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 has gained significant traction for its efficacy in handling asynchronous and event-driven scenarios, particularly in distributed systems. This paradigm emphasizes responsiveness, scalability, and resilience, making it an ideal choice for real-time applications. However, distributing reactive programming introduces challenges such as consistency, coordination, fault tolerance, and communication overhead. This thesis investigates these challenges and proposes a system designed to address them. By leveraging a shared-nothing architecture and deploying an Erlang cluster, we develop a runtime for the purely functional reactive language Haai. This system aims to eliminate known issues in distributed reactive programming, such as the Reactive Thread Hijacking Problem and Reactive/Imperative Impedance Mismatch. We detail the implementation of the Haai virtual machine (Hvm), the deployment of reactors, and the management of distributed environments to enhance scalability, fault tolerance, and performance in distributed reactive systems.

Contents

Chapter 1

Introduction

Reactive programming has been around for a while, gaining traction in various domains due to its ability to handle asynchronous and event-driven scenarios effectively. One of its notable advantages lies in its suitability for distributed systems. By its nature, reactive programming facilitates the development of systems that can easily scale across distributed environments, allowing components to communicate asynchronously and react to events in real-time. However, distributing reactive programming poses several challenges (Bainomugisha et al., 2013) such as consistency, coordination, fault tolerance and many more. These challenges highlight the need for further research and study in understanding how reactive programming principles can be effectively applied and scaled in distributed environments. By investigating these challenges, we can gain insights into best practices and architectural approaches for building robust and scalable distributed reactive systems.

1.1 Reactive programming

Reactive Programming (RP) is a broader programming paradigm that encompasses the idea of reacting to changes and events in a system. It involves modeling the flow of data as streams of events and using declarative and composable abstractions to handle asynchronous and event-driven scenarios. The canonical reactive program model (ref?) 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. Both can be seen as time-varying values but behaviors are said to be continuous over time whilst events are discrete over time. Behaviors have a value at any moment (continuous). 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 behavior that continuously contains the result of a1 * a2. For each time that an event occurs updating a1 or a2, A3 will be automatically updated. One could say that the tempo or rhythm of the propagation of change for the behavior A3 depends on the speed that events have a new discrete value.

```
(defr (A3 a1 a2)
(out (* a1 a2)))
```

Listing 1.1: Haai code

1.2 Problem Statement

The distribution of reactive programming (RP) introduces a myriad of complexities and challenges. In this work we would like to focus on developing a framework to further study the distribution of first-class reactors in a cluster environment. In this work we want to tackle the creation of a declarative specification to deploy the distributed reactive program and all underlying infrastructure to deploy the first-class reactors into different sized cluster environments.

1.3 Overview

The Structure of this thesis starts with a situation of (distributed) reactive programming. After explaining the strategy of the thesis we explain purely reactive programming in section 2.3. More detail on the runtime is found in chapter 2.4. Chapter 3 explains the distribution of the runtime and the tools used to build it. Before ending with the conclusion we have chapter 4 with the evaluation of the distribution and the musical use case of the distributed reactive melody generator.

1.4 Strategy and Methodology

In order to run the first-class reactors in a distributed setting we setup a cluster environment. The cluster environment is build as a shared nothing architecture (sna) and hosts an Erlang cluster. We have chosen this architecture for basic simplicity as it allows to easily scale and monitor fault tolerance since each node is independent and self-sufficient.

To deploy the system (combination of firs-class reactors) we make use a declarative domain specific language to define which reactor needs to run on what node or group of nodes. We have build a runtime for the purely functional reactive language Haai on top of the sna. Each node represents a runtime towards which a (master) node can send the code to be deployed.

All the different nodes or runtimes in the cluster have a rhythmical organization (eg: the loop time of a reactor is a fixed value in milliseconds).

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, explained in section 1.1, showing there similarities. Two fundamental differences instead are 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 exiting 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 a glitch, 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

In this section and the rest of this thesis we will focus on reactive programming. Reactive programming has many applications we will show tree 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

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, that are reactive definitions of individual processing steps triggered by incoming events.

```
 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

The domain-specific language Yampa (Courtney et al., 2003) embedded in Haskell for programming hybrid (mixed discrete-time and continuous-time) systems. Listening 2.4 shows a signal function that describes the behavior of a bouncing ball with position (p) and velocity (v) that are continuously updated over time. When the discrete event 'hitGround' occurs the velocity of the ball reverses.

```
-- 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

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 could make stop a reactive program from being reactive. In section 2.3 we will introduce a novel reactive language Haai that by design excludes both problems mentioned above.

2.2 Distributed reactive programming

Different distributed systems can be thought of as a reactive system. The distributed system reacts to events in a similar way a non distributed reactive system would do. For example an internet of things application with many connected sensors. Or a distributed reactive system with strong computational differences for different behaviors that should finish in a certain requested time. The distribution can optimize multiple nodes with different computational capacities related to the application.

Once working in a distributed setting, typical problems arise like data inconsistencies and node disconnects and crashes. In the literature inconsistencies in the data are called glitches. A glitch is a momentary inconsistency in a time-varying value. For example in a financial application we do not want to read inconsistent values. We need to be sure that all dependencies for a certain value are correct before reading that value. Being free of glitches is called glitch-freedom (Margara and Salvaneschi, 2018).

To date, REScala has been the only reactive language providing 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 2.5 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 2.5: REScala, shared calendar application

2.3 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.

2.3.1 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. We can write a single reactor or combine multiple reactors in one 'main' reactor or program. The main program is build out of reactors, these can be native or user defined. Every computation in a reactor 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 host language.

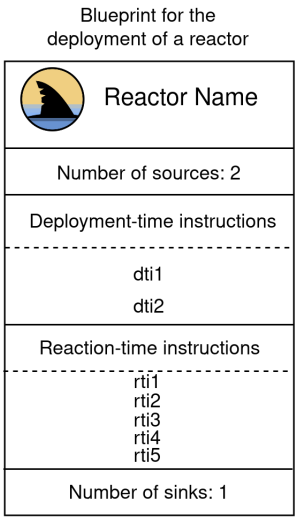

Figure 2.1: Graphical representation of a reactor

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 2.4.1.

2.3.2 Haai

We are making 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.

Haai Code example

This Haai code in listing 2.6 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 2.6: Haai code

The reactor in listing 2.6 has two sources (f and bpm) and two sinks defined with the keyword out. To port the Haai code (listing 2.6) into the cluster environment we compile the Haai code into a byte code representation (listing 2.7) that was developed for the Remus virtual machine. "a virtual machine that has been carefully designed to be usable for running reactive programs in low-powered 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

Bytecode example

After parsing the bytecode that was compiled form the original Haai code in listing 2.6 with the Remus compiler into an Elixir readable format of nested lists a program looks like listing 2.7. In this example 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 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 2.7: Remus bytecode as Elixir nested lists

2.4 Haai virtual machine

To run the Haai code or compiled bytecode we implement in this thesis a virtual machine for Haai. 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 computer 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).

2.4.1 Deploying a reactor

A reactor can be seen as a blue print 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) exists out of four 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 blue print 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 block we are constructing. From the deployment time instructions in the reactor we can prepare the deployment time memory (dtm). The Dtm part of the memory block we are constructing will reserve space for the deployment of the reactors found in the dti.

Sofar 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. 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.

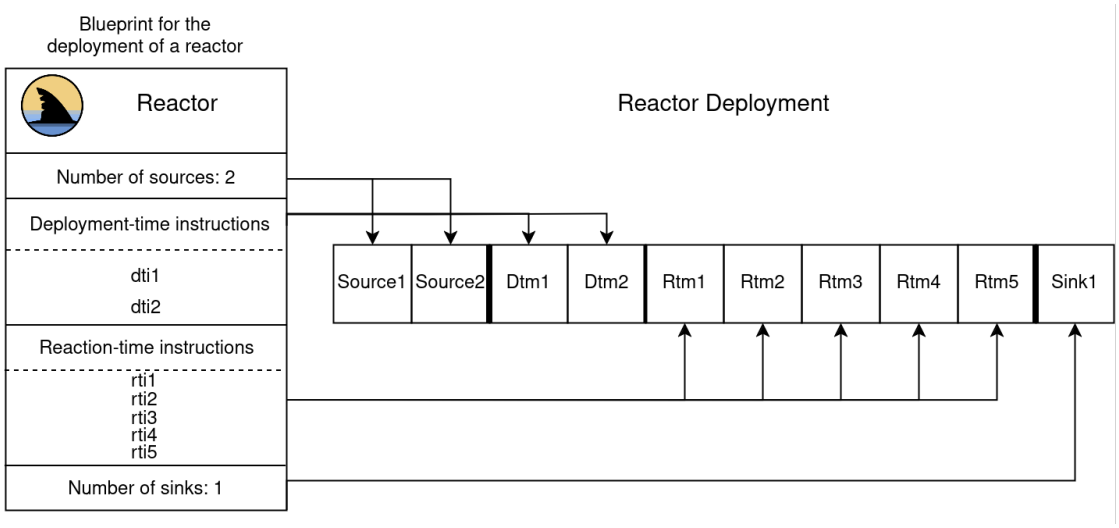

Figure 2.2: Reactor deployment, from blueprint to memory block

2.4.2 Starting the deployment

The deployment needs to be its sources and sinks to the runtime in order to communicate with the system. 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 required way for the overall program. Section 4 presents a musical use case where the values found in the sink serve as values for OSC messages.

2.4.3 Managing the reactors state

Managing the state of a reactor is handled by a single actor per program or reactor deployment. This actor is built around the GenServer module in Elixir, a behavior module that provides a generic server implementation. In this implementation, the GenServer is named 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 GenServer.

Memory primarily accepts calls from the runtime instructions found in the reactor deployment. These calls are mapped to the Haai instruction set. Additionally, there are calls to initialize the Memory GenServer, as well as to get or set sources and sinks within it. One instance of Memory represents the memory block of a deployed reactor.

The GenServer module in Elixir operates with calls or casts. In this context, we use calls, which are synchronous requests to the GenServer. By making synchronous calls to Memory, we ensure that the values within Memory are accessed by only one operation at a time, thereby maintaining the integrity and correctness of these values.

2.4.4 Native reactors

The Haai virtual machine makes use of native reactors to perform simple arithmetic operations like addition, division, multiplication and subtraction. Native reactors are recognized by the Hvm and applied when required. Thanks to the native reactors we do not need to implement the most basic reactors when composing a program or one could add native reactors to the system

if needed. Native reactors provide us a possibility to define the capabilities of the system by providing a certain set of native reactors equipped for basic tasks in a domain.

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 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. Different instances of the Hvm, each running on one Elixir node, can run physically on the same system. An idle Elixir node is not resource-intensive, making it feasible to run tenfold of nodes on a single laptop machine to simulate one or more cluster environments. All the nodes in one cluster environment have the same code base. Once the cluster is up and running all instances of the Hvm are waiting to receive the start command with the required configuration details.

To organize the deployment configurations for a cluster environment we developed a small domain specific language. This allows us to describing the deployed cluster environment in a declarative way.

3.1 Domain specific language

As input file to the DSL we use 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 in a declarative syntax with purpose-specific abstractions, how the deployment should take place. Listing 3.1 shows a small example, where two deployments are defined with purpose-specific abstractions like task, reactor, node, connector ans sinks. This deployment configuration file will start the same reactor p1 with different connector functions in the cluster environment.

```
deployments:
  - task: start
    reactor: p1
    node: node2@0.0.0.0
    connector1: f1
    connector2: t2
```

```
    sinks: s1
  - task: start
    reactor: p1
    node: node3@0.0.0.0
    connector1: f3
    connector2: t1
    sinks: s1
```

Listing 3.1: YAML code

The YAML file is parsed and validated in the translation process form YAML to Elixir data structure (nested list). For each abstraction a list with valid items exist and the input will be validated against that list. When the deployment configuration succeeds validation the translated configuration information will be transmitted to a distribution module on the master node in the elixir cluster environment that can send start commands to Hvm nodes.

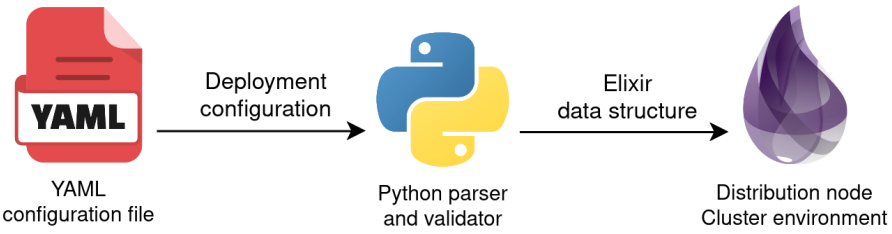

Figure 3.1: The DSL in YAML is parsed with Python into Elixir's list data structure

Figure 3.1 visualize the process of sending the configuration file trough the python translation and validation pass towards the distributed module 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. we have 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 reactor.

Deploying a reactor on the cluster involves initiating the Hvm on the selected node with the bytecode of the requested reactor. This allows each node to run either the same reactor or different reactors, depending on the bytecode it receives. The ability to send the actual request to deploy a reactor allows us to dynamically alter the cluster environment. At this point no failure detection has been build into the reactors but replacing a node that has gone down or balance the load by adjusting the number of nodes running a particular reactor are thought to become part of the reactive program. For now any management of the cluster environment is not part of the reactive program and is done with the Erlang OTP functionality that comes with the cluster.

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 for as long as the worker node exists. 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 model into a chosen number of nodes. One reactor can be deployed to an 'unlimited' number of nodes. Each deployment counts as an individual instance of the configured reactor.

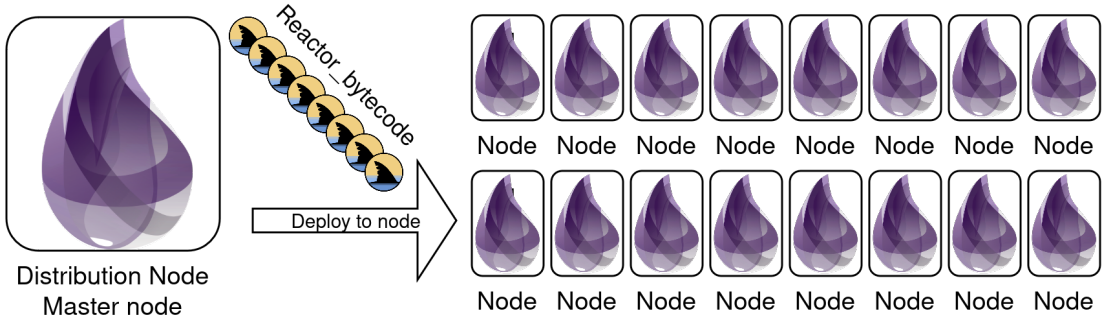

Figure 3.2: Distributing reactor deployment on Elixir cluster

3.3 Cluster environment

The cluster environment exists out of actors. In our context we can see the actors as the fundamental units of the reactive programming environment. The actor model of computation (Hewitt, 2015) is a mathematical model where actors are universal primitives of computation. Each actor has its own state and behavior and can communicate with each other using messages.

At present, in our cluster environment the communication between actors only happens inside of each individual node. A worker node in the cluster exists out of two actors. One actor is the process of the Hvm, the second actor is the process that manages the state (called memory) of the Hvm. The Hvm and memory actor communicate when the state of the Hvm changes. To make sure that the memory of the Hvm is always consistent the communication happens synchronously making sure that each call into the memory has to finish before a next call can be accepted. The actor model of Erlang where a process is represented by an actor also allows for asynchronous communication, both can be used depending on the required situation.

The cluster environment is ready to have communication between worker nodes. But at present the Hvm has no functionalyty to exchange messages between worker nodes.

Chapter 4

Evaluation

4.1 Evaluating the cluster environment

The single master node with an open amount of worker nodes used in our distribution is well suited. The individual nature of the nodes in the cluster gives us guarantees we do not need to worry about when running the cluster environment. 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. Also form a performance stand point the cluster environment is not resource hungry. The number of nodes that simulate the cluster can easily be a tenfold on one machine (eg: not a physically spaced cluster). The reactors itself have a very light computational cost existing out of some basic arithmetic operations. One Elixir node consumes about 90MB of memory and virtually no processor time when idle and about 3% when performing in the musical use case presented in the next section. This allowed to simulate a cluster on one laptop machine with up to a hundred nodes. Potentially allowing for big cluster environments with over a thousand nodes. In the scope of this work only cluster environments running on one laptop where tested. The size of the cluster was never a problem but when the hardware resources of the test machine became limited to boot up the cluster. That limit was hit slightly above 100 nodes.

In the (next) section 4.2 we present the Distributed reactive melody generator as a use case for the cluster environment. The instrument generates sound on a continuous basis. The sounds never stop as the rectors that generate values for those sound never stop either. The values produced by the reactors are send over the network to a sound server that generates synthesized sound.

4.2 Distributed reactive melody generator

The distributed firs-order reactors in the cluster environment, existing out of many reactors provides the sink values form the reactors as values inserted into OSC messages (Schmeder et al., 2010) that are send to a sound server to trigger the synthesis of some 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.

The flow of rhythmical messages is related to the duration of each sound. That same duration that makes the sound disappear after it was started determines the iteration speed for the reactor or program that runs on each individual node. Each reactor iterates over and over but each

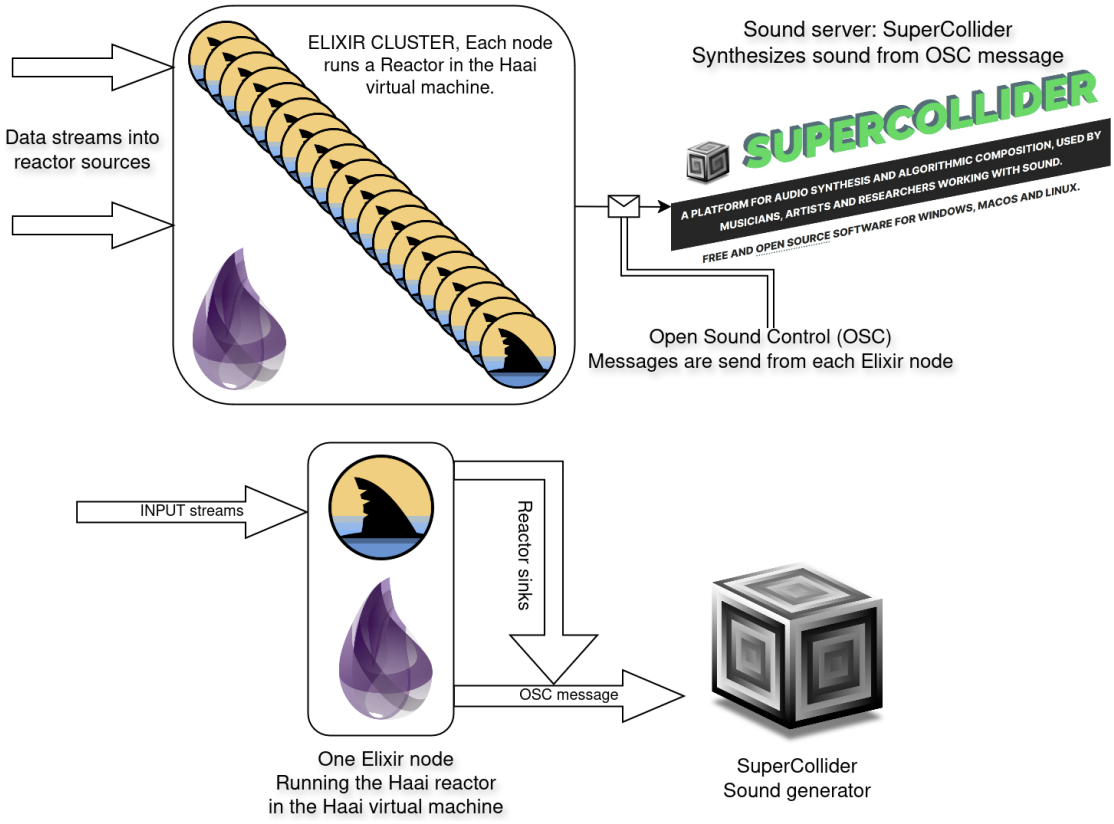

Figure 4.1: Distriuted Reactive Melody Generator

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. At this point the Hvm calls a function that it was given to handle the sink values. For this use case that function constructs the message and sends it to the sound server that propagates it to the requested synthesizer.

4.2.1 Sound server

The synthesizer that produces the sound is a Synthdef (the definition of a synthesizer) 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 of Supercollider include:

Audio Synthesis Supercollider offers a wide range of synthesis techniques, including additive, subtractive, granular, and physical modeling synthesis. Users can create complex sounds by combining these techniques and modulating parameters in real-time.

Real-time Processing One of Supercollider's key strengths is its ability to process audio in

real-time. This makes it suitable for live performances, interactive installations, and other time-sensitive applications.

Algorithmic Composition Supercollider provides tools for generating music algorithmically, allowing users to create compositions based on mathematical algorithms, rulesets, or generative processes.

Integration with External Hardware Supercollider can interface with external MIDI controllers, audio interfaces, and other hardware devices, enabling users to incorporate physical instruments and sensors into their sound projects.

Community and Documentation Supercollider has a vibrant online community of users who share code, tutorials, and resources. Additionally, comprehensive documentation is available, including tutorials, reference guides, and examples to help users learn and master the software.

SuperCollider has been and still is actively used in academia for many purposes, for example:

Research in Music Technology Digital signal processing, human-computer interaction, and machine learning for music. Researchers leverage its flexibility and programmability to prototype new algorithms, experiment with novel synthesis techniques, and investigate the perceptual and cognitive aspects of sound.

Composition and Sound Design Supercollider is used to create innovative works that push the boundaries of traditional music and sound art. Its ability to generate complex and evolving textures, as well as its support for algorithmic composition, makes it a valuable tool for exploring new sonic territories.

Teaching and Learning Supercollider is increasingly incorporated into music technology and computer music curricula at universities and colleges worldwide. It provides students with hands-on experience in sound synthesis, programming, and digital audio processing.

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

5.1 Further work

Extending the Hvm to allow for (actor) communication between worker nodes will make it possible for first-class reactors to communicate over the network.

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., & 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

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

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.