework for distributing first-class reactive programs in a cluster environment. The novel reactive language Haai, which operates independently of a base language, addresses key challenges in reactive programming by preventing the base language from interfering with the reactivity of the program.

The Haai virtual machine, implemented in Elixir and leveraging the Erlang Virtual Machine (BEAM), enables the distribution of reactive programs across a cluster. The evaluation of this cluster environment has demonstrated its flexibility, scalability, and fault tolerance, allowing for the simulation of large cluster environments on a single machine. This capability is crucial for developing and testing distributed reactive systems in a controlled setting before deploying them in real-world scenarios.

A practical use case, the distributed reactive melody generator, showcases the framework's application in transforming streams of data into synthesized sound. This use case involves sending network messages to a Supercollider sound server, highlighting the potential of reactive programming in creating innovative and dynamic systems that respond to real-time events. The ability to transform data streams into audible outputs underscores the versatility and creativity that reactive programming can bring to various domains, including music and multimedia.

The canonical reactive program model, exemplified by spreadsheets, demonstrates the power of time-varying values and the propagation of change. The Haai language, with its clear and concise syntax, allows for the creation of complex behaviors from simple building blocks, promoting readability and maintainability. The use of behaviors and events as fundamental abstractions in RP ensures that systems can handle both continuous and discrete changes efficiently.

Overall, this work contributes to the advancement of reactive programming by providing a robust framework for distributed reactive programming and demonstrating its practical applications in a musical context. The Haai language and its virtual machine offer a promising avenue for further research and development in the field of reactive systems. Future work could explore additional use cases, optimize the performance of the Haai virtual machine, inter-connectivity between cluster nodes and investigate the integration of reactive programming with other emerging technologies.

Bibliography

Bainomugisha, E., Carreton, A. L., Cutsem, T. V., Mostinckx, S., & Meuter, W. D. (2013). A survey on reactive programming. *ACM Comput. Surv.*, *45*(4), 52:1–52:34. https://doi.org/10.1145/2501654.2501666

Courtney, A., Nilsson, H., & Peterson, J. (2003). The yampa arcade. In J. Jeuring (Ed.), *Proceedings of the ACM SIGPLAN workshop on haskell, haskell 2003, uppsala, sweden, august 28, 2003* (pp. 7–18). ACM. https://doi.org/10.1145/871895.871897

Elliott, C. (1998). *Programming for greater freedom of expression*. https://jacobfilipp.com/DrDobbs/articles/DDJ/1998/9807/9807a/9807a.htm

Elliott, C., & Hudak, P. (1997). Functional reactive animation. In S. L. P. Jones, M. Tofte, & A. M. Berman (Eds.), *Proceedings of the 1997 ACM SIGPLAN international conference on functional programming (ICFP '97), amsterdam, the netherlands, june 9-11, 1997* (pp. 263–273). ACM. https://doi.org/10.1145/258948.258973

Hewitt, C. (2015). Actor Model of Computation. In *Inconsistency Robustness*. https://hal.science/hal-01163534

Logan, M., Merritt, E., & Carlsson, R. (2010). *Erlang and otp in action* (1st). Manning Publications Co.

Margara, A., & Salvaneschi, G. (2018). On the semantics of distributed reactive programming: The cost of consistency. *IEEE Trans. Software Eng.*, *44*(7), 689–711. https://doi.org/10.1109/TSE.2018.2833109

Meisinger, G. (2021). *Book of yampa*. https://yampa-book.readthedocs.io

Mogk, R., Baumgärtner, L., Salvaneschi, G., Freisleben, B., & Mezini, M. (2018). Fault-tolerant distributed reactive programming. In T. D. Millstein (Ed.), *32nd european conference on object-oriented programming, ECOOP 2018, july 16-21, 2018, amsterdam, the netherlands* (1:1–1:26). Schloss Dagstuhl - Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPICS.ECOOP.2018.1

Oeyen, B., De Koster, J., & De Meuter, W. (2022). Reactive programming on the bare metal: A formal model for a low-level reactive virtual machine. *Proceedings of the 9th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*, 50–62. https://doi.org/10.1145/3563837.3568342

Oeyen, B., De Koster, J., & De Meuter, W. (2024). Reactive programming without functions. *The Art, Science, and Engineering of Programming*, *8*(3), 11. https://doi.org/10.22152/programming-journal.org/2024/8/11

Schmeder, A., Freed, A., & Wessel, D. (2010). Best practices for open sound control. *Linux Audio Conference*, *10*.

Sterz, A., Eichholz, M., Mogk, R., Baumgärtner, L., Graubner, P., Hollick, M., Mezini, M., & Freisleben, B. (2021). Reactifi: Reactive programming of wi-fi firmware on mobile devices. *Art Sci. Eng. Program.*, *5*(2), 4. https://doi.org/10.22152/PROGRAMMING-JOURNAL.ORG/2021/5/4

Stonebraker, M. (1986). The case for shared nothing. *IEEE Database Eng. Bull.*, *9*(1), 4–9. http://sites.computer.org/debull/86MAR-CD.pdf

Trust, T. L. (1999). *A one-thousand year long composition*. https://longplayer.org/

Vonder, S. V. d., Renaux, T., Oeyen, B., Koster, J. D., & Meuter, W. D. (2020). Tackling the awkward squad for reactive programming: The actor-reactor model [ISSN: 1868-8969]. In R. Hirschfeld & T. Pape (Eds.), *34th european conference on object-oriented programming (ECOOP 2020)* (19:1–19:29). Schloss Dagstuhl–Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2020.19

Wilson, S., Cottle, D., & Collins, N. (2011). *The supercollider book*. The MIT Press.