



VRIJE
UNIVERSITEIT
BRUSSEL



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

DISTRIBUTING REACTIVE HAAI

Haai Virtual machine

Hans

June 2024

Promotors: prof. dr. Wolfgang De Meuter Supervisor: Bjarno Oeyen

sciences and bioengineering sciences

Abstract

We are looking into the distribution of a purely reactive programming language Haai.

Context . inquiry . Approach . Knowledge . Grounding . Importance

Contents

Abstract	iii
1 Introduction	1
1.0.1 Statement of the problem	1
1.0.2 Definition of terms	1
1.0.3 Remaining chapters	2
2 Haai	3
2.0.1 compare with other rx	3
2.1 Haai instruction set	3
2.2 Reactors	4
2.2.1 Reactors memory	4
3 Haai virtual machine	5
3.1 Example bytecode program	5
3.2 Native reactors	7
4 Distribution	9
4.1 Shared-nothing architecture	9
4.2 Erlang virtual machine	9
4.3 Actor model	10
4.4 Distributed Erlang mechanisms	10
5 Distributed reactive melody generator	11
5.0.1 Sound server	11
5.0.2 Sound server communication	12
5.0.3 Musical values	12
Further work	15
Conclusion	17

Chapter 1

Introduction

This thesis steps into the world of distributed reactive programs. Reactive programming revolves around the concept of reacting on streams of data. In reactive programming the program reacts on a stream of data. The stream of data travels through the reactive program to exit in a possibly altered state. We will investigate possibilities to distribute the reactive programming language Haai. (Oeyen et al., 2024) Haai is a purely reactive language consisting of only reactors. The literature review on distribution models for reactive programs lead us to the actor model (De Koster et al., 2016). We have implemented a virtual machine that opens the possibilities for the Haai language to be distributed on a Elixir (Juric, 2024) cluster.

1.0.1 Statement of the problem

To distribute a reactive program different approaches can be taken, We will select a distribution model to experiment with the distribution of reactive programming languages or reactors written in Haai. Glitch avoidance in reactive programs has been achieved on most modern implementations that run on a single node. In a distributed setting glitch avoidance is not an easy task due to the parts the distributed system is build of. Delays, network failures and other problems make it complicated to reach glitch freedom in a distributed setting. (Bainomugisha et al., 2013)

1.0.2 Definition of terms

Reactive programming language A computer language to write reactive programs.

Reactive programming Reactive programming is a programming paradigm that focuses on asynchronous data streams and the propagation of changes. In reactive programming, you model your application as a series of data flows and transformations, where changes in one part of the system can automatically trigger updates in other parts. This paradigm is particularly useful for building interactive and responsive applications, such as user interfaces, real-time analytics, and event-driven systems.

Distribution the deployment of reactive systems across multiple computing nodes or devices. This distribution enables the system to handle a large volume of data, support high concurrency, and scale horizontally to meet increasing demands.

1.0.3 Remaining chapters

In the next chapter we give a description of the purely reactive programming language Haai. Chapter 3 about the main implementation of this thesis describes the Haai virtual machine (Hvm). The distribution chapter explains why we selected Elixir as a platform and how we build the distributed cluster that runs multiple independent Hvm's. Before concluding and reflecting on further work we have a description of the use case of the thesis.

Chapter 2

Haai

We are making use of the novel purely reactive language Haai. Haai is reactive all-the-way-through. 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)

Two common problems encountered within reactive programming are by design eliminated. These two problems are called the *Reactive Thread Hijacking Problem* and the *Reactive/Imperative Impedance Mismatch*. (Vonder et al., 2020) Two problems that arise from mixing reactive code with sequential code in a two-layered reactive programming language.

Reactive Thread Hijacking Problem

This happens in a two-layered reactive programming language when the reactive runtime or top layer calls a function in the bottom layer and that call never returns. The host has hijacked the control and the reactive runtime will not be able to regain control.

Reactive/Imperative Impedance Mismatch

When the reactive runtime in a two-layered reactive program used different side effects by calling functions on the first layer. It might be so that the order in which they are executed is not as expected. Creating faults in the programs that might not be easy to detect.

2.0.1 compare with other rx

2.1 Haai instruction set

After compiling a Haai program into its bytecode everything is based around this instruction set. The instruction set defines the set of low-level operations that the virtual machine can execute. Each instruction corresponds to a specific operation. the Haai virtual machine presented in the next chapter is build around this instruction set.

[**I-ALLOCMONO**, **reactor**] Allocate memory for the given reactor.

[**I-LOOKUP**, **signal**] Lookup the value for signal in a signal table.

[I-SUPPLY, from, destination, index] Supply the value found in from to destination at index.

[I-REACT, memory_location] Apply the reaction required at memory_location.

[I-CONSUME, memory_location, index] Move the value form REACT to the run-time-memory.

[I-SINK, rti_location, sink_index] Remove value from run-time-memory and store in the reactor sink.

2.2 Reactors

In Haai the only construct available for expressing computations is a reactor. Such a reactor consists of the following parts. There are the sources and the sinks, they can be seen as what goes in and what comes out of the reactor Or the sinks will hold the altered value that was found in the source. The reactor has two possibilities to react on the values found in the source. The reactor can use a user defined reactor or a native reactor. For example in the Haai virtual machine we have native reactors for basic arithmetic. The memory size of a reactor is predefined by the number of instructions it needs to run.

2.2.1 Reactors memory

The in size fixed memory block of a reactor exists of the following four parts: sources, deployment-time-memory (dtm), reaction-time-memory (rtm) and sinks.

FIGURE

Chapter 3

Haai virtual machine

The Haai virtual machine (Hvm) is developed in the Elixir computer language. elixir runs on top of the Erlang Virtual Machine called BEAM. The BEAM virtual machine provides features such as lightweight processes, scheduling, memory management, and fault tolerance. This will allow us to explore the possibilities for working with the Haai computer language in a distributed environment.

For the development of this virtual machine inspirations was found in a different virtual machine for Haai called Remus. Remus is a virtual machine that has been carefully designed to be usable for running reactive programs in low-powered computing environments. (Oeyen et al., 2022b)

3.1 Example bytecode program

After parsing the bytecode that was compiled from the original Haai code with the Haai compiler into an Elixir readable format of nested lists a program could look like this. In this example we see two user defined reactors named `consonance` and `note_length` followed by the main reactor that functions as the main reactor to be executed.

```
co_n1 = [
  [: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]
  ],

  [:note_length, 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", :lm],
  ["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]
]
]
]

```

The above is an example of bytecode that is send to the Haai virtual machine. This bytecode is a compilation of the following Haai program defining two user defined reactors and joining them together within the main reactor. The main reactor could be seen as the 'main' function in C. The starting point or the call that kicks off the programs execution.

```

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

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

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

```

3.2 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.

Chapter 4

Distribution

4.1 Shared-nothing architecture

To organize the distribution of the Haai virtual machine we have opted for a shared-nothing architecture. (Stonebraker, 1986) This allows us to build the cluster with the following characteristics, suitable for the first time distribution of the reactive language Haai.

Independent Nodes Each node can execute its own specific code and perform computations independently since no memory, storage or processing power is shared.

No shared state Each node manages its state independently. Cluster communication between nodes is done with messages. we use Erlang distributed mechanisms in the implementation.

Horizontal scalability Shared-nothing architectures are highly scalable horizontally, this makes it straightforward to add or remove nodes from the cluster. It allows us to run clusters of all sizes. Practically the cluster size will be bound to the available hardware.

Fault isolation Faults or failures in one node do not affect the operation of other nodes in the system. We can monitor node failure with Erlang distributed mechanisms and act accordingly.

Flexibility and Modularity Changes to one node do not impact the operation of other nodes, allowing for maintenance, upgrades, and system evolution of the cluster.‘

4.2 Erlang virtual machine

We implemented the shared-nothing architecture in Elixir a functional, concurrent, and fault-tolerant programming language built on the Erlang Virtual Machine (BEAM). The Erlang Virtual Machine provides a robust platform due to several key features and design principles:

Concurrency model Erlang has a lightweight concurrency model which is based on actor-like processes. Since each process runs independently and can communicate with other processes through message passing the Erlang virtual machine is inherently decentralized by design.

Fault tolerance The BEAM VM provides mechanisms for process supervision, fault isolation, and hot code swapping, allowing systems to recover from failures without affecting the overall operation of the cluster.

Distribution transparency Erlang’s distribution features enable processes to communicate effortlessly across network boundaries. Within an Erlang cluster, nodes can seamlessly interact through remote procedure calls (RPCs) and message passing. Different types of distribution models can be build on the Erlang platform.

Scalability Horizontal scaling or adding nodes to a cluster is straightforward. This makes it easy to scale clusters to need.

Tooling Common tasks are simplified thanks to a comprehensive tool set. tools like the Mix build tool and the OTP framework are core tools used in the implementation of the distributed Haai cluster.

The example program is explained in more detail in the use case chapter. The same program or reactor can be run many times. The distributed setup we make exists out of a number of elixir nodes that all send there messages to one sound server. The bigger the distributed network is the more messages that are send to the sound server. The sound server it self has virtually no limit to receive messages. The main parts of the distribution are taken care of by the underlying Beam virtual machine. One main reason to develop the Hvm on this platform is to be able to distribute in a solid and proven way.

4.3 Actor model

The actor system in Erlang implements the processes actor model. (De Koster et al., 2016) These actors are modeled as processes and can define there receive primitive to specify what messages it responds to. Thanks to the isolated turn principle the actor model guarantees to be free of data races and deadlocks by design. In Erlang actors are typically used when you need to encapsulate state and behavior within a single unit of computation., actors are spawned using constructs like GenServer, Task, or Agent within a single node. All actors communicate with each other using message passing within the same node. One can use these actors for modeling concurrent and stateful components in an application.

4.4 Distributed Erlang mechanisms

Distributed Erlang mechanisms come into play when you need to communicate between actors residing on different nodes within a cluster. Nodes communicate with each other using distributed Erlang protocols, allowing actors on different nodes to exchange messages. Distributed Erlang mechanisms provide fault tolerance, distribution transparency, and location transparency across the cluster.

Distributed Erlang provides location transparency, allowing actors to communicate regardless of their physical location within the cluster. This transparency hides the underlying network details, promoting loose coupling (Carreton et al., 2010) between nodes and actors. Actors on different nodes can interact seamlessly through message passing, without needing to be aware of the specific nodes they are communicating with. Loose coupling in distributed Erlang enables the system to scale horizontally by adding or removing nodes without significantly impacting the overall architecture.

Chapter 5

Distributed reactive melody generator

The use case of this thesis makes sound. The distributed Haai cluster, existing out of many distributed reactors provides the sink values from 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.

FIGURE

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 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.

5.0.1 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 super collider 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.

5.0.2 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.

5.0.3 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 eighth 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.

Further work

Try other distribution models

Conclusion

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>
- Carreton, A. L., Mostinckx, S., Cutsem, T. V., & Meuter, W. D. (2010). Loosely-coupled distributed reactive programming in mobile ad hoc networks. In J. Vitek (Ed.), *Objects, models, components, patterns, 48th international conference, TOOLS 2010, málaga, spain, june 28 - july 2, 2010. proceedings* (pp. 41–60). Springer. https://doi.org/10.1007/978-3-642-13953-6_3
- De Koster, J., Van Cutsem, T., & De Meuter, W. (2016). 43 years of actors: A taxonomy of actor models and their key properties. *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, 31–40. <https://doi.org/10.1145/3001886.3001890>
- Juric, S. (2024). *Elixir in action*. Simon; Schuster.
- Oeyen, B., De Koster, J., & De Meuter, W. (2022a). 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.
- Oeyen, B., De Koster, J., & De Meuter, W. (2022b). 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.
- 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.