UCL
Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

Lasp on Grisp: Implementation and evaluation of a general purpose edge computing system for Internet of Things.

Dissertation presented by
Alexandre Carlier , Igor Kopestenski , Dan Martens

for obtaining the Master's degree in
Computer Science and Engineering

Supervisor
Peter Van Roy

Readers
Ramin Sadre, Étienne Rivière , Adam Lindberg , Igor Zavalyshyn

Academic year 2017-2018

Declaration of Authorship

We, Alexandre CARLIER, Igor KOPESTENSKI, Dan MARTENS declare that this thesis titled "Lasp on Grisp: Implementation and evaluation of a general purpose edge computing system for Internet of Things." and the work presented in it are our own. we confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where we have consulted the published work of others, this is always clearly attributed.
- Where we have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely our own work.
- We have acknowledged all main sources of help.
- Where the thesis is based on work done by us jointly with others, We have made clear exactly what was done by others and what we have contributed ourselves.

Signed:

Date:

UNIVERSITÉ CATHOLIQUE DE LOUVAIN

Lasp on Grisp: Implementation and evaluation of a general purpose edge computing system for Internet of Things.

Abstract

Alexandre CARLIER - Igor KOPESTENSKI - Dan MARTENS

The cumulus needs to become stratus. In other words, the current cloud computing infrastructure needs to dissipate and relegate some of its workload to nodes that are closer to clients, at the edge of networks. This requirement is a direct consequence of the exponential growth of the amount of Internet of Things (IoT) devices. Eventhough this need for a new edge computing paradigm is acknowledged by all, there is currently no implementation of it and IoT applications still resort to traditional designs.

One of the biggest challenges for a production grade edge model is outperforming cloud datacenters for general purpose applications. Edge networks can imply both high heterogeneousness and unreliability, and nodes can be severely resource constrained.

LightKone[1] is an European project that aims at solving this issue by bringing a novel extension to the cloud, where all levels starting from edge nodes coalesce their efforts to alleviate the burden of the layers above them.

By combining synchronization-free programming and hybrid gossip algorithms, LightKone introduces a design for general purpose tasks at the edge. This model turns the 50 billions of additional IoT devices that are predicted for 2025 into supplementary resources for the cloud. They will be able to provide similar services as currently available cloud computers, but directly on the edge.

In this context, we propose GrispLasp, a prototype implementation of an edge node that embodies LighKone's vision of an edge device. The key objective of this work is to provide a concrete general purpose edge network, and evaluate its overall efficiency and performance. We have implemented a Lasp distributed application framework running on GRiSP boards. It is augmented with a task model that enables generic computations and distributed storage through Lasp.

We have made real-life deployments under the forms of wireless sensor networks and were able to successfully demonstrate that IoT sensor data gathered from the GRiSP boards could be aggregated, processed and stored directly in the GrispLasp edge.

Our prototype was thoroughly benchmarked against its counterpart, that would run identical applications but would rely on the services of Amazon EC2 instances for storage and processing. Our first results indicate that this new programming paradigm could alleviate the hardness of building applications on the edge. We expect that future edge nodes become more powerful and confirm our results, and increase scalability for generic edge computations.

[1]lightkone.eu

Contents

Declaration of Authorship iii

Abstract v

1 Introduction 1
- 1.1 The myth of the never-ending cloud . . . 1
- 1.2 A statement of the problem . . . 1
- 1.3 Implementation and evaluation of the proposed solution . . . 1
  - 1.3.1 Porting Lasp on GRiSP . . . 2
  - 1.3.2 The generic Task model . . . 2
  - 1.3.3 Evaluation . . . 2

I Edge Computing Foundations 3

2 Related Work 5
- 2.1 Context . . . 5
- 2.2 Technical Background . . . 5
- 2.3 Strong Eventual Consistency . . . 6
- 2.4 Edge and Fog computing . . . 6
- 2.5 Internet of Things . . . 7
  - 2.5.1 Data aggregation . . . 7
- 2.6 Classic cloud application . . . 7

3 Methods Used 9
- 3.1 Requirements for the prototype model . . . 9
  - 3.1.1 Hardware . . . 9
    - 3.1.1.1 GRiSP boards . . . 9
  - 3.1.2 Software . . . 9
    - 3.1.2.1 Erlang/OTP . . . 9
    - 3.1.2.2 Lasp . . . 10
    - 3.1.2.3 Algorithms and data structures : CRDTs . . . 10
- 3.2 Methodology . . . 11
  - 3.2.1 Overview of the approach . . . 11
  - 3.2.2 Research for qualitative and quantitative aspects . . . 11

II Experimental Framework: Prototype Edge Model 13

4 Edge IoT Devices 15
- 4.1 Introduction . . . 15
- 4.2 GRiSP board . . . 15
  - 4.2.1 Specifications . . . 15
    - 4.2.1.1 CPU . . . 15

4.2.1.2 Internal memory . . . 15
4.2.1.3 External memory . . . 15
4.2.1.4 Network . . . 15
4.2.1.5 User Interface . . . 15
4.2.1.6 I/O . . . 16
4.3 The GRiSP environment . . . 16
4.3.1 Introduction . . . 16
4.3.2 GRiSP Erlang Runtime Library . . . 17
4.3.3 Rebar3 build tool . . . 17
4.3.4 Rebar plug-in for GRiSP . . . 18
4.3.4.1 Build provider . . . 18
4.3.4.2 Deployment provider . . . 18
4.3.5 GRiSP toolchain . . . 19
4.4 Deployment . . . 19
4.4.1 Introduction . . . 19
4.4.2 Deployed application overview . . . 20
4.4.2.1 Release . . . 20
4.4.2.2 BEAM file . . . 20
4.4.3 Post deployment boot . . . 20
4.5 Sensor Devices . . . 20
4.5.1 Introduction . . . 20
4.5.2 Sensor integration for GRiSP . . . 21
4.5.2.1 Data exchange between the sensor and the application 21
4.5.2.2 How to implement a sensor driver? . . . 22
4.5.3 Pmod MAX Sonar . . . 23
4.5.4 Pmod Nav . . . 23
4.5.5 Pmod Als . . . 23

5 Edge Network layer 25
5.1 Introduction . . . 25
5.2 GRiSP Networking layer . . . 25
5.2.1 Infrastructure mode vs Ad-Hoc mode . . . 25
5.2.1.1 High membership dynamics/scalability . . . 25
5.2.1.2 Area covering . . . 26
5.2.1.3 Reliability . . . 26
5.3 Setting Ad-Hoc mode on GRiSP . . . 26
5.4 Partisan and membership management . . . 27
5.4.1 Introduction . . . 27
5.4.2 Gossip protocols . . . 28
5.4.3 Plumtree . . . 28
5.4.4 HyParView . . . 28
5.5 Porting Partisan on GRiSP . . . 29

6 Edge Application Layer 31
6.1 Lasp programming language . . . 31
6.1.1 Introduction . . . 31
6.1.2 Lasp architecture . . . 31
6.2 Lasp computation model . . . 32
6.2.1 Data structures . . . 32
6.2.1.1 Conflict-free replicated data types . . . 32
6.3 Lasp API . . . 33

6.3.1 Join two nodes . . . 33
6.3.2 Cluster membership . . . 34
6.3.3 Declare a set . . . 34
6.3.4 Update a set . . . 34
6.3.5 Query a set . . . 34
6.4 Porting Lasp on GRiSP . . . 34
6.4.1 Introduction . . . 34
6.4.2 Launching the Lasp programming system . . . 35

7 Experimental implementation of an edge computation model 37
7.1 Edge Computation Model . . . 37
7.1.1 Introduction . . . 37
7.1.1.1 Presentation of the Edge Computation model . . . 37
7.2 Distributed application design . . . 37
7.2.1 Preamble . . . 37
7.2.2 Additional security concerns . . . 38
7.2.3 Architecture overview . . . 38
7.2.3.1 Properties . . . 38
7.2.3.2 Supervision tree . . . 38
7.2.3.3 Node subsystem . . . 40
7.2.3.4 Edge node design pattern . . . 42

8 Task Model 45
8.1 Introduction . . . 45
8.2 Overview of the Task Model . . . 45
8.2.1 Task Model portrayal . . . 45
8.2.2 Task data structure . . . 45
8.2.3 High order generic functions . . . 46
8.2.3.1 Anatomy of the Task Framework . . . 46
8.2.3.2 Task Framework Load Balancer . . . 47
8.2.3.3 Task Framework API . . . 47
8.3 Example Applications . . . 48
8.3.1 Meteorological Task . . . 48
8.3.2 Generic MAX Sonar Sensor Task . . . 48
8.3.2.1 Detection algorithm . . . 48

9 Edge Server and Dashboard 51
9.1 Introduction . . . 51
9.2 Local Edge Server . . . 51
9.2.1 Overview . . . 51
9.2.2 Local Edge Server API . . . 51
9.3 Dashboard . . . 52
9.3.1 Introduction . . . 52
9.3.2 Overview . . . 52
9.3.3 Dynamic sensor data display . . . 53
9.3.4 Layout of the Dashboard . . . 53

10 Engineering of the GrispLasp infrastructure 55
10.1 Preamble . . . 55
10.2 Interlevel design . . . 55
10.3 Wireless network frequency . . . 56

10.4 Memory limitations . . . 56
10.4.1 Analysis of the Erlang memory allocators . . . 57
10.4.2 Tuning the BEAM memory allocation . . . 57
10.4.2.1 Memory allocation strategy . . . 58
10.4.3 Avoiding large memory segment cache size . . . 59
10.4.4 Forcing garbage collection manually . . . 59
10.4.5 Persisting and flushing Lasp CRDTs . . . 60
10.4.5.1 CRDT flushing . . . 60
10.4.5.2 Loading persisted CRDT's . . . 60

III Evaluation **63**

11 Evaluation of two computing models **65**
11.1 Introduction . . . 65
11.2 Computing Environments . . . 65
11.2.1 Lasp on Edge . . . 65
11.2.2 Lasp on Cloud . . . 66
11.3 Measurements methodology . . . 66
11.3.1 Meteorological task . . . 66
11.3.2 Computations at the edge . . . 67
11.3.2.1 Environment Setup . . . 67
11.3.2.2 Execution flow of the task function . . . 67
11.3.2.3 Measurement process . . . 68
11.3.3 Computations on the cloud . . . 71
11.3.4 Environment setup . . . 71
11.3.4.1 Cloud Server . . . 71
11.3.4.2 Cloud Infrastructure . . . 72
11.3.4.3 Measurements setup . . . 72
11.3.4.4 Execution flow for xcloudlasp . . . 73
11.3.4.5 Execution flow for cloudlasp . . . 73
11.3.4.6 Measurement process . . . 73
11.4 Experiments . . . 74
11.4.1 Experiments at the edge . . . 74
11.4.1.1 Experiment Workflow . . . 74
11.4.2 Results . . . 75
11.4.2.1 GrispLasp . . . 75
11.4.2.2 CloudLasp . . . 79
11.5 Comparing the Edge and Cloud computing models . . . 82
11.5.1 The big four . . . 82
11.5.1.1 Cost . . . 82
11.5.1.2 Availability . . . 83
11.5.1.3 Speed . . . 83
11.5.1.4 Latency . . . 83
11.5.1.5 Energy Consumption . . . 84
11.5.2 Complementarity of the models . . . 84
11.6 Conclusion . . . 84

12 Conclusions **85**
12.1 Results . . . 85
12.1.1 GrispLasp . . . 86
12.1.2 CloudLasp . . . 86
12.2 Future Work . . . 86
12.2.1 GRiSP . . . 86
12.2.1.1 Additional sensor integration . . . 86
12.2.1.2 Performance and memory enhancements . . . 86
12.2.2 Task model . . . 87
12.2.3 Local Edge Server and Dashboard . . . 87
12.2.3.1 Regression analysis . . . 87
12.2.3.2 Local decision making . . . 87

A Source code and User's Manual **89**
A.1 Pmod MAX sonar . . . 90
A.2 Pmod ALS driver . . . 92
A.3 UART Erlang driver . . . 94
A.4 UART C driver . . . 95

B Node core modules source **97**
B.1 Node storage utility . . . 97
B.2 Node Generic Task Server . . . 100
B.3 Node Generic Task Worker . . . 102
B.4 Node Generic Task Functions . . . 106
B.5 Node Benchmark Server . . . 110
B.6 Node Generic Task Functions Benchmark . . . 111
B.7 Node Ping Worker . . . 114

Bibliography **117**

List of Figures

2.1 Cloud application . . . 7

4.1 GRiSP board . . . 16
4.2 Dependency tree . . . 18
4.3 SPI modes . . . 22

5.1 Fanout of 3 . . . 28

6.1 Lasp architecture . . . 31

7.1 Supervision structure . . . 40
7.2 GRiSP Node . . . 41
7.3 Improved Supervision Overview . . . 42
7.4 Edge Node . . . 43

9.1 DashBoard . . . 53
9.2 DashBoard . . . 54

10.1 Edge Levels . . . 55
10.2 2.4GHz Wireless Network Channels . . . 56
10.3 BEAM Memory Allocators . . . 57
10.4 GrispLasp Stack . . . 58

11.1 Monotonic Read on the cardinalities of node's CRDTs . . . 69
11.2 Convergence Acknowledgement Process . . . 70
11.3 Cloud infrastructure . . . 72
11.4 Update mean time . . . 80
11.5 CPU utilization . . . 80
11.6 Network Bandwidth . . . 81
11.7 Convergence time . . . 81

List of Tables

8.1 Generic Server API 47
8.2 Generic Worker API 48

10.1 Visual representation of the eheap allocator and its allocated blocks. . . 59

List of Abbreviations

API	**A**pplication **P**rogramming **I**nterface
BEAM	**B**ogdan/Björn's **E**rlang **A**bstract **M**achine
BSP	**B**oard **S**upport **P**ackage
CSP	**C**loud **S**ervice **P**rovider
EC	**E**ventual **C**onsistency
ERTS	**E**rlang **R**un **T**ime **S**ystem
GERL	**G**RiSP **E**rlang **R**untime **L**ibrary
IoT	**I**nternet **O**f **T**hings
IT	**I**nformation **T**echnology
PMOD	**P**eripheral **MOD**ule
SEC	**S**trong **E**ventual **C**onsistency
SoC	**S**ystem **o**n **C**hip
SPA	**S**ingle **P**age **A**pplication
UART	**U**niversal **A**synchronous **R**eceiver **T**ransmitter
WSN	**W**ireless **S**ensor **N**etwork

Chapter 1

Introduction

1.1 The myth of the never-ending cloud

Since the first commercial appearances of cloud-based services, providers have rivaled to find superlatives to advertise the unlimitedness of their infrastructures. However, potential clients are seldom told that behind the vague notion of cloud, there are complex systems that may face anomalies like any other. After all, *the cloud is just someone else's computer*. In this chapter, we discuss how emerging devices put severe strain on providers and the traditional cloud model, and that solutions are under active research.

1.2 A statement of the problem

In a context of pandemic proliferation of IoT devices[47], CSPs are aware that the current cloud computing model is not sustainable much longer. However, since the number of these devices has still not reached a critical treshold and that the increase is still at its very beginning, the need for a new model in production does not prevent the growth of traditional cloud infrastructures[11]. Also, there is still no research that has developed a new model that is both cost-efficient and at least as performant as the current one that could also handle the considerable forthcoming growth.

Hence, the global state of the art has not reached sufficient knowledge to provide an alternative that will be vital for all the stakeholders that are *in the clouds*. The purpose of this work is to complement the work of the LightKone project[21], and its unique approach to offload cloud datacenters. These are currently mainly responsible for storage and processing of cloud applications, and will require adaptions to handle billions of new IoT devices and their data.

1.3 Implementation and evaluation of the proposed solution

In this sense, we propose GrispLasp, an Erlang prototype deployment of an edge device that demonstrates how a share of the work can be done at the peripheral extreme of a network. More importantly, our implementation is embedded with a task model that does not limit the edge nodes to perform *a* part of the work, but **any** work. Following LightKone's vision of the constitution of the future cloud infrastructure[44, 45], we demonstrate how the Lasp library and its communication protocol[33, 35, 34] can provide general purpose computations and distributed storage directly on the light edge.

1.3.1 Porting Lasp on GRiSP

The light edge encompasses devices such as smartphones, laptops and any kind of IoT devices. As opposed to the heavy edge[45], where there can exist intermediate points of presence that are not in the spectrum of user devices anymore. In this work, we have ported the Lasp software only on the light edge, with a particular focus on GRiSP boards[20] although without excluding computers and laptops.

We have made deployments of wireless adhoc networks of GRiSP boards running the GrispLasp framework, such that each node could be able to cooperate with its pairs. Additionally, we have not excluded the ability for other devices running Erlang applications to work with GrispLasp nodes. Lasp comes with the Partisan communication protocol that we have used in replacement of the default distributed Erlang solution. It implements HyParView[27], a membership protocol that we have used to construct a virtual overlay over the physical network. This provides each node with a clusterized view of the network, while it can only directly communicate with its direct neighbors.

1.3.2 The generic Task model

We propose and implement a generic task model aiming to provide a mechanism for managing and orchestrating distributed tasks in an edge network comprised of GRiSP boards with the use of Lasp CRDT's [34]. Tasks are units of computations that rely on Erlang high order generic functions. The latter are a powerful means of abstraction that we use to enable devices running our framework to execute virtually any possible types of computations at the edge.

1.3.3 Evaluation

We examine in detail how the GrispLasp prototype performs in real-world setups, and evaluate it on several aspects to investigate use cases where our edge prototype could be a better choice over a regular cloud setup. Thus we use time metrics to be able to determine what use cases could favor our prototype over a regular cloud solution where GRiSP boards perform very few of the work.

The study of these evaluations aspires to bring for closing the current knowledge gap that prevents actual large-scale edge networks from being deployed.

The rest of this dissertation is organized in three main parts. First, we provide the foundations of Edge Computing. Part II introduces all the building blocks of our experimental framework, starting from the hardware level and going upwards through the application design, to end with the engineering considerations for real-world deployments. Part III displays all our evaluation process and its outcome, and proposals for further improvements of our prototype to expand the body of knowledge in edge computing.

Part I

Edge Computing Foundations

Chapter 2

Related Work

2.1 Context

During the early 2000s, companies were relying increasingly on IT infrastructures in order to establish links between separate local networks [18]. By doing so, they were able to decrease operational costs over time as the productivity gains yielded by software tools would surpass the expenses of building and maintaining the underlying infrastructures.

Nonetheless, companies with insufficient funds or individuals willing to found a business were unable to provide the substantial investment necessary for IT infrastructures. Moreover, there is no guarantee that the estimates made beforehand will match the actual resource requirements. Companies would expose themselves to unnecessary overhead or inadequate infrastructures, ultimately defeating the purpose of investing in the first place [53].

In 2006, Amazon launched its Elastic Compute Cloud [5], followed by the Google App Engine in 2008 [22]. These were the two first industrial grade cloud computing platforms that offered companies the ability to benefit from IT services without owning expensive infrastructures. The cloud computing business model introduces the notion of *pay-as-you-go*, meaning that clients are charged only for what they use when they use it. This approach is more cost-effective in contrast to owning a private infrastructure since upfront investment implies that there is no elasticity [45].

Since the beginning of the decade, cloud providers are facing a new challenge to keep providing the same guarantees and quality for their services: the Internet of Things (IoT)[45]. Cisco dates the *birth* of IoT between 2008 and 2009, when the device to human ratio became higher than 1 for the first time i.e. more devices connected to the Internet than humans on Earth [17]. This increasing ratio forces service providers to adapt their cloud computing model such that it can scale and still maintain high availability and low latency everywhere.

2.2 Technical Background

The following thesis work relies on hardware and software kindly provided by the LightKone project [1]. The global approach consists in combining synchronization-free programming and hybrid gossiping to provide a generic edge model [45]. The authors state that there is currently no solutions for general-purpose computations

[1] https://www.lightkone.eu/

on large edge networks, which makes research in that direction particularly relevant.

2.3 Strong Eventual Consistency

Maintaining consistency has shown to be a key challenge in planetary scale IoT applications. Since cloud applications using storage services must always be provided with low latency, data is globally geo-replicated to make it steadfastly available to all clients. This introduces new challenges for preserving the consistency property of data as multiple replicas can store conflicting operations.

Eiger [29] is a framework proposed as a scalable solution for geo-replicated storage. It is stated that experimental results show little to no overhead compared to other approaches, while providing a richer data model than a simple key-value store with low latency and ensuring *causal consistency*. Cassandra is another novel system introduced by Facebook [24]. It has been successfully tested on datasets containing entries from more than 100M Facebook users, but authors of [29] argue that only certain configurations ensure *stronger-than-eventual* consistency.

The *eventual* aspect of the SEC property relates to the convergence of all nodes towards the same state. [43] introduce Conflict-free Replicated Data Types (CRDT) that guarantee eventual consistency by nature. [34] provide an implementation of a distributed application with the Lasp library, where the Lasp State-based Observe-Remove Set that also relies on join-semilattice theory described by [43] is presented with its proof of eventual consistency.

Basho Technologies have also introduced an implementation of CRDTs in Riak [41]. But it has been shown that the first implementations were suffering performance issues at scale [9]. Bigsets [8, 9] have subsequently been engineered by Basho to overcome these issues for set CRDTs, as these were the ones affected.

2.4 Edge and Fog computing

In 2012, the need for a new cloud computing paradigm was assessed by [6]. It is argued that IoT applications bring non-trivial requirements that cannot be met in the long term by the current cloud computing model. Edge computing refers to the concept where computing resources, data storages or services in general are available closer to the data sources. The difficulty resides in the fact that the devices connected at the edge of the network often lack in connectivity and have low energy available. This decentralized computing model can be placed at different levels of the network. The two following approaches implement the edge computing model at different levels of the network.

The *Fog Computing* model is situated between the cloud infrastructures and the sensor devices; it serves as gateway for the edge devices to communicate with the cloud servers. It is used as computing power and data storage for pre-aggreated data received from the sensor network.

Another proposed approach that goes even further in offloading the cloud is *Mist Computing*, and relies on moving the processing at the very edge of the network directly on IoT devices. It has already been adopted in industry cases and has

shown encouraging experimental results [25, 31, 39]. For example in hard real-time embedded systems, it is really useful to make local decision.

2.5 Internet of Things

The Internet of Things refers to the network of all physical devices that are connected and allows them to share data. The current number of IoT devices is estimated at 23.14 billions in 2018 [2]. It is expected to keep growing exponentially in the next years.

2.5.1 Data aggregation

Data generated on IoT devices at the edge can be stored at the edge for a limited interval but aggregates and processed data can be forwarded to the cloud. It is argued that in most cases aggregates are sufficient for majority of applications, and that raw data processing in data centers could be avoided [45].

2.6 Classic cloud application

Let's introduce the architecture of a "classic" cloud based application using the Amazon Web Services (AWS) which stores and analyzes data in the cloud.

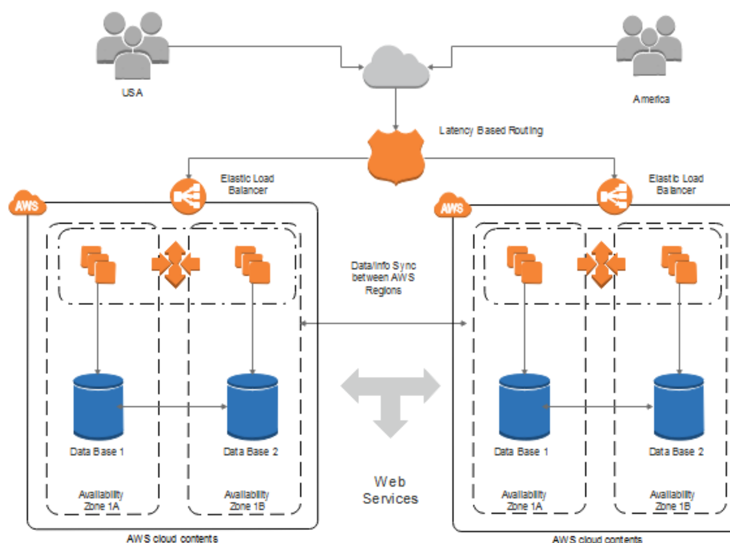


FIGURE 2.1: Cloud application

In a current classic cloud architecture, everything is done on the cloud. Clients from all around the world send data that needs to be analyzed. The data are sent to a region where several EC2 instances are running and can scale up or down depending on the traffic load. These instances apply computation on the data received, for instance the computation could be an aggregation of the data. Once the data has been aggregated, it is stored in one of the available form of database on the cloud. The scalability of these EC2 instances is their main force, as you may not know in

[2] https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/

advance the traffic load needed for your application, the cloud infrastructure is a good solution to follow your needs without having to invest in long term hardware.

Chapter 3

Methods Used

3.1 Requirements for the prototype model

In this section we aim to provide the main requirements of the system created for this thesis. This list of requirements is non-exhaustive, as the goal here is not to detail all the features but rather to give the reader an idea of what the system consists in and what main goals have been achieved. The second section will explain how the requirements were filled by the system.

3.1.1 Hardware

3.1.1.1 GRiSP boards

The GRiSP boards kindly provided by LightKone's partner will serve as a starting point for our edge devices.

3.1.2 Software

3.1.2.1 Erlang/OTP

Erlang is a concurrent, functional programming language developed by Ericsson which was released as an open-source software in 1998. OTP stands for Open Telecommunication Protocol and was a proprietary language for Ericsson before they released the open source version: Erlang/OTP where OTP is the name for the Erlang standard library. Erlang was originally designed to support the development of telephony applications and was used by Ericsson itself for the purpose of programming Ericsson cellular data carrier nodes. It is used nowadays to create highly scalable, distributed, fault-tolerant and concurrent applications requiring very high uptime. It can be found on machines going from embedded systems to large, performant servers. Erlang is often used by companies that require systems that can scale to millions or even billions of users.

The Erlang runtime system has the following desirable properties:

- Distributed
- Fault-tolerant
- Concurrency
- Soft real-time
- Robustness

- High availability
- Code hot swapping
- Worker-supervisor message-passing model
- Garbage collection

Erlang uses the "Actor Model" where each actor is a separate, concurrent process running in a virtual machine. These concurrent processes are extremely lightweight and can be spawned in very large quantities. Moreover, processes communicate with each other using asynchronous message passing and share no resources.

Erlang is a functional programming language and thus uses the Functional Programming paradigm where functions and operations are executed in a similar fashion to mathematical calculations. A function or operation takes one or more inputs and returns an output value after evaluation. Executing the latter twice will always give the same result. This type of programming, called Declarative Programming, is useful because it eliminates side effects (i.e the program does not change state or use mutable data) and makes it easier to understand and predict the behavior of programs. The opposing programming paradigm is Imperative Programming in which functions are able to modify the program's state during execution meaning that multiple executions of the same program using the same input values can lead to different results depending on the program's state during execution. One of the main aspects of Functional programming is that high order functions are possible, an high order function is a function that can take one or several other functions as parameter. This allows us to use functions like *Map* or *Filter* which can be especially efficient when working with sets or lists. Let's give an example of high order function to demonstrate the power of functional programming:

```
1 Double = fun(X) -> 2*X end,
  Result = map(Double,[1,2,3,4,5]),
3 Result = [2,4,6,8,10]
```

As a function can be stored in a variable, it is easier to be able to use several times, and it makes the syntax clearer.

3.1.2.2 Lasp

The Lasp set of libraries is a cornerstone in our implementation. It gives part of its name to our prototype as it makes for a half of its core, that is the distributed part.

3.1.2.3 Algorithms and data structures : CRDTs

We use Conflict-free Replicated Data Types, a form of datatypes specifically designed to abstract the notion of concurrent operations and conflicts when developping distributed applications.

3.2 Methodology

3.2.1 Overview of the approach

Our stance towards analyzing the overall performance of our prototype is mainly quantitative. Using time metrics, we attempt to determine how it behaves compared to its counterpart on an EC2 instance deployment.

3.2.2 Research for qualitative and quantitative aspects

We perform calculations for the most part, but we also provide a fair qualitative factor for some aspects, as the deployment complexity or the global impressions that we have for future work on the current model.

Part II

Experimental Framework: Prototype Edge Model

Chapter 4

Edge IoT Devices

4.1 Introduction

4.2 GRiSP board

In the following sections, a brief summary on the materials employed is given. The below mentioned hardware and software components used are mandatory requirements of the thesis work and thus were not subject to change. A detailed specification of the GRiSP board components is given hereafter.

4.2.1 Specifications

The central piece of hardware used in the project is the **GRiSP-Base board**. The GRiSP board is a small and compact circuit board, i.e System-On-Chip, consisting of a built-in 802.11 b/g/n WLAN antenna, a Micro USB serial port for supply and external connectivity and a MicroSD socket for standard MicroSD Cards. The board also features a series of I/O connectors and pins which can host a variety of Digilent Pmod^{TM} peripheral modules.

4.2.1.1 CPU

The GRiSP board includes an Atmel SAM V71 (SAMV71Q21ES3) processor with an ARM Cortex M7 single core running at 300 MHz clock speed.

4.2.1.2 Internal memory

GRiSP has a 2048Kb flash memory which is used by the Boot loader as well as 384 Kb SRAM that is used as internal cache by the CPU.

4.2.1.3 External memory

GRiSP hosts a 64Mb SDRAM, a 2Kb EEPROM and has a built-in MicroSD Socket for standard MicroSD Cards.

4.2.1.4 Network

GRiSP has a built-in 802.11 b/g/n 2.4 GHz single-chip realtek RTL8188etv Wifi antenna.

4.2.1.5 User Interface

GRiSP has two RGB LEDs, 5 DIP switches and a boot/reset button key.

4.2.1.6 I/O

As aforementioned, the GRiSP board comes with seven connectors/interfaces that can host a variety of Digilent Pmod™ interface boards that allow functionality expansion of the GRiSP by providing environment sensing and actuating capabilities. Pmod devices are developed and produced by Digilent which is a leading electrical engineering products company [1].

The details of the built-in connectors on GRiSP are provided below:

- A Dallas 1-Wire via 3-pin connector
- A Digilent Pmod compatible I2C interface
- Two Digilent Pmod Type 1 interfaces (GPIO)
- One Digilent Pmod Type 2 interface (SPI)
- One Digilent Pmod Type 2A interface (expanded SPI with interrupts)
- One Digilent Pmod Type 4 interface (UART)

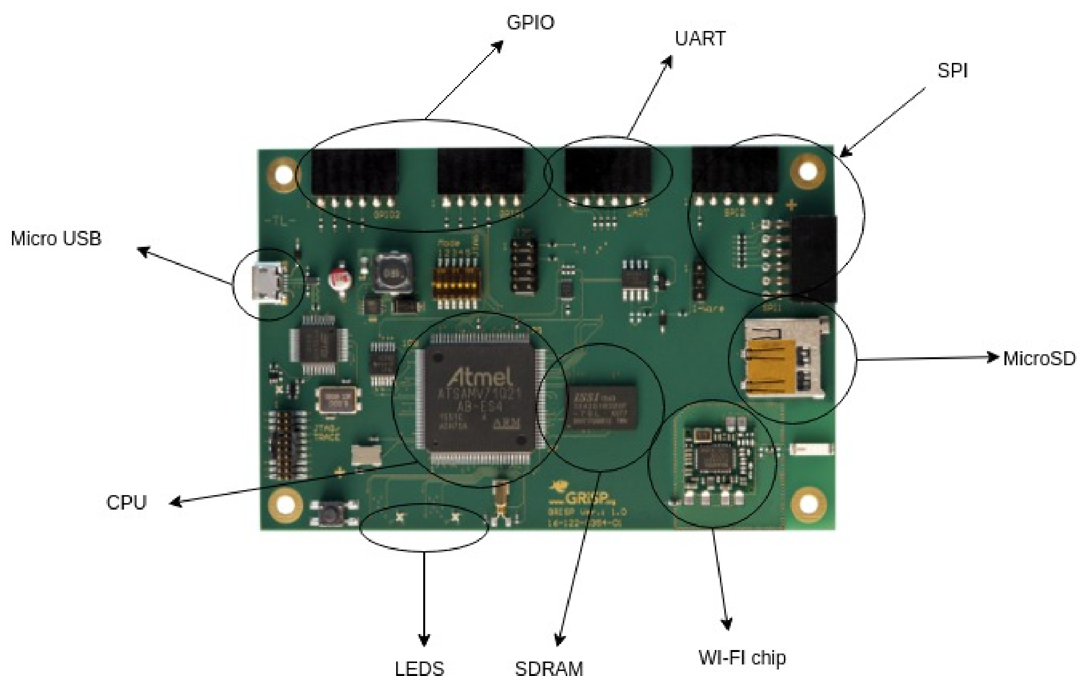


FIGURE 4.1: GRiSP board, reprinted from grisp.org

4.3 The GRiSP environment

4.3.1 Introduction

The section below provides an in-depth apprehension of the GRiSP development environment. It consists in 3 main components that are required in order to deploy applications on the boards, thus a crucial part for porting Lasp on GRiSP.

[1] https://store.digilentinc.com/about-us/

4.3.2 GRiSP Erlang Runtime Library

The **GRiSP Erlang Runtime library** (GERL) is an Erlang application which provides an interface for communicating with the hardware of the GRiSP board. The library consists of two layers of drivers.

The first layer is composed of C low-level drivers that communicate directly with the RTEMS(Real Time embedded military system) OS API which provides a bridge to interact with the actual hardware. These drivers are typically interfaces to different hardware protocols that are present on the module ports of the GRiSP board. The GERL currently implements C drivers that interface with the SPI, GPIO, I2C and UART protocols.

The second layer is composed of Erlang high-level drivers that provide hardware abstraction. These drivers are used to create a binding between the high-level and the low-level drivers. Moreover, the purpose of the high-level drivers is to send and receive messages to and from the low-level drivers using a simple Erlang interface and ultimately for the programmer to focus on building applications using those abstractions.

In practice, the C low-level layer typically deals with sending and receiving binaries to and from the hardware module ports while the Erlang high-level layer is used to process these binaries and offer an interface that the programmer can use. Additionally, the GERL also holds configuration, script and system files that are patched to the custom Erlang/OTP application of GRiSP when executing the Rebar3 GRiSP plugin, as described in Section 4.3.4.

4.3.3 Rebar3 build tool

Rebar3 is the build tool used to build, compile and deploy Erlang/OTP applications on GRiSP boards. It provides a standardized structure for Erlang/OTP applications that enhances interoperability between projects. Upon creation, applications built using rebar3 will contain a `rebar.config` configuration file at the root of the project. It provides an easy way to add Lasp as a dependency inside a GRiSP application. Snippet 4.1 shows the necessary dependency configuration for GRiSP applications that require the Lasp libraries.

```
1 {deps, [
    grisp,
3   {epmd, {git, "https://github.com/erlang/epmd", {ref, "4d1a59"}}},
    lasp
5 ]}.
```

SNIPPET 4.1: Including Lasp as a dependency of a GRiSP application

Once the dependencies are resolved, the dependency tree obtained when executing the `rebar3 tree` command is the in Figure 4.2. `grisplasp` is the top-level project, the first-level dependencies are the ones that were explicitely stated, and their nested requirements. Nevertheless, it has been observed that this procedure was only sufficient when the target environment is a regular computer. In fact, the configuration does not include the source dependencies once the applications are deployed to GRiSP boards. Hence, the Lasp modules are not successfully ported to the destination SD card and will be unusable. Section **??** describes how this issue can be resolved.

```
└─ grisplasp─0.1.0 (project app)
   ├─ epmd─1.0.0 (git repo)
   ├─ grisp─1.1.0 (hex package)
   │  └─ mapz─0.3.0 (hex package)
   └─ lasp─0.8.2 (hex package)
      ├─ gen_flow─0.0.5 (hex package)
      ├─ lager─3.6.2 (hex package)
      │  └─ goldrush─0.1.9 (hex package)
      ├─ lasp_support─0.0.3 (hex package)
      ├─ partisan─1.3.3 (hex package)
      │  └─ acceptor_pool─1.0.0-rc.0 (hex package)
      ├─ plumtree─0.5.0 (hex package)
      ├─ rand_compat─0.0.3 (hex package)
      ├─ time_compat─0.0.1 (hex package)
      └─ types─0.1.8 (hex package)
```

FIGURE 4.2: Porting Lasp on GRiSP as a dependency.

4.3.4 Rebar plug-in for GRiSP

The `rebar3_grisp` plug-in extends rebar3 with commands for building and deploying GRiSP applications. Rebar3 plug-ins rely on the concept of providers, that are Erlang modules executed upon rebar3 command-line calls [10].

4.3.4.1 Build provider

The `rebar3_grisp_build` provider offers the ability to build a custom Erlang/OTP system that will be shipped on GRiSP boards [40]. This provides a custom version that integrates features of the GRiSP Erlang runtime making it possible to interface with GRiSP hardware. It is called with the following command line at the root of a GRiSP application folder :

```
rebar3 grisp build
```

4.3.4.1.1 Patching OTP The build provider is first responsible of patching the Erlang/OTP `Makefile.in` configuration file. The latter is an output of the `automake` software used to automate parts of the compilation process. GRiSP drivers are referenced inside it with the help of a git patch if the `git-diff` software detects their absence.

4.3.4.1.2 Building OTP The final step of the provider is to regenerate the configuration scripts for Erlang/OTP using the `otp_build` utility function that performs the adequate calls to the `autoconf` software. After executing the automake and autoconf scripts, the provider has generated the necessary input files for a call to `configure`. It will produce the `Makefile` that will allow Erlang/OTP to be installed and copied into the GRiSP project.

4.3.4.2 Deployment provider

The deployment provider implements the second feature of the rebar3 GRiSP plug-in by combining the previously built Erlang/OTP system and user written applications. Upon a call of the following :

```
rebar3 grisp deploy -relname <release name> -relvsn <release
                        version>
```

It will build a release of the application specified by its name and version and copy it on the target SD card altogether with the custom Erlang/OTP system and is further explained in Section 4.4.

4.3.5 GRiSP toolchain

The main components of the GRiSP toolchain are a cross compiler, a linker and a set of librairies.

4.3.5.0.1 Cross compiler The cross compiler is able to produce executables for various architectures, and makes it possible to execute GRiSP applications both on computers and GRiSP boards. Since GRiSP boards and embedded systems seldom need a compiler, nor have sufficient resources to support one, the applications must be compiled on a more powerful development environment then ported on the target system.

RTEMS' features can be ported on numerous different CPU architectures, but it must know beforehand about the rest of the hardware that is on the embedded system. A **Board Support Package** is a set of libraries that allow operating systems to work for specific embedded systems by providing access to hardware-specific features.

4.3.5.0.2 Linker script RTEMS excepts a configuration for the linker used to produce executables for the target platform. The GRiSP toolchain contains a script that specifies the board's memory configuration e.g. what region will be allocated for the bootloader [28].

4.3.5.0.3 Libraries Software and libraries that are involved in developing applications for GRiSP are added as modular components of the toolchain. It includes the `libgrisp` library that contains `C` programs for interfacing with the EEPROM or LEDs on the board, and an initialization program. The toolchain also contains a `rtems-libbsd` library that replaces the built-in FreeBSD networking module in the RTEMS kernel [42].

4.4 Deployment

4.4.1 Introduction

The deployment of an Erlang application on a GRiSP board is the process of generating and copying a **release** of the application along with startup scripts and more importantly the **BEAM.bin file** which contains the binary code of the BEAM virtual machine, into an SD card. The deployment starts once the "rebar3 grisp deploy" command is called, as explained in the previous section on the Rebar3 Grisp Plugin 4.3.4.

4.4.2 Deployed application overview

4.4.2.1 Release

A release is composed of a set of applications and libraries that are needed in order to start and execute the Erlang application on the board in an Erlang Virtual Machine which is commonly referred to as Erlang Run Time System (ERTS). [48] By default, a release includes boot files and scripts that are used by ERTS to start the application as an Erlang node on the board.

4.4.2.2 BEAM file

A BEAM file (which has the .beam suffix) is a virtual machine running compiled Erlang code. It is usually composed of both the Erlang/OTP runtime system as well as the application made by the developer. In the case of GRiSP, after deployment, the SD card will contain a grisp.ini file that specifies the path to the beam.bin executable on the SD card. This file is not a standard BEAM file but instead is generated by the compiler on the programmer's development environment and contains binary machine code which can be executed directly by the GRiSP board.

When the BEAM file is compiled during the execution of the "rebar3 grisp build" command, it is also cross-compiled to the RTEMS ARM architecture. This essentially means that the BEAM file will run the RTEMS OS kernel in the same executable file that will run the Erlang/OTP runtime system and the application. In addition to that, the BEAM will also contain the GRiSP Runtime which will communicate directly with the RTEMS kernel to send and receive messages to and from the GRiSP module ports.

4.4.3 Post deployment boot

Once the application has successfully deployed on the SD card, the latter can safely be inserted in the MicroSD socket of the GRiSP board which will subsequently start its boot loader upon pressing the power button of the GRiSP board. The boot loader will initialize and mount the SD card. If successful, it will try to read the grisp.ini configuration file from the root directory of the SD card. This configuration contains the boot image path to the beam.bin executable file and will be executed by the GRiSP board to boot the entire system.

4.5 Sensor Devices

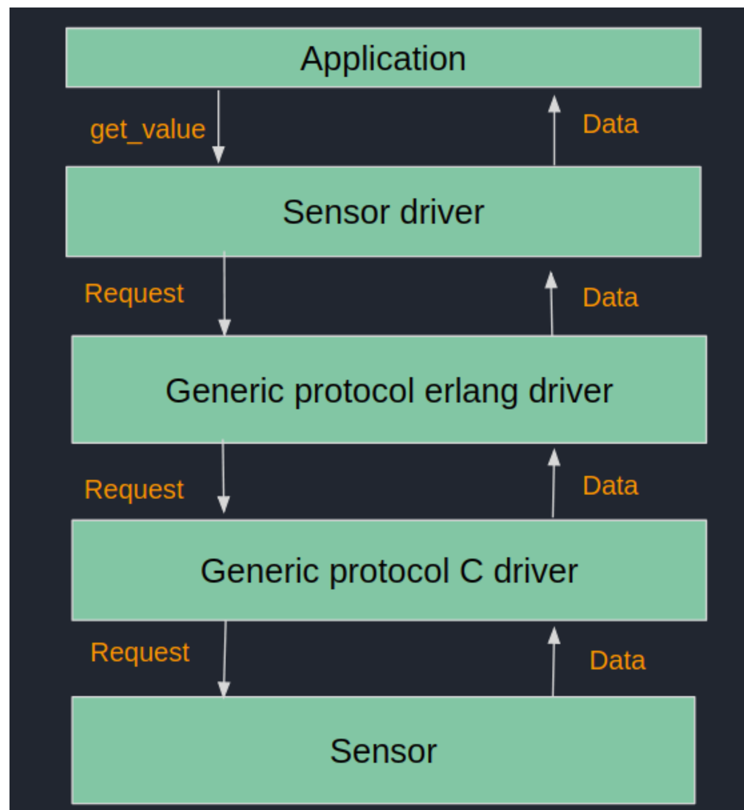
4.5.1 Introduction

With the different interfaces it is possible to plug-in several sensors on the same board as long as the sensor is set on the correct interface to be able to communicate with the GRiSP board. The interfaces are not completely generic because they only allow sensors of type *Pmod* [2] to be connected. Pmod stands for "Peripheral module", it is an open standard that has been designed by Digilent Inc.[3]
There is a large variety of sensors from simple push buttons, to complex multi-touch LCD screens. The following section details how the sensors are integrated in the GRiSP boards and what their functions are in the application.

[2] https://en.wikipedia.org/wiki/Pmod_Interface
[3] https://store.digilentinc.com/

4.5.2 Sensor integration for GRiSP

Since the only way to interact with the GRISP board is by using the Erlang virtual machine, it is needed to be able to communicate with the sensors through an Erlang file. This is what the sensors' drivers are for, they give a high-level API to retrieve data from the sensors without having to understand how it works in the lower layers. The following scheme shows the sensor architecture.



Let's explain this architecture by starting from the bottom of the picture. As mentioned earlier there are different connectors on the GRISP boards, for each connector there is a generic C driver that establishes the hardware communication between the sensor and the board. Once the information has been read by the board the data needs to be sent one layer above, to the Erlang virtual machine, to allow the application to have access to the data.
The data is first transmitted to a generic Erlang driver that corresponds to the same interface and then is transmitted to the driver corresponding to the specific sensor.
A practical example is shown to clarify this explanation. As an example, we will demonstrate the *Pmod MAX_Sonar*, a detailed description of it will be given later:

4.5.2.1 Data exchange between the sensor and the application

1. The *Pmod MAX_Sonar* uses the UART interface to communicate with the GRiSP board. This means that the data is extracted from the sensor using the generic C UART drivers which corresponds to a file called "grisp_termios_drv.c" A.4, this file simply uses the correct C functions to read the sensor registers following the UART protocol.

2. Once the data has been read, it is sent to a generic Erlang process, called "grisp_termios_drv.erl" A.3 which is generic for all the sensors using the UART protocol.

3. From there the data is sent to the specific driver for the sensor which is called "pmod_maxsonar.erl" A.1, where the data pulled from the sensor can be processed in the correct way. In most cases a sensor can be configured and needs to be calibrated, this configuration is done by writing values to specific register on the sensor. The data exchange flow explained in this chapter can also be reversed and come from the sensor driver to the sensor itself to configure the sensor.

4. The drivers often implement a "read" function which is called by the application to output the data in a graph for instance.

4.5.2.2 How to implement a sensor driver?

Writing a high level sensor driver still requires some hardware knowledge. Even if the generic Erlang drivers provide an API to communicate with the sensor, the user needs to know what and where to communicate with the sensor. A Pmod sensor can be seen as a set of registers with their own respective addresses. Each register has a specific function, either it stores data generated by the sensor,like temperature of pressure, or data to change the configuration of the sensor. Let's go through the creation of the driver for the Pmod ALS sensor which only has one register. By reading the user guide [4], we can figure out enough information to know how to use the sensor.

First the protocol used by the sensor is SPI. SPI stands for "synchronous peripheral interface", it is a synchronous serial communication interface used for short distance communication. As mentioned in the user guide the register is read-only, so let's use the generic read function provided by the Erlang SPI driver to communicate with this register:

$$grisp_spi : send_recv(slot, mode, req, skip, pad)$$

- slot: the slot corresponds to where the sensor is plugged in the GRiSP board, it can be either SPI1 or SPI2.4.1
- mode: This is the tricky part when using SPI, the protocol has 4 different modes corresponding to different combinations of clock polarity and phase.

SPI mode	Clock polarity (CPOL/CKP)	Clock phase (CPHA)	Clock edge (CKE/NCPHA)
0	0	0	1
1	0	1	0
2	1	0	1
3	1	1	0

FIGURE 4.3: SPI mode

- req: This contains the request, the value of the bytes that will be sent on SPI interface, the request usually has the following format «operation,message,address»,

[4] https://reference.digilentinc.com/pmod/pmod/als/user_guide

where the operation is either read or write, the message contains the bits sent on the interface and the address corresponds to where to read or write the message.

- skip: This is a parameter to skip a part of the response bits as they may not be useful and contain information.
- pad: The padding is useful to fill in the request with ones.

The following information in the user guide help us understand how to fill the parameter of the function to be able to retrieve the data:

- The on-board ADC, which we are communicating with via the system board, is set up so that data is "clocked out" on the falling edge of the serial clock and is "clocked in" or "read" by the system board just before or on the subsequent rising edge of the serial clock.
- The data itself comes in a set of 15 bits the first 3 bits are leading zeros, bits 4-11 are the information about the light level with the MSB first, and bits 12-15 are trailing zeros.

The first sentence indicates which SPI mode to use, as the data is "clocked out" at the falling edge, the clock polarity is high, so $CPOL = 1$ and the data is read before the rising edge, the clock phase is "trailing", $CPHA = 1$, by looking at the figure 4.3 here above, the SPI mode is the last one. The second information explains how to parse the response to retrieve the data, to receive 2 bytes of data we need to send 2 bytes of data, but since it is one register that is read-only the operation parameter is not needed and the address neither. Using Erlang it is easy to parse binary information using pattern matching, as the request will send back 2 bytes we can use the following line to parse the useful information.

$$<< _ : 3, Resp : 8, Pad : 5 >>= grisp_spi : send_recv(spi1, 3, << 0 : 8 >>, 0, 1)$$

The complete version of the driver is available in the appendix A.2.

4.5.3 Pmod MAX Sonar

As mentioned before this sensor communicates with the GRiSP board using UART, it is an ultrasonic range detector with one-inch resolution. A practical example of how it can be used will be given at section 8.3.2.

4.5.4 Pmod Nav

The Pmod Nav stands for navigation as this sensor is able to give accelaration, angular rate, magnetic field and the pressure. It uses the SPI interface to communicate with GRiSP.

4.5.5 Pmod Als

ALS stands for Ambient Light Sensor, this sensor is able to give a 12-bit output value of the luminosity, the driver had to be created to use it. It also uses the SPI interface.

Chapter 5

Edge Network layer

5.1 Introduction

This section will explain the different components of the network layer. We will first explain why and what network topology was the best to fit the required conditions to respect the edge computing model. We will also present and explain the different distributed applications that enable our system to be reliable and fault-tolerant.

5.2 GRiSP Networking layer

Here are some challenging aspects for Edge computing that need to be taken into account when choosing the topology of the network:

1. high membership dynamics/scalability
2. Area covering
3. Reliability

The goal was to find the network architecture that fits best for the different challenges announced above. The GRiSP boards come with a built-in 802.11 b/g/n 2.4 Ghz WiFi antenna. For the WiFi, there are two possible modes that can be configured: *infrastructure* and *Ad-Hoc or Peer-to-Peer*. Let's compare those two modes by using the criteria defined earlier.

5.2.1 Infrastructure mode vs Ad-Hoc mode

Infrastructure mode is the classical Wi-Fi architecture that can be found in most houses. The router is the centralized access point through which all the messages of the network come and go. Whereas in Ad-Hoc mode it is decentralized, the devices communicate directly with each other if there are near enough, or they multi-hop if the receiver of the messages is too far. Let's compare those two modes for each criteria:

5.2.1.1 High membership dynamics/scalability

High membership dynamics means that the topology of the network changes a lot. A lot of nodes join or leave the network at all time and some nodes even become unavailable. If the number of devices connected simultaneously surpasses the available connection slots on the router then no extra devices will be able to connect. Which means that for the infrastructure mode we need to know in advance the number of nodes in the cluster to know how much hardware is needed, whereas with the

Ad-Hoc mode the network can scale up and down easily without needing extra hardware.

5.2.1.2 Area covering

Using the infrastructure mode the area covered by the access point is limited to the maximum distance the router can transmit a message to. As the system can have nodes moving away from each other it should be possible to keep the communication up even if the nodes are far from the router. With the Ad-hoc mode this problem can be solved since a node can keep the communication up as long as it is near enough from just one node of the cluster.

5.2.1.3 Reliability

The system should keep working as long as one node keeps running in the cluster. With an infrastructure architecture if the router fails then the whole system fails since any other node is not able to communicate anymore, the router is a single point of failure. By setting a Peer-to-Peer topology the nodes will always be able to communicate with the neighbors as long as they are close enough to each other.

With the arguments above it is easy to see that the Ad-Hoc topology is a better fit for the Edge computing model.

5.3 Setting Ad-Hoc mode on GRiSP

In their original configuration the GRiSP boards use the infrastructure mode and communicate with each other using a central router as explained earlier. To be able to use the boards in the Ad-Hoc configuration, changes needed to be made in the C code and not in the Erlang code, more precisely in the erl_main.c file. The code snippet hereunder illustrate the changes made to the file to allow the GRiSP boards to connect to an Ad-Hoc network.

```
create_wlandev(void)
{
  int exit_code;
  char *ifcfg[] = {
                    "ifconfig",
                    "wlan0",
                    "create",
                    "wlandev",
                    "rtwn0",
                    "wlanmode",
                    "adhoc",
                    "channel",
                    "6",
                    "up",
    NULL
  };

  char *ifcfg_adhoc_params[] = {
                    "ifconfig",
                    "wlan0",
                    "inet",
                    ip_self,
                    "netmask",
                    "255.255.0.0",
                    "ssid",
                    "edge",
                    NULL
  };
```

Let's explain the different parameters of this configuration. This C code represents a FreeBSD command executed by the board, the command is the following:

```
ifconfig wlan0 create wlandev rtwn0 wlanmode adhoc channel 6 up
```

This FreeBSD command is used to configure the network interface. The goal is to create a cloned interface called *rtwn0*, which is activated in the ad-hoc mode, using the channel 6 (2,437 Ghz) to communicate. When the interface has been created it needs to be configured by assigning an IP address to the board, a netmask and a SSID for the network name. This is done using the following command.

```
ifconfig wlan0 inet ip_self netmask 255.255.0.0 ssid edge
```

Of course all the board need the same configuration to be able to communicate except for the IP address which is unique for each board. The *ip_self* value is set in another file called *grisp.ini.mustache*. This file is transfered to the SD card during the deployment of the application.

5.4 Partisan and membership management

5.4.1 Introduction

The underlying networking layer of Lasp involves a multitude of protocols that rely on each other to disseminate messages to nodes across the network and to retain and repair the network's membership in the event of failures. In this section, an overview of the different algorithms that the Lasp framework relies upon is given. The distribution layer used is called Partisan [33], it supports different network

topology, for the Peer-to-Peer toopolies the backend is built upon two main algorithms: *Plumtree*[26] and *Hyparview*[27], both are hybrid gossip algorithms

5.4.2 Gossip protocols

Gossip protocols, also called epidemic protocols, are computer-to-computer communication protocols that are inspired by the way that gossips or rumors are spread in a societal environment or how an epidemic disease spreads in a population. Upon the reception of a message for the first time each node will forward this message to a number of neighbors chosen randomly, this number is called the *fanout* factor.

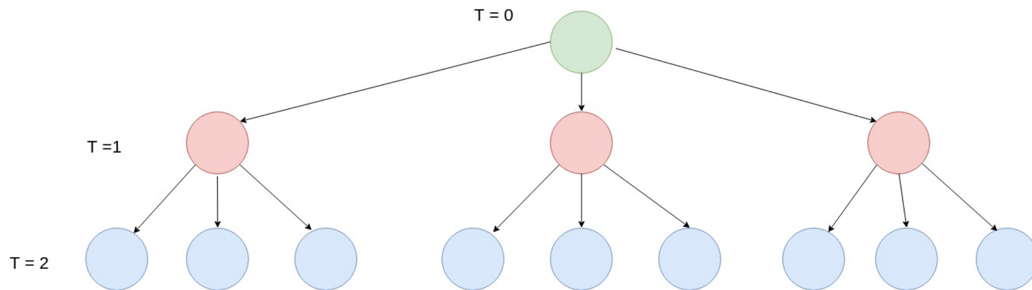


FIGURE 5.1: Fanout of 3

Gossip have many advantages especially for distributed applications. It is scalable since the time complexity for a message to reach all the nodes will be $\mathcal{O}(\log_f n)$ where f is the fanout factor. Gossip algorithms are also fault-tolerant, as long as a node is not isolated and is connected to several nodes, there will be a high probability of receiving the messages. In distributed applications, robustness is an important feature

5.4.3 Plumtree

The idea of Plumtree is to combine both a tree-based and gossip algorithm to take advantages of both type to achieve low message complexity and high reliability. Plumtree main goal is to build a spanning tree with all the nodes of the network and keep this spanning tree up to date even in case of nodes failure or network partitions. Due to the high number of nodes that may constitute the cluster, it is often hard for gossip protocols to scale up because of the cost of maintaining a complete view of the system. For that matter, Plumtree relies on *Partial views* in order to keep a scalable solution. A partial view consist on a small subset of nodes that are selected as gossip neighbors. The downside of using partial views is that if a large number of nodes fail the partial view will be severely damaged and it may create a lot of network partitions, to solve this issue we introduce an algorithm in the next paragraph called *Hyparview*.

5.4.4 HyParView

As explained previously a solution to implement highly scalable and resilient broadcast consist in using those partial views. But under a high rate of nodes failures this feature can be hard to maintain as it can take a long time to restore desirable properties for those views such as *degree distribution* and *clustering coefficient*. Hyparview uses two distinct types of partial views to solve this problem: *passive view* and *active view*

5.5 Porting Partisan on GRiSP

The Partisan source code had to be inserted as dependency when deploying the GRiSP application. This was done using the "_checkouts" feature of rebar3. This feature allows us to work locally on the application that are placed in the directory. Even if the dependency is still present in the rebar.config file the _checkout dir will take precedence over the same application. As Partisan is an Erlang library it contains a file called "partisan_sup" which is a supervisor process for the Partisan application. Once our application was deployed on the SD card, it was possible to call the Partisan functions, for instance to start the Partisan application it was possible to call: *partisan_sup:start_link()*. This supervisor will start all the other services needed for the proper running of Partisan.

Chapter 6

Edge Application Layer

6.1 Lasp programming language

6.1.1 Introduction

The Lasp programming language was designed to facilitate distributed application on a large-scale where the different nodes status can vary a lot between online and offline. The complexity of distributed application has been growing in the past few years as the number of connected devices is increasing exponentially. Implemented as an Erlang library, Lasp provides the ability to deterministically compose set into large scale computation.

6.1.2 Lasp architecture

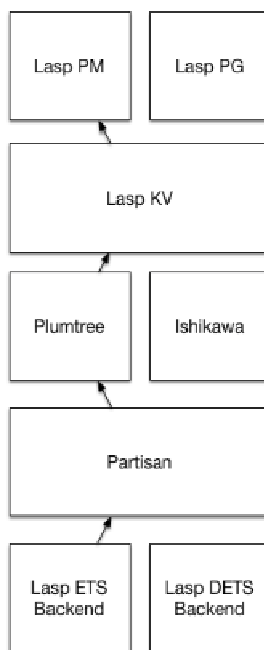


FIGURE 6.1: Lasp deployed stack architecture

The figure 6.1 here above shows the different layer that compose the Lasp programming language[36], here is a short explanation on their purpose. The most important one, Partisan, has been detailed in section 5.4:

1. Lasp PM: Stands for Lasp programming model, it is the higher level where client interact directly with the sets he has created. The API available for Lasp PM will be explained in more details later in the chapter.

2. Lasp KV: Stands for Lasp Key-Value, this is the CRDT based data store that provides full replication of the CRDT's inside the cluster of nodes.

3. Plumtree: It is an implementation of the plumtree algorithm which combines tree based and gossip protocols to achieve low message complexity and high availability more details are given in section 5.4.4).

4. Partisan: Partisan is the communication layer for Lasp, it is a distributed layer that supports several network topologies, and scales better then Distributed Erlang when the number of nodes exceed 512. [33]

5. Lasp ETS: This is the Erlang ets storage used to store all the variables locally in the Erlang VM.

6.2 Lasp computation model

Lasp's inherent goal of providing "a new programming model designed to simplify large-scale distributed programming" ([34]) comes with a number of challenges regarding the manner in which data is computed and distributed across nodes in the network but also how the network itself and its actors interact with each other. Improving from traditional distributed system computation models, the Lasp framework integrates a number of novel data structures and algorithms that are the subject of this section.

6.2.1 Data structures

The following subsection gives an overview of the data structures used at the heart of Lasp. Synchronization, consistency and replication of data across distributed systems being the foundation stone of any large distributed system, a careful selection of accordingly data structures and distributed programming models must be done.

6.2.1.1 Conflict-free replicated data types

Conflict-Free Replicated Data Types (CRDTs) are novel data types proposed by Shapiro et al. as part of a new formalism which resolves distributed data whose value are updated concurrently across a distributed system in a deterministic manner and without coordination between the replicas. This formalism defines a new property on data consistency in distributed systems, **Strong Eventual Consistency**, as a newer and better approach to **Eventual Consistency cite article here**. Eventual Consistency in distributed systems defines the case where different entities own a copy of some data object. Each replica i.e the entity itself, a node in a distributed system, is able to update its own copy of the data object locally and without immediate synchronization, possibly resulting in later conflicts where different entities have different values for a same data object if data was updated concurrently and once the updated data is sent to the other replicas. In Eventual Consistency, conflicts can arise, but may be arbitrated requiring a consensus between entities in order for them to agree upon the definitive value. When no more changes are applied to a particular data object for a certain period of time, all copies of that object will *eventually* converge to the same state on all entities.

Strong Eventual Consistency improves the previous consistency model by avoiding the complexity of conflict resolution. By avoiding conflict resolution, the Strong Eventual Consistency property hides the latter complexity to the programmer who does not depend on implementing the resolution logic himself anymore rendering the property highly desirable in distributed systems ([34]). To avoid conflict resolution, Strong Eventual Consistency leverages simple mathematical properties that ensure absence of conflict, while preservering safety and liveness properties of concurrent systems (cite article here). Resolutions of concurrent updates are done deterministically on each entities without the need of manual resolution by the programmer: given all updates to an object are eventually delivered in a distributed system, all copies of that object will converge to the same state ([34]).

In conclusion, CRDTs are data types that inherit the SEC property allowing updates on data objects to happen concurrently and independently without the need of explicit coordination between replicas i.e synchronization does not happen on data object updates. CRDTs support temporary divergence on each replica but guarantee that a given data object's value will eventually converge on all replicas after a certain period of time, monotonically and deterministically ([34]).

The GrispLasp prototype is a materialization of an edge node designed to work in a Wireless Sensor Network configuration. Since energy consumption is not a concern but as described in Section 10.4, we must address memory issues and therefore make adequate use of CRDTs. A noticeable improvement that has been implemented in GrispLasp since the updates that have been made to the GRiSP Runtime is the use of Lasp's **delta-based** dissemination protocol [35]. This propagation mode was causing unexpected behavior before and GrispLasp deployments could only achieve convergence if the state-based mode was used.

The main difference is that Lasp's delta-based dissemination decreases both the local storage requirements for CRDTs and the size of messages propagated over the edge network. In fact, Lasp uses δ-mutators that are merged with states on nodes, avoiding a monotonically increasing size in memory for CRDTs [2, 35].

6.3 Lasp API

Lasp delivers a high order API which allows the client to create a distributed system with full replicated data without having to understand or to set manually all the connections between the nodes, or keep their connections alive, all of that is done automatically by the Lasp framework. Here is a list of the main functions used for the thesis but it is possible to find the full API on (link to github).

6.3.1 Join two nodes

Before being able to share and update data in a distributed system all nodes forming the cluster need to connect to each other. This is done using **join** function of the Partisan library which is directly callable from Lasp:

$$lasp_partisan_peer_service : join('NodeName')$$

This will establish a connection between the two nodes. It is not joining just 'NodeName' but also all the other nodes that were previously connected to it.

6.3.2 Cluster membership

It is also interesting to be able to see which nodes are in the current cluster. This can be done using

$$lasp_partisan_peer_service : members()$$

. As mentioned earlier Partisan is a distribution layer which can be configured for several topologies. For the thesis it was more efficient to set the nodes in a peer-to-peer mode which means that the **partisan peer manager** used is *Hyparview*. When *Hyparview* is used the member function will not show all the connected nodes but only the ones in the active view of the current node. An explanation of Hyparview is given at section 5.4.4

6.3.3 Declare a set

Lasp allows the client to easily create its own set which will be fully replicated on all the nodes of the cluster, this is done using the following function:

$$lasp : declare(id, set_type)$$

. The **Id** is a tuple of two values:{name,type} where the name is a random atom of the client's choice and type needs to correspond to the **set_type** argument from the function.

6.3.4 Update a set

The update function allows the client to add or remove an element to the set:

$$lasp : update(id, operation, node)$$

. The update function can only add one element at the time in the set, but this element can be any data type. The id is the same as mentioned previously: a tuple of form {name,set_type}. The operations available are add or remove but they are not always available for all the different types of set; for instance a **state_gset** which stands for grow only set will not allow a remove operation on it.

6.3.5 Query a set

One other essential function that Lasp API provides is the ability to query the current state of a set, of course this will return the state of the set on the node where the function call is made. If the node is not yet up to date, this version of the set might not be the most recent one but will eventually converge to the most recent version.

$$lasp : query(id)$$

6.4 Porting Lasp on GRiSP

6.4.1 Introduction

Porting applications relying on the Lasp framework to GRiSP boards consists in being able to run them as scalable distributed programs with Lasp as a substitute for the Erlang disterl native framework. In contrast to disterl, Lasp is a set of libraries that is still under development and subject to change. This implies that a trade-off

exists between innovation and stability. This statement also holds for the GRiSP boards, and therefore obviously for the combination of both.

6.4.2 Launching the Lasp programming system

Nodes can be launched via a shell call to Rebar that creates an instance running Lasp, or using a function in the Lasp support module. The first option is unapplicable when using GRiSP boards, since there is no Unix-like shell because the boot procedure directly runs an Erlang shell. The other possibility is to rely on the Lasp support program to start nodes on each GRiSP host. It contains functions that are specifically designed to automatically create and operate a set of Lasp nodes, but those rely on a **runner node** that acts as a manager for the cluster. In terms of scalability, a large network of GRiSP hosts could not run Lasp using this approach since each node only communicates with in-range neighbors and none could manage the whole set.

Consequently, the proposed method works around this concern by including the Lasp and Partisan frameworks as top-level children in the root supervision tree. The module that is started once the boot process is over handles the creation of child supervisors from the Lasp programming system. By doing so, each GRiSP board is able to automatically start the Lasp suite at the beginning of it's execution process.

Chapter 7

Experimental implementation of an edge computation model

7.1 Edge Computation Model

7.1.1 Introduction

The following chapter introduces the *edge computing* model, sometimes also referred to as *fog* or *mist* computing [7, 32, 39].

7.1.1.1 Presentation of the Edge Computation model

The edge computing model can be viewed as an extension of the well-known cloud computing model that addresses specific use cases such as IoT applications [6].

7.1.1.1.1 Motivations The amount of IoT devices is expected to grow exponentially in the next few years and could go from 23 billions to over 75 billions by 2025 [47]. For certain types of cloud applications that provide services based on data retrieved from these IoT devices, such as urban services, this growth represents a risk of decreased service quality if not handled properly. Relying on the current cloud computing model would require guaranteeing availability and low latency to users regardless of their location, while the underlying infrastructure might not sustain such heavy workloads [45].

7.2 Distributed application design

7.2.1 Preamble

The following section describes the design of the experimental distributed application that has been built to run on GRiSP boards. In Section 4.5, we have provided an insight at the sensor data that could be collected using Digilent Pmod[38, 37] sensors. Since the GRiSP Runtime allows direct access to these measurements from Erlang, it opens the path to *Smart Home* and even *Smart City* applications. Several studies have already provided promising approaches for urban services relying on IoT[23, 51, 55], but they all stress the difficulty of treating heterogeneousness and unreliability.

We have implemented the GrispLasp distributed application framework as a program enabling any kind of generic computing, but the IoT nature of the GRiSP hardware combined with Pmod sensors makes it a perfect match for *Smart* urban

applications. These have been a starting point for the design described in this chapter.

7.2.2 Additional security concerns

IoT devices have introduced a new generation of cyberattacks[12], that requires a matching set of Security measures. Since IoT devices for Smart Homes and Cities collect and process data directly from our environment, it also implies that a security breach on these devices can lead to sensitive data being manipulated or misused. Also, attackers are able to perform perpetual analysis of vulnerable devices on the Internet[4], and forming armies of destructive botnets. Currently these aspects are not yet addressed in the GrispLasp prototype, neither at network nor application levels. But it is we do insist on the importance of security when it comes to edge nodes. More IoT devices forcibly means more possibles vulnerabilities. And *it should* also imply according measures. But this has been omitted from the prototype implementation due to time constraints and mostly as it is not the primary objective of this work.

7.2.3 Architecture overview

7.2.3.1 Properties

There are several properties that must be held at all times by the application.

- Fault tolerance: failures such as connectivity errors, hardware crashes and unhadled exceptions are considered as normal behavior. A distributed application running on WSN nodes should expect and manage high heterogeneity, unreliability and churn.
- Full replication: the storage must not only be distributed, but replicated on each node such that data is available with up to $N - 1$ nodes down, when there are N nodes.
- Eventual convergence: Each replica should eventually converge towards an identical state. There is no strict time constraints for convergence, but only a liveness property guarantee required on all nodes.

7.2.3.2 Supervision tree

There are 2 types of processes in an Erlang supervision tree, namely **supervisors** and **workers**. Supervisors are responsible for starting, stopping and monitoring their child processes [15]. These children can either be other supervisors or workers. *Workers* are children that perform tasks such as computations i.e. all processes but supervisors are referred to as workers.

Supervision trees can be built in a way that enables them to recover from failure, and thus hold the fault tolerance property. At first the GrispLasp application could not be deployed to SD cards and launched on GRiSP boards using the same boot sequence as the one on our computers. The first instances of successful deployments were achieved through integration of Lasp and Partisan supervision modules, that were orchestrated by a GrispLasp master supervisor as shown in Figure 7.1.

7.2.3.2.1 Fault tolerance In this initial release, the application was scinded into 3 subsystems that were all supervised by the **master** supervisor. These subsystems are Partisan, Lasp and Node and the first-level leaves of the supervision tree in Figure 7.1 correspond to their respective top-level supervisors. Hence, each independent component is specified as a permanent child process of the master supervisor. Snippet 7.1 shows an example specification for the node supervisor.

```
-define(NODE_SPEC,
  #{id => node_sup, start => {node_sup, start_link , []},
    restart => permanent, type => supervisor, % restarted in all cases
    shutdown => 15000, modules => [node_sup]}).
```

SNIPPET 7.1: First-level Supervisor specification

7.2.3.2.2 Supervision flags When an Erlang supervisor is initialized, it can be provided with flags that are applied during the restarting process of its children. The master supervisor is provided with the flags shown in Snippet 7.2 and a specific children starting order. As described in the Erlang documentation, `rest_for_one` strategy corresponds to the following behavior [15] :

> If a child process terminates, the rest of the child processes (that is, the child processes after the terminated process in start order) are terminated. Then the terminated child process and the rest of the child processes are restarted.

The order in which the subsystems are to be started is :

1. Partisan
2. Lasp
3. Node

When the master supervisor restarts one of its children from the list above, he will first terminate all the processes in the subsystems that are posterior siblings. Then the target child and terminated siblings are all restarted in the specified list order. This order guarantees that a subsystem is restarted only if necessary, since each list item can be restarted without affecting its anterior siblings.

But thanks to continuous development to GRiSP, Lasp and Erlang tools during our research, we were able to produce a second supervision model that was able to successfully run on boards without requiring direct manipulation of Lasp and Partisan's supervisors. The new model allowed us to specify the boot sequence on a per-application basis. It has enhanced the GrispLasp node with a more semantically valid supervision tree where each depending application uses its own specifications to provide the best restart strategy.

As a result, the distributed application design that we have used for the further development of GrispLasp allows much more coexisting supervision trees. Each one of these trees corresponds to an application, and the result matches a correct architecture of an OTP application. The improved supervision trees for the `node`, `lasp` and `partisan` applications is shown in Figure 7.3. The key difference is observable

with an overlay of the supervision trees view provided by `observer`. In the shown state, the report displays a variety of applications with their individual supervision trees. This has been a great improvement for the distributed application, as it allowed both simpler and more efficient process management.

```
init(all) ->
    SupFlags = #{strategy => rest_for_one, intensity => 1,
     period => 20},
    {ok, {SupFlags, [?PARTISAN_SPEC, ?LASP_SPEC, ?NODE_SPEC]}};
```

SNIPPET 7.2: Master Supervisor initialization function

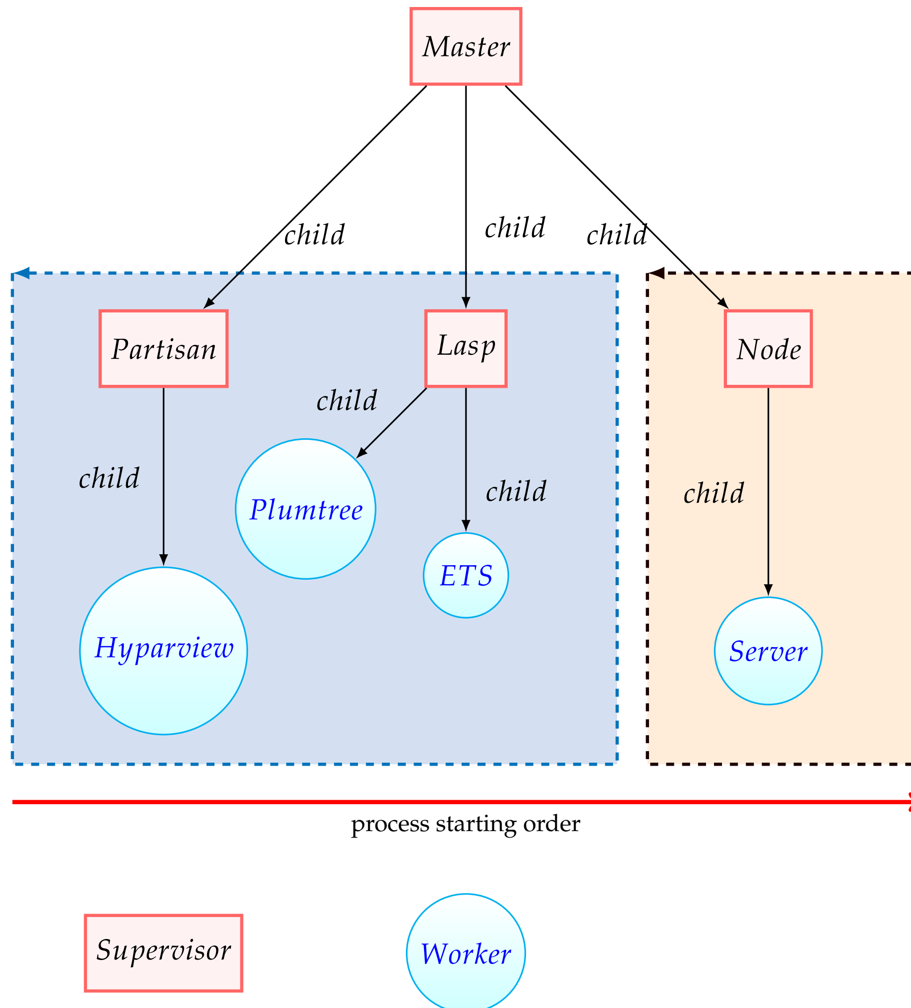


FIGURE 7.1: High-level supervision overview of a node.

7.2.3.3 Node subsystem

The node subsystem is the set of modules that are designed to handle all the processing that can be executed locally, regardless of network connectivity. This design principle ensures that nodes can continue functioning in case a node is isolated, and thus contribute to fault tolerance.

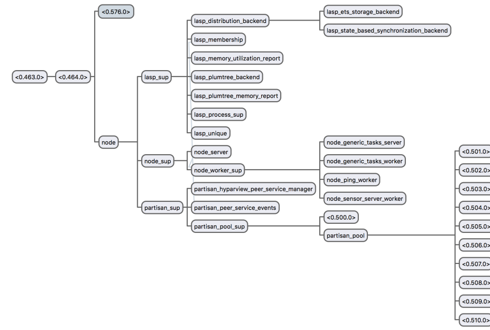


FIGURE 7.2: Supervision tree of the node application process.

7.2.3.3.1 Node server The node server is an implementation of the Erlang `gen_server` behavior. The latter is an interface that exposes functions for a client-server model [14]. *Clients* are any other modules that send requests to the server using the Erlang native functions. The server holds a key position in the node as it handles requests for spawning workers. Also, the server initializes a supervisor for workers as its sibling in the supervision tree. This supervisor will be used to manage all workers spawned on clients' requests. Snippet 7.3 demonstrates how requests are handled in the node server. Upon receiving a call for a specific worker type, the server will lookup for the corresponding specification. If the worker is found the new instance is started as a child of the worker supervisor and the new PID is added to the set of running workers. Otherwise, a reply indicating that the worker type was not found is returned instead of the PID is sent back to the request originator. Figure 7.2 is the representation of the full supervision tree that can be visualized using the Erlang `observer` module. It can be observed that the `node_server` process has built a subset of the supervision tree where the worker supervisor is its only sibling who is managing all the workers.

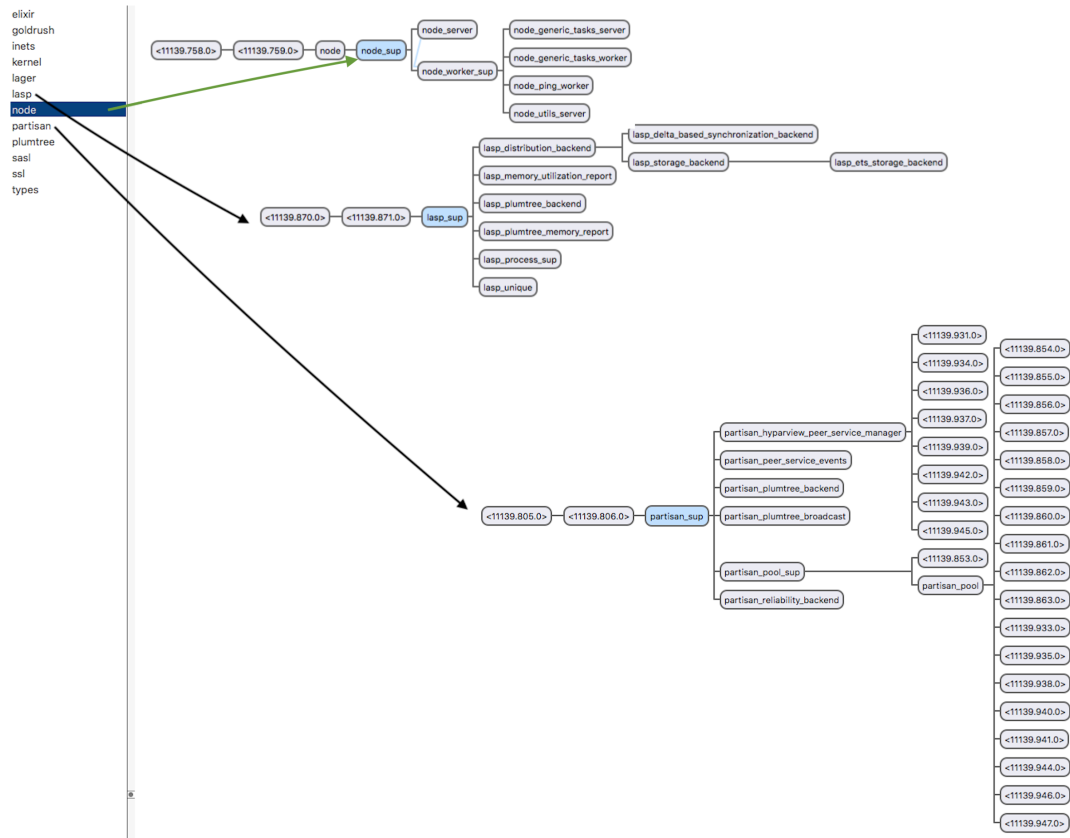


FIGURE 7.3: Supervision trees of Node, Lasp and Partisan applications on a GrispLasp node.

```
handle_call({start_worker, WorkerType}, _From,
      S = #state{worker_sup = WorkerSup, workers = W}) ->
    case maps:get(WorkerType, get_worker_specs_map()) of
      {badkey, _} ->
        {reply, {badkey, worker_type_not_exist}, S};
      ChildSpec ->
        {ok, Pid} = supervisor:start_child(WorkerSup, ChildSpec),
        Ref = erlang:monitor(process, Pid),
        {reply, {ok, Pid}, S#state{workers = gb_sets:add(Ref, W)}}
    end;
```

SNIPPET 7.3: Worker spawning request handling

7.2.3.4 Edge node design pattern

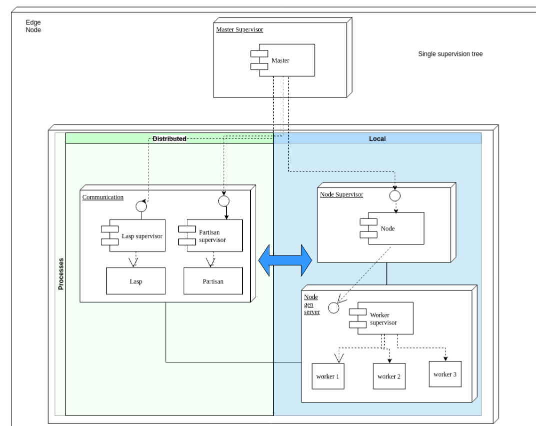


FIGURE 7.4: Design pattern for edge nodes.

Chapter 8

Task Model

8.1 Introduction

Edge applications running on IoT devices often need to execute a variety of different tasks concurrently. A **task** may been seen as an independent subprogram (or process) that is heterogeneous by nature due to the fact that the goal it aims to achieve is always unique. In edge applications and cloud computing in general, a task often needs to be executed by multiple nodes in a cluster. Therefore, the task needs to be distributed on the all the nodes that are required to execute it. An important aspect of distributed tasks is the manner in which they are managed and orchestrated. In this section, we describe an experimental implementation of a task framework and explain how the latter helps in providing a way to manage and execute distributed tasks in a cluster of IoT devices.

8.2 Overview of the Task Model

8.2.1 Task Model portrayal

The Task Model is an abstraction for managing distributed tasks in a cluster of nodes and is implemented as a Erlang high order framework which is referred to as **Task Framework**. Each node in the cluster has the ability to use the framework when required as it is included in the node's application by default. As tasks need to be distributed and replicated on all the nodes in the cluster, an efficient solution is to use Lasp CRDT state-based Observed-Remove Sets [1]. Tasks are thus stored in CRDT's and are eventually replicated on other nodes after performing the anti-entropy round. Nodes are then able to retrieve tasks from the CRDT and to subsequently execute them.

8.2.2 Task data structure

In the framework, a task is an Erlang tuple comprised of 3 elements:

- Name: The name of the task
- Targets: An Erlang list containing the names of the distributed Erlang nodes that are subjects to executing that particular task. A task can be executed by all the nodes or by a subset of nodes.
- Function: The actual computational unit, an Erlang function, that will be executed by a node.

[1] `https://github.com/lasp-lang/types/blob/master/src/state_orset.erl`

This tuple is the main data structure that will be stored in Lasp CRDTss and replicated on all the nodes that are part of the cluster. This offers resilience in the case that one or more nodes become unreachable or defunct.

8.2.3 High order generic functions

Erlang, being a functional programming language, offers the ability to use what is referred to as **high order functions**. A high order function is a function comprised of variables which are functions themselves. The latter also has the ability to pass a function as a parameter which can in turn be used as a variable inside that function. These types of functions are a powerful means of abstraction as one single function is able to self-contain an entire program.

The function defined in a task is itself a high order function. We complete the term by adding the word "generic" to indicate that the function's inherent parameters are tailorable depending on the requirements and needs of the node executing the task. In essence, a function of a task is altogether complete, self-contained and customizable.

8.2.3.1 Anatomy of the Task Framework

The Task Framework consists of three co-depent entities which are found in three different Erlang modules:

- **Task Server**: The Task Server can be seen as a module implementing all the CRUD operations for tasks. Possible operations include adding, removing and finding tasks. Each operation uses the Lasp API to query or update CRDT's. When updating a CRDT, the newly updated value will eventually converge on all the nodes that are part of the cluster. Additionally, the function which takes care of adding a task is function-agnostic meaning that the node is able to pass any valid Erlang function as the function of the task.

- **Task Worker**: The Task Worker module is responsible for starting tasks on a node. This module can be compared to a *supervisor* for the execution of tasks. It holds a list of running and finished tasks in order to avoid starting a task multiple times. Starting a task consists in a four step operation:

  1. The first step is to find one or more tasks depending on the node's needs by querying the Task Server.
  2. The second step filters the task(s) and verifies whether or not the node that wants to start a task is a possible candidate for executing the latter. This verification procedure is done by checking if the node is indeed present in the target list of the task and also if that task is not currently running on the node.
  3. The third step extracts the generic function from the task tuple and spawns a new Erlang process which will in turn run the generic function. Moreover, when the process is spawned, its execution is monitored by keeping a reference to the agent monitoring the process. Whenever the spawned process terminates for any reason (because of a crash or in the best case because the process terminated gracefully), the monitor agent notifies the

Function	Arguments	Description
add_task	{Name, Targets, Fun}	Adds the task tuple {Name, Targets, Fun} to the tasks CRDT
remove_task	TaskName	Removes the task named {TaskName} from the tasks CRDT
remove_all_tasks	nil	Removes all the tasks from the tasks CRDT
get_all_tasks	nil	Fetches all the tasks from the tasks CRDT
find_task	Name	Finds the task named {Name} from the tasks CRDT

TABLE 8.1: Generic tasks server API functions.

Task Worker module and the task is subsequently removed from the running tasks list.

4. The fourth and final step is acknowledging that the task has started executing and is then added to the list of running tasks.

- **Generic Task functions**: The Generic Task functions module contains premade generic high order Erlang functions that can be added and used directly in the Task Framework. As stated before, the Task Server can accept any kind of Erlang function that will be inserted in the Task data structure.

8.2.3.2 Task Framework Load Balancer

A cluster of nodes may consist of nodes running on various types of hardware each having different computing capacity. In this regard, and to avoid overloading nodes, the use of a Task load balancer comes naturally as a pragmatic approach to solve this problem. The Task Framework incorporates a load balancer who's responsibility is to control the load on nodes that are part of the cluster. The number of running tasks as well as the current CPU usage are used as control variables and are checked each time a node is willing to start a new task. A threshold on the number of running tasks is defined based on the computing capacity of the node's hardware. In the experimental Edge model prototype, there are two different types of nodes: GRiSP boards and laptops. Both of these types of devices take part in the execution of tasks in the cluster. The computing power of a laptop is orders of magnitude higher than the one of a GRiSP board and thus has a higher threshold on the allowed number of running tasks. Moreover, tasks requiring a greater amount of computing power are automatically assigned to laptop's as their host targets.

8.2.3.3 Task Framework API

Table 8.1 and Table 8.2 describe the API calls that resp. provide access to capabilities of the generic tasks server and worker modules.

Function	Arguments	Description
`start_task`	`Name`	Starts the task named **`Name`**
`find_and_start_task`	`nil`	Fetches and starts a random task from the tasks CRDT
`start_all_tasks`	`nil`	Starts all tasks in the tasks CRDT
`is_running`	`TaskName`	Returns true if the task named **`TaskName`** is already running

TABLE 8.2: Generic tasks worker API functions.

8.3 Example Applications

8.3.1 Meteorological Task

This task will be thoroughly described in Chapter 11

8.3.2 Generic MAX Sonar Sensor Task

Using the Pmod Max sonar it is possible to detect movement in range from 0 to 255 inches (about 6 meters). An example application using this sensor would be to be able to count the number of people coming in and out of a room. Since the sensor is only able to detect movement but not direction, two GRiSP boards are needed to figure out if people are coming in the room or leaving the room.

8.3.2.1 Detection algorithm

The issue was to cover as much different use cases as possible, since there is no way to identify the person passing through the system, by using RFID for instance, a smart way of using timer was the best way to go. The two boards use Erlang processes to communicate with each other, here is the pseudo-code for the task:

```
Process A:
if sensor_see_something()
    -> send message saw_something(in/out) to other board

Process B:
receive saw_something -> timer 2s
if sensor_see_something() -> update count
else -> drop message
```

To fit in the task model explained previously those two processes need to be started in a single task which can be started and stopped at any moments as long as the boards have a Pmod Max sonar.

Boards need to be designated as in or out to be able to tell the direction of the person passing through the system. The **sensor_see_something()** function is more complex than it seems. Indeed, let's take a case where a message is sent every time the sensor sees something closer than a certain threshold, as long as the person stands in front of the sensor it will keep sending message to the other board. Since Erlang process store messages in a mailbox, the other board will treat every new message

as a person passing through, so this is not a possible solution.

```
fun sensor_see_something(List,Threshold) ->
if read_sensor < Threshold -> NewList = add _to_list(read_sensor,List),
    sensor_see_something(NewList,Threshold)

else
  if head(List) < Threshold && size(List) > 5 -> trigger message, NewList
    = [],sensor_see_something(NewList,Threshold)

     else sensor_see_something(NewList,Threshold)
```

The idea is to keep a list of values that are under the threshold, as long as the sensor value is under the threshold it keeps adding value to the list. When the value read is greater than the threshold, the function checks that the last value in the list was under the threshold and that the current list size is greater than 5. If it is the case then it means that a message needs to be sent to the other board. This means that the message is only sent when the person has left the sensor field of vision.

This function has been implemented in the sensor driver itself to optimize the rate at which the sonar could detect movement.

8.3.2.1.1 Limits of the system Using just timers it was complicated to cover all the possible cases that this problem implies, for instance since the second board only has 2 seconds to acknowledge the passage of a person, if the person stays in front of the sensor for more than two seconds the timer will expire and the second board will not count it. As long as the objects pass the system one by one there will not be any problems.

Chapter 9

Edge Server and Dashboard

9.1 Introduction

IoT devices located on the edge are responsible for executing tasks with goals ranging from sensing their environment to doing distributed computations. In either case, some data is generally processed and ultimately saved to Lasp CRDT's to achieve global replication on participating nodes of an edge cluster. Moreover, as discussed in chapter X, data aggregation is also performed on nodes periodically for the purpose of summarizing data as well as optimizing the memory on nodes. Resulting processed data is intended to be interpreted programmatically and visually by external parties. In this regard, an intermediary between the producer of data and the end user is needed. This intermediary is an actor whose responsibility is to further process data and ultimately to offer the end user a visual representation of the latter. This intermediary, referred to as **Local Edge Server**, and a data vizualizer interface, referred to as **Dashboard** are the subject of following section.

9.2 Local Edge Server

9.2.1 Overview

The Local Edge Server is a server implemented in **Elixir**[1], a general purpose functional programming language that runs on the Erlang virtual machine. Elixir can be seen as an improved extension of Erlang providing the user with a richer, more elegant syntax as well as tools for organizing code and removing boilerplate allowing the programmer to gain in productivity. Moreover, Elixir is built on top of Erlang and compiles down to BEAM bytecode. As a result, Elixir is able to use all the features and abstractions provided by Erlang due to its inherent interoperability.

The Local Edge Server is part of the edge cluster along with the GRiSP boards. Upon starting, and also periodically, the server will attempt to join each node in the cluster via Lasp in order to be able to retrieve the nodes CRDT's data at a later point in the future as explained below.

9.2.2 Local Edge Server API

The Local Edge Server exposes an HTTP API endpoint that is reachable by external clients interested in retrieving aggregated sensor or computational data collected and processed by the GRiSP boards. Upon receiving a request from a client, the

[1] https://elixir-lang.org/

Local Edge Server will query defined Lasp CRDT's, convert the Erlang data objects retrieved to the JSON standard file format and return this result to the client. The JSON format [2] is used because of it's simplicity, ease of reading for humans and also because it is a standard format used in traditional API's.

The following table depicts the Local Edge Server API:

Edge Server API		
URL	Method	Description
/api/nodes	GET	Fetches the data of all the nodes.
/api/node/:name	GET	Fetches the data for the node named **:name**
/api/alive-nodes	GET	Returns the nodes alive in the cluster

9.3 Dashboard

9.3.1 Introduction

Sensor and computational data collected, processed and replicated on the GRiSP boards are generally not meaningful to humans if their content is not interpreted programmatically or visually. Additionally, GRiSP boards running for a long period of time could generate significant amount of data, even if the latter is generally aggregated, that would otherwise be difficult to interpret without an automated software that is aware of the structure of the corresponding data. In the following section we describe our experimental implementation of a dynamic web user interface application presenting data generated by the GRiSP boards in an orderly and visual manner.

9.3.2 Overview

The Dashboard is a web application that displays the sensor's data of each GRiSP board individually and also global sensor data summaries for all the nodes. The dashboard is implemented using the **React** [3] Javascript framework. React is a popular open-source Javascript library used to create dynamic user interfaces and more specifically to build Single Page Applications (SPA) [4]. Single page applications are web applications written solely in Javascript that dynamically change the content of a web page on user or system events without the need of loading the entire page from a server. The SPA's entire application comprising of HTML, JS and CSS ressources is retrieved on the first load. Additional resources needed during the user's visit on the SPA are dynamically fetched using API calls. Overall, SPA's enable web applications to appear seemingly smoother and faster to the end user as a result of the lack of interruption in the latter's navigation.

To retrieve and display the sensor data of the edge nodes, the Dashboard's web application makes a request to the Local Edge Server's API and retrieves the JSON containing the data of all the nodes. Next, the data is interpreted and used by the dashboard to display accordingly graphical and/or numerical information related to each sensor present on each node.

[2] https://en.wikipedia.org/wiki/JSON
[3] https://reactjs.org/
[4] https://en.wikipedia.org/wiki/Single-page_application

9.3.3 Dynamic sensor data display

The web application refreshes the board's sensor data displayed on the dashboard every 60 seconds. The new values are dynamically updated on the dashboard's interface using React allowing the user viewing the dashboard to see the chart progression with updated data in pseudo-realtime. We have implemented a display page for each of the following sensors:

- Ambient light (Pmod ALS)
- Temperature (Pmod Nav)
- Pressure (Pmod Nav)
- Magnetic field (Pmod Nav)
- Gyroscope (Pmod Nav)

9.3.4 Layout of the Dashboard

The Dashboard layout is depicted in the figure below:

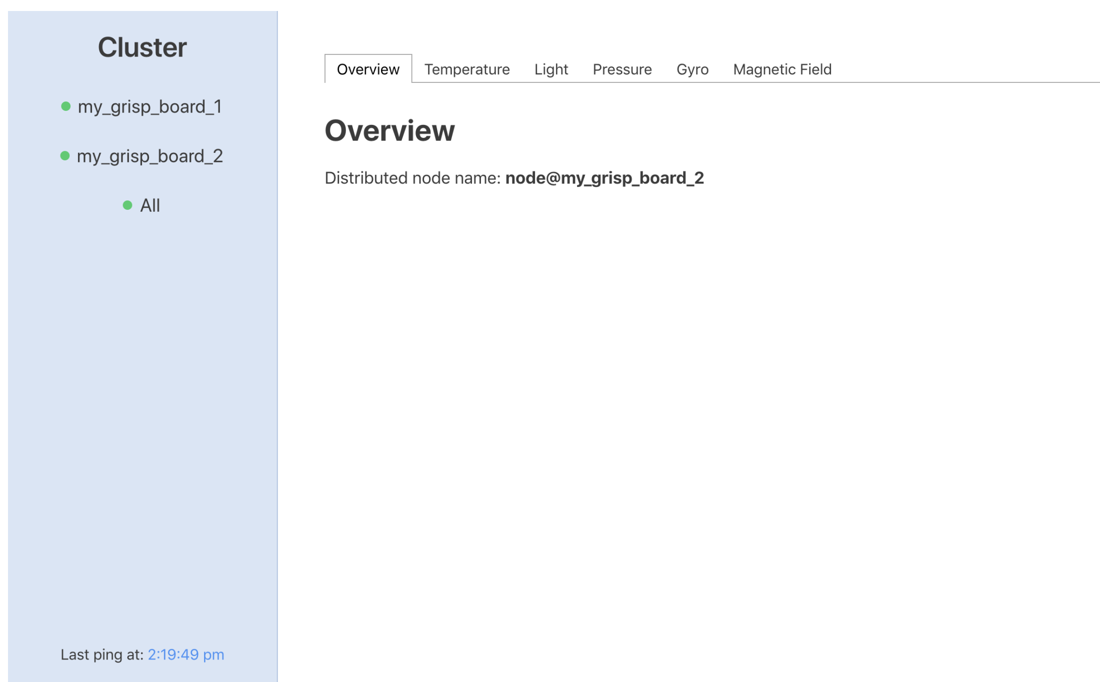


FIGURE 9.1: Dashboard Layout

The layout consists of a menu on the left which shows the GRiSP boards that are part of the Lasp cluster. A small green or red bullet next to the board's name indicates whether or not the latter is online or offline. The last time that the local edge server pinged the nodes part of the cluster is shown at the bottom of the menu. The name of a board is a link that is clickable. When a user clicks on one of the boards on the menu, the user gets presented with a list of tabs that show the gathered data for a particular sensor as depicted on the image below:

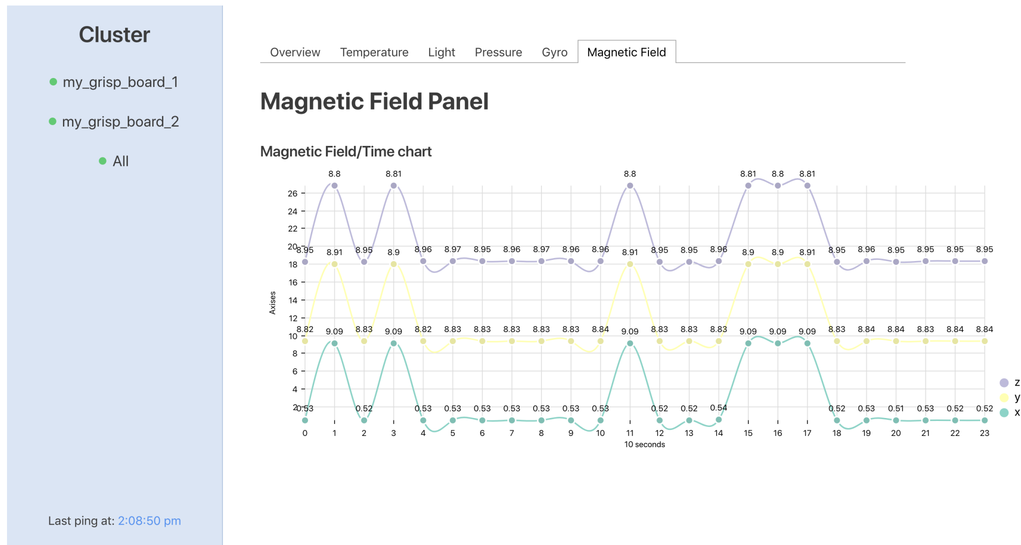


FIGURE 9.2: Dashboard magnetic field sensor data of node "node@my_grisp_board_2"

An interesting feature of the dashboard is the ability for a user to see the sensor's data of a node, even if the latter has become unreachable. This is due to the inherent properties of Lasp's CRDTs which provide resilience in the case of failure. Since node's data is stored in CRDTs that are replicated on all the nodes that are part of the cluster, if at least one node survives, the latter will hold the data of all the other nodes until the moment they crashed.

Chapter 10

Engineering of the GrispLasp infrastructure

10.1 Preamble

The present chapter introduces the modules that have been created in order to address the specific requirements of light edge nodes in terms of storage. The LightKone project has presented a taxonomy for a classification of all possible components in the edge model. Figure 10.1 provides the ontological perspective that has been shown in the 5th deliverable of the LightKone project [44].

As described in Section 1.3.1, the main objective of this work is a real-life general purpose edge network. The requirements include porting the Lasp libraries on GRiSP boards, and the rest of this chapter is dedicated to the software engineering process that has enabled such a deployment.

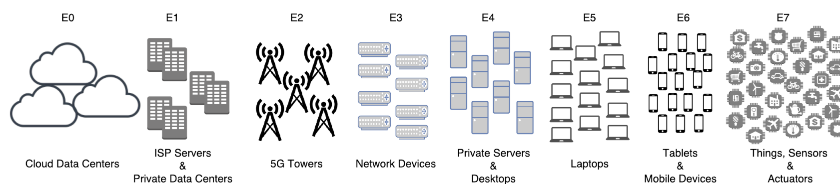


FIGURE 10.1: A vision of the scope of edge levels in the LightKone project.

10.2 Interlevel design

In the pattern shown in Figure 10.1, we are not able to distinguish wether the GrispLasp prototype fits level E6 or E7. The GRiSP boards have approximately 10× less computation power and 100× less memory than currently manufactured mobile devices. Hence it would imply that GRiSP boards are a relevant solution as E7 level devices, that could be responsible for holding local cache storage and filtering data [44]. But the proposed edge implementation uses Lasp as a direct substitute for higher level cloud storage solutions and provides a generic task model that is an entry point for any kind of data aggregation operations. We take the initiative to augment LightKone's taxonomy to fit GrispLasp in a new level labeled **E6.5**. Level E6.5 intended to combine semantics of level E6 and E7 such that it encompasses the uses of

level E7 and E6 operations that are adapted for the GRiSP hardware.

Therefore we classify GrispLasp accordingly to its capabilities and resources :

- Distributed storage and general purpose computations at the edge
- Limited, but fixed and well-known memory, storage and processing power

This separation from the initially known levels comes as a result of the numerous adaptations we have made for the implementation and instrumentation of GrispLasp.

10.3 Wireless network frequency

Configuring wave modulation on GRiSP boards such that the network is located in channel 6 of the 2.4GHz frequency band has resulted in improvements in terms of maximum range and obstacle resilience of the adhoc network. By default, the network was created in channel 10, which lead to a serious amount of interferences considering that most of the Wi-Fi access points observed during experiments are in channel 11.

Figure 10.2 displays the 11 usable frequency channels in the 2.4GHz band. It can be distinctly observed that only channels 1, 6 and 11 are non-overlapping.

Since each 802.11b/g/n channel is 22 MHz wide, but the spacing between each channel is only 5 MHz [52], it follows that there needs to be strictly more than 4 channels between 2 non-interfering channels.

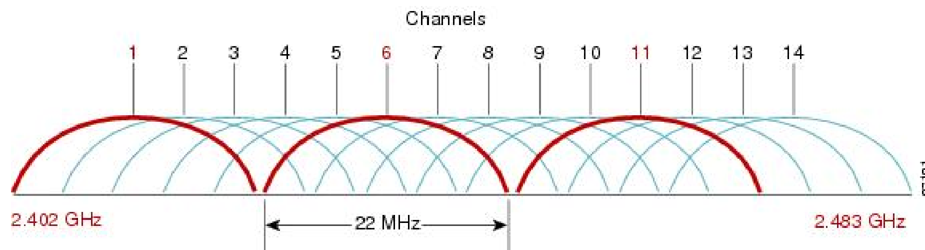


FIGURE 10.2: The 11 channels available in the 2.4GHz band and a representation of their frequency spectrum.

10.4 Memory limitations

The experimental framework that we have built for applications at the edge required a substantial amount of additional configurations and implementations to be evaluated against a regular cloud solution. This is firstly due to the added complexity of *Dockerizing* and deploying applications using AWS services as described in Section 11.3.4.1. Since our evaluations are intended for comparison with these environments, we must record metrics on both counterparts. In this section, we elaborate the challenges that were overcame to alleviate the fierce memory constraints for GrispLasp.

Memory issues are recurrent in deployments where BEAM nodes are required to live for an extended period of time[30, 19, 50, 54, 16, 49]. Since GrispLasp aims at building edge networks where nodes are fault tolerant and able to keep running even in case of failures of all sorts, it has been a concern we could not ignore in our prototype implementation. Considering the E6.5 level that we have previously defined, we optimize our prototype to the fullest possible extent, as the available memory is much smaller than most of machines hosting BEAMs.

10.4.1 Analysis of the Erlang memory allocators

Eventhough GrispLasp runs on a BEAM i.e. an *Abstract Machine*[48], and not strictly a virtual machine such as in other Erlang implementations like Erlang on Xen[1] or Erjarng[2], the memory allocation process is very sophisticated. Figure 10.3 shows that there are two main allocators, and several sub-allocators that coexist in a single scheduler.

Memory areas can be allocated to GrispLasp in 2 ways :

- `sys_alloc` : through *malloc* and *free*
- `mseg_alloc` : through *mmap* if the host system permits it

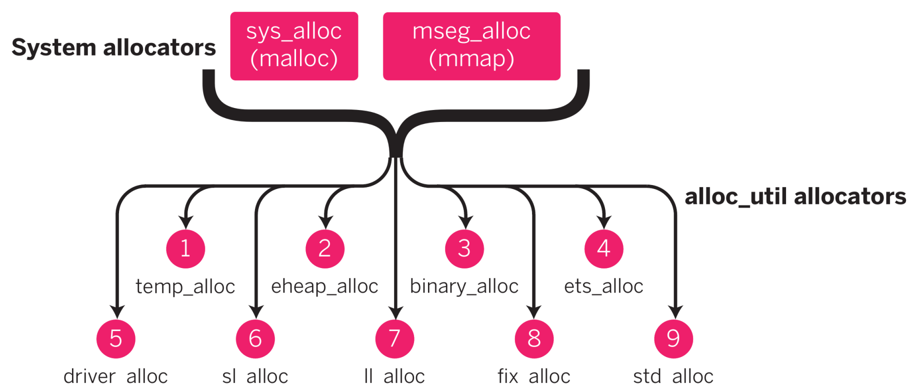


FIGURE 10.3: Memory allocator hierarchy in the BEAM. Reprinted from *Stuff Goes Bad: Erlang in Anger*[46].

10.4.2 Tuning the BEAM memory allocation

After porting the Lasp on Grisp, we have encountered severe heap memory limitations when applications were deployed on the nodes. The generic task model provides an API that puts little to no restriction on what software can be executed on a GrispLasp edge network. But the first deployments we've made have shown that even simple tasks would systematically crash every node after a short period

[1] http://erlangonxen.org
[2] http://www.erjang.org

of time, depending on the intensity of the tasks. The errors consistently originated from the Erlang heap allocator, that would at some point attempt to allocate more memory than the Erlang VM could provide.

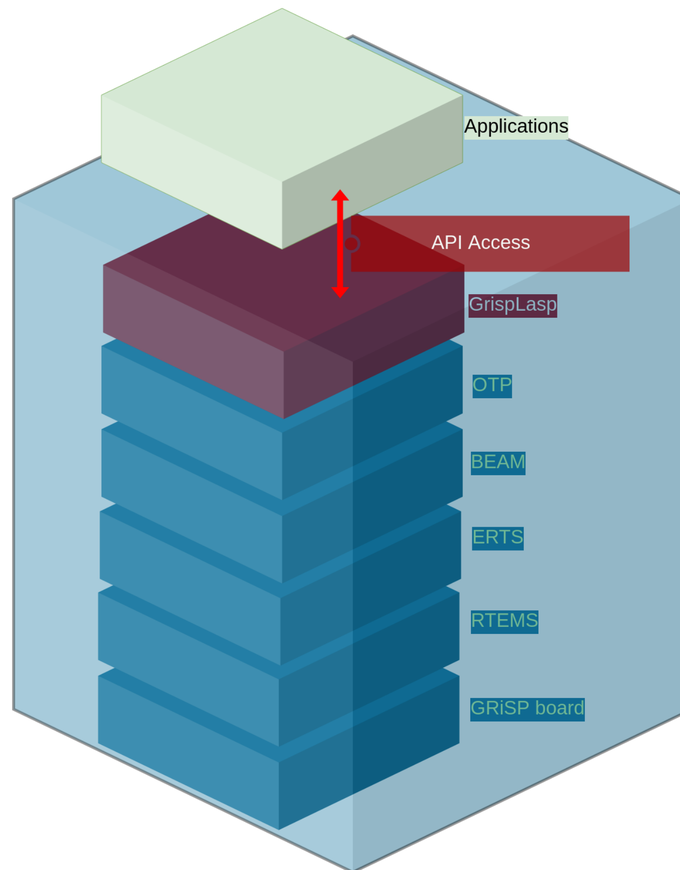


FIGURE 10.4: A stacked view of the GrispLasp prototype.

Figure 10.4 shows a GrispLasp node in the form of a blackbox that provides access to lower levels through the task model API. More than simply offering unlimited possibilities of tasks that can be sent by applications, the task model also exposes the BEAM's vulnerabilities as it is very sensitive when running on GRiSP boards. This has lead us to implement a series of changes to the regular behavior for additional robustness of the system. Attempting to solve such issues requires an extensive and in-depth understanding of the way memory is managed is the Erlang VM. Fred Hebert, the famous author of *Learn You Some Erlang for Great Good!*[3], has encountered similar issues while working at Heroku [49]. It has given us an insight for the adjustments that were required for a functional deployment of a GrispLasp edge network, and the underlying complexity.

10.4.2.1 Memory allocation strategy

In Table 10.1 is an example of what eheap allocations could look like at some point in the execution of GrispLasp. Each *mbcs* is a carrier that can contain multiple blocks of allocated memory. Red areas indicate currently allocated space in a carrier. The

[3] https://learnyousomeerlang.com/

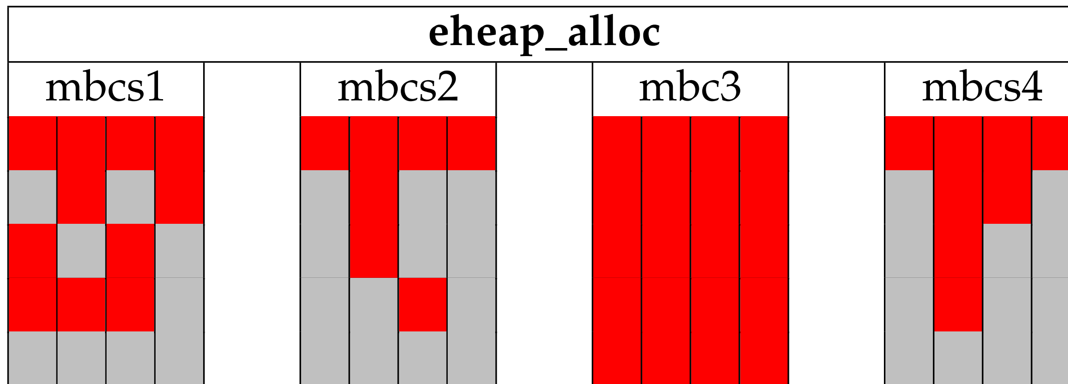


TABLE 10.1: Visual representation of the eheap allocator and its allocated blocks.

allocation strategy dictates the way an allocator determines where exactly a block will be positioned inside a carrier. We have set GrispLasp's strategy to **address order best fit** for the `eheap_alloc` allocator. This translates as :

> When the Erlang heap allocator needs to allocate a block of size n in memory, it will search for a block b such that:
>
> $$b = min_size(\forall block | size(block) \geq n)$$
>
> And if there are multiple blocks that can satisfy the previous predicate, the one with the lowest address is chosen. We have adopted this allocation strategy to avoid as much fragmentation as possible in the heap.[13].

10.4.3 Avoiding large memory segment cache size

By default, when a memory segment is freed, the Erlang VM does not make it immediately available and available for subsequent allocations. Instead, a certain amount of segments are cached by the *mseg_alloc* so that in case of peaks of allocation requests, it could distribute a steady amount of segments. But GrispLasp seldom requires allocations of several big segments of memory, and therefore the cache size has been reduced from 10 to 5 maximum segments.

10.4.4 Forcing garbage collection manually

All the previous adjustments were made through emulator flags that exist for fine-grained optimizations or environments that do not perform well under classic scenarios. But we have further increased the average lifetime of a node and the overall stability by using Erlang's `garbage_collect` function.

We have made periodic per-process garbage collections for the long running processes that were inevitable such as the ones that were necessary for benchmarks. Also, one of the core modules of a GrispLasp node that is shown in Appendix B.1 contains a function that performs a major garbage collection sweep on all processes in the BEAM[46, 49].

10.4.5 Persisting and flushing Lasp CRDTs

Due to their inherent unreliability, edge IoT devices must integrate mechanisms for protecting data by persisting the latter to cope with eventual failures that may occur. The probability of failure of an edge device grows the longer it stays alive in the network. To this end, we created a module referred to as **Node Storage Utility** specifically targeting this matter and whose purpose is to save the node's CRDTs to its SD card periodically. The benefit of this module is dual:

1. Saving CRDT data to the node's SD card allows to protect the former from eventual occurring crashes.
2. The process of saving also has the benefit of freeing memory on the node, improving its overall stability and reliability.

10.4.5.1 CRDT flushing

SEC guarantees abstract all the conflict resolution process such that it is no longer a burden for distributed application design. But it requires merging δ-mutators or entire states in algorithms that are responsible for convergence. Obviously, this procedure requires monotonically increasing memory to store the states which can be a limiting factor on resource constrained devices. We propose an extension to the CRDT implementation in Appendix B.1, that is designed to help devices with limited RAM. We have implemented it in the GrispLasp framework so that at any point in time, a node can persist its current state of a CRDT to the SD card of the GRiSP board.

Currenty, two persistence modes exist for writing on the SD card :

- **`save_no_rmv_all`** : upon call, the node will *flush* the local state by swapping it out of memory to a file on the SD card. Each time the operation is called for a same CRDT variable, GrispLasp will create a new *part* that will be persisted in a file with an incremented index. Each CRDT will be stored in its own directory on the SD card, in separate parts in their increasing numerical order e.g. :
  1. data
     (a) temperature
        i. `part_001`
        ii. `part_002`
        iii. `part_003`
- **`save_rmv_all`** : very similar to the previous operation, but before flushing, the node performs one last update telling all other cluster members to remove every element in the set. This can be later used as a trigger for a local flush on nodes that hear that signal. This can be used to lighten the memory usage on a cluster-wide scale.

10.4.5.2 Loading persisted CRDT's

We have also implemented the reverse equivalent of the `flush` operation, to enable loading parts that were previously persisted. It relies on Erlang ETS utilitary functions to recreate the tables that maintained the state of CRDTs. The call can be

used to combine several parts and merge them with operations from their respective implementations. This way, a device with larger amounts of memory can retrieve fragments of states from smaller ones.

Part III

Evaluation

Chapter 11

Evaluation of two computing models

11.1 Introduction

The previous part of this thesis work presented the general approach that was taken to create and deploy a cluster of GRiSP boards in an edge network running applications with the purpose of executing tasks ranging from environment sensing to general computations. The nodes constituting this edge network heavily rely on the use of the Lasp programming model and its underlying librairies to achieve global replication of data on the nodes with the use of CRDT's as well as management of cluster membership with the use of Lasp's Partisan library.

The present chapter focuses on measuring and evaluating how these programs perform in an edge computing environment but also how they would perform in a cloud computing environment. This evaluation inherently tackles two different computing models, the edge computing and cloud computing models. At the end of this part, we present the results of our experiments, discuss them and finally compare the two models.

11.2 Computing Environments

11.2.1 Lasp on Edge

This environment ,referred to as **GrispLasp**, is the general environment that was used during the realization of this thesis work and was described in the previous chapters. In this environment, a cluster of GRiSP boards is deployed in an edge network and runs applications relying on Lasp to replicate gathered data on each node that are part of the cluster. The nodes in the cluster periodically execute edge computing tasks.
The nodes are located close to the end users and gathered data is only local to the network i.e. it is not stored on the cloud. Lasp allows an unreliable edge network to be highly resilient in the face of failure as data is replicated on each nodes that are part of the cluster.
Additionally, as IoT devices present in the network have low computing power and storage capacity, data is periodically aggregated to reduce the global data size which can be significant. This allows to offload applications running at the edge while reducing the network load.

11.2.2 Lasp on Cloud

In this environment, referred to as **CloudLasp**, a cluster of servers hosted on the cloud receives data from the boards located at the edge. Specifically, the server closest to the edge network is the intermediary between the edge nodes and the cloud nodes. This server is responsible for receiving, processing and aggregating data sent from the edge nodes and to replicate the processed data on the other servers using Lasp.
Naturally, this intermediary is considered to be a single point of failure. In the rare case of a downtime, data sent from the edge to this server could be lost forever if no precautions are taken in the face of such event.

11.3 Measurements methodology

In this section, a description of the approach taken at measuring both applications running completely on the edge and edge applications communicating with the cloud is given.

11.3.1 Meteorological task

The basis of measurements is the same in both of the scenarios presented above. Data generation is done at the edge and is either replicated locally in the edge network or is sent to a server on the cloud which will in turn further process and replicate that data on other cloud servers that are part of the cloud cluster. In other words, aggregation and replication are either done completely on the edge, or completely on the cloud.

Our experiments focus on a single application running on GRiSP boards in the edge network. This application runs a task through the API described in Tables 8.1 and 8.2. It does meteorological data sensing, gathering temperature and pressure periodically using the Pmod Nav sensor device. The task's function is self-contained and performs a predefined amount of sampling iterations.

This task consists of three different variables that may impact the execution of the experiment:

1. **Loop count**: The number of iterations of the task's function.
2. **Sample count**: The number of samples of temperature and pressure pairs to gather in a single iteration.
3. **Time interval**: The time in milliseconds between each sampling of a {temperature,pressure} pair.

During each iteration and after sampling the desired amount of *{temperature,pressure}* pairs, an aggregation is performed on the gathered samples. This aggregation is done either on the edge in the case of the edge computing model or on the cloud in the case of the cloud computing model. The aggregation consists of calculating the mean, variance and covariance of the samples of {temperature,pressure}. Furthermore, the resulting aggregation is then replicated on participating nodes of the cluster. In both scenarios, we measure two notably important variables:

- The aggregation computation time.

- The convergence time i.e. the elapsed time between the moment the resulting aggregation has been added to a Lasp CRDT and the moment that the resulting aggregation has converged on all the other nodes that are part of the cluster.

11.3.2 Computations at the edge

In the following section, we describe the general setup and method we used to perform measurements on boards in an edge network.

11.3.2.1 Environment Setup

Three boards were used to perform the measurements. Each board was placed in a same room at different positions and heights with the Pmod Nav sensor device plugged in. Since GRiSP's requires external power in order to function, the latter was provided via micro usb cable connected to a usb power adapter plugged into an electrical outlet inside a wall. Thus, each board was placed close to an electrical outlet at different places in the room forming a scalene triangle. An overview of the positions of each board is given hereafter:

- Board 1 was placed on a table at approximately 1 meter above the floor.
- Board 2 was placed on a stairstep at approximately 1.2 meter above the floor.
- Board 3 was placed directly on the floor.
- Board 2 was placed between board 1 and board 3. The former is approximately 3 meters away from board 1 and 3.5 meters away from board 3.
- The distance between board 1 and 3 is approximately 4.5 meters.

The boards formed an edge network in peer-to-peer mode each running the same Erlang application. Boards would be started manually one after the other. Upon booting, each board would first connect to their counterparts via Lasp using HyparView to form a cluster. Three minutes after booting, each board would start the task described above. After the task performed all its iterations, the measurements would then be saved to a file on the board's SD card.

11.3.2.2 Execution flow of the task function

An iteration of the task function consists of three parts:

1. Sampling: X samples of temperature and pressure is gathered at Y time interval.
2. Aggregation: After sampling, an aggregation of the X temperatures and pressures is performed. This aggregation consists of calculating the mean, variance and covariance of both types of samples.
3. Replication: After the aggregation has been performed, the resulting aggregated data is added to a Lasp CRDT of type **state_gset** [1]. By default, each node performing the task will put the resulting aggregated data in a Lasp CRDT named after the node's name e.g `node@my_grisp_board_1`. The grow-only CRDT state_gset type is a CRDT whose value can only monotically increase with time. This CRDT is then replicated on all the nodes that are part of the edge cluster.

[1] https://github.com/lasp-lang/types/blob/master/src/state_gset.erl

11.3.2.3 Measurement process

11.3.2.3.1 Aggregation computation time This variable is the simplest to measure. For each iteration, after sampling data and right before the start of the aggregation computation on the sampled data, the current time is stored in a variable T1. After the computation has completed, the current time is stored in a variable T2 and the duration of the computation can be measured with the simple substraction depicted below.
Since the nodes form a peer-to-peer network, they do not have internet access and thus are not aware of the real world time. To circumvent this issue, the $erlang : monotonic_time(millisecond)$ function is used. When a device starts an Erlang VM, by default, Erlang defines a random unspecified point in time from which it will monotonically increase as time goes on. This function can be used to retrieve the current monotonic time value and is used to measure time locally on a node that has no knowledge of the real world time.

$$Aggregationcomputationtime = T2 - T1$$

11.3.2.3.2 Convergence time The convergence time can be seen as the time elapsed between the moment a node updates a CRDT with a new value and the moment this updated value converges on another node in the network. Measuring the convergence time poses two challenges:

1. How does a node know when a CRDT has converged on itself?
2. How can a node know how long it took for a CRDT to converge on itself?

As explained above, it is not possible to get the real world time if a node is part of a peer-to-peer network and has no access to the internet. Furthermore, Erlang monotonic clocks are different on each runtime systems. This renders measuring convergence times on nodes difficult as a node detecting that a CRDT has been updated cannot simply save the time at which it has detected that update since Erlang monotonic clocks are different on each nodes. The proposed solution to tackle these two challenges is described hereafter.

11.3.2.3.3 Detecting that a CRDT has converged on a node The duration of the execution of the task's function is governed by its number of iterations which is defined before the experiment. During each iteration, a Lasp CRDT update is performed after sampling and aggregating the resulting sample. This update uses the *add* operation to add the resulting aggregation to the node's *state_gset* CRDT. Thus, the size of this CRDT, referred to as **cardinality**, grows monotonically after each iteration. Lasp provides a mechanism for triggering an event once the value of a grow-only CRDT reaches a particular point in time.
This operation is called **monotonic read** and is a blocking instruction meaning that a process doing a monotonic read will stop its execution until a particular value is read and bound by Lasp. Inter alia, the monotonic read allows one to wait until a CRDT reaches a particular cardinality. The cardinality specifies the number of elements in a set and is a measure of the size of a CRDT.
This feature is highly interesting in the case of detecting when a CRDT has converged on a node. Since the number of iterations as well as the name of the nodes

that will be executing the task are known beforehand, a node is able to do a monotonic read on the cardinality of each node's CRDT for each iteration of the task. When a node receives a CRDT update and because this CRDT is a grow-only set, the update indicates that the cardinality of this specific CRDT has increased and the node is able to detect the latter. This process is depicted in the figure 11.1.

```
lists:foreach(fun (Node) ->
  spawn(fun() ->
    lists:foreach(fun(Cardinality) ->
            lasp:read(node_util:atom_to_lasp_identifier(Node,
                      state_gset), {cardinality, Cardinality}),
           {convergence_acknowledgement, Node} ! {ack, node(), Cardinality}
    end, lists:seq(1, LoopCount))
  end)
end, NodesWithoutMe),
```

FIGURE 11.1: Monotonic Read on the cardinalities of node's CRDTs

When a node boots, it will run the code in 11.1 before starting the task. For each node that is not the node itself, a process iterating through a sequence of numbers from 1 to the defined number of iterations is spawned. The sequence of numbers represents the monotonic cardinality that the CRDT of each node will have at some point in time. In each iteration on this sequence of numbers, a monotonic read on the node's CRDT cardinality is performed. When the read operation detects that the specified CRDT's cardinality is reached, indicating that the updated CRDT has converged on the node, an acknowledgement message is sent to the node that triggered that update.

The Strong Eventual Consistency property of CRDT's guarantees that, eventually, the cardinality of the CRDTs of each node will monotonically increase at some point in the future and that each node will be able to detect a change in the cardinality of the CRDT's of all the other nodes in the network. This property validates the completeness of the method described above.

11.3.2.3.4 Measuring convergence time When a node detects that a CRDT converged on itself, this node should send an acknowledgement message to the node that triggered the CRDT update. The latter would then be able to measure the time it took for this CRDT to converge on an other node using its local monotonic time. In order to implement the following strategy, an additional process is spawned inside the task function. This process waits for acknowledgements from other nodes in the cluster indicating that the latter have received the update. Each time the process receives an acknowledgement, a tuple containing the sender (*From*), the current monotonic time (*TConverged*) and the cardinality (*Cardinality*) is accumulated in an Erlang list. The acknowledgement process's function is depicted in figure 11.2:

Moreover, during each iteration of the task function, the variable T2 described in paragraph 11.3.2.3.1 is added in an Erlang map of {cardinality, T2} pairs. Since each iteration adds a new aggregation in the node's Lasp CRDT, the cardinality and T2 pair is unique for all iterations. At the end of all the iterations, the list of acknowledgements is iterated and the cardinality of the triplets *{From, Cardinality, TConverged}* are matched with the cardinality of pairs *{Cardinality, T2}* to measure

```
ConvergenceAcknowledgementFun = fun CA(Acks) ->
    receive
      {ack, From, Cardinality} ->
        TConverged = erlang:monotonic_time(millisecond),
        CA([{From, TConverged, Cardinality} | Acks]);
      {done, Computations} -> % Called by the meteo process once it has
    terminated
        Self ! {done, Computations, Acks}
    end
  end
```

FIGURE 11.2: Convergence Acknowledgement Process

the time it took for the node's CRDT at cardinality *Cardinality* to converge on the node *From*. The convergence time is measured using the following substraction:

$$Convergencetime = TConverged - T2$$

11.3.2.3.5 Summarizing the results As explained above, nodes running the task will receive acknowledgements from the other nodes in the cluster. When the task on a node completes all its iterations, we measure the convergence time of every CRDT update performed at the end of each iteration on all the other nodes in the cluster. Moreover, we also calculate the aggregation computation time of each iteration. When a node has completed all the task's iterations, we perform a series of additional aggregations on the gathered data and store the results in a text file on the board's SD card. An example result is depicted in the snippet below:

```
Computations: #{1 => 1721,2 => 1709,3 => 1615,4 => 1799,
                5 => 1767,6 => 1727,7 => 1825,8 => 1806,
                9 => 1756,10 => 1825,11 => 1697,12 => 1815,
                13 => 1682,14 => 1806,15 => 1657,16 => 1800,
                17 => 1619,18 => 1895,19 => 1898,20 => 1593}
Mean Computation Time: 1750.6ms
Standard Deviation Computation Time: 88.57
Variance Computation Time: 7845.41
Nodes Convergence Time:
    #{node@my_grisp_board_1 =>
        #{1 => 362,2 => 276,3 => 421,4 => 293,5 => 321,
        6 => 302,7 => 362,8 => 446,9 => 344,10 => 390,
        11 => 572,12 => 416,13 => 574,14 => 438,
        15 => 698,16 => 492,17 => 637,18 => 477,
        19 => 578,20 => 620},
    node@my_grisp_board_3 =>
      #{1 => 330,2 => 218,3 => 422,4 => 253,5 => 251,
        6 => 268,7 => 342,8 => 334,9 => 289,10 => 317,
        11 => 571,12 => 330,13 => 568,14 => 569,
        15 => 698,16 => 374,17 => 635,18 => 384,
        19 => 431,20 => 3650}}
Nodes Convergence Calculations:
    #{node@my_grisp_board_1 =>
      {{average,450.95},
       {standard_deviation,125.74},
       {variance,15812.78}},
    node@my_grisp_board_3 =>
      {{average,561.7},
       {standard_deviation,739.95},
       {variance,547530.64}}}
Mean Convergence Time: 506.32ms
Pooled Standard Deviation Convergence Time: 731.55
Pooled Variance Convergence Time: 535176.25
```

We are particularly interested in the following two variables:

- The grand mean convergence time i.e the mean of the mean convergence time of each other node.
- The mean aggregation computation time.

11.3.2.3.6 Benchmark Server We created an Erlang module referred to as **Benchmark Server** for measuring the meteorological task. When started, this module will first perform monotonic reads on the node's CRDT cardinalities as explained in Paragraph 11.3.2.3.3 and will then proceed to start the task.

11.3.2.3.7 Latency of acknowledgements As explained before, when a node detects that the CRDT of another node in the cluster has converged on itself, the former will send an acknowledgement to the node that triggered that CRDT update. The process of sending a message to the latter naturally induces a certain latency as this message needs to be sent over the network to its recipient. The following equation depicts the computation of the true convergence time:

$$RealConvergencetime = TConverged - T2 - AcknowledgementLatency$$

We do make use of this equation since the nodes are close to each other and the *AcknowledgementLatency* can be ignored as it is not significant.

11.3.3 Computations on the cloud

In the following section, a description of the setup used to evaluate Lasp on a cloud infrastructure is given.

11.3.4 Environment setup

11.3.4.1 Cloud Server

The **Cloud Servers** are hosted on the **Amazon Elastic Compute Cloud** (EC2) platform environment 2. The chosen type of Amazon EC2 instances hosting the servers are T2.micro [2].
T2 are general purpose on-demand computation instances that are defined by Amazon as "T2 instances are for workloads that don't use the full CPU often or consistently, but occasionally need to burst to higher CPU performance.".
An instance accumulates a set rate of CPU credits per hour while the CPU is idle or in use. For the T2.micro, the amount of CPU credits that the instance can earn per hour is 6. This provides a baseline performance equivalent to 10% of a CPU core (10% * 60 minutes = 6 minutes), meaning that the instance could use 100% of a CPU core for 6 minutes with 6 CPU credits. Whenever the instance needs to burst above this baseline, it will draw from its CPU credits automatically. Furthermore, T2.micro's have 1vCPU and 1 GiB of available RAM memory and the OS used is the standard Amazon Linux AMI [3].

[2] https://aws.amazon.com/ec2/instance-types/
[3] https://aws.amazon.com/amazon-linux-ami/

The Cloud Server application hosted on the EC2 instances is implemented in Elixir, similarly to the Local Edge Server described in Section 9. Additionally, deployments of the Elixir application are done using **Amazon Elastic Container Service** (ECS). ECS is a container orchestration service that is used to deploy **Docker** containers on the EC2 instances. A Docker **container** can be seen as a standalone, self-contained package of software that provides OS-level virtualization unlike virtual machines that provide hardware and kernel virtualization.
Docker is the software allowing multiple containers to be shared on the host user-space and is used by ECS to run containers on EC2 instances. The Elixir Cloud Server application which runs on EC2 instances is containerized inside an image that hosts a lightweight Linux OS comprised of a full Erlang and Elixir installation. A single script is used to simultaneously deploy the Elixir cloud servers on EC2 using ECS and Docker Compose, a tool for running and deploying multiple Docker containers.

11.3.4.2 Cloud Infrastructure

To evaluate the convergence of Lasp on cloud, 3 servers were created in different european regions, namely, London, Frankfurt and Paris. The Elixir application running on those servers is equivalent to the one used to test the convergence time on the boards, with the exception that the library containing the Erlang functions to communicate with GRiSP hardware have been removed from the application's dependency.

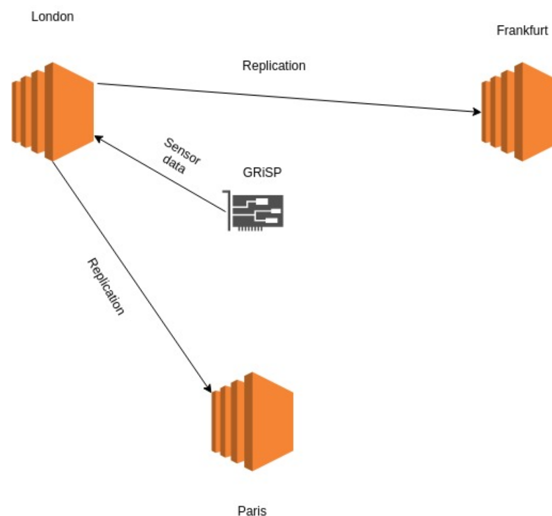


FIGURE 11.3: Cloud architecture setup

Only one server communicates with the GRiSP boards directly, the other servers receive updates from the latter. When the servers boot, they will start the Elixir application along with the same application used on the GRiSP boards and join each other in order to be part of the same cluster.

11.3.4.3 Measurements setup

The goal of the measurements is to compare Lasp running on a cluster of GRiSP board and running on a cluster of cloud instances. The measurement setup had to be as similar to the one used to measure Lasp on edge. The execution flow is almost the same except that data needs to be sent from a GRiSP board. Of course to be able

to communicate with this board, it had to be set up in infrastructure mode and the London server had to added to the list of known hosts on the board. There are two different mode used to compare Lasp on edge and Laps on cloud. One consist in sending raw data, directly from the sensor, to the server and the other one consist in sending an aggregated list of data that the server will apply calculation on. The first one is called *xcloudlasp* and the second one is called *cloudlasp*.

11.3.4.4 Execution flow for xcloudlasp

1. Raw data: Each time a sensor data is generated by the GRiSP board, it is sent to the London server
2. Sampling: Every time the server receives a data from the board, it is added to a temporary list until it reaches the size of *sample count*.
3. Aggregation: Calculating the mean, variance and covariance of both types of samples.
4. Replication: After the aggregation has been performed, the resulting aggregated data is added to a Lasp CRDT of type state_gset.

11.3.4.5 Execution flow for cloudlasp

This execution flow is identical to the one used for the Lasp on edge model, only the sampling is done on the GRiSP board, then sent to the London server which does the rest.

11.3.4.6 Measurement process

11.3.4.6.1 Aggregation computation time The computation time is measured the same way for both model since all the computation is done locally.

11.3.4.6.2 Convergence time As explained previously, the issue for the convergence time on the edge model is to synchronize the clocks together. Synchronization is not an issue when working on AWS cloud as EC2 instances clocks are synchronized using NTP. [4].

Although the clocks are synchronized it was not easy to find the right function to measure time between different servers. **erlang:monotonic_time(millisecond)** can not be used when measuring time between servers as the timestamps will depend on the booting time of the Erlang VM, another solution was to use **os:system_time()**, which gives the hardware time of the EC2 instances but since the deployment of the servers is not perfectly simultaneous the hardware clock are not perfectly synchronized either. The solution was to transform the local Erlang VM timestamp to the local time using **calendar:now_to_local_time(Now)**. The final issue to be able to measure correctly convergence time was to convert this local time to UTC time since the servers are not all in the same timezone, this was done using **calendar:local_time_to_universal_time_dst()** on the local time found previously.

The convergence time is measured for each backup servers which are Frankfurt and Paris. Meaning that the convergence does not represent the total time an update

[4]https://fr.wikipedia.org/wiki/Network_Time_Protocol

needed to propagate in the whole cluster but rather the time needed for one server to be up to date with the London server. Of course to have an idea of the global convergence time, the worst server convergence time could be used as an estimate.

To measure the convergence time, the main server will start a timer T1 right before it starts updating a Lasp set. Concurrently the two backup server run their own task where a call to **lasp:read(ID,set_type,cardinality,count)** is being executed. This function call will block until the set on the server has reached the size of the cardinality. At this moment another timer T2 is trigger by the backup server which sends a message **{Self,UpdateTime,SetName}** back to the main server:

1. Self: The name of the server sending the message
2. UpdateTime: The time at which the set has been updated on the server
3. SetName: The name of the set that has just been updated.

The main server then computes the difference between T2 and T1 to get the convergence time for each server. The results are then stored in a map to apply statistics operations on the measures.

11.4 Experiments

In the following section, we first describe the experiments we ran and later present the results we gathered during those experiments. Finally, we examine those results and discuss them.

11.4.1 Experiments at the edge

The GRiSP boards are all part of the same cluster and are instructed to update their own CRDT with gathered sensor data after each iteration of the task. They also need to wait for acknowledgements from their counterparts in order to measure the convergence time. The process of aggregating measurements and summarizing results is performed once the task finishes all its iterations as detailed in Section 11.3.2.3.5.

11.4.1.1 Experiment Workflow

The execution flow of the experiment implies a certain degree of coordination and synchronization between the nodes. This is due to the fact that there exists an inherent nondeterministic factor while running different executions of the same experiment. We describe this problematic below:

- **The boards must be part of the same cluster.** This statement is trivial. If a node is not part of the same cluster as the other nodes, then the former will not be able to detect when the CRDT of another node has converged on himself and vice-versa.
- **The boards must be booted at the same time.** As explained in **??**, when a board boots, the benchmark server is started. The latter will first launch a process that performs a sequential blocking monotonic read on the CRDT of each node in the cluster starting with a cardinality of 1 and then proceed to start the meteorological task. In the case of a board starting later then the other boards,

and that the latter have already started their task and performed some iterations, the late board will begin performing the sequential monotonic read and will instantly get results as the current cardinality of some of the other board's CRDT will already be above 1. The late board is considered to be "out-of-sync" with the concurrent execution of the task on the other boards. The resulting behavior of the late board is error-prone as the latter will immediately send acknowledgements for each of the monotonically read CRDT's cardinalities. In other words, the convergence time for different CRDT cardinalities will be the the same and will give biased results.

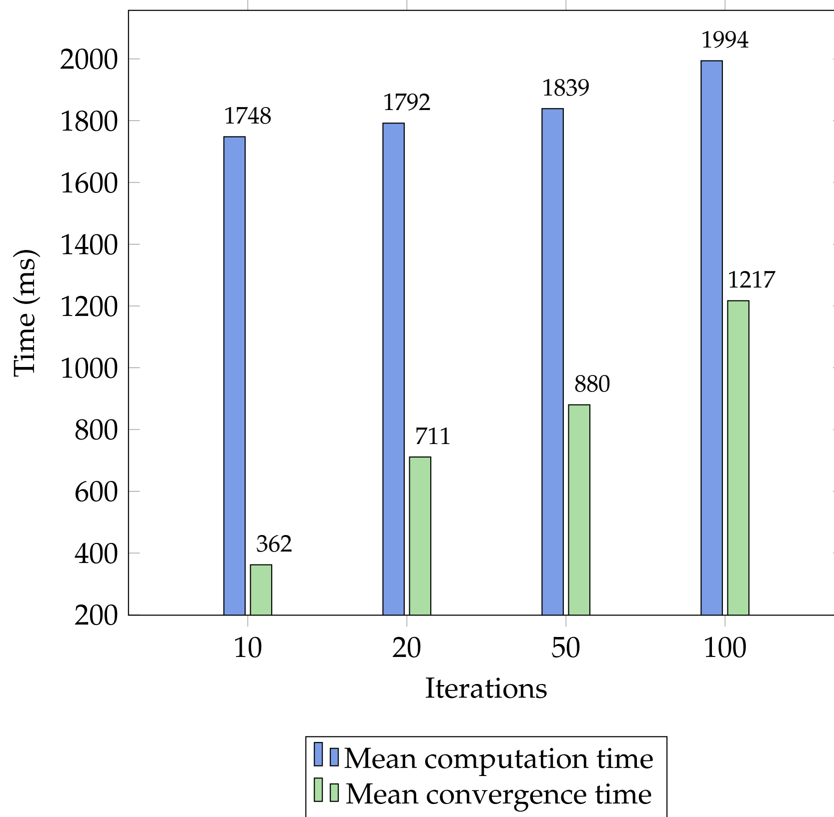
To overcome these problems, we designed a workflow for our experiments that guarantees that the above issues cannot happen:

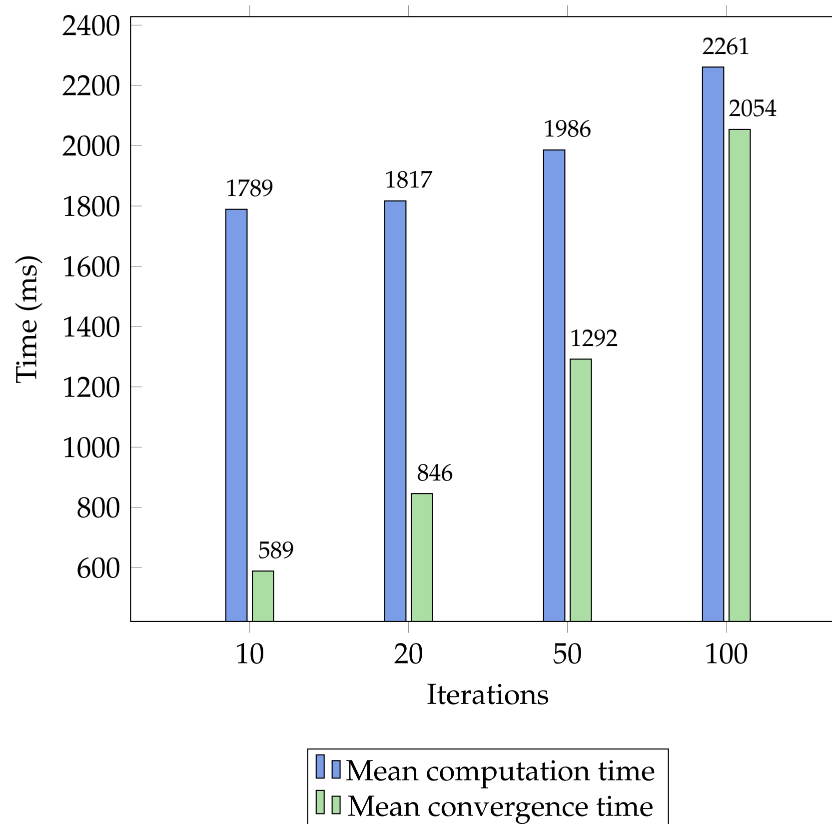
- **Booting the boards:** Fortunately, we know that the booting time of boards running the same application is near-equal. Thus, all we had to do was to rapidly start each of the 3 boards successively.
- **Forming the cluster:** When the boards had finished booting, we needed to ensure that the latter were all part of the same cluster before we could start the benchmark server since its constituents require Lasp and an already formed cluster to function. To this end, we purposely delayed the start of the benchmark server described in Section 11.3.2.3.6 until the process of clustering all the nodes together was complete. Once all the nodes were able to communicate with each other and were part of the same cluster, we could start the benchmark server and the experiment could begin.

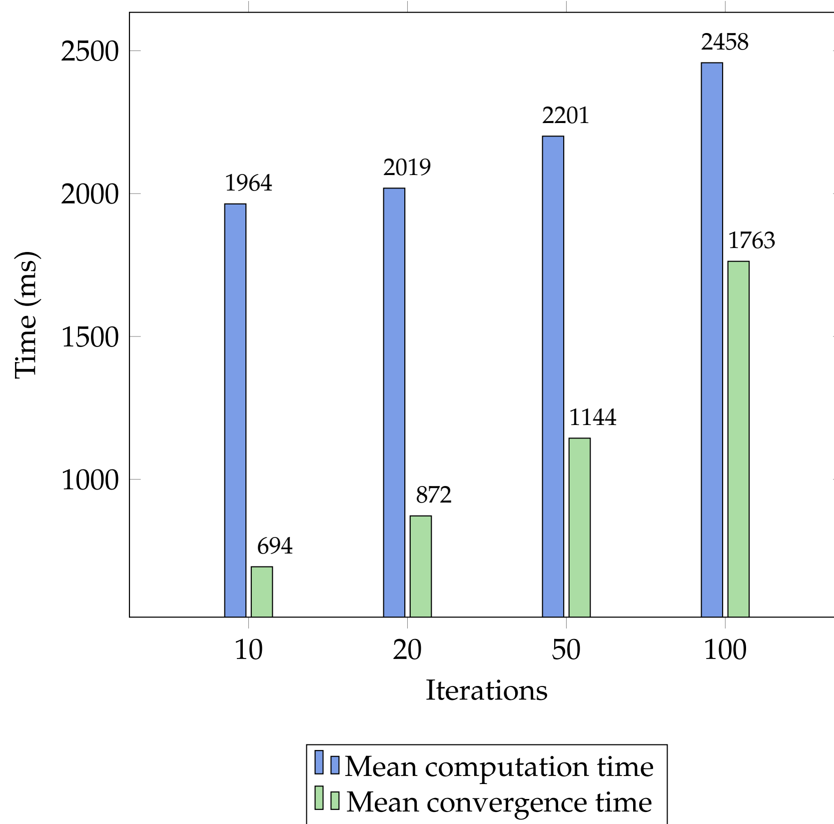
11.4.2 Results

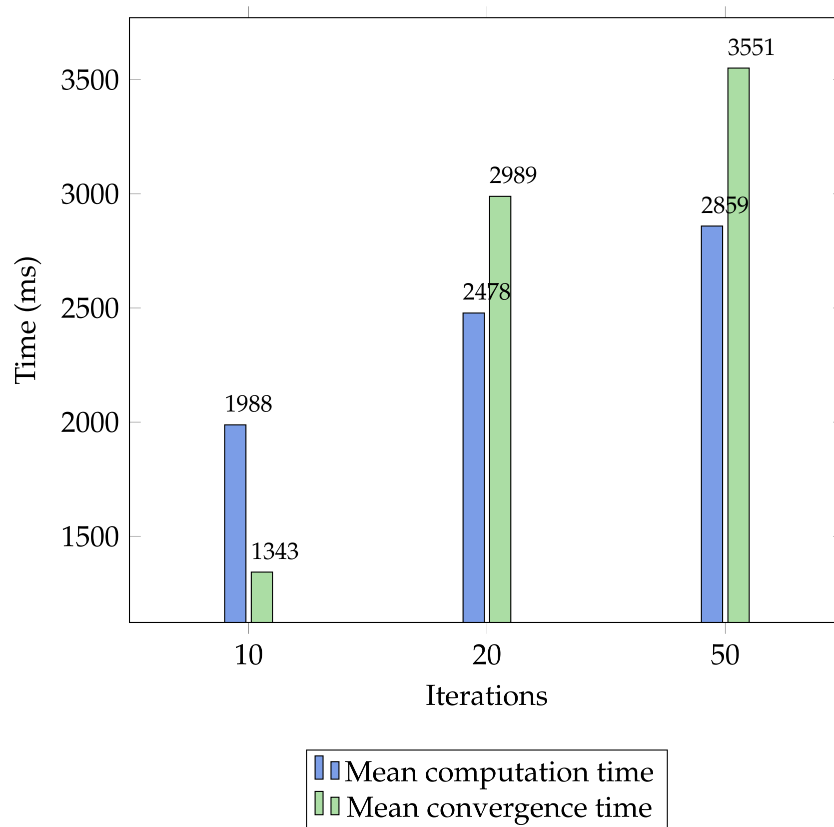
11.4.2.1 GrispLasp

As explained in Section 11.3.2.1, three GRiSP boards, part of the same cluster, were placed in a room at different positions and heights. Each of the boards were instructed with running the meteorological task described in Section 11.3.1 for a predefined number of sampling iterations. The experiment ends once all the boards have completed their task. We ran a total of 4 experiments where we tested the 3 different parameters of the task's function, namely, the number of iterations, the sample count and the time interval between each sampling. For each experiment, we tested 4 fixed number of iterations for the task: 10, 20, 50 and 100 iterations. Furthermore, we ran experiments with Lasp with the **HyparView** protocol [27] using **delta-based dissemination** strategy [2], with the option *propagate_on_update* set to true i.e once a node updates a CRDT, this update is propagated immediately to the other nodes in the cluster. At the end of each experiment, we gathered the SD cards from the GRiSP boards containing the file with the aggregated measurements as described in Section 11.3.2.3.5 and then further summarized these measurements by calculating the grand mean convergence time as well as the grand mean computation time for that experiment. These final results are the essence of our evaluations and represent the overall behaviour of the meteorological task running on a cluster of 3 GRiSP boards in an edge network. The results of these experiments are depicted in the figures below. We discuss the results shortly after.


Iterations of 5 samples with 3 seconds interval between sampling
Time (ms)
2000
1800
1600
1400
1200
1000
800
600
400
200
1748
362
1792
711
1839
880
1994
1217
10
20
50
100
Iterations
Mean computation time
Mean convergence time


Iterations of 5 samples with 1 second interval between sampling
Time (ms)
2400
2200
2000
1800
1600
1400
1200
1000
800
600
1789
589
1817
846
1986
1292
2261
2054
10
20
50
100
Iterations
Mean computation time
Mean convergence time


Iterations of **10 samples** with **1 second** interval between sampling
Time (ms)
2500
2000
1500
1000
1964
694
2019
872
2201
1144
2458
1763
10
20
50
100
Iterations
Mean computation time
Mean convergence time


Iterations of **10 samples** with **0 second** interval between sampling
Time (ms)
3500
3000
2500
2000
1500
1988
1343
2478
2989
2859
3551
10
20
50
Iterations
Mean computation time
Mean convergence time

11.4.2.1.1 Analysis Overall, we can see a general trend for both the grand mean computation time and the grand mean convergence time measured during our experiments: they grow linearly with the number of iterations.
Another interesting trend that stands out from these results is that, the faster the task runs i.e the lesser the time interval between each sampling, the worse the performances. This is true for both the grand mean convergence time and the grand mean computation time. The latter is to be expected as the processing power of the GRiSP boards is limited and thus, the higher the CPU load, the slower the overall performance of the application will be.

A surprising result however, is the grand mean convergence time of Figure 2 compared to the one in Figure 3. In these 2 experiments, the number of samples per iteration is the only factor that changes (5 samples vs 10 samples). We can observe that the grand mean convergence time in the experiment with 10 samples is lower then the one with 5 samples from 50 iterations onwards. We have not found an explanation for this behavior. Furthermore, the grand mean computation time is on average higher in the experiment with 10 samples than the one with 5 samples. This result is considered to be normal behavior as the aggregation computation takes more time when there are more samples.

In Figure 4, we did not put results for 100 iterations due to the unfortunate fact that the boards would systematically crash before reaching the desired number of iterations. We did however try to run this last experiment a number of times, without success. The reason for this crash was always the same: insufficient heap memory. This is due to the Erlang heap allocator trying to allocate more memory than the Erlang VM can provide as described in Section 10.4.2. As explained in Section 10.4, one of the major issues we faced during this thesis work was the fierce memory constraints inherent to the GRiSP boards. With only 64 Mb of available SDRAM, our application already taking close to 23 Mb and a fixed amount of 30 Mb allocated to the Erlang VM, we only had a few Mb of memory left to work with. As depicted in figure 4, the task and especially the grand mean convergence time performs worse by the number of iterations due to the pace at which the application is running.

Our goal while doing these experiments was to show the limits and the performances of a Lasp application running on GRiSP boards in an edge network. We successfully managed to push the latter to its limits by augmenting the speed at which it was running the meteorological task. However, it is important to note that in the general case, the task we evaluated is not designed to be ran at such fast pace. In a real world scenario, the temperature and the pressure which we measured do not usually change in the space of a few seconds. As such, our meteorological task should instead use much bigger intervals of time between sampling of data and aggregation to give a better representation of the reality. A practical example of a real world use-case for our meteorological task could be the following:

A GRiSP board running the meteorological task should measure samples of temperature, pressure pairs every minute and should perform an aggregation of these samples every 10 minutes.

Our experiments are encouraging however, as we showed that the bigger the interval of time between the samplings, the better the overall performances of the application. We are confident that our application could run for much longer periods of time if proper configuration and memory management of the latter are taken into consideration.

11.4.2.2 CloudLasp

To explain some of the results in the following section, we would like to show some experiments done with Lasp using the AWS EC2 instances. As the AWS instances are running on virtual machine and Amazon is hiding most of the under layer while using their instances, it is hard to understand all the results not knowing how the underlying hardware is working, that is why we would like to present this graphic representing the average update time of Lasp on one AWS EC2 instance using one parameter: *the update frequency*. It is important to mention that the *propagate_on_update* option was set to **true** during this experiment. When set to true, this parameter allows Lasp to start sending metadata information about the update before the set has converged locally on the node applying the update. The following snippet shows the Erlang code used for this experiment.

```
Func = fun(X) -> Rand = rand:uniform(),
        TimeA = erlang:monotonic_time(millisecond),
        lasp:update({test,state_gset},{add,Rand},self()),
        TimeB = erlang:monotonic_time(millisecond),
        io:format("Time to update is ~p ~n",[TimeB-TimeA]),
        timer:sleep(X) end.
```

With the first graphic below 11.4 we can observe how the update time of Lasp on the cloud servers increases as the frequency of the *update* requests decreases.

The idea is to compare the performances of our cloud infrastructure with and without the help of our edge model. Since the number of GRiSP boards was limited, we decided to emulate the functioning of those boards using an Erlang shell on a laptop. This shell will start **N** number of processes which will emulate **N** GRiSP boards. We focused on 3 different aspects comparing the two models: convergence time, CPU utilization and network bandwidth.

The first model called **XCloudlasp** represents the current classic cloud application, where the data pulled from the sensors are directly sent to the server. The second model,called **Cloudlasp** uses the edge computing by aggregating the data by sample on the sensor. When the sample has reached the count of 5 data, it is sent to the server where the computation is the same. As a result of the comparison of **Xcloudlasp** and **Cloudlasp** we would like to present the following graphs.

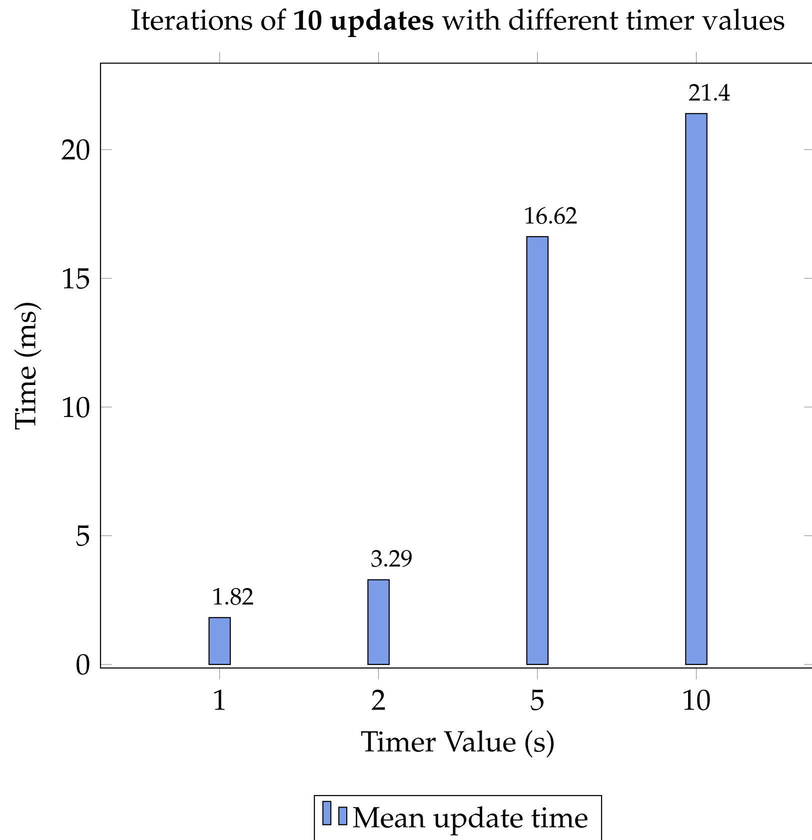


FIGURE 11.4: Update mean time

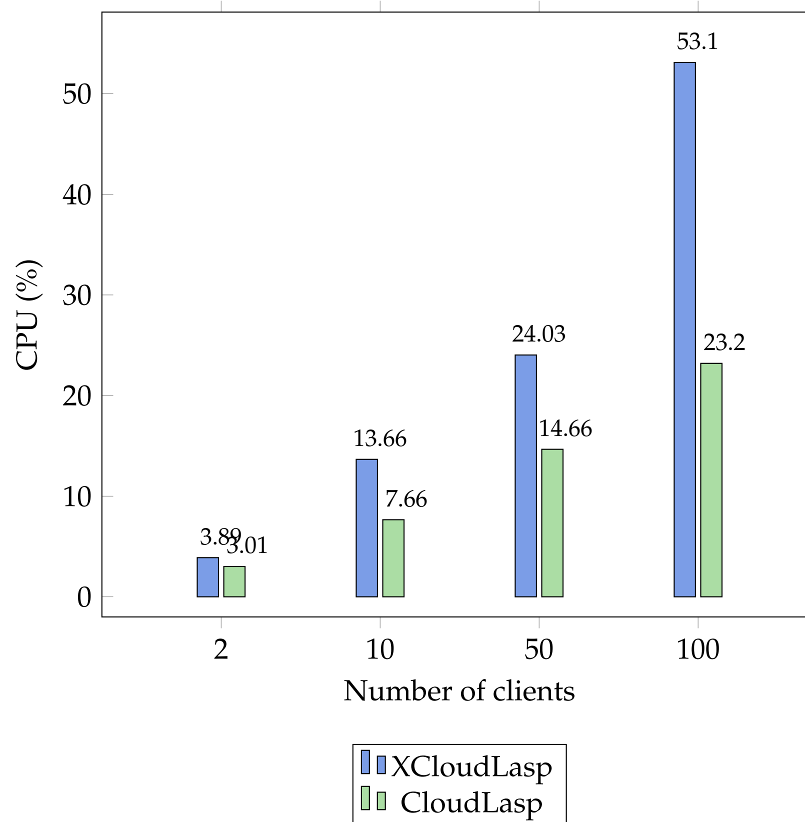


FIGURE 11.5: CPU utilization

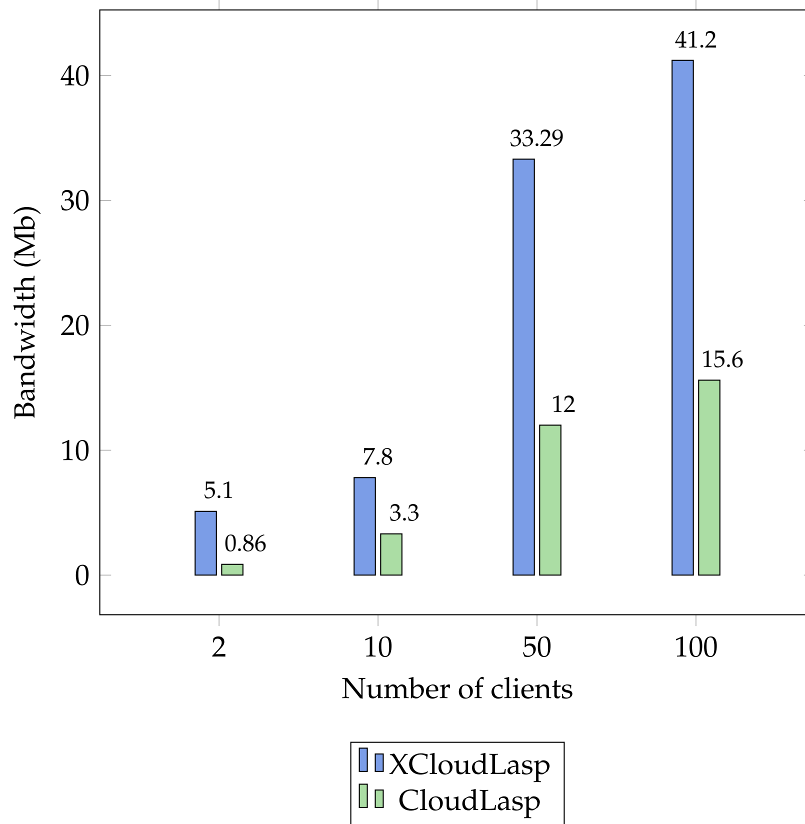


FIGURE 11.6: Network Bandwidth

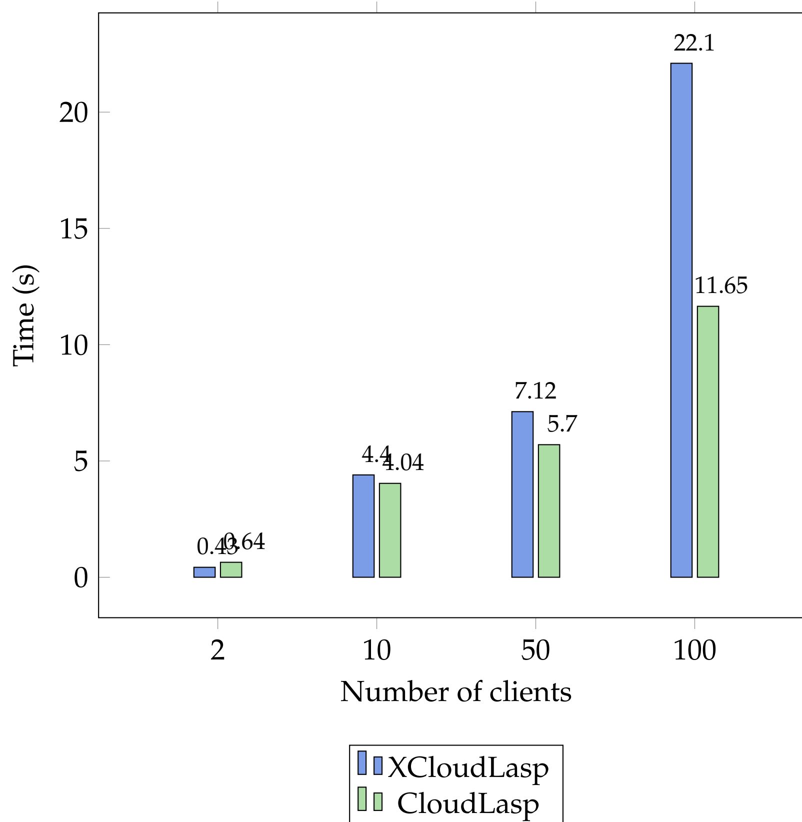


FIGURE 11.7: Convergence time

11.4.2.2.1 Analysis The values displayed on the above graphs, concerning the CPU utilization and network bandwidth were retrieved using the AWS console. It is possible to monitor different metrics on our EC2 instances using a tool called **CloudWatch**.

Let's first analyze, the graph concerning the network utilization 11.6. This represents the average network bandwidth used by the server to receive data from the sensors, as we can see on the graph using the edge model we divide by two the network bandwidth. This is not only a good result for the server but also for the edge devices too, as these edge devices usually consume more energy in communication than in computation.

On the third graph 11.5, analyzing the CPU utilization of the cloud server under the two different models, we observe a big difference as well, this is a consequence of the aggregation done by the edge network. In **XCloudLasp** the server keeps using its CPU to gather the data together whereas in **CloudLasp** the server receives samples that have been pre-aggregated and just needs to apply to statistics on it.

The last value that we would like to compare is the convergence time, as explained in this section 11.3.4.6.2. We think that as a consequence of a smaller CPU utilization when using the edge computing model, the server has more time and more CPU available to update the CRDT's efficiently, for that matter we can observe a correlation between the CPU utilization and the convergence time, for instance if we take a closer look the case of 100 simultenaous clients, we see that both the convergence time and the CPU utilization are divided by two by using **CloudLasp**.

11.5 Comparing the Edge and Cloud computing models

In the following section, we compare our experiments of the edge and cloud computing models on the basis of a 5 factors namely, Cost, Availability, Speed, Latency and Energy Consumption. The purpose of this comparison is to (1) Describe the pros and cons of the two models. (2) To explain how they can complement one another.

11.5.1 The big four

11.5.1.1 Cost

Traditional cloud infrastructures come at a cost that heavily depends on a variety of different factors. In traditional on-demand cloud computing platforms such as Amazon, a customer is able to acquire and use a portion of virtualized hardware for an unlimited amount of time and is charged by-the-usage [5]. In the case of our experiments, we used EC2 instances to evaluate our application on the cloud.

Amazon's EC2 service offers a variety of different instances and prices vary depending on a number of factors such as the region it is hosted in, the OS installed, the hardware specifications, etc. For our use-case, we used 3 T2.micro EC2 instances in different regions with an average per-hour price of **$0.0266** which approximately equates to $20 ex per month and to **$240** for the 3 instances.

[5] https://en.wikipedia.org/wiki/Amazon_Web_Services

This does not take into account additional costs due to occasional bursts in performance that may be needed when an instance's CPU usage goes above the baseline performance percentage, as explained in 11.3.4.1.
Furthermore, the total price of $240 excludes VAT and all the additional cost of other Amazon services that are used or are required for the correct functioning of our EC2 instances. We estimate that the final price per year including all the additional costs could be in the **$400 to $500** range.

Compared to the cloud, the edge infrastructure we evaluated was comprised of 3 GRiSP boards. The price per GRiSP is **$244** VAT included which gives a total of **$731** for the 3 GRiSP boards. The price per board is in fact quite high, considering that there are much cheaper and more powerful IoT devices alternatives. However, GRiSP boards are one of the first of their kind to integrate an Erlang runtime system.

From these calculations and using our estimated cost for the EC2 instances per year, we can conclude that the EC2 instances are cheaper per year than the GRiSP boards. Additionally, as explained in Section 11.4.2.2, the EC2 instances are around 100 orders of magnitude more performant then the GRiSP boards. However, it is important to put the latter conclusions into perspective.
Firstly, the GRiSP boards imply no additional cost per year because we own the latter, on the contrary of the EC2 instances which are owned by a third-party and are to be payed on a monthly-basis for as long as they are used. Secondly, it is true that the cloud servers we used are more performant than the GRiSP boards. However, with the advent and the expontential growing number of IoT devices per year, the cost of the latter will keep on decreasing while their computing capacity will continue to grow [1]. Lastly, it is important to note that the purpose of these two models are different in essence but they can complement each other as we describe in Section 11.5.2.

11.5.1.2 Availability

As far as availability is concerned, the better model between the two is the cloud computing model. Edge networks are unreliable by nature as they are composed of heterogeneous and loosely coupled computing nodes [45]. On the other hand Amazon has guarantees on their monthly uptime percentage (at least 99.99%) and will go as far as to partlially reimburse users if this threshold is reached.[3]

11.5.1.3 Speed

As explained in Section 11.4.2.2, even the smallest Amazon EC2 instance is 100 orders of magnitude more powerful then a GRiSP board in terms of computation times. (explain convergence time on cloud)

11.5.1.4 Latency

In our experiments, we did not take latency into consideration while testing the edge computing model as we only had a cluster of 3 GRiSP boards that were physically close to each other. However, in a large edge network comprising of thousands of edge devices running distributed tasks, latency might become problematic.

11.5.1.5 Energy Consumption

11.5.2 Complementarity of the models

The Edge computing model was not created as a remplacement of the cloud computing model but instead to completement the latter. With the exponential growth of the number of IoT devices, estimated to surpass the 50 billion mark by 2023 [47], a problematic arises as to how and where will the data generated by IoT devices be processed and stored.
Clearly, the cloud computing model will not be able to sustain this unprecedented amount of data. The edge computing model was proposed to alleviate this task by moving part of the storage and computations on the devices themselves at the logical extreme of the network. Furthermore, IoT devices with sufficient computational power and storage are instructed to perform aggregations on the gathered data periodically.
Eventually, aggregated data can be sent to the cloud for complementary processing and storage. In essence, IoT devices are tasked with offloading a significant fraction of the workload of data centers located in the cloud by doing local aggregations of data.[45]. There are many benefits of using the edge computing model in combination with the cloud computing model. Two notably important benefits are :

- "Reducing the amount of data handled (stored, processed, and maintained) at the data centers, which also leads to lower costs in using cloud infrastructures." [45]
- "Reducing the network load by decreasing the data to transport by processing and aggregating it at the edge" [45]

11.6 Conclusion

In this chapter, we described the experiments we did on two different computing models. Further, we presented the results that we obtained and compared the two computing models. We have also shown that the goals they aim to achieve are quite different in essence. However, used as a complementary of the other, they form an augmented computing model that may become a viable solution to the exponentially increasing number of IoT devices and amount of data generated by the latter.

Chapter 12

Conclusions

The realization of this thesis was not a trivial task. Bleeding edge technologies are inherently linked to a lack of documentation. Especially, the presented project is amongst the first ones to aim at porting Lasp on GRiSP. As such, many of the presented engineering solutions required tremendous amounts of effort and testing in order to work consistently. A summary of our main contributions is given hereafter:

- **GrispLasp**. We successfully created a stable release of an Erlang edge application comprised of Lasp running on a cluster of GRiSP boards in an edge network. This application can be used to do general sensor sensing as well as executing distributed edge computations.
- **Task Model**. We implemented a framework for managing and orchestrating distributed tasks in an edge network. Augmented with a load balancer, the task framework executes generic high order Erlang functions that can contain simple local or distributed computations or even entire self-contained programs.
- **Local Edge Server and Dashboard**. We designed and implemented an Elixir webserver comprised of a Dashboard web application aswell as exposing an API for external users to fetch sensor data from specific nodes in a Lasp cluster. The Dashboard is used to visualize the progression of sensor's data in pseudo-realtime.
- **Persistence framework**. The persistence framework is a module specifically designed to alleviate memory pressure on resource constrained edge devices by persisting CRDT states in files in a hierarchical structure.
- **CloudLasp**. We were able to deploy three Elixir servers running Lasp on the cloud and cluster them together.
- **Framework for evaluating GrispLasp**. We created a benchmarking component for measuring the performance of computations and convergences times of a generic task in an edge environment.

We believe that this thesis work still has room for further extension and improvement and we share our ideas and suggestions for the future in Section 12.2.

12.1 Results

Chapter 11 presented the experiments for which we evaluated the overall performances of an edge application running either completely on the edge (GrispLasp) or on the edge with periodical communications with the cloud (CloudLasp). A summary of the results are given hereafter:

12.1.1 GrispLasp

We measured the limits and performances of a Lasp application running on GRiSP boards in an edge network. Based on our results, we conclude that the primary limiting factor for the overall stability of an edge application using our experimental framework is the size of the allocated memory. The latter is inherently constrained by the GRiSP board's SDRAM. Nevertheless, we believe that if proper precautions are taken regarding memory management, and if the edge application is not instructed to run heavy-duty tasks that may overload the system, the latter could run for a decent period of time with increased stability.

12.1.2 CloudLasp

After comparing the current cloud architecture and a cloud architecture helped by the edge model, we can observe that as mentioned in the edge computation theory, the current model does not need to be replaced but rather delegate a part of its a work to the edge model.

12.2 Future Work

In the following section, we provide a summary of potential improvements that could be made to this thesis work.

12.2.1 GRiSP

The GRiSP project is still under heavy development and is receiving new updates regularly. In the following section we propose a few improvements that can be made to GRiSP in the context of our thesis work:

12.2.1.1 Additional sensor integration

During the realization of this thesis work, we received a variety of different Digilent Pmod™ sensors devices, offered by the Lightkone project. We integrated and used the following sensors in our applications: Nav, Als, Gyro, Max Sonar.

There are still a number of Pmod sensors that can be integrated and tested in real world scenarios using GrispLasp. We provide a non-exhaustive list of the latter: Amp2, Tmp2, BLE, 8LD

12.2.1.2 Performance and memory enhancements

One of the major issues we were confronted with during this thesis work were the memory limitations of the GRiSP boards. We tried our best to cope with these limitations by improving the Erlang memory allocation strategies and by optimizing the overall quality of the Erlang code we wrote as explained in Section 10.4.
We believe that there are still considerable improvements that can be made to the memory management part of our work.
Specifically, we had issues with GRiSP boards crashing due to insufficient heap memory. These improvements will be paramount to improve the overall stability and resilience of GRiSP boards running on the edge.
Nevertheless, memory in Erlang is a vast and complex subject that requires excellent understanding of how the latter works and will prove to be challenging.

12.2.2 Task model

As a result of this thesis work, in Section 8.1 we proposed and created a framework for orchestrating task on a network composed of GRiSP boards with the use of Lasp for global replication. Tasks are resilient to failures as they are replicated on all the nodes that are part of the cluster. This framework is central to our application as it is used to manage and execute tasks which perform local or distributed edge computations with the use of generic high order functions. This framework has many possibilities of improvements and we propose a few ideas that are presented hereafter:

- An interesting idea to improve this framework would be to allow a task to execute BEAM files directly instead of Erlang functions.
- As explained in Section 10.4.2, the task model gives no restrictions to the type of software that can be executed by a task and lead us to observe crashes among nodes running some heavy-duty tasks. An idea would be to add a mechanism for analyzing and detecting potential memory-hungry generic functions that are stored in the tasks. This is certainly a difficult task to implement but the reward would be significant in terms of preventing potential nodes from crashing due to tasks and would allow an overall more stable environment.

12.2.3 Local Edge Server and Dashboard

12.2.3.1 Regression analysis

Data generated by sensors in an IoT network can be significant. Naturally, the purpose of gathering such data is to process and analyze the latter at some point in the future. An interesting aspect of sensor data analysis is to estimate the relationship that a particular dependent variable can have with other independent variables. This statistical process of estimating relationships among variables is called **regression analysis** [1]. In the context of our thesis work, we gathered data from a multitude of different sensors and were able to present this data visually on the Dashboard presented in Section 9.3. An idea to further improve the latter would be to perform regression analysis on different gathered sensor data sets from boards in an edge network and to illustrate these regressions on the Dashboard. With regression analysis, one could then be able to predict and forecast a variable. A few possible regression analyses are proposed below:

- Regression analysis on the ambient light sensor data.
- Regression analysis on the temperature sensor data.

For example, in both cases, one could predict the ambient light or temperature at a particular minute of the day based on gathered data.

12.2.3.2 Local decision making

An important aspect of IoT devices is the notion of device control. Direct management of IoT devices is inconvenient and cumbersome, especially when the amount of devices to manage is large. To alleviate this task, an idea is to implement partial

[1] https://en.wikipedia.org/wiki/Regression_analysis

management of IoT devices through a web or desktop application interface.
In the context of our thesis work, this interface (the Dashboard presented in Section 9.3) is already created but lacks direct management of local nodes in the edge cluster. A user should be able to interact with nodes within the cluster directly from the dashboard. A practical solution is to create a new page on the the latter that displays control elements such as buttons or forms that would offer the ability to a user to trigger events on the boards directly. Naturally, the Dashboard is only the visual interface and communicates with the local edge server which is responsible for the underlying communication with the nodes in the cluster. Thus, the task of improving the existing solution implies modifications to both the visual interface (the dashboard) as well as the controller component (the local edge server).
The task framework we have implemented is a viable solution to combine with the control management interface. An event triggering button on the dashboard would create and save a task in the tasks CRDT that is subsequently replicated on the nodes in the cluster. The nodes can then fetch and execute this task's function as explained in 8.2.3. Since the latter is ran directly on a node, the former is able to control any parts of the Erlang system running on the board if needed. Furthermore, as a task can have different target nodes in the cluster, the implementation of the dashboard's control management interface would ease the control of all or a subset of nodes in the cluster. Different dashboard control management implementation ideas are proposed below in no specific order:

- Offer the ability to enable or disable one or more sensors on a defined set of nodes in the cluster.
- Offer the ability to start or stop pre-defined tasks on a defined set of nodes.
- Allow a user to create a task directly from the dashboard. This task will then be replicated on the nodes in the cluster and executed by the the nodes that are targets of this task.
- Offer the ability to control and manage the time at which persistence of data from CRDT's should be performed on the nodes as explained in Section 10.4.5.

Appendix A

Source code and User's Manual

For a complete guide on how to install and use Lasp applications ported on GRiSP boards, refer to our Github organisation which is comprised of all our repositories and the source code used during this thesis work which you can find at https://github.com/GrispLasp.

A.1 Pmod MAX sonar

```
-module(pmod_maxsonar).

-behavior(gen_server).

% API
-export([start_link/2]).
-export([get/0]).

% Callbacks
-export([init/1]).
-export([handle_call/3]).
-export([handle_cast/2]).
-export([handle_info/2]).
-export([code_change/3]).
-export([terminate/2]).

%-- Records ------------------------------------------------------------------

-record(state, {port, list_val}).

%-- API ----------------------------------------------------------------------

% @private
start_link(Slot, _Opts) ->
    gen_server:start_link({local, ?MODULE}, ?MODULE, Slot, []).

get() ->
    gen_server:call(?MODULE, get_value).

%-- Callbacks ----------------------------------------------------------------

% @private
init(Slot = uart) ->
    Port = open_port({spawn_driver, "grisp_termios_drv"}, [binary]),
    grisp_devices:register(Slot, ?MODULE),
    {ok, #state{port = Port, list_val = []}}.

% @private
handle_call(get_value, _From, #state{list_val = Val} = State) ->
    {reply, Val, State}.


% @private
handle_cast(Request, _State) -> error({unknown_cast, Request}).

% @private
handle_info({Port, {data, Data}}, #state{port = Port, list_val = Previous} = State) ->
 if
  byte_size(Data) > 5 -> NewList = Previous;
  length(Previous) > 5 -> grisp_led:color(1,green), NewList = [];
  byte_size(Data)==5 -> grisp_led:color(1,red),
                        <<_:8,Val1:8,Val2:8,Val3:8,Rest>> = Data,
                        List = [Val1,Val2,Val3],
                        {NewValue,_} = string:to_integer(List),
                        % io:format("Value is: ~p ~n",[NewValue]),
                        if
                          NewValue < 50 -> NewList = lists:append([NewValue],Previous);
                        true -> NewList = []
                        end;
  true -> NewList = Previous

  end,
  {noreply, State#state{list_val = NewList}}.



% @private
code_change(_OldVsn, State, _Extra) -> {ok, State}.

% @private
terminate(_Reason, _State) -> ok.
```

A.2 Pmod ALS driver

```
-module(pmod_als).

-behavior(gen_server).

-include("grisp.hrl").

% API
-export([start_link/2]).
-export([raw/0]).
-export([precise/0]).

% Callbacks
-export([init/1]).
-export([handle_call/3]).
-export([handle_cast/2]).
-export([handle_info/2]).
-export([code_change/3]).
-export([terminate/2]).

-define(SPI_MODE, #{cpol => high, cpha => trailing}).

%-- Records ------------------------------------------------------------

-record(state, {
    slot,
    value
}).

%-- API ----------------------------------------------------------------

% @private
start_link(Slot, _Opts) -> gen_server:start_link(?MODULE, Slot, []).

raw() ->
    Result = call(raw),
    Result.

precise() ->
     Result = call(precise),
     if
      Result < 34 -> grisp_led:color(1,blue);
      Result < 67 -> grisp_led:color(1,green);
      true -> grisp_led:color(1,red)
    end,
    Result.


%-- Callbacks ----------------------------------------------------------

% @private
init(Slot) ->
    grisp_devices:register(Slot, ?MODULE),
    erlang:send_after(5000, self(), precise),
    {ok, #state{slot = Slot}}.

% @private
handle_call(raw, _From, State) ->
    Raw = get_value(State#state.slot),
    {reply, Raw, State}.

% @private
handle_cast(Request, _State) -> error({unknown_cast, Request}).

handle_info(precise, State) ->
  Raw = get_value(State#state.slot),
  Precise = (Raw/255) * 100,
    {noreply, State,1000};

handle_info(timeout, State) ->
  Raw = get_value(State#state.slot),
  Precise = (Raw/255) * 100,
  if
   Precise < 34 -> grisp_led:color(1,blue);
```

```
    Precise < 67 -> grisp_led:color(1,green);
    true ->  grisp_led:color(1,red)
  end,
  {noreply, State, 1000}.
% @private
code_change(_OldVsn, State, _Extra) -> {ok, State}.

% @private
terminate(_Reason, _State) -> ok.

%-- Internal -----------------------------------------------------------

call(Call) ->
    Dev = grisp_devices:default(?MODULE),
    gen_server:call(Dev#device.pid, Call).

get_value(Slot) ->
    <<_:3,Resp:8,Pad:5>> = grisp_spi:send_recv(Slot, ?SPI_MODE, <<0:8>>, 0, 1),
    Resp.
```

A.3 UART Erlang driver

```
% @private
-module(grisp_termios_drv).

% API
-export([open/0]).
-export([command/2]).

%-- Macros -------------------------------------------------------------------

-define(PORT_COMMAND_TIMEOUT, 2000).

%-- API ----------------------------------------------------------------------

open() -> open_port({spawn_driver, "grisp_termios_drv"}, [binary]).

command(Port, Command) ->
    Port ! {self(), {command, <<Command/binary>>}},
    receive
        {Port, {data, Resp}} ->
            Resp
    after ?PORT_COMMAND_TIMEOUT ->
            exit({i2c_driver_timeout, Command})
    end.
```

A.4 UART C driver

```
/* grisp_termios_drv.c */

#include <assert.h>
#include <fcntl.h>
#include <rtems.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <termios.h>

#include "erl_driver.h"
#include "sys.h"

#define BUF_SIZE 256

ErlDrvData grisp_tio_start (ErlDrvPort port, char *command);
void grisp_tio_stop (ErlDrvData drv_data);
void grisp_tio_ready_in(ErlDrvData drv_data, ErlDrvEvent event);
void grisp_tio_stop_select (ErlDrvEvent event, void* reserved);

ErlDrvEntry grisp_termios_driver_entry = {
    NULL,
    grisp_tio_start,
    grisp_tio_stop,
    NULL,
    grisp_tio_ready_in,
    NULL,
    "grisp_termios_drv",
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    ERL_DRV_EXTENDED_MARKER,
    ERL_DRV_EXTENDED_MAJOR_VERSION,
    ERL_DRV_EXTENDED_MINOR_VERSION,
    0,
    NULL,
    NULL,
    grisp_tio_stop_select,
};

struct grisp_tio_data {
  ErlDrvPort port;
  int fd;
  char buf[BUF_SIZE];
};

ErlDrvData grisp_tio_start (ErlDrvPort port, char *command)
{
  struct grisp_tio_data *data;
  struct termios term;

  data = (struct grisp_tio_data *)driver_alloc(sizeof(struct grisp_tio_data));

  data->port = port;

  data->fd = open("/dev/ttyUSART0", O_RDWR);
  if (data->fd < 0)
    return ERL_DRV_ERROR_ERRNO;

  if (tcgetattr(data->fd, &term) < 0)
    return ERL_DRV_ERROR_ERRNO;

  cfsetispeed(&term, B9600);
  cfsetospeed(&term, B9600);

  term.c_lflag |= ICANON;
  term.c_lflag &= ~(ECHO | ECHOE | ECHOK | ECHONL);
```

```
  term.c_iflag |= ICRNL;

  term.c_cflag &= ~CSIZE;
  term.c_cflag |= CS8;
  term.c_cflag &= ~(CSTOPB | PARENB);


  if (tcsetattr(data->fd, TCSAFLUSH, &term) < 0)
    return ERL_DRV_ERROR_ERRNO;

  driver_select(data->port, (ErlDrvEvent)data->fd, ERL_DRV_READ, 1);

  return (ErlDrvData)data;
}

void grisp_tio_stop (ErlDrvData drv_data)
{
  struct grisp_tio_data *data = (struct grisp_tio_data *)drv_data;

  driver_select(data->port, (ErlDrvEvent)data->fd, ERL_DRV_USE, 0);
  driver_free(data);
}

void grisp_tio_ready_in (ErlDrvData drv_data, ErlDrvEvent event)
{
  ssize_t len;

  struct grisp_tio_data *data = (struct grisp_tio_data *)drv_data;

  len = read(data->fd, data->buf, BUF_SIZE);
  if (len < 0)
    driver_failure_posix(data->port, errno);
  else
    driver_output(data->port, data->buf, len);
}

void grisp_tio_stop_select (ErlDrvEvent event, void* reserved)
{
  close((int)event);
}
```

Appendix B

Node core modules source

B.1 Node storage utility

```
-module(node_storage_util).

-include_lib("node.hrl").

-compile(export_all).

-define(WILDCARD,      "_???").
-define(SEPARATOR,      "_").
-define(MAXPART,      3).

%% ==================================================================
%% API functions
%% ==================================================================

flush_crdt(Id, _FilenameOld, Mode) ->
    TableName = node_util:lasp_id_to_atom(Id),
    case Mode of
        % Save table entry to file
        % without removing the values from
        % the set. Local table is erased
        % and querying cannot retrieve previous
        % elements. State convergence is achieved
        % locally with a set containing only results
        % of subsequent operations on the CRDT.
        % If the saved state is loaded back again,
        % it replaces the current state and unsaved mutations
        % that were operated inbetween are lost.
      save_no_rmv_all ->
        % TODO : check if table is empty
          Tmp = ets:new(TableName, [ordered_set, named_table, public]),
          case ets:insert_new(Tmp, ets:take(node(), Id)) of
              true ->
                  {ok, Name} = get_filename(Id),
                  case ets:tab2file(Tmp, Name, [{sync, true}]) of
                    ok ->
                      logger:log(info, "Saved CRDT ~p to file ~p ~n", [Id, Name]),
                      true = ets:delete(Tmp),
                      true = ets:delete(node(), Id),
                      {ok, Id, Name};
                    {error, Reason} ->
                      lager:error("Could not save ~p to SD card ~n", [Id]),
                      {error, Reason}
                  end;
              false ->
                lager:error("Could not insert ~p values in tmp table ~n", [Id]),
                {error, insert}
          end;
      % Save table entry to file and remove
      % all local table entries from the state.
      % The set is updated before the flush and
      % the operations are replicated.
      % Convergence is achieved locally with
      % regular delta-based mutations since it
      % is also the global converging state.
      % If the saved state is loaded back again,
      % it replaces the current state and unsaved mutations
      % that were operated inbetween are lost.
```

```
        save_rmv_all ->
            L = values_to_list(Id),
            {ok, {NewId, _NewType, _NewMeta, _NewVal}} = lasp:update(Id, {rmv_all, L}, self()),
            Delay = application:get_env(lasp, delta_interval, ?FIVE),
            ?SLEEP(Delay + ?ONE),
            Tmp = case ets:info(TableName) of
                undefined ->
                    ets:new(TableName, [ordered_set, named_table, public]);
                _ ->
                    ets:delete(TableName),
                    ets:new(TableName, [ordered_set, named_table, public])
                end,
            New = ets:take(node(), NewId),
            case ets:insert(Tmp, New) of
                true ->
                    {ok, Name} = get_filename(Id),
                    case ets:tab2file(Tmp, Name, [{sync, true}]) of
                      ok ->
                        logger:log(notice, "Saved CRDT ~p to file ~p ~n", [Id, Name]),
                        true = ets:delete(Tmp),
                        true = ets:delete(node(), Id),
                        logger:log(info, "=== Saved state ===~n"),
                        logger:log(info, "State = ~p ~n", [New]),
                        logger:log(info, "=== Saved state ===~n"),
                        {ok, Id, Name};
                      {error, Reason} ->
                        lager:error("Could not save ~p to SD card ~n", [Id]),
                        {error, Reason}
                    end;
                false ->
                  lager:error("Could not insert ~p values in tmp table ~n", [Id]),
                  {error, insert}
            end;
          _ -> {error, unknown_mode}
    end.

load_crdt(Id, _FilenameOld, Part) ->
    case get_part(Id, Part) of
        {ok, Filename} ->
            logger:log(info, "Part number ~p of saved state of ~p successfully retrieved ~n", [
    Part,Id]),
            case ets:file2tab(Filename) of
                {ok, Tab} ->
                    L = ets:tab2list(Tab),
                    case ets:insert_new(node(), L) of
                        true ->
                            logger:log(info, "Loaded CRDT from file ~p ~n", [Filename]),
                            {ok, Id, Filename};
                        false ->
                            logger:log(info, "Overriding current state with saved state in ~p ~n
    ", [Filename]),
                            ets:insert(node(), L),
                            ets:delete(Tab, L),
                            {ok, Id, Filename}
                    end;
                {error, Reason} ->
                    lager:error("Could not load CRDT from file ~p ~n", [Filename]),
                    case Reason of
                        cannot_create_table ->
                            ets:delete(node_util:lasp_id_to_atom(Id)),
                            load_crdt(Id, _FilenameOld, Part);
                        _ -> {error, file2tab_unknown_reason}
                    end
            end;
        {error, no_such_part} ->
            logger:log(error, "Part number ~p of saved state of ~p not found ~n", [Part,Id]),
            {error, no_such_part};
        _ ->
            {error, unknown}
    end.

%% ====================================================================
%% Internal Functions
%% ====================================================================

values_to_list(Id) ->
```

```
  {ok, Var} = lasp:query(Id),
  sets:to_list(Var).

values_to_set(Id) ->
  {ok, Var} = lasp:query(Id),
  Var.

get_filename({BitString, _Type}) ->
    Filename = binary_to_list(BitString),
    Fullname = filename:join(node_config:get(data_dir, "data"), Filename),
    logger:log(info, "Full base name of destination file : ~p ~n", [Fullname]),
    case filelib:ensure_dir(Fullname) of
        ok ->
            IsDir = filelib:is_dir(Fullname),
            if
                true == IsDir ->
                    logger:log(info, "Directory ~p exists ~n", [Fullname]),
                    Part = check_part(Filename, Fullname),
                    logger:log(info, "Part is ~p ~n", [Part]),
                    {ok, filename:join([Fullname, Part])};
                true ->
                    logger:log(info, "Creating Directory ~p ~n", [Fullname]),
                    case file:make_dir(Fullname) of
                        ok ->
                            logger:log(info, "Directory ~p created ~n", [Fullname]),
                            Part = check_part(Filename,Fullname),
                            logger:log(info, "Part is ~p ~n", [Part]),
                            {ok, filename:join([Fullname, Part])};
                        {error, Reason} ->
                            logger:log(error, "Failed to create Directory ~p ~n", [Fullname]),
                            {error, Reason}
                    end
            end;
        _ ->
            {error, unknown}
    end.

check_part(Name, Dir) ->
    Basename = filename:join(Dir,Name),
    L = filelib:wildcard(unicode:characters_to_list([Basename, ?WILDCARD], utf8)),
    Len = length(L),
    % TODO : alert when MAXPART limit is reached
    Offset = integer_to_list(Len),
    Pad = lists:duplicate((?MAXPART - length(Offset)), "0"),
    Part = unicode:characters_to_list([Name, ?SEPARATOR, Pad, Offset], utf8),
    logger:log(info,"Part number ~p ~n", [Part]),
    Part.

get_part({BitString, _Type}, Num) when is_integer(Num); Num >= 0; Num =< ?MAXPART ->
    Filename = binary_to_list(BitString),
    Fullname = filename:join(node_config:get(data_dir, "data"), Filename),
    IsDir = filelib:is_dir(Fullname),
    if
        true == IsDir ->
            % return part
            Offset = integer_to_list(Num),
            Pad = lists:duplicate((?MAXPART - length(Offset)), "0"),
            Part = unicode:characters_to_list([Filename, ?SEPARATOR, Pad, Offset], utf8),
            PartFile = filename:join(Fullname, Part),
            IsFile = filelib:is_file(PartFile),
            if
                true == IsFile ->
                    {ok, PartFile};
                true ->
                    {error, no_such_part}
            end;
        true ->
            {error, no_such_directory}
    end.

%% ==============================================================
%% Utility Functions
%% ==============================================================

get_test() ->
  lasp:query({<<"test">>, state_orset}).
```

```
members() ->
  lasp_peer_service:members().

get_id() -> {<<"test">>, state_orset}.

look() ->
  ets:lookup(node(), get_id()).

update() ->
    update(add_all, ["hello", "hi", "test"]).

update(Op, Elems) ->
    lasp:update(get_id(), {Op, Elems}, self()).

gc() ->
    % TODO : compare fragmentation before and after
    _ = [logger:log(info, "Mem = ~p ~n", [X]) || X <- mem()],
    _GC = [erlang:garbage_collect(Proc, [{type, 'major'}]) || Proc <- processes()],
    logger:log(notice, "Garbage was collected manually"),
    _ = [logger:log(info, "Mem = ~p ~n", [Y]) || Y <- mem()],
    ok.

mem() ->
  [{X, erlang:round(recon_alloc:memory(X) / 1024)} || X <- [allocated, used, usage]].
```

B.2 Node Generic Task Server

```
-module(node_generic_tasks_server).
-behaviour(gen_server).

-include_lib("node.hrl").

%% API
-export([start_link/0, terminate/0, add_task/1, remove_all_tasks/0, remove_task/1, get_all_tasks
     /0, find_task/1]).

%% Gen Server Callbacks
-export([init/1, handle_call/3, handle_cast/2, handle_info/2, terminate/2, code_change/3]).

%% Records


%% ====================================================================
%% API functions
%% ====================================================================

start_link() -> gen_server:start_link({local, ?MODULE}, ?MODULE, [], []).

terminate() -> gen_server:call(?MODULE, {terminate}).

add_task(Task) -> gen_server:call(?MODULE, {add_task, Task}).

remove_task(Name) -> gen_server:call(?MODULE, {remove_task, Name}).

remove_all_tasks() -> gen_server:call(?MODULE, {remove_all_tasks}).

get_all_tasks() -> gen_server:call(?MODULE, {get_all_tasks}).

find_task(Name) -> gen_server:call(?MODULE, {find_task, Name}).


%% ====================================================================
%% Gen Server callbacks
%% ====================================================================

init([]) ->
  logger:log(info, "Starting a generic tasks server ~n"),
  %% Ensure Gen Server gets notified when his supervisor dies
```

```
  Vars = node_config:get(generic_tasks_sets_names, []),
  node_util:declare_crdts(Vars),
  process_flag(trap_exit, true),
  {ok, {}}.

handle_call({add_task, {TaskName, Targets, Fun}}, _From, State) ->
  logger:log(info, "=== ~p ~p ~p ===~n", [TaskName, Targets, Fun]),
  Task = {TaskName, Targets, Fun},
  {ok, Tasks} = lasp:query({<<"tasks">>, state_orset}),
  TasksList = sets:to_list(Tasks),
  TaskExists = [{Name, Targets, Fun} || {Name, Targets, Fun} <- TasksList, Name =:= TaskName],
  case length(TaskExists) of
    1 ->
      logger:log(info, "=== Error, task already exists ==="),
      {reply, {ko, task_already_exist}, State};
    0 ->
      lasp:update({<<"tasks">>, state_orset}, {add, Task}, self()),
      {reply, ok, State}
  end;


handle_call({remove_task, TaskName}, _From, State) ->
  {ok, Tasks} = lasp:query({<<"tasks">>, state_orset}),
  TasksList = sets:to_list(Tasks),
  TaskToRemove = [{Name, Targets, Fun} || {Name, Targets, Fun} <- TasksList, Name =:= TaskName],
  case length(TaskToRemove) of
    1 ->
      ExtractedTask = hd(TaskToRemove),
      logger:log(info, "=== Task to Remove ~p ===~n", [ExtractedTask]),
      lasp:update({<<"tasks">>, state_orset}, {rmv, ExtractedTask}, self());
    0 ->
      logger:log(info, "=== Task does not exist ===~n");
    _ ->
      logger:log(info, "=== Error, more than 1 task === ~n")
  end,
  {reply, ok, State};


handle_call({remove_all_tasks}, _From, State) ->
  {ok, Tasks} = lasp:query({<<"tasks">>, state_orset}),
  TasksList = sets:to_list(Tasks),
  lasp:update({<<"tasks">>, state_orset}, {rmv_all, TasksList}, self()),
  {reply, ok, State};

handle_call({get_all_tasks}, _From, State) ->
  {ok, Tasks} = lasp:query({<<"tasks">>, state_orset}),
  TasksList = sets:to_list(Tasks),
  {reply, TasksList, State};

handle_call({find_task, TaskName}, _From, State) ->
  {ok, Tasks} = lasp:query({<<"tasks">>, state_orset}),
  TasksList = sets:to_list(Tasks),
  Task = [{Name, Targets, Fun} || {Name, Targets, Fun} <- TasksList, Name =:= TaskName],
  case length(Task) of
    0 ->
      {reply, task_not_found, State};
    1 ->
      {reply, {ok, hd(Task)}, State};
    _ ->
      {reply, more_than_one_task, State}
  end;

handle_call(stop, _From, State) ->
  {stop, normal, ok, State}.


handle_info(Msg, State) ->
    logger:log(info, "=== Unknown message: ~p~n", [Msg]),
    {noreply, State}.

handle_cast(_Msg, State) -> {noreply, State}.

terminate(Reason, _S) ->
  logger:log(error, "=== Terminating Generic server (reason: ~p) ===~n",[Reason]),
  ok.
```

```
code_change(_OldVsn, S, _Extra) ->
  {ok, S}.


%%====================================================================
%% Internal Functions
%%====================================================================
```

B.3 Node Generic Task Worker

```
-module(node_generic_tasks_worker).
-behaviour(gen_server).

-include_lib("node.hrl").

%% API
-export([start_link/0, find_and_start_task/0, start_task/1, start_all_tasks/0, isRunning/1, stop
    /0]).

%% Gen Server Callbacks
-export([init/1, handle_call/3, handle_cast/2, handle_info/2, terminate/2, code_change/3]).

%% Records
-record(state, {running_tasks = [],
                finished_tasks = [],
                restart_interval = node_config:get(generic_tasks_restart_interval, ?MIN)}).


%% ====================================================================
%% API functions
%% ====================================================================

start_link() ->
  gen_server:start_link({local, ?MODULE}, ?MODULE, {}, []).

start_task(Name) ->
  gen_server:call(?MODULE, {start_task, Name}).

find_and_start_task() ->
  gen_server:call(?MODULE, {find_and_start_task}).

start_all_tasks() ->
  gen_server:call(?MODULE, {start_all_tasks}).

isRunning(TaskName) ->
  gen_server:call(?MODULE, {isRunning, TaskName}).

stop() ->
  gen_server:call(?MODULE, stop).

%% ====================================================================
%% Private functions
%% ====================================================================


%% ====================================================================
%% Gen Server callbacks
%% ====================================================================


init({}) ->
    logger:log(notice, "Initializing Node Server~n"),
    RestartInterval = node_config:get(generic_tasks_restart_interval, ?MIN),
    % erlang:send_after(5000, self(), {start_all_tasks}),
    % {ok, #state{}}.
    {ok, #state{running_tasks=[], finished_tasks=[], restart_interval = RestartInterval}}.



handle_call({start_task, Name}, _From, State = #state{running_tasks=RunningTasks, finished_tasks
    =FinishedTasks}) ->
```

```
    logger:log(info, "=== State is ~p ===~n", [State]),
    logger:log(info, "=== Finding task ~p ===~n", [Name]),
    CanRunTask = can_run_task(length(RunningTasks)),
    case CanRunTask of
      true ->
        Task = node_generic_tasks_server:find_task(Name),
        case Task of
          {ok, TaskFound} ->
            NewFinishedTasksList = FinishedTasks -- [TaskFound],
            TaskFun = element(3,TaskFound),
            logger:log(info, "=== Task chosen ~p ===~n", [TaskFound]),
            {Pid, Ref} = spawn_monitor(TaskFun),
            logger:log(info, "=== Spawned Task fun : PID ~p - Ref ~p ===~n", [Pid, Ref]),
            RunningTask = erlang:insert_element(4, TaskFound, {Pid, Ref}),
            logger:log(info, "=== Running Task ~p ===~n", [RunningTask]),
            {reply, RunningTask, State#state{running_tasks=RunningTasks ++ [RunningTask],
     finished_tasks=NewFinishedTasksList}};
          Error ->
            {reply, Error, State}
        end;
      false ->
        logger:log(notice, "=== Cannot run task, device is overloaded ===~n"),
        {reply, ko, State}
      end;


handle_call({find_and_start_task}, _From, State = #state{running_tasks=RTasks, finished_tasks=
     FinishedTasks}) ->
    RunningTasks = maybe_tuple_to_list(RTasks),
    logger:log(info, "=== State is ~p ===~n", [State]),
    logger:log(info, "=== Finding new task ===~n"),
    TasksList = node_generic_tasks_server:get_all_tasks(),
    logger:log(info, "=== Tasks list ~p ===~n", [TasksList]),
    FilteredTaskList = filter_task_list(TasksList, TasksList),
    logger:log(info, "=== FilteredTaskList ~p ===~n",[FilteredTaskList]),
    case length(FilteredTaskList) of
      0 ->
        {reply, no_tasks_to_run, State};
      _ ->
        RandomTaskIndex = rand:uniform(length(FilteredTaskList)),
        RandomTask = lists:nth(RandomTaskIndex, FilteredTaskList),
        CanRunTask = can_run_task(length(RunningTasks)),
        case CanRunTask of
          true ->
            NewFinishedTasksList = FinishedTasks -- [RandomTask],
            TaskFun = element(3,RandomTask),
            logger:log(info, "=== Task chosen ~p ===~n", [RandomTask]),
            {Pid, Ref} = spawn_monitor(TaskFun),
            logger:log(info, "=== Spawned Task fun : PID ~p - Ref ~p ===~n", [Pid, Ref]),
            RunningTask = erlang:insert_element(4, RandomTask, {Pid, Ref}),
            logger:log(info, "=== Running Task ~p ===~n", [RunningTask]),
            {reply, RunningTask, State#state{running_tasks=RunningTasks ++ [RunningTask],
     finished_tasks=NewFinishedTasksList}};
          false ->
            logger:log(info, "=== Cannot run task, device is overloaded ===~n"),
            {reply, ko, State#state{running_tasks=RunningTasks, finished_tasks=FinishedTasks}}
        end
    end;


handle_call({isRunning, TaskName}, _From, State = #state{running_tasks=RunningTasks,
     finished_tasks=_}) ->
  TaskRunning = [{Name, Targets, Fun, {TaskPid, TaskRef}} || {Name, Targets, Fun, {TaskPid,
     TaskRef}} <- RunningTasks, Name =:= TaskName],
  case lists:length(TaskRunning) of
    0 -> {reply, false, State};
    1 -> {reply, true, State};
    _ -> {reply, more_than_one_task, State}
  end;

handle_call(stop, _From, State) ->
  {stop, normal, ok, State};

handle_call(_Msg, _From, State) ->
  {noreply, State}.
```

```
handle_cast(_Msg, State) ->
  {noreply, State}.

handle_info({start_all_tasks}, State =
        #state{running_tasks=RunningTasks,
        finished_tasks=FinishedTasks,
        restart_interval=RestartInterval}) ->
    case start_all_tasks_periodically(RunningTasks, FinishedTasks) of
      {ko, no_tasks_to_run} ->
        logger:log(info, "=== No tasks to run ===~n"),
        {noreply, State#state{running_tasks=RunningTasks, finished_tasks=FinishedTasks},
     RestartInterval};
      {NewRunningTasksList, NewFinishedTasksList} ->
        {noreply, State#state{running_tasks=RunningTasks ++ [NewRunningTasksList],
     finished_tasks=NewFinishedTasksList}, RestartInterval}
      end;


handle_info(timeout, State =
        #state{running_tasks=RunningTasks,
        finished_tasks=FinishedTasks,
        restart_interval=RestartInterval}) ->
  case start_all_tasks_periodically(RunningTasks, FinishedTasks) of
    {ko, no_tasks_to_run} ->
      logger:log(info, "=== No tasks to run ===~n"),
      {noreply, State#state{running_tasks=RunningTasks, finished_tasks=FinishedTasks},
     RestartInterval};
    {NewRunningTasksList, NewFinishedTasksList} ->
      {noreply, State#state{running_tasks=RunningTasks ++ [NewRunningTasksList], finished_tasks=
     NewFinishedTasksList}, RestartInterval}
    end;



handle_info({'DOWN', Ref, process, Pid, Info}, State = #state{running_tasks=RunningTasks,
     finished_tasks=FinishedTasks}) ->
    logger:log(notice, "== Pid ~p has ended ===~n", [Pid]),
    RunningTasksList = [{Name, Targets, Fun, {TaskPid, TaskRef}} || {Name, Targets, Fun, {
     TaskPid, TaskRef}} <- RunningTasks, TaskPid =:= Pid],
    case length(RunningTasksList) of
      0 ->
        logger:log(info, "=== A process other than a task finished ===~n"),
        {noreply, State};
      1 ->
        {Name, Targets, Fun, {TaskPid, TaskRef}} = hd(RunningTasksList),
        case Info of
          normal -> logger:log(info, "=== Task ~p with Pid ~p finished gracefully (~p) ===~n", [
     Name, Pid, Info]);
          _ -> logger:log(error, "=== Problem: ~p ===~n", [Info])
        end,
        erlang:demonitor(Ref),
        NewRunningTasksList = lists:delete({Name, Targets, Fun, {TaskPid, TaskRef}},
     RunningTasks),
        NewFinishedTasksList = lists:append(FinishedTasks, [{Name, Targets, Fun}]),
        logger:log(info, "=== NRTL ~p , NFTL ~p ===~n", [NewRunningTasksList,
     NewFinishedTasksList]),
        {noreply, State#state{running_tasks=NewRunningTasksList, finished_tasks=
     NewFinishedTasksList}}
    end;

handle_info({'EXIT', _From, Reason}, State) ->
    logger:log(error, "=== Supervisor sent an exit signal (reason: ~p), terminating Gen Server
     ===~n", [Reason]),
    {stop, Reason, State};

handle_info(Msg, State) ->
  logger:log(notice, "=== Unknown message: ~p~n", [Msg]),
  {noreply, State}.

terminate(normal, _State) ->
  logger:log(info, "=== Normal Gen Server termination ===~n"),
  ok;

terminate(shutdown, _State) ->
  logger:log(info, "=== Supervisor asked to terminate Gen Server (reason: shutdown) ===~n"),
  ok;
```

```
terminate(Reason, _State) ->
  logger:log(info, "=== Terminating Gen Server (reason: ~p) ===~n",[Reason]),
  ok.

code_change(_OldVsn, State, _Extra) ->
  {ok, State}.


%%====================================================================
%% Internal functions
%%====================================================================

get_cpu_load() ->
  % Load = cpu_sup:avg1(),
  Load = 5,
  PercentLoad = 100 * (1 - 50/(50 + Load)).

get_device() ->
  os:getenv("type").

can_run_task(RunningTasksCount) ->
  {ok, CpuLoad} = node_utils_server:get_cpu_usage(),
  % CpuLoad = 20,
  logger:log(notice, "=== CPU load ~.2f ===~n",[CpuLoad]),
  DeviceType = get_device(),
  logger:log(notice, "=== Device is ~p ===~n",[DeviceType]),
  TresholdReached = case DeviceType of
    "grisp" ->
      if RunningTasksCount =< 2 -> false;
        true -> true
      end;
    "laptop" ->
      if RunningTasksCount =< 5 -> false;
        true -> true
      end
  end,
  logger:log(info, "=== Is threshold reached? ~p ===~n",[TresholdReached]),
  CanRun = if CpuLoad < 50, TresholdReached =:= false -> true;
    true -> false
  end,
  CanRun.


filter_task_list(TasksList, RTasks) ->
    RunningTasks = maybe_tuple_to_list(RTasks),
    logger:log(info, "Task List in filter_task_list = ~p ~n ", [TasksList]),
    logger:log(info, "Running List in filter_task_list = ~p ~n ", [RunningTasks]),

    FilteredTaskList = lists:filter(
    fun ({Name, Targets, _}) ->
    IsTarget = case Targets of
      all -> true;
      List -> lists:member(node(), List)
    end,
    TaskIsRunning = case length(RunningTasks) of
      0 -> false;
      _ ->
        lists:any(
        fun({ProcessingTaskName, _, _, _}) ->
          if ProcessingTaskName =:= Name -> false;
            true -> true
          end
        end, RunningTasks)
    end,
    logger:log(info, "=== Task is already running : ~p - Node is target : ~p ===~n", [
     TaskIsRunning, IsTarget]),
    IsCanditate = if TaskIsRunning =:= false, IsTarget =:= true -> true;
       true -> false
    end,
    IsCanditate
    end , TasksList),
    FilteredTaskList.

start_all_tasks_periodically(RunningTasks, FinishedTasks) ->
  logger:log(info, "=== Finding new task ===~n"),
```

```
  TasksList = node_generic_tasks_server:get_all_tasks(),
  logger:log(info, "=== Tasks list ~p ===~n", [TasksList]),
  FilteredTaskList = filter_task_list(TasksList, RunningTasks),
  case lists:length(FilteredTaskList) of
    0 ->
      {ko, no_tasks_to_run};
    _ ->
      NewFinishedTasksList = FinishedTasks -- FilteredTaskList,
      StartedTasks = lists:mapfoldl(
        fun(Task, StartedTasks) ->
          CanRunTask = can_run_task(length(RunningTasks)),
          case CanRunTask of
            true ->
              TaskFun = element(3, Task),
              logger:log(info, "=== Task chosen ~p ===~n", [Task]),
              {Pid, Ref} = spawn_monitor(TaskFun),
              logger:log(info, "=== Spawned Task fun : PID ~p - Ref ~p ===~n", [Pid, Ref]),
              RunningTask = erlang:insert_element(4, Task, {Pid, Ref}),
              logger:log(info, "=== Running Task ~p ===~n", [RunningTask]),
              {Task, StartedTasks ++ RunningTask};
            false ->
              logger:log(notice, "=== Cannot run task, device is overloaded ===~n"),
              {Task, StartedTasks}
          end
        end
      , [], FilteredTaskList),
      NewRunningTasksList = element(2, StartedTasks),
      {NewRunningTasksList, NewFinishedTasksList}
end.

maybe_tuple_to_list(Var) ->
    case is_tuple(Var) of
        true ->
            tuple_to_list(Var);
        false when is_list(Var) ->
            Var
    end.
```

B.4 Node Generic Task Functions

```
-module(node_generic_tasks_functions).
-compile(export_all).

-include("node.hrl").

nav_sensor(Comp, Register) ->
  % grisp:add_device(spi1, pmod_nav),
  % logger:log(info, "Value = ~p ~n", pmod_nav:read(alt, [press_out])),
  logger:log(notice, "Value = ~p ~n", pmod_nav:read(Comp, [Register])).


meteorological_statistics(SampleCount, SampleInterval, Trigger) ->
    % Must check if module is available
    {pmod_nav, Pid, _Ref} = node_util:get_nav(),
    % meteo = shell:rd(meteo, {press = [], temp = []}),
    % State = #{press => [], temp => [], time => []},
    State = maps:new(),
    State1 = maps:put(press, [], State),
    State2 = maps:put(temp, [], State1),
    State3 = maps:put(time, [], State2),

    FoldFun = fun
        (Elem, AccIn) when is_integer(Elem) andalso is_map(AccIn) ->
            timer:sleep(SampleInterval),
            T = node_stream_worker:maybe_get_time(),
            % T = calendar:local_time(),
            [Pr, Tmp] = gen_server:call(Pid, {read, alt, [press_out, temp_out], #{}}),
            % [Pr, Tmp] = [1000.234, 29.55555],
            #{press => maps:get(press, AccIn) ++ [Pr],
            temp => maps:get(temp, AccIn) ++ [Tmp],
```

```
            time => maps:get(time, AccIn) ++ [T]}
    end,

    M = lists:foldl(FoldFun, State3, lists:seq(1, SampleCount)),
    [Pressures, Temperatures, Epochs] = maps:values(M),

    Result = #{measures => lists:zip3(Epochs, Pressures, Temperatures),
        pmean => 'Elixir.Numerix.Statistics':mean(Pressures),
        pvar => 'Elixir.Numerix.Statistics':variance(Pressures),
        tmean => 'Elixir.Numerix.Statistics':mean(Temperatures),
        tvar => 'Elixir.Numerix.Statistics':variance(Temperatures),
        cov => 'Elixir.Numerix.Statistics':covariance(Pressures, Temperatures)},
    % {ok, {Id, _, _, _}} = hd(node_util:declare_crdts([meteostats])),
    % {ok, {NewId, NewT, NewM, NewV}} = lasp:update(Id, {add, {node(), Result}}, self()),
    {ok, {Id, _, _, _}} = hd(node_util:declare_crdts([node()])),
    % {ok, {ExecId, _, _, _}} = hd(node_util:declare_crdts([executors])),
    % {ok, {ChunksId, _, _, _}} = lasp:declare({"<<chunks>>", state_gcounter}, state_gcounter),
    {ok, {ExecId, _, _, _}} = lasp:declare({"<<executors>>", state_gset}, state_gset),
    % {ok, {ExecId, _, _, _}} = lasp:declare({"<<executors>>", state_gset}, state_gset).
    {ok, {NewId, NewT, NewM, NewV}} = lasp:update(Id, {add, Result}, self()),

    % {ok, {NewId, NewT, NewM, NewV}} = lasp:update({<<"test">>, state_gset}, {add, "hello"},
     self()),
    % lasp:update({<<"executors">>}, {add, NewId}, self()),
    % {ok, {_, _, _, _}} = lasp:update(ExecId, {add, NewId}, self()),
    lasp:update(ExecId, {add, NewId}, self()),
    % lasp:update({"<<chunks>>", state_gcounter}, increment, self()),
    % node_app:add_task_meteo().
    % {ok, Set} = lasp:query({<<"node@GrispAdhoc">>, state_gset}).
    % sets:to_list(Set).
    spawn(fun() ->
        lasp:read(ExecId, {cardinality, Trigger}),
        % ExecId = node_util:atom_to_lasp_identifier(executors,state_gset),
        % NOTE : lasp:read() WILL block current process regardless
        % if the minimim value has been reached, must ALWAYS be spawned in subprocess
        {ok, Set} = lasp:query(ExecId),
        % {ok, Set} = lasp:query({"<<executors>>", state_gset}),
        L = sets:to_list(Set),
        % io:format("Values = ~p ~n", L),
        [ io:format("Set = ~p ~n", [X]) || X <- L],
        [H|T] = L,
        UnionFold = fun
            (SetName, AccIn) ->
                {UID, Num} = AccIn,
                % Right = node_util:atom_to_lasp_identifier(SetName,state_orset),
                UNum = list_to_bitstring("union" ++ integer_to_list(Num)),
                {ok, {UnionId, _, _, _}} = lasp:declare({UNum, state_orset}, state_orset),
                lasp:union(UID, SetName, UnionId),
                {UnionId, Num + 1}
        end,
        lists:foldl(UnionFold, {H, 1}, T)
    end),

    {ok, {NewId, NewT, NewM, NewV}}.

%% Can provide current union of measurements
%% even when sensing task is still measuring
statistics_union(Id) ->
    {ok, Set} = lasp:query(Id),
    L = sets:to_list(Set),
    [H|T] = L,
    MergeFun = fun(K, V1, V2) ->
        lists:flatten([V1,V2])
    end,

    MapsMergeFun = fun
        (M, AccIn) ->
            node_util:maps_merge(MergeFun, AccIn, M)
    end,

    UMap = lists:foldl(MapsMergeFun, H, T),
    TempMean = 'Elixir.Numerix.Statistics':mean(maps:get(tmean, UMap)),
    PressMean = 'Elixir.Numerix.Statistics':mean(maps:get(pmean, UMap)),
    UMap2 = maps:put(tmean_union, TempMean, UMap),
    UMap3 = maps:put(pmean_union, PressMean, UMap2),
    Tid = ets:new(stats, [public, ordered_set, named_table, {heir, whereis(node), []}]),
```

```
    % Tid = ets:whereis(stats),
    % io:format("Map size = ~p ~n", [maps:size(UMap3)]),
    ets:insert(Tid, {UMap3}).
    % UMap3.
    % spawn(fun() -> lasp:read(ExecId, {size, 3}), io:format("3 reached") end).
% L = lists:map(MapFun, lists:seq(1, SampleCount)),
% MapFun = fun
%     (Elem) when is_integer(Elem) ->
%         timer:sleep(SampleInterval),
%         T = node_stream_worker:maybe_get_time(),
%         Mes = gen_server:call(Pid, {read, alt, [press_out, temp_out], #{}}),
%         {T, Mes}
% end,
%#{press := Press, temp := Temp, time := Time}
% L = lists:map(MapFun, lists:seq(1, SampleCount)),

% Pressures = maps:get(press, M),
% Temperatures = maps:get(temp, M),
% Epochs = maps:get(time, M),
% [ X || X <- maps:values(M)],
    % ReceiverFun = fun (State#meteo{press = PList}) ->
    %     receive
    %
    %     after
    %         SampleInterval ->
    %             P = gen_server:call(Pid, {read, alt, [press_out], #{}}),
    %             NewState = #meteo{press = PList ++ P, temp = []},
    %             self() ! NewState
    %     end.
    %
    % ok.
        % ReceiverFun = fun () ->
        %     receive
        %         Data when is_list(Data) ->
        %             disconnect(),
        %             idle();
        %         {connect, B} ->
        %             B ! {busy, self()},
        %             wait_for_onhook()
        %     end,
        %
        %
        % PFun = fun
        %     (SampleInterval) ->
        %         [PressOut] = erlang:send_after(SampleInterval, Pid, {read, alt, [press_out], [
    ]})
        % end
        % wait_for_onhook() ->
        %     receive
        %         Data when is_list(Data) ->
        %             disconnect(),
        %             idle();
        %         {connect, B} ->
        %             B ! {busy, self()},
        %             wait_for_onhook()
        %     end.



temp_sensor({Counter, Temps}, PeriodicTime) ->

  WaitFun = fun(State) ->
      receive
      after (PeriodicTime) ->
        logger:log(info, "State is ~p and periodicTime is ~p ===~n", [State, PeriodicTime]),
        temp_sensor(State, PeriodicTime)
      end
  end,

  Sum = fun F(List) ->
          SumFun = fun ([H|T]) -> H + F(T);
              ([]) -> 0
          end,
          SumFun(List)
        end,
```

```
  Average = fun(List) -> Sum(List)/length(List) end,

  SensorFun = fun() ->
    logger:log(info, "=== Counter is at ~p ===~n", [Counter]),
    % logger:log(info, "=== Temp list : ~p ===~n",[Temps]),
    logger:log(info, "=== Temp list : ~p ===~n",[Temps]),
    case Counter of
      5 ->
        logger:log(info, "=== Timer has ended, aggregating data and updating CRDT... === ~n"),
        AverageTemp = Average(Temps),
        logger:log(info, "=== Average temp in past hour is ~p ===~n", [AverageTemp]),
        {ok, TempsCRDT} = lasp:query({<<"temp">>, state_orset}),
        TempsList = sets:to_list(TempsCRDT),
        % logger:log(info, "=== Temps CRDT : ~p ===~n", [TempsList]),
        logger:log(info, "=== Temps CRDT : ~p ===~n", [TempsList]),
        OldCrdtData = [{Node, OldAvg, HourCounter, HourAvg, HourData} || {Node, OldAvg,
    HourCounter, HourAvg, HourData} <- TempsList, Node =:= node()],
        % logger:log(info, "=== Old CRDT data is ~p ===~n",[OldCrdtData]),
        logger:log(info, "=== Old CRDT data is ~p ===~n",[OldCrdtData]),
        case length(OldCrdtData) of
          0 ->
            lasp:update({<<"temp">>,state_orset},{add,{node(), AverageTemp, 1, [AverageTemp], [
    AverageTemp]}}, self());
          1 ->
            {Node, OldAvg, HourCounter, HourAvg, HourData} = hd(OldCrdtData),
            NewAverageTemp = ((OldAvg * HourCounter)/(HourCounter+1))+(AverageTemp*(1/(
    HourCounter+1))),
            logger:log(info, "=== New average temp : ~p ===~n",[NewAverageTemp]),
            lasp:update({<<"temp">>, state_orset}, {rmv, {Node, OldAvg, HourCounter, HourAvg,
    HourData}}, self()),
            lasp:update({<<"temp">>,state_orset},{add,{node(), NewAverageTemp, HourCounter+1,
    HourAvg ++ [NewAverageTemp], HourData ++ [AverageTemp]}}, self())
        end,
        {0, []};
      _ ->
        {AnswerTemp,Temp} = node_sensor_server_worker:read(temp),
        TempList = case AnswerTemp of
          read -> lists:append(Temps,[Temp]);
          sensor_not_created -> exit(sensor_not_created)
        end,
        {Counter+1, TempList}
    end
  end,
  WaitFun(SensorFun()).


sonar_sensor(Mode, NodeTarget) ->
  SonarSensor = fun A() ->
    receive
      true ->
        {sonar_listener, NodeTarget} ! trigger,
        % {sonar_listener, node@my_grisp_board_2} ! {fuck}.
        spawn(fun () ->
          case Mode of
            in ->
              lasp:update({"<<enters>>", state_gcounter}, increment, self());
            out ->
              lasp:update({"<<exits>>", state_gcounter}, increment, self())
          end
        end),
        grisp_led:color(1,blue),
        grisp_led:color(2,blue),
      % timer:sleep(500),
      % grisp_led:color(1,green),
      % grisp_led:color(2,green),
        A()
    end
  end,


  SonarListener = fun B() ->
    receive
      Msg ->
      % logger:log(info, "=== received ~p ===~n", [Mode]),
        grisp_led:color(1,red),
        grisp_led:color(2,red),
```

```
        PidSonar = whereis(pmod_maxsonar),
        erlang:suspend_process(PidSonar),
        logger:log(notice, "suspending_process~n"),
        timer:sleep(750),
        erlang:resume_process(PidSonar),
        B()
    end
  end,

  PidSensor = spawn(SonarSensor),
  register(sonar_sensor, PidSensor),
  PidListener = spawn(SonarListener),
  register(sonar_listener, PidListener).



all_sensor_data(Nav, Als) ->
    [Press, Temp] = gen_server:call(Nav, {read, alt, [press_out, temp_out], #{}}),
    Mag = gen_server:call(Nav, {read, mag, [out_x_m, out_y_m, out_z_m], #{}}),
    Gyro = gen_server:call(Nav, {read, acc, [out_x_g, out_y_g, out_z_g], #{}}),
    Raw = gen_server:call(Als, raw),
    {_,{H,Mi,_}} = node_util:maybe_get_time(),
    {ok, {_, _, _, _}} = lasp:update(node_util:atom_to_lasp_identifier(node(),state_gset), {add,
      [H*60 + Mi,Raw,Press,Temp,Mag,Gyro]}, self()),
    timer:sleep(1000),
    all_sensor_data(Nav,Als).
```

B.5 Node Benchmark Server

```
-module(node_benchmark_server).

-behaviour(gen_server).

-include_lib("node.hrl").

%% API
-export([start_link/0, terminate/0]).
-export([benchmark_meteo_task/1]).

%% Gen Server Callbacks
-export([code_change/3, handle_call/3, handle_cast/2,
    handle_info/2, init/1, terminate/2]).



%% ====================================================================
%% API functions
%% ====================================================================

start_link() ->
    gen_server:start_link({local, ?MODULE}, ?MODULE, [],
        []).

benchmark_meteo_task(LoopCount) -> gen_server:call(?MODULE, {benchmark_meteo_task, LoopCount}).

terminate() -> gen_server:call(?MODULE, {terminate}).

%% ====================================================================
%% Gen Server callbacks
%% ====================================================================



init([]) ->
  logger:log(notice, "Starting a node benchmark server"),
  erlang:send_after(90000, self(), {benchmark_meteo_task, 10}),
  {ok, {}}.
```

```
handle_call(stop, _From, State) ->
    {stop, normal, ok, State}.


handle_info({benchmark_meteo_task, LoopCount}, State) ->
  EvaluationMode = node_config:get(evaluation_mode, grisplasp),
  logger:log(notice, "=== Starting meteo task benchmark in mode ~p ===~n", [EvaluationMode]),
  SampleCount = 5,
  SampleInterval = 3000,
  node_generic_tasks_server:add_task({tasknav, all, fun () ->
    LoopSeq = lists:seq(1, LoopCount),
    case EvaluationMode of
      grisplasp ->
        % NodesWithoutMe = lists:delete(node(),NodeList),
        NodesWithoutMe = lists:delete(node(),?BOARDS(?ALL)),
        % logger:log(notice, "Node list ~p", [NodesWithoutMe]),
        lists:foreach(fun (Node) ->
          logger:log(notice, "Spawning listener for  ~p", [node_util:atom_to_lasp_identifier(
     Node, state_gset)]),
            spawn(fun() ->
              lists:foreach(fun(Cardinality) ->
                lasp:read(node_util:atom_to_lasp_identifier(Node, state_gset), {cardinality,
     Cardinality}),
                % logger:log(notice, "CRDT with cardinality ~p from node ~p converged on our node!
      Sending Acknowledgement", [Cardinality, Node]),
                {convergence_acknowledgement, Node} ! {ack, node(), Cardinality}
              end, LoopSeq)
            end)
        end, NodesWithoutMe),
        node_generic_tasks_functions_benchmark:meteorological_statistics_grisplasp(LoopCount,
     SampleCount, SampleInterval)
      end
    end }),
  node_generic_tasks_worker:start_task(tasknav),
  {noreply, State};

handle_info(Msg, State) ->
    logger:log(info, "=== Unknown message: ~p~n", [Msg]),
    {noreply, State}.

handle_cast(_Msg, State) -> {noreply, State}.

terminate(Reason, _S) ->
    logger:log(info, "=== Terminating node benchmark server (reason: ~p) ===~n",[Reason]),
    ok.

code_change(_OldVsn, S, _Extra) -> {ok, S}.

%%====================================================================
%% Internal functions
%%====================================================================
```

B.6 Node Generic Task Functions Benchmark

```
-module(node_generic_tasks_functions_benchmark).
-include_lib("node.hrl").
-compile(export_all).


% ==> Aggregation, computation and replication with Lasp on Edge
meteorological_statistics_grisplasp(LoopCount, SampleCount, SampleInterval) ->


  % logger:log(notice, "Starting Meteo statistics task benchmark with Lasp on GRiSP ~n"),

  Self = self(),
  SampleSeq = lists:seq(1, SampleCount),
  MeteoMap = #{press => [], temp => [], time => []},
  MeteorologicalStatisticsFun = fun MSF (LoopCountRemaining, AccComputations) ->
```

```
logger:log(notice, "Meteo Function remaining iterations: ~p", [LoopCountRemaining]),

% Must check if module is available
{pmod_nav, Pid, _Ref} = node_util:get_nav(),

% (Elem, AccIn) when is_integer(Elem) andalso is_map(AccIn) ->
FoldFun =
    fun (Elem, AccIn) ->
        timer:sleep(SampleInterval),
        % T = node_util:maybe_get_time(),
        % T = calendar:local_time(), % Replace by erlang:monotonic_time(second) to reduce
 size
        [Pr, Tmp] = gen_server:call(Pid, {read, alt, [press_out, temp_out], #{}}),
        % logger:log(notice, "Getting data from nav sensor pr: ~p tmp: ~p", [Pr, Tmp]),
        #{press => [Pr] ++ maps:get(press, AccIn),
        temp => [Tmp] ++ maps:get(temp, AccIn),
        time => [erlang:monotonic_time(second)] ++ maps:get(time, AccIn)}
end,

M = lists:foldl(FoldFun, MeteoMap, SampleSeq),
% logger:log(notice, "Done Sampling data"),

T1Computation = erlang:monotonic_time(millisecond),

[Pressures, Temperatures, Epochs] = maps:values(M),
Result = #{measures => lists:zip3(Epochs, Pressures, Temperatures),
    pmean => 'Elixir.Numerix.Statistics':mean(Pressures),
    pvar => 'Elixir.Numerix.Statistics':variance(Pressures),
    tmean => 'Elixir.Numerix.Statistics':mean(Temperatures),
    tvar => 'Elixir.Numerix.Statistics':variance(Temperatures),
    cov => 'Elixir.Numerix.Statistics':covariance(Pressures, Temperatures)},

T2Computation = erlang:monotonic_time(millisecond),
lasp:update(node_util:atom_to_lasp_identifier(node(), state_gset), {add, Result}, self()),

Cardinality = LoopCount-LoopCountRemaining+1,
ComputationTime = T2Computation - T1Computation,
NewAcc = maps:put(Cardinality, {T2Computation, ComputationTime}, AccComputations),

if LoopCountRemaining > 1 ->
  MSF(LoopCountRemaining-1, NewAcc);
true ->
  timer:sleep(30000), % Give time to the CAF process to finish receiving acks
  convergence_acknowledgement ! {done, NewAcc}
end
end,

ConvergenceAcknowledgementFun = fun CA(Acks) ->
  receive
    {ack, From, Cardinality} ->
      TConverged = erlang:monotonic_time(millisecond),
      CA([{From, TConverged, Cardinality} | Acks]);
    {done, Computations} -> % Called by the meteo process once it has terminated
      Self ! {done, Computations, Acks}
  end
end,

logger:log(notice, "Spawning Acknowledgement receiver process"),
% https://stackoverflow.com/questions/571339/erlang-spawning-processes-and-passing-arguments
PidCAF = spawn(fun () -> ConvergenceAcknowledgementFun([]) end),
register(convergence_acknowledgement, PidCAF),
timer:sleep(20000),
PidMSF = spawn(fun () -> MeteorologicalStatisticsFun(LoopCount, #{}) end),
register(meteo_stats, PidMSF),
register(meteo_task, self()),

receive
    {done, Computations, Acks} ->
      grisp_led:color(1, blue),
      grisp_led:color(2, green),
      logger:log(notice, "Meteo task is done, received acks and computations. Calculating
 computation time + convergence time..."),
      % logger:log(notice, "Computations: ~p - Acks: ~p", [Computations, Acks]),

      NodesConvergenceTime = lists:foldl(
```

```
        fun(Ack, Acc) ->
          {From, TConverged, Cardinality} = Ack,
          {T2Computation, _} = maps:get(Cardinality, Computations),
          ConvergenceTime = TConverged - T2Computation,
          case maps:find(From, Acc) of
            {ok, NodeMap} ->
              NewNodeMap = maps:put(Cardinality, ConvergenceTime, NodeMap),
              maps:update(From, NewNodeMap, Acc);
            error ->
              maps:put(From, #{Cardinality => ConvergenceTime}, Acc)
          end
        end, #{}, Acks),
        % logger:log(notice, "Nodes convergence times is ~p", [NodesConvergenceTime]),
        NodesConvergenceTimeCalculations = maps:map(fun(Node, ConvergenceTimes) ->
          Convergences = maps:values(ConvergenceTimes),
          SD = 'Elixir.Numerix.Statistics':std_dev(Convergences),
          Var = 'Elixir.Numerix.Statistics':variance(Convergences),
          Avg ='Elixir.Numerix.Statistics':mean(Convergences),
          {{average, Avg}, {standard_deviation, SD},{variance, Var}}
        end, NodesConvergenceTime),

        SumConvergenceTimes = lists:foldl(fun({{_, Mean},{_,_},{_,_}}, Acc) ->
          Acc + Mean
        end, 0, maps:values(NodesConvergenceTimeCalculations)),
        MeanConvergenceTime = SumConvergenceTimes / length(maps:keys(NodesConvergenceTime)),

        Variances = lists:foldl(fun({{_, _},{_,_},{_, Var}}, Acc) ->
          Acc ++ [Var]
        end, [], maps:values(NodesConvergenceTimeCalculations)),

        SampleSize1 = length(maps:keys(maps:get(lists:nth(1, maps:keys(NodesConvergenceTime)),
    NodesConvergenceTime))),
        SampleSize2 = length(maps:keys(maps:get(lists:nth(2, maps:keys(NodesConvergenceTime)),
    NodesConvergenceTime))),
        PooledVarianceConvergence = (((SampleSize1-1)*lists:nth(1,Variances)) + ((SampleSize2
  -1)*lists:nth(2,Variances)))/((SampleSize1+SampleSize2)/2),
        PooledStandardDeviationConvergence = math:sqrt(PooledVarianceConvergence),
        ComputationFiltered = maps:map(fun(Cardinality, {T2Computation, ComputationTime}) ->
          ComputationTime
        end, Computations),
        MeanComputationTime = 'Elixir.Numerix.Statistics':mean(maps:values(ComputationFiltered
  )),
        StandardDeviationComputationTime = 'Elixir.Numerix.Statistics':std_dev(maps:values(
  ComputationFiltered)),
        VarianceComputationTime = 'Elixir.Numerix.Statistics':variance(maps:values(
  ComputationFiltered)),
        % logger:log(info, "nodes total converges averages = ~p ~n", [
  NodesConvergenceTimeAverage]),

     %%%%%%%%%%%%%%%%%%%%%%%%
     % Save results to ETS %
     %%%%%%%%%%%%%%%%%%%%%%%%
     % TableName = list_to_atom(unicode:characters_to_list([atom_to_list(node()), "_results"]
  , latin1)),
     % Tmp = ets:new(TableName, [ordered_set, named_table, public]),
     % ResultSave = case ets:insert_new(Tmp, {node(), {ComputationFiltered}, {
  MeanComputationTime}, {VarianceComputationTime}, {NodesConvergenceTime}, {
  NodesConvergenceTimeCalculations},{MeanConvergenceTime},{PooledVarianceConvergence}}) of
     %      true ->
     %        FileName = node(),
     %        case ets:tab2file(Tmp, FileName, [{sync, true}]) of
     %          ok ->
     %            logger:log(notice, "Saved results to SD card."),
     %            true = ets:delete(Tmp),
     %            ok;
     %          {error, Reason} ->
     %            logger:log(error, "Could not save results to SD card"),
     %            {error, Reason}
     %        end;
     %      false ->
     %        logger:log(error, "Could not insert results in ets"),
     %        {error, insert}
     % end,
     %%%%%%%%%%%%%%%%%%%%%%%%%
     % Save results to file %
     %%%%%%%%%%%%%%%%%%%%%%%%%
```

```
        file:write_file(atom_to_list(node()), io_lib:fwrite("Computations: ~p ~nMean Computation
     Time: ~pms~nStandard Deviation Computation Time: ~p~nVariance Computation Time: ~p~nNodes
    Convergence Time: ~p ~nNodes Convergence Calculations: ~p ~nMean Convergence Time: ~pms~
    nPooled Standard Deviation Convergence Time: ~p~nPooled Variance Convergence Time: ~p~n", [
    ComputationFiltered, MeanComputationTime, StandardDeviationComputationTime,
    VarianceComputationTime, NodesConvergenceTime, NodesConvergenceTimeCalculations,
    MeanConvergenceTime, PooledStandardDeviationConvergence, PooledVarianceConvergence])),
        logger:log(info,"Wrote results to file. ~n"),
        grisp_led:color(2, red),
        timer:sleep(2000),
        exit(PidCAF, kill),
        exit(PidMSF, kill)
    end.
```

B.7 Node Ping Worker

```
-module(node_ping_worker).

-behaviour(gen_server).

%% API
-export([full_ping/0, ping/1, start_link/0,
    terminate/0]).

%% Gen Server Callbacks
-export([code_change/3, handle_call/3, handle_cast/2,
    handle_info/2, init/1, terminate/2]).

-include("node.hrl").

%% ====================================================================
%% API functions
%% ====================================================================

start_link() ->
    gen_server:start_link({local, node_ping_worker},
        ?MODULE, {}, []).

ping(N) ->
    gen_server:call(node_ping_worker, {ping, N}, infinity).

full_ping() ->
    gen_server:call(node_ping_worker, {full_ping},
        infinity).

terminate() ->
    gen_server:cast(node_ping_worker, {terminate}).

%% ====================================================================
%% Gen Server callbacks
%% ====================================================================

init({}) ->
    logger:log(info, "Initializing Node Pinger~n"),
    process_flag(trap_exit,
     true), %% Ensure Gen Server gets notified when his supervisor dies
    erlang:send_after(30000, self(),
          {full_ping}), %% Start full pinger after 5 seconds
    % self() ! {full_ping},
    {ok, []}.

handle_call({terminate}, _From, CurrentList) ->
    logger:log(info, "=== Ping server terminates with Current "
        "list of Node pinged correctly (~p) ===~n",
        [CurrentList]),
    {reply, {terminate}, CurrentList};
handle_call(_Message, _From, CurrentList) ->
    {reply, {ok, CurrentList}, CurrentList}.

handle_info({full_ping}, CurrentList) ->
```

```
    logger:log(info, "=== Starting a full ping ===~n"),
    T1 = os:timestamp(),
    PingedNodes = ping(),
    T2 = os:timestamp(),
    Time = timer:now_diff(T2, T1),
    logger:log(info, "=== Time to do a full ping ~ps ===~n",
        [Time / 1000000]),
    logger:log(notice, "=== Nodes that answered back ~p ===~n",
        [PingedNodes]),
    {noreply, PingedNodes, 60000};
handle_info(timeout, CurrentList) ->
    logger:log(info, "=== Timeout of full ping, restarting "
        "after 90s ===~n"),
    T1 = os:timestamp(),
    PingedNodes = ping(),
    T2 = os:timestamp(),
    Time = timer:now_diff(T2, T1),
    logger:log(info, "=== Time to do a full ping ~ps ===~n",
        [Time / 1000000]),
    logger:log(notice, "=== Nodes that answered back ~p ===~n",
        [PingedNodes]),
    {noreply, PingedNodes, 180000};
handle_info(Msg, CurrentList) ->
    logger:log(info, "=== Unknown message: ~p~n", [Msg]),
    {noreply, CurrentList}.

handle_cast(_Message, CurrentList) ->
    {noreply, CurrentList}.

terminate(_Reason, _CurrentList) -> ok.

code_change(_OldVersion, CurrentList, _Extra) ->
    {ok, CurrentList}.

%%====================================================================
%% Internal functions
%%====================================================================

%% @doc Macros are defined in node.hrl
%% so calling ?BOARDS() with <em>?DAN</em> argument
%% will return a list of the hostnames of the boards Dan usually runs
%% deafult macros are : ?ALL,?ALEX,?DAN,?IGOR
%% but ?BOARDS(X) will return a list of hostnames
%% with any other supplied sequence
%% @end
ping() ->
  Remotes = maps:fold(fun
    (K, V, AccIn) when is_list(V) ->
      AccIn ++ V
  end, [], node_config:get(remote_hosts, #{})),
    % List = (?BOARDS((?IGOR))) ++ ['nodews@Laymer-3'],

    % List = Remotes,

    List = (?BOARDS((?ALL))),
    % List =  (?BOARDS(?ALL)) ++ Remotes,
    ListWithoutSelf = lists:delete(node(), List),
    lists:foldl(fun (Node, Acc) ->
      case net_adm:ping(Node) of
        pong ->
          IsARemote = lists:member(Node, Remotes),
          if IsARemote == true ->
            logger:log(info, "=== Node ~p is an aws server", [Node]),
            Acc ++ [Node];
          true ->
            logger:log(info, "=== Attempting to join the node ~p with lasp", [Node]),
            lasp_peer_service:join(Node),
            Acc ++ [Node]
          end;
        pang ->
          logger:log(info, "=== Node ~p is unreachable", [Node]),
          Acc
      end
    end, [], ListWithoutSelf).
```

Bibliography

[1] *About Moore's Law, Metcalfe's Law and the IoT*. en. https://www.nokia.com/en_int/blog/moores-law-metcalfes-law-iot.

[2] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. "Delta State Replicated Data Types". In: *Journal of Parallel and Distributed Computing* 111 (Jan. 2018), pp. 162–173. ISSN: 07437315. DOI: 10.1016/j.jpdc.2017.08.003. arXiv: 1603.01529.

[3] *Amazon Compute Service Level Agreement*. en-US. https://aws.amazon.com/compute/sla/.

[4] Kishore Angrishi. "Turning Internet of Things(IoT) into Internet of Vulnerabilities (IoV) : IoT Botnets". In: *arXiv:1702.03681 [cs]* (Feb. 2017). arXiv: 1702.03681 [cs].

[5] *Announcing Amazon Elastic Compute Cloud (Amazon EC2) - Beta*. en-US. https://aws.amazon.com/about-aws/whats-new/2006/08/24/announcing-amazon-elastic-compute-cloud-amazon-ec2—beta/.

[6] Flavio Bonomi et al. "Fog Computing and Its Role in the Internet of Things". en. In: ACM Press, 2012, p. 13. ISBN: 978-1-4503-1519-7. DOI: 10.1145/2342509.2342513.

[7] Flavio Bonomi et al. "Fog Computing and Its Role in the Internet of Things". In: *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*. MCC '12. New York, NY, USA: ACM, 2012, pp. 13–16. ISBN: 978-1-4503-1519-7. DOI: 10.1145/2342509.2342513.

[8] Russell Brown and Torben Hoffmann. "BigSets: Scaling CRDTs to Large Sizes in Riak". en. In: (), p. 3.

[9] Russell Brown, Zeeshan Lakhani, and Paul Place. "Big(Ger) Sets: Decomposed Delta CRDT Sets in Riak". In: *arXiv:1605.06424 [cs]* (2016), pp. 1–5. DOI: 10.1145/2911151.2911156. arXiv: 1605.06424 [cs].

[10] *Building Plugins*. https://www.rebar3.org/docs/plugins.

[11] *Cloud Growth Rate Increased Again in Q1; Amazon Maintains Market Share Dominance | Synergy Research Group*. https://www.srgresearch.com/articles/cloud-growth-rate-increased-again-q1-amazon-maintains-market-share-dominance.

[12] "Cyberattack". en. In: *Wikipedia* (Aug. 2018). Page Version ID: 854697194.

[13] *Erlang – Erts_alloc*. http://erlang.org/doc/man/erts_alloc.html#M_as.

[14] *Erlang – Gen_server Behaviour*. http://erlang.org/doc/design_principles/gen_server_concepts.html.

[15] *Erlang – Supervisor Behaviour*. http://erlang.org/doc/design_principles/sup_princ.html.

[16] *Erlang 19.0 Garbage Collector | Erlang Solution Blog*. https://www.erlang-solutions.com/blog/erlang-19-0-garbage-collector.html.

[17] Dave Evans. "How the Next Evolution of the Internet Is Changing Everything". en. In: (2011), p. 11.

[18] *Evolution of IT Infrastructure | LinkedIn*. https://www.linkedin.com/pulse/evolution-infrastructure-paul-m-veillard/.

[19] *Garbage Collection and Memory Management in Erlang*. https://stackoverflow.com/questions/1022 collection-and-memory-management-in-erlang.

[20] *GRiSP — Home — Bare Metal Hardware for the Internet of Things, Specially Designed for Erlang*. https://grisp.org/.

[21] *https://www.lightkone.eu/*. en-US.

[22] *Introducing Google App Engine + Our New Blog*. Apr. 2008.

[23] J. Jin et al. "An Information Framework for Creating a Smart City Through Internet of Things". In: *IEEE Internet of Things Journal* 1.2 (Apr. 2014), pp. 112–121. ISSN: 2327-4662. DOI: 10.1109/JIOT.2013.2296516.

[24] Avinash Lakshman and Prashant Malik. "Cassandra: A Decentralized Structured Storage System". en. In: *ACM SIGOPS Operating Systems Review* 44.2 (Apr. 2010), p. 35. ISSN: 01635980. DOI: 10.1145/1773912.1773922.

[25] Joao Leitao. "Towards Enabling Novel Edge-Enabled Applications$_*$". en. In: (), p. 9.

[26] Joao Leitao, Jose Pereira, and Luis Rodrigues. "Epidemic Broadcast Trees". In: *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems*. SRDS '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 301–310. ISBN: 978-0-7695-2995-0.

[27] Joao Leitao, Jose Pereira, and Luis Rodrigues. "HyParView: A Membership Protocol for Reliable Gossip-Based Broadcast". In: *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. DSN '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 419–429. ISBN: 978-0-7695-2855-7. DOI: 10.1109/DSN.2007.56.

[28] *Linker Script What Is a "Linkcmds" File?* https://docs.rtems.org/releases/rtemsdocs-4.6.2/share/rtems/html/bsp_howto/bsp_howto00018.html.

[29] Wyatt Lloyd et al. "Stronger Semantics for Low-Latency Geo-Replicated Storage". en. In: (), p. 16.

[30] Erlang Solutions Ltd. *Memory Management Battle Stories*.

[31] Raka Mahesa. *How Cloud, Fog, and Mist Computing Can Work Together*. en-US. Mar. 2018.

[32] Raka Mahesa. *How Cloud, Fog, and Mist Computing Can Work Together*. en-US. Mar. 2018.

[33] Christopher Meiklejohn and Heather Miller. "Partisan: Enabling Cloud-Scale Erlang Applications". In: *arXiv:1802.02652 [cs]* (Feb. 2018). arXiv: 1802.02652 [cs].

[34] Christopher Meiklejohn and Peter Van Roy. "Lasp: A Language for Distributed, Coordination-Free Programming". In: *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*. PPDP '15. New York, NY, USA: ACM, 2015, pp. 184–195. ISBN: 978-1-4503-3516-4. DOI: 10.1145/2790449.2790525.

[35] Christopher S. Meiklejohn et al. "Practical Evaluation of the Lasp Programming Model at Large Scale - An Experience Report". In: *arXiv:1708.06423 [cs]* (2017), pp. 109–114. DOI: 10.1145/3131851.3131862. arXiv: 1708.06423 [cs].

[36] *Planetary Scale Applications · The Lasp Programming System*. en. https://lasp-lang.readme.io/docs/what-is-the-lasp-suite.

[37] *Pmod Modules - Page 1 - Digilent*. https://store.digilentinc.com/pmod-modules/.

[38] *Pmod Standard [Reference.Digilentinc]*. https://reference.digilentinc.com/reference/pmod/specificati

[39] Jurgo S. Preden et al. "The Benefits of Self-Awareness and Attention in Fog and Mist Computing". en. In: *Computer* 48.7 (July 2015), pp. 37–45. ISSN: 0018-9162. DOI: 10.1109/MC.2015.207.

[40] *Rebar3_grisp: Rebar Plug-in for GRiSP*. June 2018.

[41] *Riak KV*. en-US.

[42] *Rtems-Libbsd: RTEMS BSD Porting Project*. May 2018.

[43] Marc Shapiro et al. "Conflict-Free Replicated Data Types". en. In: *Stabilization, Safety, and Security of Distributed Systems*. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Oct. 2011, pp. 386–400. ISBN: 978-3-642-24549-7 978-3-642-24550-3. DOI: 10.1007/978-3-642-24550-3_29.

[44] Ali Shoker et al. "LightKone: Deliverable 5.2 : Report on Generic Edge Computing". en. In: (), p. 12.

[45] Ali Shoker et al. "LightKone: Towards General Purpose Computations on the Edge". en. In: (), p. 12.

[46] *Stuff Goes Bad: Erlang in Anger*. https://www.erlang-in-anger.com/.

[47] * All products require an annual contract Prices do not include sales tax. *IoT: Number of Connected Devices Worldwide 2012-2025*. en. https://www.statista.com/statistics/471264/io number-of-connected-devices-worldwide/.

[48] *The Erlang Runtime System*. https://happi.github.io/theBeamBook/#_preface.

[49] *Troubleshooting Down the Logplex Rabbit Hole*. en. https://blog.heroku.com/logplex-down-the-rabbit-hole.

[50] Robert Virding. *[Erlang-Questions] Problem with Beam Process Not Freeing Memory*. Tue Sep 17 16:53:40 CEST 2013.

[51] P. Vlacheas et al. "Enabling Smart Cities through a Cognitive Management Framework for the Internet of Things". In: *IEEE Communications Magazine* 51.6 (June 2013), pp. 102–111. ISSN: 0163-6804. DOI: 10.1109/MCOM.2013.6525602.

[52] *Voice over Wireless LAN 4.1 Design Guide - Voice over WLAN Radio Frequency Design [Design Zone for Mobility]*. en. https://www.cisco.com/c/en/us/td/docs/solutions/Enterprise/ book/vowlan_ch3.html.

[53] *What Is Cloud Computing? - Amazon Web Services*. en-US. https://aws.amazon.com/what-is-cloud-computing/.

[54] JESPER WILHELMSSON. "Efficient Memory Management for Message-Passing Concurrency". en. In: (), p. 114.

[55] Andrea Zanella et al. "Internet of Things for Smart Cities". en. In: *IEEE Internet of Things Journal* 1.1 (Feb. 2014), pp. 22–32. ISSN: 2327-4662. DOI: 10.1109/JIOT.2014.2306328.