DIAL
Digital access to libraries

"Doing large-scale computations on an Internet of Things network"

Banken, Julien ; Xanthos, Nicolas

ABSTRACT

As you may notice, the Internet-of-Things devices are everywhere: in our factories, in our cities and in our houses. The challenge of tomorrow will be to store and process data generated by these devices. In this master thesis, we tried to tackle this challenge. We implemented a MapReduce algorithm designed to run on the extreme edge of the network. With the edge computing, the IoT devices can exchange, store and process data without depending on an external infrastructure. As part of this master thesis, we developed a resilient MapReduce algorithm that work in a peer-to-peer network composed of IoT devices.

CITE THIS VERSION

Banken, Julien ; Xanthos, Nicolas. *Doing large-scale computations on an Internet of Things network.* Ecole polytechnique de Louvain, Université catholique de Louvain, 2020. Prom. : Van Roy, Peter. http://hdl.handle.net/2078.1/thesis:26489

Le dépôt institutionnel DIAL est destiné au dépôt et à la diffusion de documents scientifiques émanents des membres de l'UCLouvain. Toute utilisation de ce document à des fin lucratives ou commerciales est strictement interdite. L'utilisateur s'engage à respecter les droits d'auteur lié à ce document, principalement le droit à l'intégrité de l'oeuvre et le droit à la paternité. La politique complète de copyright est disponible sur la page Copyright policy

DIAL is an institutional repository for the deposit and dissemination of scientific documents from UCLouvain members. Usage of this document for profit or commercial purposes is stricly prohibited. User agrees to respect copyright about this document, mainly text integrity and source mention. Full content of copyright policy is available at Copyright policy

Available at: http://hdl.handle.net/2078.1/thesis:26489

[Downloaded 2020/08/16 at 11:21:10]

UCLouvain

epl

École polytechnique de Louvain

Doing large-scale computations on an Internet of Things network

Authors: **Julien Banken, Nicolas Xanthos**
Supervisor: **Peter Van Roy**
Readers: **Igor Kopestenski, Etienne Rivière**
Academic year 2019–2020
Master [120] in Computer Science

Acknowledgements

We would like to thank Igor Kopestenski and Professor Peter Van Roy. Igor helps us a lot to get started with the project. He gave us useful advice from his own experience. He also gave us many resources to learn more about Erlang. He answered many of our questions. Professor Peter Van Roy was very enthusiast about the project. He really managed to give meaning to our work and make us want to get involved. He also responds to many of our questions and he put us back on track several times.

Contents

1 Introduction **1**

I System and dependencies **5**

2 GRiSP **6**
2.1 Picture 6
2.2 Technical specifications 6
2.3 RTEMS 8
2.4 Erlang/OTP 8

3 Dependencies **10**
3.1 Partisan 10
3.2 Lasp 13

II Lynkia **16**

4 Algorithms **17**
4.1 MapReduce 17
4.2 Task model 25

5 Implementation **31**
5.1 MapReduce 31
5.2 Task model 35

6 Usage **38**
6.1 Use Lynkia 38
6.2 Configure Lynkia 42
6.3 Extend Lynkia 42

III Evaluation **44**

7 Test scenarios **45**

7.1 MapReduce 45
7.2 Task model 56

8 Limits **62**
8.1 Hardware 62
8.2 Software 63

9 Future work **64**
9.1 Possible improvements 64

10 Conclusion **66**

IV Appendix **70**

A Lynkia **71**
A.1 Getting started 71
A.2 Modules 74
A.3 Variables 75
A.4 API 76
A.5 Examples 77

B Configuration of Lynkia **79**

C Gallery **80**

D FAQ **81**

E Other approaches **82**
E.1 Dataflow 82
E.2 Stream computation 89
E.3 Stochastic approach 92

Acronyms **98**

Glossary **99**

Abstract

As you may notice, the Internet-of-Things devices are everywhere: in our factories, in our cities and in our houses. The challenge of tomorrow will be to store and process data generated by these devices. In this master thesis, we tried to tackle this challenge. We implemented a MapReduce algorithm designed to run on the extreme edge of the network. With the edge computing, the IoT devices can exchange, store and process data without depending on an external infrastructure. As part of this master thesis, we developed a resilient MapReduce algorithm that work in a peer-to-peer network composed of IoT devices.

Chapter 1

Introduction

According to Cisco [4], the number of connected devices has grown exponentially since 2008. To process and store the data produced by these devices, many actors have turned to the cloud. However, the cloud has some drawbacks:

- First of all, the cloud hosting services are not free. Each cloud provider will have its own pricing scheme. Most of the time, the cloud provider will charge you for the resources that you are using. If the devices produce a lot of data, the system will have to use more resources to process them. As a result, the cloud hosting service can be expensive.
- Second of all, the cloud is a black box from the end-user standpoint. When we are uploading data, we do not know what the provider will do with them and we do not know if the provider will respect the GDPR.
- Finally, the IoT devices should have an internet connection to be able to communicate with the datacenter. If the devices have an intermittent and an unstable internet connection, the system may have down times which can be critical in some applications. Sometimes, the devices might not have an internet connection at all. In that case, the system may become unusable.

The cloud has its limits and these limits are too restrictive in the context of Internet-of-Things. The solution would be to decentralize partially or totally the processing and storage of data on the edge. There are two architectures: the fog computing and the edge computing allowing to decentralize the data processing on the edge. They are often confused even though they have different philosophies.

The fog computing paradigm extends the cloud computing paradigm to the edge [1] [2]. The IoT network has a connection to the cloud. Data processing and storage are not completely decentralized on the edge. The cloud still has an important role in data processing. This approach makes it possible to process part

of the data on the edge of the network as close as possible to the data sources. And it can also rely on the computing power of the cloud to process larger amounts of data. The fog can be used to reduce the amount of data sent to the cloud by sending aggregated data, and thus reduce the costs related to the cloud, but without removing them completely.

The edge computing paradigm totally moves the execution of the algorithms and the data storage on the edge [13]. With this decentralized approach, the IoT network does not depend on a cloud infrastructure. In our project we use the edge computing and more specifically the extreme edge computing that means that we execute the algorithms and store the data directly on the sensors boards themselves. The extreme edge is the most cost-effective because since it does not need any additional hardware to work. But extreme edge is also the hardest case because the available resources are very limited and the sensors are not as reliable as a computer.

The algorithms that will run the edge will have to consider the environment in which the IoT network is and the constraints of the system. The system has an open and dynamic membership where the devices can join and leave the network at any time. The devices may crash at any time and cause partition in the network. The devices are limited compared to recent computers, they have less computing power and less storage.

For these reasons, we cannot just reuse the algorithms that works well in a cloud environment. We will have to design new ones. In this project, we will try to implement a new MapReduce algorithm that will consider the constraints and the environment of the connected devices. Our goal is to have a functional implementation that can be deployed in practice on a network of GRiSP boards.

We have chosen to implement the MapReduce because it is one of the best-known algorithms of big data, we would like to provide the proof-of-concept that big data algorithm can run on the extreme edge.

As the IoT devices have less computing power and less storage, it would not be fair to compare the performance of our system with existing ones. To be clear, the system will not be able to store and process terabytes of data. However, the system will be able to process a sufficient amount of data for some real-world application. With the future technological advances, the devices will have more and more storage and more and more computing power. Therefore, the amount of data we that can process will increase over time. Someday, the IoT devices be able to solve big data problems as is it the case today with the cloud.

The rest of this master thesis will be organized as following: In the first part, we will present the system and the dependencies that we will use. In the second part, we will present Lynkia. In a few words, Lynkia is an open-source library that we developed. The library will contain our implementation of the MapReduce as well as a generic task model. In the third part, we will test Lynkia in real world condition. Lynkia will be deployed on a set of GRiSP board. Here, we will assess the resilience of Lynkia by injecting faults.

The main contributions of our master thesis are:

Lynkia: Lynkia is an open-source library that we develop as part of this project. The library will be configurable and modular. It includes a task model and a MapReduce model.

- **Task model**: The task model allows the nodes to distribute their workload seamlessly with their neighbors. The model comes with several load balancing strategies that can be configure.
- **MapReduce model**: The MapReduce allows the nodes to aggregate data stored on the devices.
  - The MapReduce relies on our own task model.
  - The MapReduce tolerates up to $n - 1$ failures.
  - The MapReduce is partition tolerant.
  - The messages exchange can be omitted, delayed, reorder and duplicated

Evaluation: As part of this project, we assessed Lynkia in real conditions by deploying the code on a network of GRiSP boards. In the evaluation, we tested the reliability of the system, compared different load balancing strategies and measure the efficiency of the task model.

- **Reliability tests**: To test the reliability of the MapReduce, we ran a synthetic example on a network of GRiSP boards and we injected faults manually. Here, we test different network topologies namely the full-mesh topology, the butterfly topology and the line topology. In all failure scenarios, the MapReduce works as expected and returns the right result.
- **Load balancing**: We assessed the efficiency of the two load balancing strategies we implemented i.e. the pull and the push strategy. Here, we shown that our pull strategy distributes the load more efficiently than our push strategy.

- **Benchmarks**: We assessed the efficiency of the task model. We showed that our task model gives better results when the provided tasks are computationally intensive.

Part I

System and dependencies

Chapter 2

GRiSP

In this chapter, we will present the technical specifications of the GRiSP boards that we will use to test our system. Unlike the classic IoT devices, these boards can run Erlang programs out-of-the box via RTEMS. In this chapter, we will also introduce RTEMS and give the specificities of Erlang.

2.1 Picture

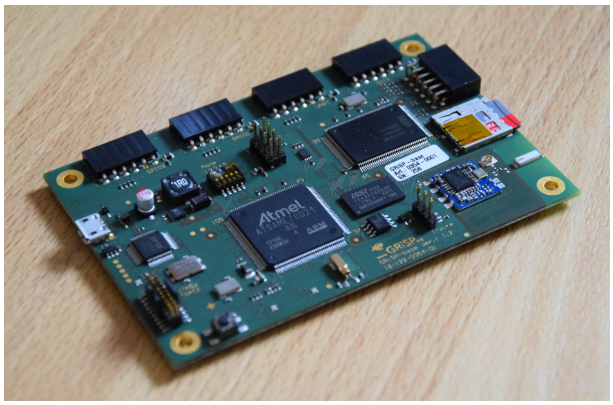


Figure 2.1: GRiSP board

2.2 Technical specifications

CPU: The GRiSP board has an Atmel SAM V71 (SAMV71Q21ES3) processor based on an ARM Cortex M7 running at up to 300 MHz. It has a single- and double-precision hardware Floating Point Unit (FPU).

Internal memory: The GRiSP board has 2048 Kbytes of flash memory which is used by the Bootloader and 384 Kbytes of SRAM which is used as internal cache by the CPU.

External memory: The GRiSP board has 64 Mbytes of SDRAM, a 2K bytes of EEPROM and a built-in MicroSD Socket for standard MicroSD Cards.

Network: The GRiSP board has a Realtek RTL8188ETV single chip integrating 802.11 b/g/n WLAN. Two modes are available: the infrastructure mode is the classical Wi-Fi architecture, each node is connected to a router and all messages pass through the router. The second mode is the ad-hoc mode, nodes are directly connected with other nodes and directly send messages to their neighbors.

User interface: The GRiSP board has two RGB LEDs, 5 DIP switches and a boot/reset button key.

I/O: The GRiSP board support Digilent Pmod interface boards that allow to extend the capabilities of the GRiSP board with sensors and actuators.

- 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)
- One Digilent Pmod Type 4 interface (UART)

It is possible to buy on the internet some modules to extend the capabilities of the GRiSP board. For the GPIO protocol, we can find modules to add a keypad, a push button, an infrared light detector, etc. For the SPI protocol, we can find modules to add an ambient light sensor, a screen, a microphone, an accelerometer, a gyroscope, a barometer, etc. And finally, for the UART protocol, we can find modules to add a USB port, a Bluetooth interface, a GPS receiver, an ultra-sonic range finder, etc.

If you intend to buy one of these modules, we advise you to make sure that there exists a driver for the GRiSP boards. If there is no driver, it is possible to build a tool chain to develop a module yourself in C.

2.3 RTEMS

GRiSP is running RTEMS which is a hard-real-time operating system designed for embedded systems supporting API like POSIX. It is not a conventional operating system, it does not provide any management of the memory. From the application point of view, it looks like a single Unix process with multi-threads. RTEMS is much lighter than most Linux distribution.

2.4 Erlang/OTP

Erlang/OTP is a programming language developed in 1987 by Ericsson and released in 1998 as an open-source software. OTP comes from Open Telecommunication Protocol which is a set of middlewares and libraries to create and structure Erlang systems. Today, OTP is an integral part of Erlang/OTP. Erlang code is compiled in bytecode and executed by BEAM which is the virtual machine of Erlang. BEAM is also used by other language like Elixir.

Erlang is probably one of the best existing languages for creating distributed systems. It has many interesting properties that many languages do not have:

- Erlang is a functional programming language. In this paradigm, functions are first-class data type that means that they can be bound to a variable, we can pass a function as parameter to another function and a function can be returned by another function.
- Erlang is concurrent. The Erlang processes are extremely lightweight. They are created and managed by the virtual machine. Each Erlang process is independent from the others. It has its own memory space, its own heap and its own stack. In Erlang, we can have thousands of processes that can run concurrently. Processes communicate with each other through asynchronous message passing. The message passing between two processes running on the same virtual machine is extremely fast: the virtual machine will simply copy the messages from the memory of one process to the memory of the other process.
- Erlang/OTP is fault tolerant. Erlang uses supervisor tree architecture to create a hierarchy between the processes. The idea is simple, we create a supervisor whose role is to start children processes that are either workers or lower level supervisors. The supervisor will be able to restart a child process that crashes.

- Erlang is distributed. We can create a cluster of nodes that will be able to communicate with each other. Once connected, a node can, as example, send a message to another node. The message will be forwarded to a given process.
- Erlang can scale both horizontally and vertically with ease. Vertical scaling means that the nodes can spawn many processes and use all its resources. Horizontal scaling means that the node can connect to other nodes and send them part of its workload.

Although the language was developed in the 80's, it is still used today in applications requiring a high level of availability and reliability. Nowadays, there are plenty of well-known software that used Erlang like WhatsApp, CouchDB, Amazon SimpleDB and RabbitMQ.

Chapter 3

Dependencies

Lynkia uses two libraries namely Partisan and Lasp. In a few words, Partisan is a library that allows the nodes to exchange messages. Lasp is a library that allow the nodes to create shared data structure. Here, Lynkia will be able to use these structures as input. Lynkia will use Partisan to exchange and broadcast messages efficiently over the network. In this part, we will present these two libraries and we will show you how to use their key features.

3.1 Partisan

Partisan is a library that allows the nodes to exchange messages [10]. The library presents itself as an alternative to Distributed Erlang with additional features and improvements. Unlike Distributed Erlang, Partisan supports 3 different models: the client-server model (1), the peer-to-peer model (2) and the publish-subscribe model (3). Some models are more suitable than others depending on the context considered.

For this project, we will use the peer-to-peer model. With this model, the network can have any topology and each node will be able to communicate with all the nodes without having to maintain a connection with every one of them: When a node sends a message to another node, the message may pass through several intermediate nodes before reaching its destination.

By reducing the number of connections, the system will become more scalable as the nodes do not have to constantly exchange keep-alive messages to every other node. However, the system will become more sensitive to network partition: In some cases, a node can be the only intermediary between two sets of nodes. It means that all messages sent from one partition to another will pass through that node. If that node crashes, the nodes coming from two different partitions will no

longer be able to communicate.

The backend module of the peer-to-peer topology is based on the HyParView membership protocol and the Plumtree epidemic broadcast protocol.

HyParView will be used to manage the connectivity of the nodes. Each node will maintain an active and a passive view:

- The active view is a pool containing the identifiers of the nodes that will be used for message dissemination.
- The passive view is a pool containing the identifiers of the nodes that can replace an unresponsive node from the active view. Unlike the active view, the passive view is not used for message dissemination.

The role of the HyParView membership protocol is to update the active view and the passive view dynamically in order to maintain a low-cost global system connectivity even though there are a lot of failures.

Plumtree will be used to generate a spanning tree. This data structure will be used to broadcast a message efficiently. The node can also use the spanning tree to send a message to a node that is not its active view.

For the peer-to-peer topology, the causal order of the messages is not guaranteed: If a node sends several messages, a node may receive the messages in an arbitrary order even if the underlying communication protocol guarantees a point-to-point in-order delivery like TCP. The reason is the messages may not necessarily take the same path by passing by several intermediate nodes.

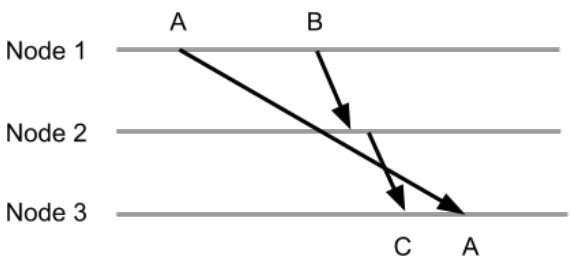


Figure 3.1: Causal order

In this example, the node 1 sends the message A to node 3. Then, the node 1 sends the message B to node 2. When the node 2 receives the message, it produces

a new message C. Then, the node 2 sends the message to node 3. In this example, we can see that the node 3 receives the messages C before the message A which violate the causal order property.

3.1.1 Key functions

Membership management

We can connect two nodes with the following function:

```
partisan_peer_service:join("node@host").
```

To see the active view of the node:

```
partisan_peer_service:members().
```

Messages

Receiver:

```
-module(mymodule).
-behavior(gen_server).

% ...

handle_cast({say_hello_to, Name}, State) ->
    io:format("Hello ~p~n", [Name]),
    {noreply, State}.
```

Sender:

```
partisan_peer_service:cast_message(
    Node,
    mymodule,
    {say_hello_to, "Joe"}
).
```

3.2 Lasp

Lasp is a library that defines helper functions to create and manipulate CRDTs variables [11]. A Conflict-free replicated data type (CRDT) can be seen as a shared variable: Each node will have a local copy of each variable and each node will be able to update them independently. When a node updates a variable, the node will transform it locally. After having applied an update, the state of the variable of the node will be different from the other nodes. To maintain the consistency between the nodes, each node will propagate its state to its neighbors. When a node receives a state, the node will merge it with its own state and obtain a state that reflects all the changes observed by the two nodes.

If you are already familiar with distributed systems, you may know that conflicts will arise when two nodes perform a concurrent update. With CRDTs, we do not have to care about conflicts because they do not exist. When a node updates a variable, the node modifies the state of the variable and not its representation. The representation of the variable is computed from the variable state by using the `query` function.

There exist many types of CRDTs. Each CRDT variable defines a `merge` function, a `query` function as well as functions to transform the variable. These functions will be specific to each CRDT variable. The `merge` will be commutative, associative and idempotent.

- **Commutative**: Thanks to this property, a node can merge several states in an arbitrary order. It means that the causal order of the messages does not have to be guarantee. In other words, the nodes do not have to wait that all messages that causally precedes the given message have been delivered before delivering the given message.
- **Associative**: Thanks to this property, a node can merge a state capturing several updates. It means that the nodes do not necessarily have to merge individually each intermediate state.
- **Idempotent**: Thanks to this property, a node can merge a several times the same state without producing any side effect. It means that the messages can be duplicated.

The mechanism for spreading the changes over the network works exactly like an epidemic. When a node transforms a variable, the node will transform the state of the variable locally. Then, the node will propagate its state to its neighbors. When a neighboring node receives the updated state, it will merge it

with its own state and obtain a state that reflects all the changes observed by the two nodes. The neighboring nodes of the initial node will become infected. They will in turn propagate their state to their own neighbors. By doing so, the node will propagate the changes of the initial node to the other nodes of the network.

When a node updates a variable, there is two possible outcomes: Either, the update will eventually be propagated to all the nodes of the network and every node will sooner or later see the update. Either, the nodes carrying the update crashes and the update will be lost. Of course, there exists different strategies to avoid losing data. One strategy would be to regularly save the state of the node on its persistent storage and recover it when the node is restarted.

The probability of the second scenario occurring decreases as the number of infected nodes increases because each infected node will back up the update: When a node crashes, the node can pull the state of another node. If there exist at least one node in the network having a trace of the update, the update has still a chance of being propagated to all the other nodes. For the update to be lost, all nodes carrying the update will have to be down during the same period. If the number of nodes carrying the update is large, it is unlikely that they will all crash at the same time.

The number of infected nodes over time can be represented with a logistic curve where the x axis corresponds to the time and the y axis corresponds to the number of infected nodes. At the beginning, the number of infected nodes is quite small. Then, it increases exponentially for a short period of time. Then, the function reaches an inflection point and the growth start to decrease until it settles down when the number of infected is close to the total number of nodes.

Lasp supports two different synchronization strategies namely the state-based approach and the delta-based approach. Both approaches have pros and cons.

State-Based With the state-based approach, the node will propagate their entire state. The communication channel does not have to be reliable, the messages can be dropped, duplicated and reordered.

The state-based approach may use a lot of bandwidth since the nodes need to send their entire state frequently. For the CRDT variable that get bigger on each mutation, the node will have to send more and more data. At some point, congestion may appear on the network.

Delta-Based With the delta-based approach, the node will encode the recent mutations in an incremental way by using delta-mutators. A delta-mutator is a function that takes an operation and a state as input and returns as output a small state that represents the effect of the given mutation on the given state. In the literature, this state is called a delta-state. After having generated a delta-state, the node will send it to its neighbors. The neighboring nodes will then merge that delta-state with their own state.

The delta-based approach aims to reduce the quantity of information. Current delta-propagation algorithms can still propagate redundant state between replicas. In some cases, according to [3], it is no better than the state-based approach.

With the Lasp library, it is possible to write a program based on the dataflow programming paradigm by using CRDT variables as input and output. If you want to learn more about dataflow programming, we wrote an additional section in the appendix E.1.

3.2.1 Key functions

Create a variable:

```
lasp:declare({<<"name">>, state_gset}, state_gset).
```

Retrieve a variable:

```
{ok, Value} = lasp:query({<<"name">>, state_gset}).
```

Update a variable:

```
lasp:update({<<"name">>, state_gset}, {add, 42}, self()).
```

Part II

Lynkia

Chapter 4

Algorithms

4.1 MapReduce

MapReduce is a programming model to perform massive and parallel computation on a cluster of nodes. The model is mainly used to process and aggregate a large volume of data. In the context of the Internet-Of-Things, the model could be used to aggregate data measured by the sensors of the devices. We can imagine many use-cases where the model can be used. As example, the devices could regularly measure the temperature with their sensors and compute the maximum temperature observed by each node over the last three days with the MapReduce model.

The model has been designed to be generic. To write a query, the programmer will have to provide two functions namely the `map` and the `reduce` function: in a few words, the `map` function will transform the raw input data. The `reduce` function will be used to generate aggregate results. The function `map` and `reduce` will contain the logic of the query. The programmer does not have to worry about how the system will balance the work between the nodes or how the system will handle the node failures. It is all going to be handled transparently by the system.

4.1.1 Requirements

For this project, we decided to implement a MapReduce algorithm for the GRiSP boards. Our implementation is slightly different from the original MapReduce algorithm. We adapted the algorithm to better consider the constraints of the system as well as its environment:

- As you may notice in section 2, the GRiSP boards have a limited computing power and a limited amount of memory. As part of this project, we tried to

design an algorithm that will be able to process as much data as possible given the technical limitations of the GRiSP boards.

- We will assume that the nodes can leave and join the network at any time for many different reasons. In that conditions, the system is very exposed to network partitioning. As part of this project, we tried to design an algorithm that will be fault and partition tolerant. The algorithm should be able to tolerate up to $n - 1$ failures while ensuring the correctness of the result.
- We will assume that the communication between the nodes is not reliable: The messages can be reordered, duplicated and deleted. As part of this project, we tried to design algorithm that will not be sensitive to that.

4.1.2 Overview

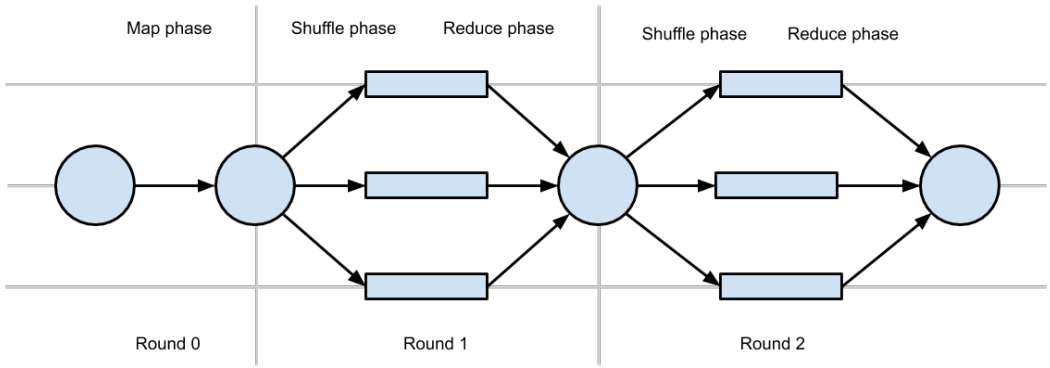


Figure 4.1: MapReduce - Phases

In our implementation, a MapReduce takes place in several rounds. In round 0, the `map` function provided by the programmer will be applied to the raw data to produce key-value pairs. This round corresponds to the map phase. The other rounds are composed of two phases: the shuffle phase and the reduce phase. During the shuffle phase, the pairs will be split in sub-groups. During the reduce phase, the `reduce` function provided by the programmer will be applied on each sub-group. Each sub-group will produce new key-values pairs. The output pairs will become the input pairs of the next round. The shuffle phase and the reduce phase will be repeated successively until they no longer produce any changes on the given pairs.

We distinguish two roles: the leader role and the observer role. The leader will oversee the execution of the MapReduce. The leader will be responsible of the division and the distribution of tasks. The observers will observe the progress of

the MapReduce. The observers will be able to resume the MapReduce from the last observed checkpoint. For each MapReduce, the node will be either a leader or an observer.

4.1.3 Leader

Map phase

During the map phase, the leader will extract raw data from several sources and will produce a list of key-value pairs by using the `map` function provided by the programmer. The map phase ends when all data has been transformed into key-value pairs. The leader will then broadcast a checkpoint containing all information required to resume the MapReduce i.e. the key-value pairs, the round number, the reduce function and some options.

Shuffle phase

During the shuffle phase, the leader will split the input pairs in many smaller sets called batches. Each batch will have a maximal capacity set by the programmer. To minimize the number of rounds, the leader will have to group as much as possible the pairs having the same key within the same batch. To minimize the number of batches, the leader will be allowed be group pairs having a different key in the same batch. In the algorithm we implemented, the shuffle phase takes place in two stages:

During the first stage, the leader will from batches of size M containing pairs having the same key. To form the batches, the node will start by creating an empty key-value dictionary. Each key will be mapped to a list of pairs having that key. Then, the node will insert the input pairs to the dictionary one after the other. When a list contains M pairs, the node will form a new batch from these pairs. Then, the node will remove them from the dictionary.

Each time the node forms a new batch, the node will schedule a new task that will be responsible to the reduction of the considered batch. With this approach, the node that will process the task will be able to start the reduction immediately.

In the following example, the algorithm will perform the first stage of the shuffle phase over the pairs $[\{a, 1\}, \{b, 2\}, \{a, 2\}, \{c, 4\}]$. Here, the node will form batches of size 2.

Pairs	Groups

`[`
`  {a, 1},`
`  {b, 2},`
`  {a, 2},`
`  {c, 4}`
`]`
	`#{}`

`[`
`  {a, 1}, <-`
`  {b, 2},`
`  {a, 2},`
`  {c, 4}`
`]`
	`#{`
	`  a => [{a, 1}]`
	` }`

`[`
`  {a, 1},`
`  {b, 2}, <-`
`  {a, 2},`
`  {c, 4}`
`]`
	`#{`
	`  a => [{a, 1}],`
	`  b => [{b, 2}]`
	` }`

`[`
`  {a, 1},`
`  {b, 2},`
`  {a, 2}, <-`
`  {c, 4}`
`]`
	`#{`
	`  a => [{a, 1}, {a, 2}],`
	`  b => [{b, 2}]`
	` }`

`[`
`  {a, 1},`
`  {b, 2},`
`  {a, 2}, <-`
`  {c, 4}`
`]`
	`#{`
	`  b => [{b, 2}]`
	` }`

`[`
`  {a, 1},`
`  {b, 2},`
`  {a, 2},`
`  {c, 4} <-`
`]`
	`#{`
	`  b => [{b, 2}],`
	`  c => [{c, 4}]`
	` }`

In this example, the node will form one batch: $[\{a, 1\}, \{a, 2\}]$. There will remain some residual pairs that will be grouped during the second stage.

During the second stage, the node will form batches from the remaining pairs. The number of remaining pairs per key will be strictly smaller than M. To form the batches, the node will start by generating a list from the values of the dictionary. The list will therefore be a list of lists. Then, the node will sort the lists by size in descending order. To form the batches, the node will start with an empty batch. Then, it will iterate over the list of lists. The node will add the considered pairs to the batch if the size of the batch will not exceed M by adding them. When a batch is full or no other list can be added to the batch, the node will form a new batch and repeat that process until there is no more remaining pairs.

In the following example, the algorithm will perform the second stage on some remaining pairs. Here, the node will form batches of size 3.

First iteration:

Pairs	Groups

`[`
`  [{a, 1}, {a, 42}],`
`  [{d, 4}, {d, 2}],`
`  [{b, 5}]`
`]`
	`[]`

`[`
`  [{a, 1}, {a, 42}], <-`
`  [{d, 4}, {d, 2}],`
`  [{b, 5}]`
`]`
	`[{a, 1}, {a, 42}]`

`[`
`  [{d, 4}, {d, 2}], <-`
`  [{b, 5}]`
`]`
	`[{a, 1}, {a, 42}]`

`[`
`  [{d, 4}, {d, 2}],`
`  [{b, 5}] <-`
`]`
	`[{a, 1}, {a, 42}, {b, 5}]`

`[`
`  [{d, 4}, {d, 2}]`
`]`
	`[{a, 1}, {a, 42}, {b, 5}]`

Second iteration:

Pairs	Groups

`[`
`  [{d, 4}, {d, 2}]`
`]`
	`[]`

`[`
`  [{d, 4}, {d, 2}] <-`
`]`
	`[{d, 4}, {d, 2}]`

`[]`	`[{d, 4}, {d, 2}]`

In this example, the algorithm will produce two batches: $[\{a, 1\}, \{a, 42\}, \{b, 5\}]$ and $[\{d, 4\}, \{d, 2\}]$. At the end of the second stage, there will be no remaining pairs left.

One important remark is that the solution obtained by using the algorithm mentioned above will not necessarily be the most optimal one. The problem is actually similar to the bin packing problem. Unfortunately, the problem has an NP-hard computational complexity. To get the optimal solution, we would have to try many combinations. As the search space is large and grows exponentially with the number of lists considered, it might take too much time to compute the optimal solution. Here, we proposed a heuristic that will find an acceptable solution in a reasonable time.

Reduce phase

During the reduce phase, the node will wait to receive the result of all tasks produced during the shuffle phase. Once the node gets all the results, the node will broadcast a new checkpoint containing the new key-value pairs, the new round number, the reduce function and some options. Then, the node will start a new round if the input pairs and the output pairs are different. If they are not, an additional shuffle and reduce phase will not produce any changes. The node can therefore stop the MapReduce.

In this project, we implemented our own task model. With this model, the tasks will be executed by the node itself or by one of the other nodes of the network.

The task model will guarantee that the node will receive the result of the tasks it scheduled if it does not crash. If the node does not have any neighboring nodes, the node will execute every reduction itself. With this approach, the leader does not depend on the other nodes. The system will make progress as long as there is at least one leader in the network.

As the resources of the GRiSP boards are limited, we put a restriction on the maximal amount of memory that a task can use during its execution. We also put a limit on the execution time of a task. If the task consumes too much memory or takes too long to compute, the task will be killed. If the task associated to a batch has been killed, the leader will split the batch in half and will schedule two new tasks. The node will keep splitting the batches until they no longer cause problems.

4.1.4 Observer

The observer will observe the activity of the MapReduce. When an observer no longer observes any activity for a certain period, the observer will take the lead and become a leader to resume the MapReduce from the most advanced checkpoint.

4.1.5 Fault management

To detect leader inactivity, each observer will maintain a timer. The timer will be reset each time the observer receives a new message from a leader. The leader will regularly broadcast heartbeat messages to prevent the observers to timeout. If the timer of an observer expires, the observer can reasonably assume that the leader has crashed since it no longer observes any activity. In that case, the observer will become a master and resume the MapReduce from the most advanced control point.

Unfortunately, the failure detector is not perfect. Sometimes, the timer of an observer can expire for many reasons: the leader or the network may experience slowdowns, the message can be delayed, etc. In that case, there could be several leaders within the same partition. Having several leaders will not break the system. However, it will slow the system down as each leader will produce independent tasks that will have to be executed.

When there are several leaders within the same network partition, the leaders will compete against each other. When a leader sees that another leader reached a more advanced round, the leader will let the other leader finish the job and it will become an observer. With this approach, we will eliminate concurrent leaders and reduce the total number of tasks produced.

To prevent all observers to become leader at the same time, each observer will have a random timeout. That way, the observers are less likely to all become leader at the same time. When an observer becomes a leader, the node will start broadcasting heartbeat messages before the other nodes timeout.

If an observer crashes, it is not a big deal: the observers can be seen as backup nodes. The leader will still be able to make progress and the other observers will be able to resume the MapReduce when needed.

As we said earlier, we assumed that the communication channel between the node is not reliable: the messages can be omitted, delayed, reorder and duplicated. Our algorithm will not be sensitive to these problems. When an observer receives a checkpoint, the observer will cache the checkpoint if it is more advanced than the cached one. If the observer received an older checkpoint or the same checkpoint, the node will ignore it. If the observer does not receive the most advanced checkpoint, the observer will resume the MapReduce from an older checkpoint. It will not affect the correctness of the algorithm. The nodes will have a little more work to do.

4.2 Task model

In the last section, we saw that the leader will produce one task per batch during the shuffle phases. Now, we would like to have a module that will have the responsibility to process these tasks and to distribute them.

4.2.1 Related work

Erlang BIF

Those of you who are familiar with Erlang may have notice that there exists a build-in function named spawn that can be used to execute a function on a remote node. This function will take two arguments: The node and the function that the node should execute.

```
erlang:spawn(Node@localhost, fun() ->
    io:format("Hello world~n")
end).
```

Unfortunately, this function does not have a fallback strategy: if the remote node crashes or does not respond, the function will simply print a warning message

and the given function will not be executed. Moreover, the workload management is left to the responsibility of the programmer: The function will be executed on the specified node even if it is overloaded. The node will not forward the function to another node.

Achlys

Achlys is a library that implements helper functions to create tasks that will be backed up in a CRDT variable and executed by one or several nodes [6]. There are many application cases where this feature can be used especially in the context of Internet-of-Things. For instance, a node could ask every node to measure the temperature with its sensor and add their results to a given CRDT variable.
Example:

```
Name = "Node@localhost",
Task = achlys:declare(Name, Node, single, fun() ->
    io:format("Starting the task~n")
end),
achlys:bite(Task).
```

For the MapReduce, we decided not to use the task model of Achlys for several reasons. Achlys will store the tasks in a grow-only set. As its name suggest, the state of the grow-only sets will keep growing as new elements are inserted. With the grow-only sets, it is not possible to remove an element once it has been inserted. It means that Achlys will not be able to remove a task that have been completed. The tasks will hence accumulate over time. This accumulation is problematic because it can cause network congestion and crashes:

- Network congestion: If the nodes propagate their states frequently and use the state-based propagation approach, congestion may appear on the network since the nodes will have to transfer more and more data as the state of the variable grows.
- Crashes: If we keep adding tasks, the nodes will at some point run out of memory. The node will crash and will no longer be available.

Using another type of set will not completely solve the problem: The state of all CRDT variable characterizing a set are monotonic. It means that the state of the variable is not allowed to return to a former state. Most sets that expose a remove operation will internally keep track of all the elements that have been removed.

With the observe-remove set, it is possible to remove an element from the set without using a remove set (or tombstones set). This optimization aims to reduce the memory usage of the variable while maintaining the monotonicity property.[7]

Unfortunately, Lasp does not provide any function to delete a CRDT variable. To get rid of the completed tasks, we cannot create a new variable, copy all the non-completed tasks into it and delete the original variable.

The other problem is that execution delay will depends on the propagation delay of the CRDTs and the look up interval of the workers. These delays can be modified in the configuration file. With the default parameters, the execution delay will be high. If we reduce the value of these parameters, it will put more pressure on the nodes as they will have to propagate and merge their states more often. It will then probably be faster to execute the function locally rather than asking another node to execute it.

For these reasons, the task model of Achlys should be used wisely: The Achlys task model is great to execute a recurrent task on several nodes. However, it should not be used to speed up the execution of a program.

4.2.2 Overview

As we could not find a model that fully meets our needs, we decided to implement our own task model based on Partisan. The task model will be separated from the MapReduce algorithm. That way, the task model can easily be reused. To schedule a task, the programmer will have to provide a function and its arguments.

Each node will maintain a queue. When the node schedules a task, the node will add a new task to the end of its queue. The node will then process the tasks that have been scheduled one after the other. Here, the node will always execute the first task of its queue. When the task has been completed, the node will remove the task from its queue.

When the node is overloaded, the node can forward tasks to its neighbors. Here, the node will forward the last tasks of the queue. Once a task has been forwarded, the task will remain in the queue. That way, the node will be able to process the task itself if the remote node does not respond.

As it takes time for the remote node to process the task, it will not be efficient to forward the tasks that are about to be processed: If the node uses this strategy, the nodes will execute the same tasks. The node will hence perform redundant work which will not help the node to go faster.

When a node receives a task, it will add the task to its queue. The node will then process the task as its own. If the remote node is also overloaded, the node can forward tasks to its neighbors. Hence, a task can be forwarded several times before being executed.

The specificity of our task model is that the nodes will not be allowed to forward the same task to different nodes. This constraint aims to reduce the speed of propagation of the task over the network. Here, we assume the nodes will forward a task to an idle node. If the node is idle, it will be able to process the task quite rapidly. It will not be necessary to forward the task to more nodes.

With this constraint, the task will propagate linearly over the network. The task will go from one node to another and will not be allowed to pass twice by the same node. The nodes that received the task form a chain. In the worst case, the task will pass through every node if the graph is traceable.

All the nodes belong to the chain will eventually process the given task. When a node completed a task, the node will send its result to its predecessor and ask its successor to discard the task. When a node receives a result, the node will send the result to its predecessor and remove the task from its queue. When a node receives a message `kill`, the node will send the message to its successor and remove the task from its queue.

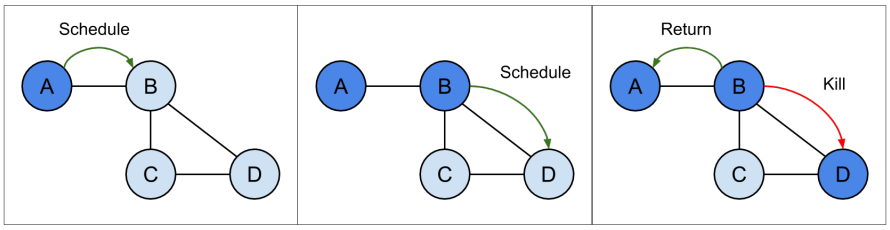


Figure 4.2: Task propagation

In this example, the node A schedules a task. Then, the node decides to forward the task to node B. Node B will decides to forward the task to node D. At this point, the node A, B and D will have the task in their queue. Eventually, each

node will process the scheduled task. Here, the node B get the result first. The node will return the result to the node that scheduled the task (i.e. Node A) and the node will send a kill message to the node it forwarded the task to (i.e. Node D).

In a first iteration of the algorithm, we allowed the nodes to forward a task to several nodes. We notice that this approach was quite inefficient:

- A node could receive several times the same tasks from different nodes. Sometimes, the node completes the task, returns its result and receives the tasks from another node a few seconds later. To avoid processing the task repeatedly, we cached the result of the task. The node was then able to return the cached result if the task has already been processed. Unfortunately, this approach consumes a lot of memory which is already limited.
- To discard a task, the nodes had to exchange many messages. We noticed that the nodes took longer to send messages to schedule and kill the task than they did to execute it.
- When we scheduled a lot of tasks to test the limit of the system, we end up with the worst-case scenario where all nodes get all the tasks. In that case, the system was extremely slow.

Regarding of the properties of the task model, it will guarantee that the node will receive the result of the tasks it scheduled if it does not crash. With our task model, the nodes will be independent: if a node forwards a task to one of its neighbors and the neighboring node does not do the requested work, the node will sooner or later end up doing the work itself.

4.2.3 Load balancing

In our task model, the nodes will be able to balance their load by forwarding tasks. Every task that will be process by another node will not have to be processed by the node itself. There are different strategies of load balancing. The two main ones are the push strategy and the pull strategy. Each strategy will have pros and cons.

With the push strategy, the node will give tasks to its neighbors. Here, the nodes will decide which node will have to process a given task. To minimize the computation time of the tasks, the nodes will try to identify the idle nodes and

send them some tasks. To identify these nodes, the nodes may measure statistics like the response time for instance.

With the pull strategy, the node will steal tasks to its neighbors. Here, the idle nodes will manifest themselves naturally. The nodes will have more control on their load by explicitly indicating the number of tasks they want to steal.

The main problem of the push strategy is that the node can wrongly assume that a node is idle. As a result, the chosen node may be overloaded and may take times to complete the given task which will affect the efficiency of the task model. However, the push strategy will be better to absorb load spikes as the node can forward the task instantly.

4.2.4 Fault management

If a node forwards a task to a slow or overloaded node, the node that scheduled the task initially may complete the task first. In that case, the node will ask the other node to discard the task by sending a `kill` message.

If an intermediate node of the chain crashes, the result of the task cannot be traced back to the node that scheduled the task initially. As the forwarded tasks remain in the queue, the forwarded task will eventually be processed by the node itself.

As we have seen, the topology of the network can change at any time. Thanks to Partisan, a node will still be able to send messages to its successor and its predecessor even if the nodes are no longer direct neighbors. The messages will transparently pass through intermediate nodes.

In the worst case, the node will re-schedule the task. When the node completes the task and send the result to its predecessor, the predecessor will ignore the message and stop the propagation of the message if the task has already been processed.

Sometimes, the node can schedule a task that loop endlessly and never return a result. Unfortunately, the node cannot determine whether a function stop or not. It has been proven that this problem is undecidable (see the halting problem). The function can also use a lot of memory and may slow the board down. Here, we limit the computation time and the memory usage of the tasks. If a task exceeds these limits, the node will kill the function and return an error.

Chapter 5

Implementation

In this section, we will see how the MapReduce and the task model have been implemented in Lynkia. Here, we will give the specificities of the library and we will justify some implementation choices.

5.1 MapReduce

To make sure that a node cannot be leader and observer, we create two separated modules called `lynkia_mapreduce_leader` and `lynkia_mapreduce_observer`. These two modules are mutually recursive. When the node receives a specific message, the node can change its role. The node will behave differently depending on the role in which the node is.

5.1.1 Adapters

To load data from different sources, we implemented several adapters. An adapter is an intermediate module that allow two incompatible modules to work with each other. Here, the adapter will have the responsibility to load data from different libraries and to expose a common API. In this project, we provided three adapters:

- `lynkia_mapreduce_adapter_csv`: This adapter will be used to load data from a CSV file. Here, the file will have to be located on the disk of the node.
- `lynkia_mapreduce_adapter_file`: This adapter will be used to load data from a file. Here, the file will have to be located on the disk of the node.
- `lynkia_mapreduce_adapter_lasp`: This adapter will be used to load data from a CRDT variable like a set.

An adapter can contain several entries. Each entry will define a `map` function that will be applied to the raw input data of the corresponding data source. Here,

the leader will call the `map` function for each data entry. The `map` function will have to produce a list of key-value pairs.

```
Var = {<<"var">>, state_gset},
Adapters = [
    {lynkia_mapreduce_adapter_csv, [
        {"dataset/file_1.csv", fun(Tuple) ->
            %% (map function)
        end},
        {"dataset/file_2.csv", fun(Tuple) ->
            %% (map function)
        end}
    ]},
    {lynkia_mapreduce_adapter_lasp, [
        {Var, fun(Entry) ->
            %% (map function)
        end}
    ]}
],
%% ...
```

In this example, the leader will parse the files `file_1.csv` and `file_2.csv` from the `dataset` directory. For each CSV file and each row of these files, the node will call the corresponding `map` function. Here, the argument of the anonymous function will correspond to a row of the csv file. The leader will also load data from a CRDT variable called `var`. For each entry of the grow-only set, the node will call the corresponding `map` function. Here, the argument of the anonymous function will correspond to an element of the set.

A programmer can easily implement a new adapter to load data from a different source. To do that, the programmer will have to create a new module that implements the function `get_pairs/3` taking three arguments: the entries, the options and a callback function. Then, the programmer can start using its adapter by adding a new tuple in the variable `Adapters`.

With the adapters, the code base of Lynkia is easy to extend and easy to maintain: if the API of the library changes, we will have to modify the corresponding adapter without having to change the logic of the algorithm.

Once the leader gets all the pairs produced by the `map` function, the leader will broadcast the pairs and all information required to resume the MapReduce. Then, the leader pass to the first shuffle phase.

5.1.2 Broadcast

To broadcast messages, we used a gossip protocol. These protocols are known to be efficient for large networks. Partisan provides different gossip protocols. In this project, we will apply some changes to the "Demers et al.'s Rumor-Mongering" protocol [9]. The implementation of the algorithm can be found on the official repository of Partisan (see here).

When a node broadcasts a message, the node will generate a unique id. Here, the node will form a tuple containing its name and a monotonic number. The number will initially be set to 0 and will be incremented for each new message. Then, the node will select k nodes from its local view and will send them the message to broadcast. Here, the parameter k corresponds to the *fanout*.

When a node receives a message to broadcast, the node will check if the message has already been processed. If that is the case, the node will ignore the message. Otherwise, the node will deliver the message to the indicated module. Then, the node will select k nodes from its local view and send them the message to broadcast. Here, the node will not be allowed to select the node that sent the message.

Example:

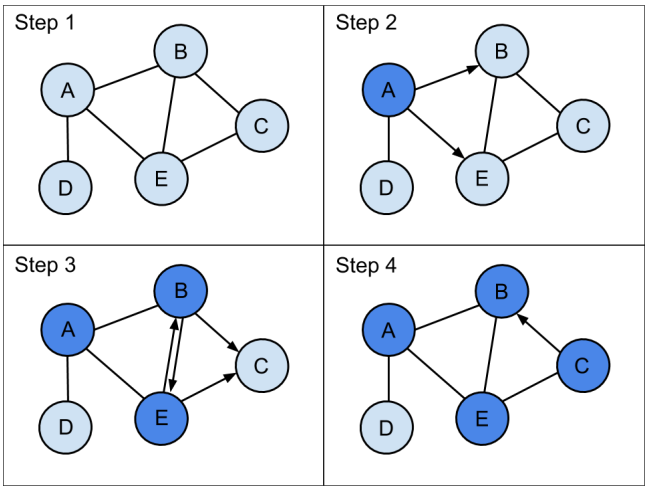


Figure 5.1: Broadcast - Message propagation

In this example, we fixed the fanout to 2. As you can see, the node D did not receive the message. The gossip protocol is hence not always reliable: when a message reaches the node A, the node D has a two out of three chance of receiving the message from A given that the neighboring nodes of A have the same chance of being selected.

To make sure that all nodes eventually receives the message, we should increase the fanout so that each node sends the message to all its neighboring nodes. In the given example, the fanout should be equal to 3. That way, the node D will always be selected. By increasing the fanout, the nodes will have to send more messages. For large system, it can cause network congestion and slowdowns. There is a trade-off between the scalability and the reliability of the system.

In the original implementation, the node will add a new entry in `ets` whenever the node receives a new message. That way, the node can easily determine whether a given message has been processed. Here, the node will store the entire message in `ets` which can use a lot of memory.

The main flaw of the original implementation is that the node will not remove the old messages from its memory. It means that the size of the `ets` table will keep growing as the node receives new messages. To fix this problem, we implemented a garbage collector and we change the format of the `ets` entry as follow:

- The key of the entry will be a tuple containing the name of the node, the monotonic number and a hash of the message. Here, we will use the function phash2 to hash the message.
- The value of the entry will be the timestamp of the message. Here, we will assume that a message cannot circulate for more than x seconds in the network. With the value, the garbage collector that we implemented will know whether the message can be removed from memory.

In Lynkia, the leader will use that algorithm to broadcast a new checkpoint. In the previous chapter, we have seen that the MapReduce algorithm is tolerant to message omission. To reduce the quantity of messages exchanged over the network, the leader could broadcast the checkpoints every n rounds. With this approach, the nodes may lose more progression if the leader crashes. Nevertheless, the observers will still be able to resume the MapReduce from an oldest checkpoint. If the nodes are reliable, it would be fine to have a n larger than 1. The programmer will be able to fix n in the configuration file (see `mapreduce_broadcast_interval`).

5.1.3 Garbage collector

In Erlang, we do not have an explicit function to free a memory block. To remove data from a process, we implemented a function `gc` that will ask the process to remove all expired data from its state. Here, the process will produce a new state that no longer contain the expired data. When the process passes from one state to another, the garbage collector of Erlang will eventually remove the old state from memory.

In Lynkia, we used the following pattern: each time the process adds a data that can expire, the node will start a new daemon process. If the daemon process is already running, no other daemon will be spawned. The daemon process waits x seconds, calls the `gc` function and terminates. When the `gc` function is called, the node will produce a new state without the expired data. The process will restart the daemon only if there are remaining data in its state that are not been removed. With this approach, the daemon will not call the `gc` function during inactivity. The `gc` function will be called at most once every x seconds as there will be at most one daemon. We used this approach in two modules: `lynkia_mapreduce` and `lynkia_broadcast`

5.2 Task model

In the previous chapter, we have seen that the node is able to schedule tasks and to process them with or without the help of the other nodes. In Lynkia, the node will be able to process several tasks concurrently. The maximum number of tasks that can be run concurrently can be changed in the configuration file. Here, we set this number to 1 because the GRiSP boards have a single core processor. If we allow the node to execute N tasks concurrently, the scheduler will dedicate up to $\frac{1}{N}$ CPU time to each task assuming that the scheduler of Erlang is fair. Increasing N will therefore not speed up the system. In the future, the boards may have more cores. In that case, it might worth increasing N.

We also seen that the node will not be allowed to forward a task to a node that already scheduled it. To avoid cycles, the task will carry a hop list that will contain all intermediate nodes of the chain. If the node wants to forward the task, the node will have to exclude the nodes of the hop list from the potential candidates. With the hop list, the node can also determine which node sends the task. If the hop list is empty, it will mean that the function has been scheduled by the node itself.

5.2.1 Load balancing

Regarding of the forwarding strategy, we tried to decouple the forwarding strategy from the core of the task model. That way, the programmer can change the forwarding strategy with ease. The programmer will also have the freedom to create its own strategy if the provided ones do not fit his/her needs.

The core of the task model will be in the module `lynkia_spawn`. In this implementation, the node will maintain several data structures to store additional information for the running tasks, the forwarded tasks and the scheduled tasks. Each time a task is added or removed from one of these structures, the node will emit a new event. The module responsible of the forwarding strategy will listen to these events and can ask the core module to forward some tasks to a given node.

Thanks to the events, the node can count the number of tasks scheduled, the number of tasks forwarded, the total number of tasks. The node can also measure the response time of given node. Based on these data, the module can integrate a complex model that will try to predict which nodes are idle. Then, the module may decide to send some tasks to these nodes.

Push strategy

The push strategy we implemented is naive: the module will maintain a counter to count the number of tasks that have been scheduled. When the module receives an event indicating that a new task has been scheduled, the node will increment the counter. When the module receives an event indicating that a task has been completed, the node will decrement the counter.

- When the counter is lower than the given threshold, the module does not do anything. It does not ask the node to forward a task.
- When the counter is larger than the given threshold, the node will send one task to one of its neighbors. Here, the node will be chosen randomly but it will be possible to base this decision on statistics.

With this strategy, all neighboring nodes will receive on average the same number of tasks. If the node forwards its tasks to a single node, the node will have to execute all tasks itself if the remote node crashes. By sending tasks to several nodes, we are reducing that risk. Moreover, the workload will be fairly distributed if the tasks have a similar complexity. Our naive strategy is actually not that bad.

Pull strategy

The pull strategy we implement will also count the number of tasks that have been scheduled by using the same approach as the push strategy.

- When the counter is lower than a given threshold, the module will spawn a daemon process that will regularly send a message to its neighbors to ask them some work. When a neighboring node receives such message, the node will send the requested number of tasks to the node.
- When the counter is larger than the given threshold, the module will kill the daemon process. That way, the node will no longer receive tasks from its neighbors. The node will restart the daemon process once the number of scheduled tasks is below the threshold.

Chapter 6

Usage

In this section, we will show you how to use the MapReduce and the task model of Lynkia. Then, we will also show you how to configure and extend Lynkia. If you want to know how to install Lynkia, we will provide you a small guide in the appendix A.1.

6.1 Use Lynkia

6.1.1 MapReduce

To show you how to schedule a MapReduce with Lynkia, we will consider the following example. Let's suppose that we have the following data:

```
temperature;country;weather
2;Netherlands;Sunny
23;Germany;Cloudy
24;Belgium;Sunny
5;France;Sunny
0;Netherlands;Rainy
24;France;Rainy
```

Let's suppose that we want to retrieve the maximal temperature per country. To do that, we can use the MapReduce model. The first step will consist of loading data. Here, we will suppose that the data will be store in a CSV file located on the disk of the node.

To load the data from the CSV file, we will have to an adapter. As we have seen, Lynkia provide three different adapters. Here, we will use the adapter named `lynkia_mapreduce_adapter_csv`. For this adapter, we provided the path of the

CSV file as well as a `map` function that will be applied to all the records of the corresponding CSV file.

```
Adapters = [
    {lynkia_mapreduce_adapter_csv, [
        {"dataset/test.csv", fun(Tuple) ->
            case Tuple of #{
                temperature := Temperature,
                country := Country
            } -> [{Country, erlang:list_to_integer(Temperature)}];
            _ -> [] end
        end}
    ]}
],
```

For this adapter, we provided the path of the CSV file as well as a `map` function that will be applied to all the records of the corresponding CSV file. In this example, the map function produces a list of key-value pairs. The key of the pairs will corresponds to the name of the country and the value of the pairs will correspond to the temperature of the country.

The `map` function will transform these tuples:

```
#{temperature => 2, country => Netherlands, weather => Sunny}
#{temperature => 23, country => Germany, weather => Cloudy}
#{temperature => 24, country => Belgium, weather => Sunny}
#{temperature => 5, country => France, weather => Sunny}
#{temperature => 0, country => Netherlands, weather => Rainy}
#{temperature => 24, country => France, weather => Rainy}
```

To these pairs:

```
[
    {Netherlands, 2},
    {Germany, 23},
    {Belgium, 24},
    {France, 5},
    {Netherlands, 0},
    {France, 24}
]
```

Then, we will have to write a `reduce` function. This function will take two parameters: the key and the list of values associated to that key.

```
Reduce = fun(Key, Values) ->
    [{Key, lists:max(Values)}]
end,
```

Here, the `reduce` function will produce a single output pair that will correspond to the maximal value of a given country. One important remark is that the `Values` will not necessarily contain all values associated to `Key`. Some pairs might be present in other batches.

Then, we can define overwrite some default options. To use the `options` structure, we will have to include the file `lynkia.hrl` that contain the definition of the `options` structure.

```
Options = #options{
    max_round = 100,
    max_batch_size = 100,
    timeout = 3000
},
```

In this example, the algorithm will not be allowed to perform more than 100 rounds. The batches cannot hold more than 100 pairs and a task cannot take more than 3000ms to complete. Then, the node can schedule the MapReduce as follow:

```
lynkia:mapreduce(Adapters, Reduce, Options, fun(Result) ->
    case Result of {ok, Pairs} ->
        io:format("Pairs=~p~n", [Pairs])
    end
end).
```

Here, the anonymous function corresponds to the callback function. This function will be executed when the algorithm gets the final pairs. The parameter named `Result` will correspond to the final pairs. An important remark is that the function can be executed by another node.

6.1.2 Task model

To schedule a function on our task model, we can use `spawn/2` and `spawn/3`. `spawn/2` will be used to perform a synchronous call. It means that the execution thread of the process that call the `spawn/2` function will be blocked until the task model returns the result of the scheduled task. `spawn/3` will be used to perform an asynchronous call. It means that the execution thread will not be blocked. The task model will return one of these four results:

- `{ok, Result}`: When the task model returns this tuple, it means that the function terminates normally. Here, `Result` corresponds to the value returned by the provided function.
- `{error, Reason}`: When the task model returns this tuple, it means that the function performs an illegal operation and crashes. Here, `Reason` corresponds to the error that has been encountered during the execution of the provided function.
- `timeout`: When the task model returns this tuple, it means that the function has not been complete in time.
- `killed`: When the task model returns this tuple, it means that the function consumes too much memory.

Asynchronous call

For the asynchronous call, the function `spawn` of the `lynkia` module will take three arguments: the function that should be executed, its arguments and a callback function.

Example:

```
lynkia:spawn(fun() ->
    % Body of the function
    timer:sleep(1000),
    42
end, [], fun(Result) ->
    io:format("Result: ~p~n", [Result]) % Print "Result: 42"
end)
```

Synchronous call

For the synchronous call, the function `spawn` of the `lynkia` module will take two arguments: the function that should be executed and its arguments. The function will return the result of the function.

Example:

```
Result = lynkia:spawn(fun() ->
    timer:sleep(1000),
    42
end, []),
io:format("Result: ~p~n", [Result]) % Print "Result: 42"
```

6.2 Configure Lynkia

Lynkia uses a lot of constants. To make the system configurable, we defined these constants in the configuration file (see `config/test.config.src`). That way, the programmer can change them easily. The programmer could for instance increase the number of tasks that the node can process concurrently or change the frequency of the garbage collector. There are plenty of variables that can be changed. In the appendix A.3, we will give you an exhaustive list of these variables.

Example:

```
{lynkia, [
    %% Broadcast
    {fanout, 2},
    {broadcast_ets_gc_interval, 3000},
    {broadcast_ets_ttl, 4000},
    ...
]}
```

6.3 Extend Lynkia

As we said earlier, Lynkia has been designed to be modular. The programmer can choose between different forwarding strategies. To change the forwarding strategy, the programmer will have to change the variable `task_distribution_strategy` of the configuration file. This variable will have to be set to the name of the module

that will be responsible of the load balancing.

In Lynkia, we provided two different strategies: the programmer can set the variable to `lynkia_push_strategy` for the push strategy or to `lynkia_pull_strategy` for the pull strategy. If these strategies do not fit its needs, the programmer is also able to create its own forwarding strategy.

To create a new strategy, the programmer will have to create a new module that implement the `on` function. This function will take one argument that corresponds to the event that have been emitted. To store data, the programmer could spawn a new process. When the function `on` is called, the function could send to the process a new message containing the emitted event. Then, the process could update its state in response to these messages and store new data in its state.

At any time, the process can decide to forward new tasks by calling the `forward` function of the module `lynkia_spawn`. This function will take two parameters: the chosen node and the number of tasks to send to that node.

Part III

Evaluation

Chapter 7

Test scenarios

7.1 MapReduce

7.1.1 Experimental protocol

In this section, we will assess the reliability of the MapReduce. We will run the tests on 5 GRiSP boards. Each GRiSP board will have the configuration mentioned in the section 2. The board will be connected via an ad-hoc Wi-Fi connection. During the test, each board will write logs to their micro SD card with Logger. After the experience, we copied the logs from the micro SD card to our laptop. For the following graph, we wrote a Python script to parse the logs and draw visuals.

When a GRiSP board boot, the board will set the system time to a default value. Each board will have a different system time. In the following example, we will schedule a MapReduce on one board. The first log written by the MapReduce will be our reference block: the time 0 will correspond to the moment when the node wrote that first log. Every other logs of the node will be aligned to it. The x axis will correspond to time elapsed since that first log.

In section 5.1.2, we seen that the broadcast is not always reliable. The node may write their first log several seconds apart. As a result, the time 0 of one node may not be the same as the time 0 of another. To fix this problem, we added an offset manually to synchronize the nodes.

7.1.2 Test

To test the reliability of the system, we will create a synthetic test A.5.2. With this test, the node will produce many rounds. Here, we want to see the nodes change

roles when a failure occurs.

The `map` function will produce some key-value pairs:

Key	Value
0	1
0	2
0	3

The `reduce` function will apply a map function that will increment the key of the pair. The function will take n pairs as input and will produce n pairs as output.

Key	Value
1	1
1	2
1	3

The node will stop incrementing the keys if the keys is larger or equal than a given limit. This limit corresponds to the number of reduce phase we want.

Regarding of the `reduce` function, we added a delay to simulate a heavy workload. The delay is fixed to 100 milliseconds. That way, the MapReduce will take longer to complete and we will have more time to disconnect the nodes manually. The computation time of the following example is therefore not representative. Here, we want to test the reliability of the system and not its efficiency.

For the tests, we pass a set of 100 key-value pairs to the synthetic test and we set the number of reductions to 10. We expect to observe twelve rounds: ten rounds for the reduction reductions, one round for the map phase and one round to verify that the MapReduce is over.

We have put the configuration of Lynkia we used for the tests in appendix B. That way, the experiments can be replicated.

7.1.3 Topologies

We ran the synthetic test on three different topologies.

Full mesh topology Each node is directly connected to the other $N - 1$ nodes. There is a direct path between any node to any other node of the network.

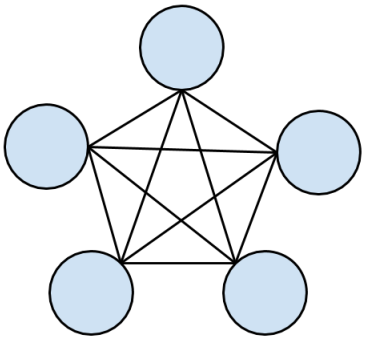


Figure 7.1: Full mesh topology

Line topology Each node has at most two neighbors and there is no cycle in the network: the first node is not connected to the last one. There is only one path between a node and another. This topology allows us to verify that the messages are propagated even to nodes that are 4 hops away from the leader.

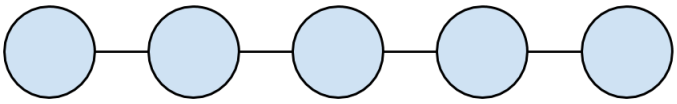


Figure 7.2: Line topology

Butterfly topology There is a center node connected to the other four nodes. The other nodes are connected to the center node and to one other node.

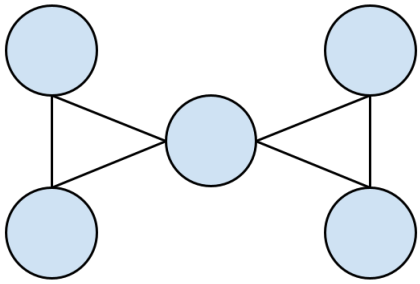


Figure 7.3: Butterfly topology

7.1.4 Scenarios

In this section, we will present different scenarios that could occurs. For each scenario, we will describe the expected result. Then we will present the observed results and discuss them. With this evaluation, we want to show that our system is resilient: it will get the right result even if some nodes crash and some messages are omitted, delayed, reordered or duplicated.

Legend of the graphs

- Orange blocks: Period of time during which the node is leader;
- Blue blocks: Period of time during which the node is observer;
- Vertical black lines:
  - For the leader, the black line represents the end of a round, the number of the ended round is the number above the line.
  - For the observer, the black line represents the reception of a checkpoint sent by the leader at the end of some rounds.
- *kill* at the end of a block: the node finished the MapReduce normally and it got the result. If there is no *kill* that means that the node crashed.

Single node

We start the MapReduce evaluation with the basic case: the single node. The purpose of this test is to show that our system works correctly in the simplest case. It can happen that a node is alone, disconnected from the network. It must still continue to work.

Expected behavior When a MapReduce is started on a single node, it become the leader and remain so until the end of the MapReduce. The leader executes all the rounds and obtain the result of the MapReduce.

Observed behavior We have run the synthetic test on a single GRiSP board.

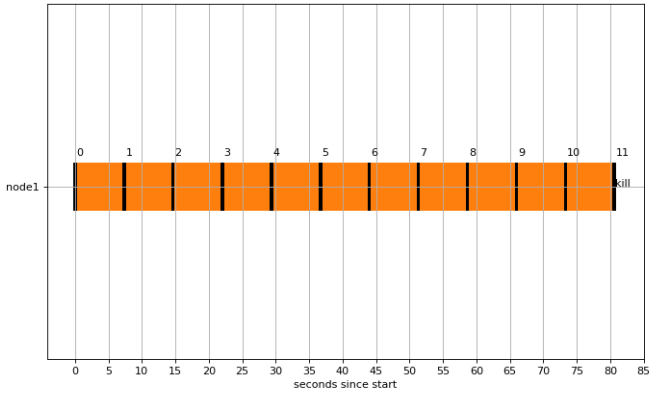


Figure 7.4: Single node

Analysis of the results The node was leader during the whole computation of the MapReduce. It executed the twelve rounds and finished the computation normally and it got the final result.

Five nodes in the network

For this evaluation, we have connected five GRiSP boards to each other in different topologies. The purpose of the following tests is to show that our system works correctly in a network of five nodes in different topologies. We want to make sure that all nodes get the final result of the MapReduce.

Expected behavior The node that started the MapReduce remains leader until the end. The leader computes the MapReduce in twelve rounds and the observers receive four checkpoints because the leader broadcasts the result of one in three rounds. The leader broadcasts the result of the MapReduce to other nodes of the network.

Observed behavior - full-mesh We have run the synthetic test on node 1.

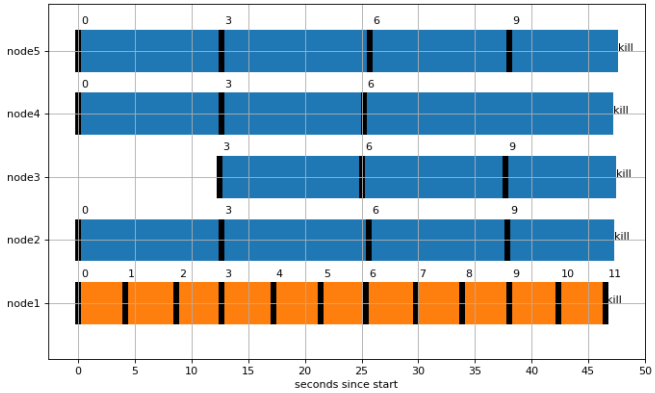


Figure 7.5: Full mesh topology

Analysis of the results The leader was the node 1 and it remained it until the end of the MapReduce. It finished the MapReduce in twelve rounds. We observe that some checkpoint messages were not received by some observers: node 3 did not receive the checkpoint of round 0 and node 4 did not receive the checkpoint of round 9. The checkpoints messages have been lost or delayed. At the end, we observe that all the nodes got the result of the MapReduce.

Observed behavior - butterfly We have run the synthetic test on node 1 which is the node at the center of the network. Node 1 is connected to the four other nodes.

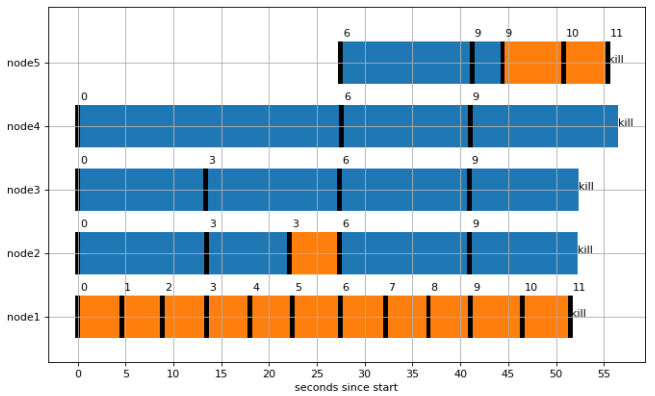


Figure 7.6: Butterfly topology

Analysis of the results In this execution, three nodes have been at one time or another leader. Node 1 was leader because it started the MapReduce and it remained leader until the end. Node 1 computed the MapReduce in twelve rounds.

Node 2 became leader because its timer timeout probably because it did not receive some `heartbeat` messages. Node 2 resumed the MapReduce at the last checkpoint (i.e. the round 3). Shortly after, the node 2 received the checkpoint of round 6 from the node 1. Node 2 became an observer again because node 1 is more advanced in the computation than itself (round 3 < round 6). Node 5 also became leader because its timer timeout probably because it did not receive some `heartbeat` messages. Node 5 resumed the MapReduce at the last checkpoint, the round 9 and finished the MapReduce. The node 5 did not become an observer again like node 2 because it has not received new checkpoints from node 1. In this execution, two nodes finished the MapReduce as leaders. We also observe that some checkpoint messages have been lost or delayed: the node 5 did not receive the first three checkpoints and node 4 did not receive the checkpoint of round 3. At the end, all the nodes got the result of the MapReduce.

Observed behavior - line We have run the synthetic test on node 1 which is a node at the edge of the network.

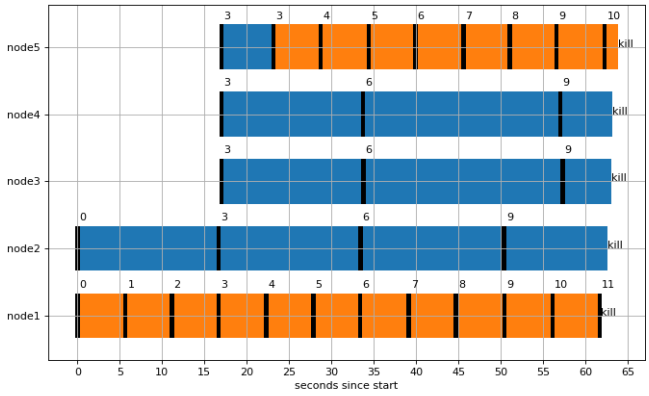


Figure 7.7: Line topology, the leader is at the beginning of the network

Analysis of the results As a reminder, in the line topology, the two nodes at the edge of the network: node 1 and node 5 are at 4 hops away.

The node 1 was leader because it started the MapReduce and it remained leader until the end. It computed the MapReduce in twelve rounds. Node 5 receives

the checkpoint of round 3 from node 1. After that, it became leader because its timer timeout probably because it did not receive some `heartbeat` messages and it resume the MapReduce at round 3. Then, node 5 no longer received any messages from node 1. And node 1 did not receive any messages from node 5. Because the nodes are far apart, messages have difficulty propagating. We also observe that nodes 3 and 4 changed their leader. They no longer observe node 1, they observe node 5. A clue on the graph is the arrival time of checkpoint 9 on nodes 3 and 4 which arrives just after the end of round 9 on node 5. There are in a way two groups in the network: {node 1 and node 2} and {node 3, node 4 and node 5}. The two groups are not subnets because they are still connected. But the leaders of both groups have difficulty to broadcast their messages to the members of the other group. But in the end, all the nodes of the network got the result of the MapReduce.

Conclusion of the "5 nodes in the network" scenario We noticed that some messages were lost or delayed, in some cases this resulted in the appearance of additional leaders. Message loss is common in IoT networks, we have to deal with it. We observe that message propagation is complicated when the nodes are at a distance of 3 hops or more. It is therefore preferable to have networks with nodes that are strongly connected to add redundancy in messages.
In all the tests, all the nodes had the result of the MapReduce.

Crash of the leader in a five nodes network

For this evaluation, we have connected five GRiSP boards to each other in different topologies. We start the synthetic test on a node that becomes the leader and we crash it during the computation of the MapReduce. The purpose of the following tests is to show that our system is resilient and it resist to the crash of the leader. We want to make sure that another node will take the lead, continue the MapReduce. At the end, we want that all the nodes, except the crashed node, got the result of the MapReduce.

Expected behavior If the leader crashes before it could broadcast the initial pairs to at least one another node, the MapReduce disappear and is never computed.

If the leader crashes after the broadcast of the initial data, another node (an observer) will timeout because it will not receive any message from the leader and in turn will become leader. The new leader will resume the MapReduce at the last checkpoint it received.

The crash of the leader could create a network partition. In this case, there will be one leader per partition and they will resume the MapReduce at the last checkpoint they received. When the MapReduce is computed the leader(s) will broadcast the result to other alive nodes of the network.

Observed behavior - full-mesh We have run the synthetic test on node 1.

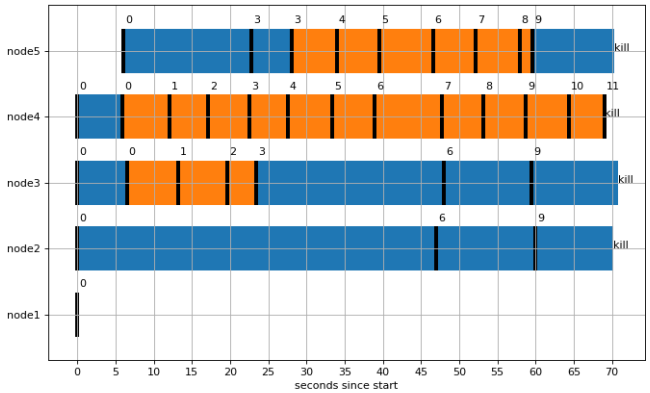


Figure 7.8: Full mesh topology, the leader crashes

Analysis of the results The leader was the node 1 because it started the MapReduce. It crashed right after broadcasting the initial pairs to other nodes.

Nodes 3 and 4 are the first two nodes to have timeout and they became leaders, they competed against each other to become the single leader and it was node 4 that won: it finished the round 3 before node 3. The new leader finished the MapReduce in twelve rounds. Node 5 received the checkpoint of round 3 from the node 4 but soon after it has timeout and became leader because its timer timeout probably because it did not receive some `heartbeat` messages and it resumed the MapReduce at checkpoint 3 and remained leader until the round 8. Then it became observer again because it received the checkpoint of round 9 from node 4 which is a more advanced leader. At the end, all the nodes got the result of the MapReduce except the node 1 that crashed.

Observed behavior - line We have run the synthetic test on node 1 which is the node at center of the network. There are nodes 2 and 3 to its left and nodes 4 and 5 to its right.

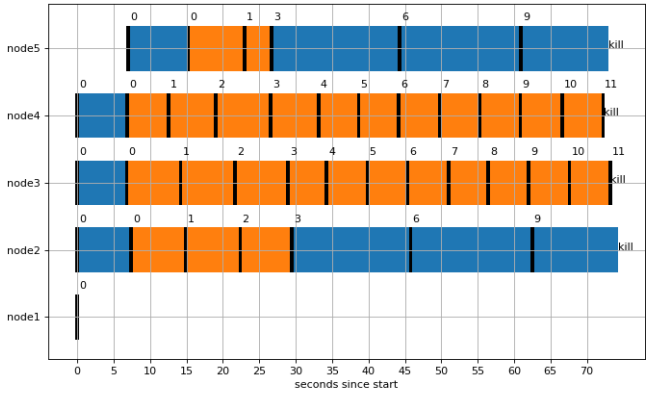


Figure 7.9: Line topology, the leader crashes creating a partition

Analysis of the results The leader was the node 1 because it started the MapReduce. It crashed right after broadcasting the initial pairs to other nodes. All the nodes received the broadcasted message. The crash created a partition in the network: nodes 2 and 3 could no longer reached nodes 4 and 5. In the two subnets the nodes resumed the MapReduce at checkpoint 0 and they competed against each other to become the leader of their subnet. There are the nodes 3 and 4 that won. They finished the execution of the MapReduce and each leader sent the result to its observer. At the end, despite the network partition, all the nodes got the result of the MapReduce except the node 1 that crashed.

Conclusion of the "Crash of the leader in a 5 nodes network" scenario As in the previous scenario, we noticed that some messages were lost or delayed and in some cases this resulted in the appearance of additional leaders.

When the timer of the observers timeout at the same time and became leaders at the same time, the leaders compete against each other until one of the leaders takes the advantage over the others. The other leader become observers again. When there are partitions in the network due to a crash, both partitions will eventually have a leader each. The two leaders will resume the MapReduce and broadcast the result into their subnet. In all the tests, all the alive nodes had the result of the MapReduce.

On the full mesh topology, we observe that the MapReduce took more time when a crash occurs (see 7.6 and 7.8). The difference in time is explained by the fact that observer should timeout before becoming a leader. Hence, there might

be idle time during the transition period. In addition to that, the new leader may resume the MapReduce from an oldest checkpoint. It means that the node may have to redo part of the completed work.

$N-1$ crashes in a five nodes network

For this evaluation, we have connected five GRiSP boards to each other in a full-mesh topology. We start the synthetic test on a node that become the leader and we crash the leader and 3 other nodes at different times. The purpose of the following test is to show that our system is resilient and it resist to $N-1$ crashes.

Expected behavior If the leader crashes before it could broadcast the initial pairs to at least one another node, the MapReduce disappear and is never computed. If the leader crashes after the broadcast of the initial data, another node (an observer) will timeout because it will not receive any message from the leader and in turn will become leader. The new leader will resume the MapReduce at the last checkpoint it received. The last remaining node will compute the result of the MapReduce. The $N-1$ crashed nodes will not receive the result.

Observed behavior We have run the synthetic test on node 1

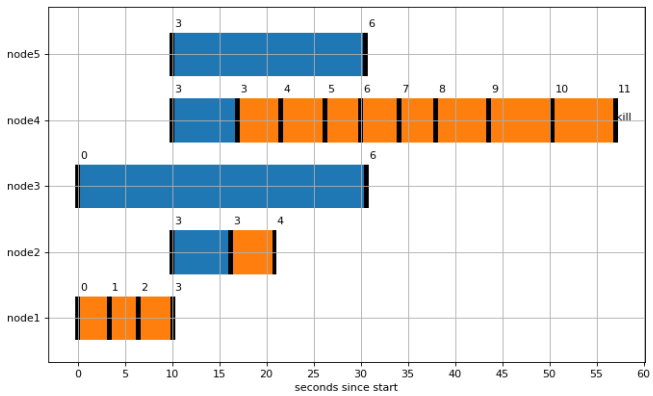


Figure 7.10: Full mesh topology, $N-1$ crashes

Analysis of the results The leader was the node 1 because it started the MapReduce. It crashed after the round 3. The timers of nodes 2 and 4 timeout and they became leaders. They resume the MapReduce at round 3 but node 2 crashed. Nodes 3 and 5 crashed too, they were observers. Node 4 was the single

leader in the network and it remained until the end because it did not crash. At the end it was the only node to have the result of the MapReduce.

7.2 Task model

In this section, we will analyze the state of the task model according to the load balancing strategy used. We will run the synthetic test describe in section 7.1 on the GRiSP boards. We will show the graphs and discuss the results.

> The task model's state is composed of a queue containing the identifiers of the scheduled tasks, a dictionary `running_tasks` containing the process id and the task id of the tasks that are running and a dictionary `forwarded_tasks` containing the forwarded task id and the node to which the task has been sent.

Single node

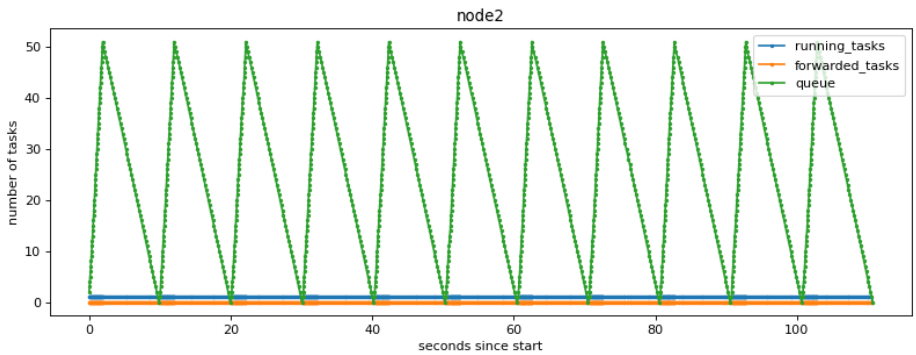


Figure 7.11: State of the task model on one node

Analysis of the result We started the test on a single node. The green line represents the number of elements in the queue is a triangle wave. Each triangle is the activity of the node during a round of the reduce phase: at the beginning of the round, the task model adds the batches one by one in the state, this is the ascending phase. The synthetic test generates 100 pairs and the maximum size of a batch is 2. So, there are $\frac{100}{2} = 50$ batches in the queue. Then, the queue empties as the tasks are computed, this corresponds to the descending phase.

The blue line is constant and represents the size of the `running_tasks`: it is always equal to 1. As explain in section 4.2, the number of concurrent process is limited to 1. As soon the result of a task is computed by its process, a new process is spawned and a new task is executed.

The orange line is constant and represents the size of the `forwarded_tasks`: it is always equal to 0 because the node did not have any neighbors, so it did not forward any tasks.

Pull strategy

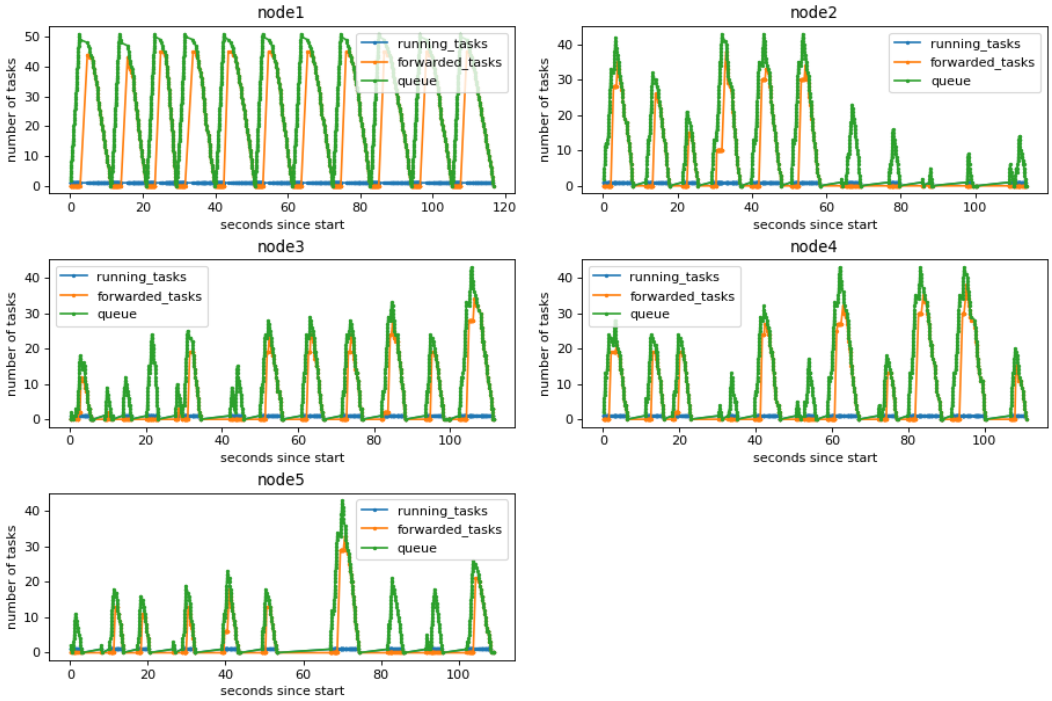


Figure 7.12: States of the task model using the pull strategy

Analysis of the results We started the test on node 1. The task model used the pull strategy with the following configuration:

- stealing threshold = 5
- maximum stolen tasks = 100.

The green line represents the number of elements in the queue is a triangle wave. Each triangle is the activity of the node during a round of the reduce phase.

The orange line represents the number of elements in `forwarding_tasks`. With the pull strategy, the neighbors stole tasks from the leader. In fact, the neighbors send a message to the leader asking to send to them some tasks. And the leader forward them tasks if the number of tasks exceeds the forwarding threshold. By looking the orange line of node 1, we observe that the sender forwarded more than 40 tasks per round to its neighbors. The observers stole tasks to the leader but also to other observers. For example, the node 2 received about 40 tasks (green line) but it forwarded about 30 tasks to other nodes.

From a general point of view, the workload is well distributed among all the nodes. Considering all the rounds, the four observers had on average the same number of tasks in their queue.

Push strategy

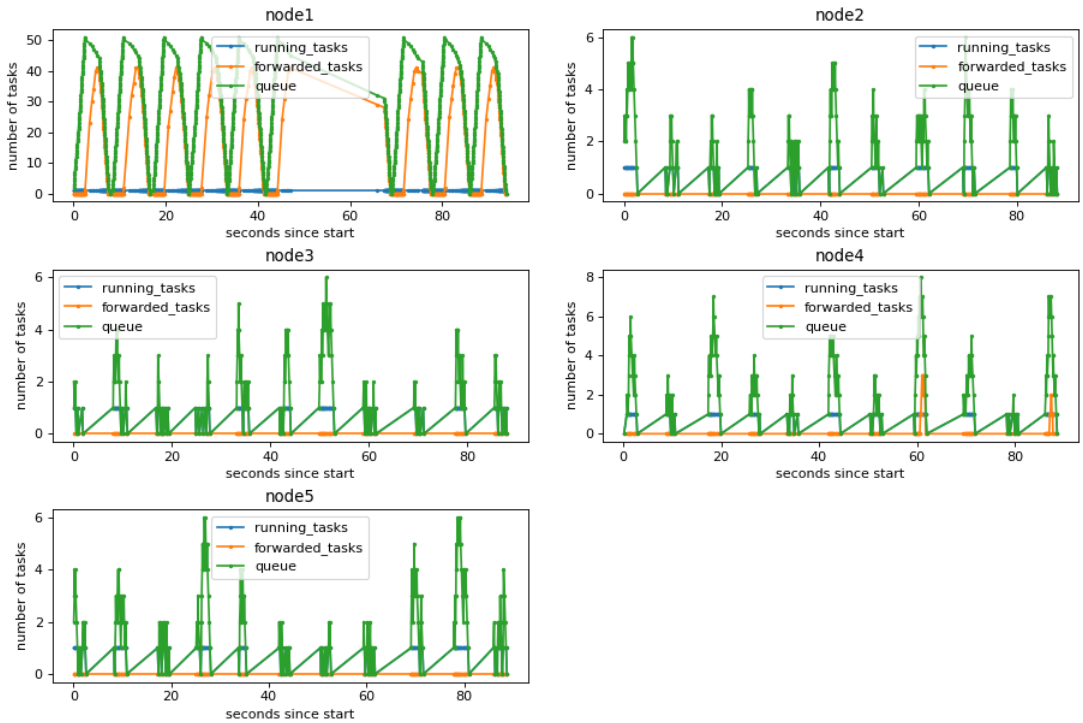


Figure 7.13: States of the task model using the push strategy

Analysis of the results In the push strategy, it is the leader that forwards tasks to its neighbors. As soon as the number of tasks exceeds the forwarding threshold, the leader forwards the $N - threshold$ tasks one by one to its neighbors.

We observe that the graph of node 1 does not have the expected shape between 40 and 70 seconds. There should be eleven picks corresponding to rounds 1 to 11. Like we had in the previous two tests. We have run the test several times and we always got the same graph. We assume that at this point in time the node was overloaded and it did not write all the logs to the file.

By looking the orange line of node 1, we observe that the leader forwards about 40 tasks. If we add up the number of tasks in the queue of the other four nodes, we get $6 + 2 + 8 + 4 = 20$ tasks. This can be justified by the fact that the leader sent a `kill` message to remove a task from the queue of its neighbor.

Compared to the pull strategy, the workload distribution is not as good, the observers had no more than six tasks per round. Considering all the rounds, the four observers had on average the same number of tasks in their queue.

7.2.1 Benchmark

To measure the efficiency of the task model, we will test it independently of the MapReduce. In the new test, a node will schedule 100 tasks and measure the time needed to complete them. Each task will contain a sleep instruction that will block the execution thread of the task during a given period. This instruction aims to simulate a heavy workload. In the following tests, we will vary the execution delay of the task and we will see how the system behaves. Then, we will compare the results in different configurations.

In the first configuration, the node that schedules the tasks will have no neighboring nodes. The node will hence execute all tasks on its own. In the second configuration, the node that schedules the tasks will have 4 neighboring nodes and each node will be connected to every other node. In that configuration, the node will be able to send some task to the other nodes. Here, we expect that the node that scheduled the tasks get their results more quickly when it has neighboring nodes.

One node

Delay	Average	Standard deviation	Minimum	Maximum
10ms	4971.05	47	4912	5083
20ms	5858	151.75	5676	6220
50ms	8772.15	112.14	8584	9042
100ms	13465.9	86.78	13300	13590

Analysis of the results We observe that the average execution time increases as the delay increases.

Five nodes with pull strategy

Delay	Average	Standard deviation	Minimum	Maximum
10ms	7202.9	350.78	6523	8280
20ms	7476.75	513.84	6745	8837
50ms	7632.9	432.51	6708	8223
100ms	8203.9	657.49	7202	9936

Analysis of the results If the tasks take 10ms and 20ms to complete, the node should rather execute the tasks locally instead of forwarding tasks to another node. If the tasks take more than 50ms to complete, the task model becomes interesting. For 50ms, we observe an average speed up of 12 percent. For 100ms, we observe an average speed up of 39 percent.

Five nodes with push strategy

Delay	Average	Standard deviation	Minimum	Maximum
10ms	12581.65	307.19	12099	13190
20ms	12676.40	223.39	12327	13794
50ms	12777.75	172.42	12428	13044
100ms	12820.95	288.56	12156	13387

Analysis of the results If the tasks take between 10ms and 50ms to complete, the node should rather execute the tasks locally instead of forwarding the task to another node. From 100ms, we observe an average speed up of 4.8% percent compared to the average execution time on a single node.

We observe that the push strategy is slower than the pull strategy. The explanation is that with the push strategy, the leader forwards the tasks one by one (cf. section 5.2.1). Whereas with the pull strategy, the leader can forward several

tasks at the same time (cf. section 5.2.1). The leader therefore has more outgoing messages to manage which may explain his longer execution time.

The node may have completed the task before the remote node receives the task, executes it, and returns its result. As a result, the remote nodes may perform redundant work which will not speed up the MapReduce.

To resume, our experimentation shows that the push strategy is less efficient than the pull strategy. In both strategies, the task model becomes relevant only if the tasks take time to be completed. The programmer that uses our task model will have to keep that in mind. For the MapReduce, it is therefore recommended to form large batches. That way, the `reduce` function will take time to complete and the task model will speed up the execution of the MapReduce. With larger batch, the node will also send less messages.

Chapter 8

Limits

In this project, we implemented a MapReduce algorithm designed to work in a dynamic environment where the nodes can leave and join the network at any time and the communication between the nodes is not reliable. In this section, we will talk about the limits of the system.

8.1 Hardware

To test our implementation, we implemented several examples. In one of them, we implemented a hello-world example which consist of counting for each word the number of times it appears in a given text. The example will be provided in the appendix. With 5 nodes, we achieved to process a text of 1000 words. When we tried bigger example, the board crashes and displays the following error message:

```
eheap_alloc: Cannot allocate 1272748 bytes of memory (of type "heap").
Crash dump is being written to: erl_crash.dump...done
fatal extension: source=5, is_internal=0, error=1
```

This error message means that the board does not achieve to allocate a memory block of the given size. We have three different hypothesis: (1) the memory is completely full, (2) there is enough memory but the free memory slots are fragmented and the board is not able to fragment the given block to fill the gaps, (3) there is a hard limit on the maximal amount of memory that the board can allocate at once.

We think that our algorithm may have a large memory footprint. In our implementation, the leader will send regularly checkpoints and heartbeat messages. If the nodes do not process the messages fast enough, the node may retain several checkpoints in memory which can take up a lot of space. Unfortunately, broadcasting the checkpoints is the only way to have redundancy. It is the price to pay to

have a resilient algorithm.

8.2 Software

Sending or broadcasting a message is not instantaneous. It is possible that the leader crashes just before broadcasting the first checkpoint. In that case, no other node will be able to resume the MapReduce as none of them will have a checkpoint. They may not even notice that a MapReduce has been scheduled on the faulty node.

On the benchmarks, we have seen that the MapReduce is not necessarily quicker when we are connecting more nodes. The task model will speed up the MapReduce only if the reduction tasks are computationally intensive. Due to the memory limitation that we have encountered, we cannot form larger batches to increase the computation time of the `reduce` functions. In these conditions, the task model might not be relevant.

Chapter 9

Future work

During the development of Lynkia, we had many improvement ideas that we did not implement due to time constraint. Some of these ideas aims to improve the efficiency of the system. In this section, we will explain the most relevant ideas that could be implemented later.

9.1 Possible improvements

We notice that, most of the time, the `reduce` function will produce less pairs. As a result, the number of key-value pairs tends to decrease with each round. To put less pressure on the leader, we could nest the MapReduce recursively: the leader could split the input pairs in groups and schedule one MapReduce per group. Then, the leader can let the other nodes becoming the leader of the sub-MapReduce and collect the key-value pairs produced by each sub-MapReduce. Once the leader gets all the pairs, the leader can pass to the next round.

We have seen that each leader will produce tasks. When a leader is killed, the leader will not ask the other nodes to delete the tasks its scheduled. For the task model, it would be great to be able to give a group id to a task and to be able to delete all tasks associated to a given group id. By setting a group id for each round, the leader will be able to ask the other nodes to delete all tasks it scheduled for a given round by sending a single message.

We have seen that the system can work even if the messages are omitted, delayed, reorder and duplicated. There may exist other libraries than Partisan. One possible improvement would be to make Lynkia network agnostic. That way, the programmer could use the library of its choice.

We wish we could test our implementation on a large network composed of hundreds of nodes. Unfortunately, we do not have enough GRiSP boards to do it. One possibility would be to simulate the boards on our laptops. With this approach, we achieved to simulate around 20 boards. To emulate more boards, we planned to use the cloud for experiment purposes.

In the tests, we shown that the system was able to work even when there is a network partition. Here, we could add more tests with other topologies. We could also try to compare the efficiency of the MapReduce with different parameters. We could, for instance, compare the computation time of the MapReduce for different batch sizes.

Chapter 10

Conclusion

In this project, we have succeeded in developing a MapReduce algorithm that is highly resilient and can run on the extreme edge of the network. We also developed a generic and modular task model that allow the nodes to balance their workload with the other nodes of the network. The task model can be used as a building block of other distributed algorithms.

In the evaluation, we tried to break the system by introducing faults manually. It turns out that the MapReduce is resilient and tolerates up to $n-1$ failures. The results that we obtained are rather encouraging for the future: Here, we do not achieve to process a large amount of data because of the technical limitations of the GRiSP boards. However, the technical limitations of today will not be those of tomorrow. Thanks to the new technological advances, the devices of tomorrow will have more computing power and more memory. Therefore, they will be able to process more and more data.

As evidence, the next generation of GRiSP board that will be released soon will have twice as much memory as the GRiSP boards we used. The board will pass from 64MB to 128MB of RAM. The board will also have a more powerful CPU. We expect that the new boards will be able to process more data.

In the future, we think that the IoT devices will be as powerful as the computers we are using today. They will also be way more affordable and hence more numerous. The IoT devices will have enough computing power and storage to process a large amount of data. The devices will be autonomous and will no longer depends on a cloud infrastructure.

The good news is that the Lynkia code base is future proof: We have seen that we can easily adjust some parameters to take full advantage of the hardware of the

new devices. As example, we could increase the number of tasks that the nodes can be executed concurrently to use all cores of the CPU.

In this project, we have been confronted with new problematic: we noticed that it is difficult to design distributed systems. Here, we had to deal with their limits and their constraints. By playing with them, we have acquired better expertise in the design of distributed systems. We also learned to use Erlang, a rich programming language having many programming paradigms. For the first time, we took part in a major project. We learned to organized despite the health crisis.

We hope that this project may guide further scientific research. We also hope that this project has made you want to learn more about distributed systems and IoT. In the future, we may continue maintaining Lynkia to fix bugs and add new features. As the project is open source, the code base is open to everyone and you will be welcome to contribute to our project.

Bibliography

[1] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16, 2012.

[2] O. Consortium et al. Openfog reference architecture for fog computing. *Architecture Working Group*:1–162, 2017.

[3] V. Enes, P. S. Almeida, C. Baquero, and J. Leitão. Efficient synchronization of state-based crdts. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 148–159. IEEE, 2019.

[4] D. Evans. How the next evolution of the internet is changing everything. In 2011.

[5] I. Kopestenski and P. Roy. Erlang as an enabling technology for resilient general-purpose applications on edge, iot networks, Aug. 2019.

[6] I. Kopestenski and P. Van Roy. Achlys: towards a framework for distributed storage and generic computing applications for wireless iot edge networks with lasp on grisp. In *2019 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 875–881. IEEE, 2019.

[7] C. Meiklejohn. Applied Monotonicity: A Brief History of CRDTs in Riak. http://christophermeiklejohn.com/erlang/lasp/2019/03/08/monotonicity.html, 2019. [Online; accessed 06-08-2020].

[8] C. Meiklejohn. Partisan: testable, high performance, large scale distributed Erlang. https://codesync.global/media/partisan-testable-high-performance-large-scale-distributed-erlang, 2019.

[9] C. Meiklejohn. Using Fault-Injection to Evolve a Reliable Broadcast Protocol. http://christophermeiklejohn.com/erlang/partisan/2019/04/20/fault-injection-reliable-broadcast.html, 2019. [Online; accessed 06-08-2020].

[10] C. Meiklejohn and H. Miller. Partisan: enabling cloud-scale erlang applications. *arXiv preprint arXiv:1802.02652*, 2018.

[11] C. Meiklejohn and P. 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, pages 184–195, Siena, Italy. Association for Computing Machinery, 2015. ISBN: 9781450335164. DOI: 10.1145/2790449.2790525. URL: https://doi.org/10.1145/2790449.2790525.

[12] C. Meiklejohn and P. Van Roy. The implementation and use of a generic dataflow behaviour in erlang. In *Proceedings of the 14th ACM SIGPLAN Workshop on Erlang*, pages 39–45, 2015.

[13] A. Shoker, J. Leitao, P. Van Roy, and C. Meiklejohn. Lightkone: towards general purpose computations on the edge. *White Paper published on http://www.lightkone.eu*, 40, 2016.

Part IV

Appendix

Appendix A

Lynkia

For this project, we created a new Github organization around the Lynkia project. The source code of the project is available on this repository and is provided under the Apache-2.0 License.

A.1 Getting started

Step 1: Install Erlang
Step 2: Follow the GRiSP guide to install the dependencies.
Step 3: Clone the Lynkia repository

```
git clone https://github.com/lynkia/lynkia.git
```

A.1.1 Test Lynkia on your computer

Step 1: Start the nodes.
Open two different terminals.
On the first one, type:

```
rebar3 shell --name lynkia1@127.0.0.1 --apps lynkia
```

On the second one, type:

```
rebar3 shell --name lynkia2@127.0.0.1 --apps lynkia
```

These commands start two instances of Lynkia.
Step 2: Connect the nodes
You can either:

- Enter `lynkia_utils:join('lynkia1@127.0.0.1').` on lynkia2

- Enter `lynkia_utils:join('lynkia2@127.0.0.1').` on lynkia1

Step 3: Verify that your two nodes are connected

```
lynkia_utils:members().
```

Step 4: Run a MapReduce test

```
test:synthetic_test().
```

A.1.2 Test Lynkia on GRiSP board

Step 1: Modify `rebar.config` with respect to your OS:
MacOS:

```
{deploy, [
% ==========
% Mac OS X :
% ==========
    {pre_script, "rm -rf /Volumes/GRISP/*"},
    {destination, "/Volumes/GRISP"},
    {post_script, "diskutil unmount /Volumes/GRISP"}
]}
```

Linux:

```
{deploy, [
% ==========
% Linux :
% ==========
    {pre_script, "rm -rf /media/user/GRISP/*"},
    {destination, "/media/user/GRISP"},
    {post_script, "umount /media/user/GRISP"}
]}
```

Step 2: Modify `grisp/grisp_base/files/erl_inetrc` by adding the IP addresses of your GRiSP boards. (More details in GRiSP guide)
Step 3: Deploy Lynkia on boards:
Insert the micro SD card of the the GRiSP board in your computer and deploy Lynkia on the card:

```
NAME=my_grisp_board_1 IP=169.254.16.1 rebar3 grisp deploy -n lynkia -v 0.1.0
```

The `IP` must correspond to a IP addresses you added in `erl_inetrc`. When the deployment is complete, remove the micro SD card from the computer and insert it in your GRiSP board and boot it. You can of course deploy Lynkia on other nodes by modifying the `NAME` and `IP`.
Step 4: Connect your GRiSP boards to your computer via the serial port:

1. The GRiSP board must be connected to your computer via USB
2. Type `ls /dev/*usb*` a list of usb device is displayed:

   ```
   /dev/cu.usbserial-001030  /dev/tty.usbserial-001030
   /dev/cu.usbserial-001031  /dev/tty.usbserial-001031
   ```

   We want the TTY devices ending with a 1: `/dev/tty.usbserial-001031`
3. Use `screen` to connect your board: `screen /dev/tty.usbserial-001031 115200`

> There is a bug when we use the serial port to connect the GRiSP: the typed text does not appear in the Erlang shell until you press enter. It is recommended to type the command in a text editor and copy/paste the command into the terminal.
>
> You can also connect your computer to the GRiSP boards via Wi-Fi, a tutorial is available on GRiSP guide

Step 5: Connect the nodes

```
lynkia_utils:join('lynkia@my_grisp_board_X').
```

Step 6: Verify that your two nodes are connected

```
lynkia_utils:members().
```

Step 7: Run a MapReduce test:

```
test:synthetic_test().
```

A.2 Modules

- `lynkia.erl`: The module will expose the major functions of the project.
- `lynkia_config.erl`: The module will be used to retrieve some constants defined as environment variables. These variables can be changed by updating the configuration file.
- `lynkia_mapreduce.erl`: The module will be responsible of each MapReduce process. The module will receive the messages sent via Partisan and forward them to the right process.
- `lynkia_mapreduce_dispatcher.erl`: The module will have the responsibility to split a batch in many smaller batches and to create one task per batch. Then, the module will block the execution thread until all the scheduled functions return a result. The module will also be able to split a batch in half if the tasks has been killed.
- `lynkia_mapreduce_leader.erl`: This module will contain the logic of the leader. The module will spawn a process to send heartbeat messages to all the other nodes.
- `lynkia_mapreduce_observer.erl`: The module will contain the logic of the observer. The module will start a timeout to become a leader.
- `lynkia_spawn.erl`: The module will contain the logic of the task model.
- `lynkia_utils.erl`: The module will contain helper functions.

Adapters:

- `lynkia_mapreduce_adapter_csv.erl`: This adapter will be used to load data from a CSV file located on the disk of the node.
- `lynkia_mapreduce_adapter_file.erl`: This adapter will be used to load data from a file.
- `lynkia_mapreduce_adapter_lasp.erl`: This adapter will be used to load data from a CRDT variable.

Strategies:

- `lynkia_pull_strategy.erl`: This module will contain the logic of the pull strategy.
- `lynkia_push_strategy.erl`: This module will contain the logic of the push strategy.

A.3 Variables

Broadcast:

- `fanout`: Fanout of the broadcast.
- `broadcast_ets_gc_interval`: Delay between each execution of the garbage collector.
- `broadcast_ets_ttl`: Time-to-live of the broadcasted messages.

MapReduce:

- `mapreduce_broadcast_interval`: The leader will broadcast a checkpoint every x round.
- `mapreduce_gc_interval`: Delay between each execution of the garbage collector.
- `mapreduce_ttl`: Time-to-live of the cached result.
- `mapreduce_max_round`: Maximum number of round.
- `mapreduce_max_batch_size`: Maximum batch size.
- `mapreduce_round_timeout`:
- `mapreduce_observer_min_timeout`: Minimum delay before timeout
- `mapreduce_observer_max_timeout`: Maximum delay before timeout
- `mapreduce_leader_heartbeat_delay`: Delay between each heartbeat message

Task model:

- `task_distribution_strategy`: Name of the module that will be responsible of the load balancing.
- `pull_stealing_threshold`
- `pull_steal_interval`: Delay between each steal.
- `pull_number_of_tasks_to_steal`: Total number of task that the pull strategy can steal.
- `forwarding_threshold`
- `task_workers`: Number of tasks that can be run concurrently.
- `task_max_computation_time`: Maximum computation time of a task.
- `task_max_memory_usage`: Maximum memory usage of a task

A.4 API

- `lynkia:spawn/2`: The function will allow the programmer to schedule a function on our task model. The function will take as parameters the function to execute and a list containing all arguments that will be passed to the given function. Here, the function call is synchronous.

- `lynkia:spawn/3`: The function will allow the programmer to schedule a function on our task model. The function will take as parameters a function, a list containing all arguments that will be passed to the given function and a callback function that will be called once the task has been completed. Here, the function call is asynchronous.

- `lynkia:mapreduce/3`: The function will allow the programmer to schedule a new MapReduce. The function will take as parameters an adapter, a reduce function and a callback function.

- `lynkia:mapreduce/4`: The function will allow the programmer to schedule a new MapReduce. The function will take as parameters an adapter, a reduce function, some options and a callback function. With the `options` variable, the programmer can change the maximum number of rounds, the maximal batch size and the timeout of each reduction.

A.5 Examples

A.5.1 Word counting

The word counting is a popular example for MapReduce. With this example, we can count the number of occurrences of each word in a text.

```
Path = "file",
Adapters = [
    {lynkia_mapreduce_adapter_file, [
        {Path, fun(Lines) ->
            lists:flatmap(fun(Line) ->
                Separator = " ",
                Words = string:tokens(Line, Separator),
                lists:map(fun(Word) -> {Word, 1} end, Words)
            end, Lines)
        end}
    ]}
],

Reduce = fun(Key, Values) ->
    [{Key, lists:sum(Values)}]
end,

Options = #options{
    max_round = 100,
    max_batch_size = 100,
    timeout = 3000
},

lynkia:mapreduce(Adapters, Reduce, fun(Result) ->
    case Result of
        {ok, Pairs} ->
            io:format("Pairs = ~p~n", [Pairs]);
        {error, Reason} ->
            io:format("Error: Reason = ~p~n", [Pairs])
    end
end).
```

A.5.2 Synthetic Test

```
Max = 100,
Var = {<<"var">>, state_gset},
lasp:bind(Var, {state_gset, lists:seq(0, Max)}),
lasp:read(Var, {cardinality, Max}),

Adapters = [
    {lynkia_mapreduce_adapter_lasp, [
        {Var, fun(Value) ->
            [{0, Value}]
        end}
    ]}
],

Reduce = fun(Key, Values) ->
    timer:sleep(100),
    case Key < 10 of % Number of rounds = 10
        true ->
            lists:map(fun(Value) ->
                {Key + 1, Value}
            end, Values);
        false ->
            lists:map(fun(Value) ->
                {Key, Value}
            end, Values)
    end
end,

Options = #options{
    max_round = 100,
    max_batch_size = 2,
    timeout = 5000
},

lynkia:mapreduce(Adapters, Reduce, Options, fun(Result) ->
    io:format("Result=~p~n", [Result])
end).
```

Appendix B

Configuration of Lynkia

```
{lynkia, [
    %% Broadcast
    {fanout, 2},
    {broadcast_ets_gc_interval, 3000},
    {broadcast_ets_ttl, 4000},

    %% MapReduce
    {mapreduce_broadcast_interval, 3},
    {mapreduce_gc_interval, 3000},
    {mapreduce_ttl, 4000},
    {mapreduce_max_round, 10},
    {mapreduce_max_batch_size, 10},
    {mapreduce_round_timeout, 3000},
    {mapreduce_observer_min_timeout, 5000},
    {mapreduce_observer_max_timeout, 10000},
    {mapreduce_leader_heartbeat_delay, 1000},

    %% Task model
    {task_distribution_strategy, lynkia_push_strategy},
    {pull_stealing_threshold, 5},
    {pull_steal_interval, 1000},
    {pull_number_of_tasks_to_steal, 100},
    {forwarding_threshold, 5},
    {task_workers, 1},
    {task_max_computation_time, 5000},
    {task_max_memory_usage, 0}
]}
```

Appendix C

Gallery

Figure C.1: GRiSP boards network

Appendix D

FAQ

Where does the name Lynkia come from ?

Lynkia refers to the blue linckia which is a starfish having regenerative capabilities: When the starfish is bitten by a predator, the starfish is able to regenerate itself completely from one of its arms. Lynkia is a library that is highly fault tolerant. The library mimics this ability: when there is only one node in the network that have a checkpoint of a MapReduce, the node will sooner or later propagate it to the other nodes. Eventually, every other node will have a copy of the checkpoint. The system will then again be able to tolerate further failures.

Where is the source code of Lynkia ?

The source code of Lynkia will be host on github.

Is it possible to contribute to Lynkia ?

Yes, you can. The source code of Lynkia is open-source and provided under the Apache-2.0 License. If you encounter a problem, do not hesitate to open a new issue. The pull requests are also greatly appreciated.

Appendix E

Other approaches

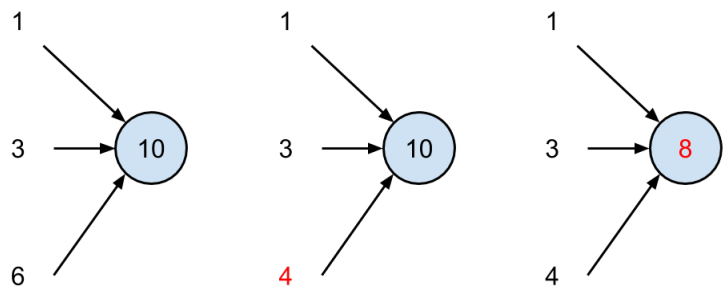
In this project, we explored other approaches to perform large-scale computation on the GRiSP boards. Among them, we can cite the dataflow programming approach (1), the stream computation approach (2) and the stochastic approach (3). This appendix should be considered as a bonus.

E.1 Dataflow

E.1.1 Concept

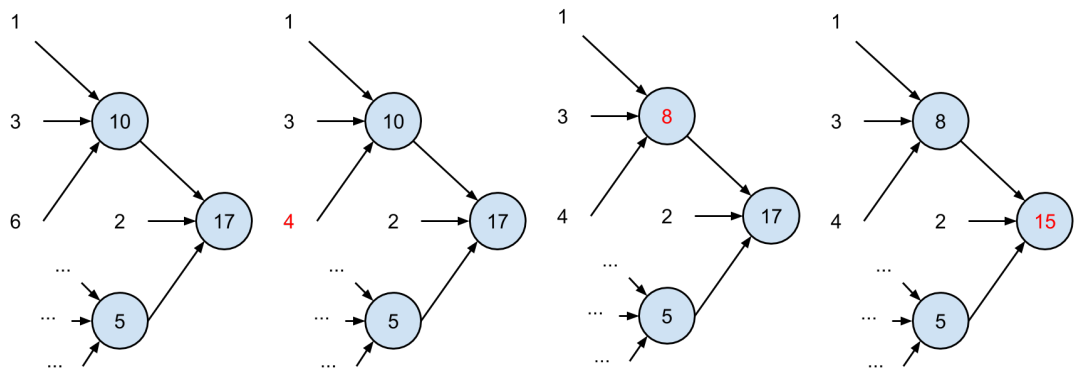
A dataflow program can be represented with a directed acyclic graph (DAG) where each node represents a processing element and each link represents a dependency. In this model, each processing element maintains a value that results of a computation involving other values.

The processing elements are designed to be reactive: When an input of a processing element has changed, the processing element will have to update its value to reflect that change. This value is expected to be eventually equal to the result obtained by applying the function calculated by the processing element over its inputs.



In the figure above, we have a processing element taking 3 values as input. In this example, the processing element will compute the sum of all its input values. When the value 6 is updated, the processing element will dynamically update its value to reflect that change.

A processing element can also take as input the value maintained by another processing element. By combining the processing elements like that, we can build more complex dataflow programs. The changes will then propagate as follows. When an input value of a processing element is updated, the processing element will update its value to reflect that change. When the value of the processing element is updated, another processing elements using this value will also have to be updated.



In the figure above, we can see that some intermediate values have been reused. It is not always necessary to re-evaluate the values of every node when a change occurs. The number of nodes that need to be updated depends on the structure of the graph. In the graphical representation, the direction of the links indicates how the values are propagated through the graph when a change occurs. If the graph has a cycle, the graph may never reach a stable state since a single change may trigger an infinite chain reaction.

E.1.2 Gen Flow

The `gen_flow` abstraction can be used for building dataflow computations. To model a processing element, the programmer will have to create a new module with the `gen_flow` behavior. With this behavior, the module will have to specify three functions: The `init` function, the read function and the process function.

- `init/1`: The `init` function will be called to initialize a local state. This state can be used to pass data to the process functions.
- `read/1`: This function will be called to retrieve a list of read functions. The result of each reach function will correspond to an input of the processing element.
- `process/2`: This function will be used to perform a computation whenever one of the read functions returns a result.

Each processing element will maintain a cache to store the last result of each read function. Initially, the cache of each read function will be set to a bottom value. Here, the bottom value will be represented with an atom `undefined`.

The `init` function of the `gen_flow` process will call the `init` function of the module having the `gen_flow` behavior in order to initialize the local state of the module. Then, the main thread of the `gen_flow` process will perform several iterations. At each iteration, the main thread of the `gen_flow` process will spawn one thread per read function and wait for a response from one of them. Each thread will have the responsibility to execute a read function and send its result to the main thread.

When the main thread receives a result, the main thread will be unblocked and will set the cache value of the given read function to the given result. The main thread will process a single message. It means that the main thread will update the cache value of the read function that returns a result first. The other cache values will not be updated even if the other read functions return a result.

Once the cache has been updated, the main thread will call the process function defined in the module of the programmer with a copy of the values present in the cache. This function will have to return a tuple containing a flag and the next state of the module. The flag will be used to indicate whether the action has been processed. Then, a new iteration will be performed.

To introduce laziness, each read function will have to do a synchronous blocking call: The function should return a result only if the current input value is newer

than the value that has been used to evaluate the value of the processing element (i.e: the cache value associated to the given read function).

If several read functions return a result within the same iteration, the main thread will only update the cache value of the read function that return a result first. The cache value of the other read functions will be updated in a future iteration: When the cache value of a given read function is outdated, the read function returns its new input value immediately. After several iterations, the cache will eventually be up to date.

At the beginning of each new iteration, the main thread of the `gen_flow` module will first kill all functions that have been spawned at the previous iteration and clear all exit messages from the inbox. This ensures that the main thread will not receive a message from a previous iteration.

E.1.3 Lasp process

The `lasp_process` module implements the `gen_flow` behavior. This module has been built to work specifically with CRDT variables.

The function `start_dag_link` of the `lasp_process` module can be used to create a DAG link. The function will take three arguments: A list of read functions, a transformation and a write function.

1. The read functions will have the same responsibility as described in the `gen_flow` section. Each read function will deal with a CRDT variable. The programmer can use the read function of Lasp to detect changes that may occur on a CRDT variable.

2. The transformation function will be called when one of the input variable has changed. The transformation function will have the responsibility to transform the outputs of the read functions.

3. The write function takes the results returned by the transformation function and make something out of this result. The programmer can bind this result to a CRDT variable by using the bind function of Lasp. To do that, the transformation function should return a valid state with respect to the considered output variable.

The local state used by the `lasp_process` will contain the transformation function and the write function. When one input variable of the processing element is updated, the `gen_flow` process will call the process function of `lasp_process`.

The process function will call the transform and the write function if all read functions have been triggered at least once.

The read function of Lasp will perform a monotonic read on a CRDT variable by using the value that has been cached by the `gen_flow` process. If a most recent value is observed, the read function returns a result immediately. Otherwise, the function call block until the variable is updated. This technique ensures that all changes will eventually be captured and that the computation will converge deterministically to the correct result. The `gen_flow` process might perform several iterations before reaching a stable state.

The variables used as input and output will have to be monotonic. Among the different available variables, we can use:

- The grow-only set
- The or-set
- The awset with provenance semiring
- The grow-only counter
- The pn-counter
- The two-phase set

```
IVar = {<<"ivar">>, state_orset},
OVar = {<<"ovar">>, state_orset},

lasp:declare(IVar, state_orset),
lasp:declare(OVar, state_orset),

lasp_process:start_dag_link([
    [{IVar, fun(ID, Threshold) -> % Read function
        lasp:read(ID, Threshold) % Synchronous read
    end}],
    fun(Tuple) -> % Transformation function
        case Tuple of {ID, Type, Metadata, Values} ->
            Values
        end
    end,
    {OVar, fun(ID, Values) -> % Write function
        lasp:bind(ID, Values)
```

```
    end}
]).
```

E.1.4 Lasp functions

Functional programming primitives

The following functions take as input another function. To guarantee a deterministic result, the function given as argument will have to be deterministic: It means that the function should always return the same output for the same input.

Map: The algorithm will apply the given function over each value of the input variable. The output variable will be bound to the obtained result. The output variable is expected to have the same cardinality as the input variable.

```
Fun = fun(K) -> K + 1 end,
lasp:map(IVar, Fun, OVar).
```

Filter: The algorithm will iterate over each value of the input variable. The algorithm will call the given predicate function over the current value: If the predicate function returns true, the current value will be added to the output variable.

```
Fun = fun(Value) -> Value rem 2 == 0 end,
lasp:filter(IVar, Fun, OVar).
```

E.1.5 Set-theoretic functions

Product: The algorithm will compute the cartesian product of two sets: A and B. The variable C will be bound to the obtained result.

```
lasp:product(A, B, C).
```

Union: The algorithm will compute the union of two sets: A and B . The variable C will be bound to the obtained result.

```
lasp:union(A, B, C).
```

Intersection: The algorithm will compute the intersection of two sets: A and B. The variable C will be bound to the obtained result.

```
lasp:intersection(A, B, C).
```

E.1.6 Achlys process

To maintain the symmetry in case of failures, all the nodes must perform the dataflow computation. If the input and output variable are monotonic, a redundant execution will not produce any side effect on the obtained result. The input variables and the output variable will eventually converge to the same state.

To ensure that all nodes perform the dataflow computation, the programmer can use the task model of Achlys. If a node crash, the node can pull the tasks from the other nodes and execute them. The node will hence be able to recover and maintain the symmetry.

We propose a new abstraction called `achlys_process` based on the `gen_flow` behavior and the task model of Achlys. With this module, the programmer will have to specify a list of input variables, an output variable and a transformation function. The programmer does not have to specify the read and the write function.

The transformation function will take one argument: A list of values where the ith value corresponds to the cache value of the ith input variable. With this representation, we can conveniently generate an output variable by using the `foldl` function of the module `list`. It is also possible to group some results by using the `partition` function of the module `list`.

The transformation function should return a valid state with respect to the considered output variable. As for `lasp_process`, the programmer must know how the state of the output variable is represented internally by Lasp to produce a valid state.

Example:

```
Type = state_gset,
A = {<<"a">>, Type},
B = {<<"b">>, Type},
C = {<<"c">>, Type},

lasp:declare(A, Type),
lasp:declare(B, Type),
lasp:declare(C, Type),

lasp:stream(C, fun(Value) ->
    io:format("C=~w~n", [sets:to_list(Value)])
end),

achlys_process:start_dag_link(
    [A, B], C,
    fun(Results) ->
        case Results of [GSet1, GSet2] ->
            state_gset:merge(GSet1, GSet2).
        end
    end
),

timer:sleep(1000),
lasp:update(A, {add, 1}, self()),
lasp:update(B, {add, 2}, self()),
```

E.2 Stream computation

E.2.1 Eager stream

In Lasp, the `stream` function can be used to trigger a function whenever an input variable has changed. Internally, the function will create a new Lasp process. When the input variable is updated, the function will query the input variable and pass the result to the anonymous function provided by the programmer.

```
ID = {<<"set">>, state_orset},
lasp:declare(ID,  state_orset),
lasp:stream(ID, fun(V) -> V end).
```

To perform an incremental computation, we can start from an initial state. Then, we can transform that state every time the input variable is updated. The transformation applied to the state may depend on the changes that has been applied to the input variable.

To determine what has been changed, we need to cache the variable state passed to the anonymous function. When the input variable is updated, we can compare the given variable state with the variable state that have been cached at the last update. Then, we can infer all transformations that have been applied to the input variable since the last update. Finally, we can update the state based on these transformations and update the cache.

By inspecting the source code of Lasp, we noticed that Lasp already cache the input variable state to detect the variable changes. Unfortunately, it is not possible to access that cache from another module. To avoid having two copies of the variable state, we decided to create a new module called `achlys_stream_reducer`.

Let's imagine that we have a grow-only set containing numbers and that we want to compute incrementally the sum of all these numbers. To compute the sum, we can write the following piece of code:

```
ID = {<<"var">>, state_gset},
lasp:declare(ID, state_gset),
InitialState = 0,
{ok, Pid} = achlys_stream_reducer:start([
    {ID, fun(State, Operation) ->
        case Operation of
            {add, N} -> State + N;
            _ -> State
        end
    end}
], InitialState, fun(State) ->
    io:format("State = ~p~n", [State])
end).
```

As you can see, the programmer will have to specify an initial state and some anonymous functions. These functions will have the responsibility to transform the provided state based on the provided operation (see the arguments). With the

given implementation, it is possible to use several input variables. When an input variable is updated, the module will transform the state by calling every function associated to the updated input variable. On each one of the input variables is update, the module will call the provided callback function.

As we have seen earlier, the messages can be reordered. It means that the nodes can see the updates in a different order. If the computation is not commutative and associative, the nodes may obtain a different result after having applied all transformations.

The module works as follow: At each iteration, the module spawns one function per input variable. Each function will call the read function of Lasp. As a reminder, the read function will return the ID, the type, the metadata and the variable state of the given variable if a more recent state than the provided one is observed. Here, the read function will be called with the variable state that have been cached. When an input variable is updated, the read function returns the new variable state and send it to the main thread.

After having spawned the functions, the main thread will wait that one of the functions send its new variable state. When the main thread receives a variable state, the main thread will kill all spawned functions and update the cache and update the state from all transformations that have been inferred. Then, the main thread will start a new iteration.

E.2.2 Lazy stream

With the `achlys_stream_reducer` module, a new iteration is started each time a variable is updated. In some cases, it is not always necessary to keep the state up to date. Actually, we can introduce laziness by updating the state only when it is requested by the programmer. This is precisely what the module `achlys_lazy_stream_reducer` is doing.

```
ID = {<<"var">>, state_gset},
lasp:declare(ID, state_gset),
InitialState = 0,
{ok, Pid} = achlys_lazy_stream_reducer:start([
    {ID, fun(State, Operation) ->
        case Operation of
            {add, N} -> State + N;
            _ -> State
```

```
        end
    end}
], InitialState),
lasp:update(ID, {add, 5}, self()),
lasp:update(ID, {add, 2}, self()),
lasp:update(ID, {add, 3}, self()),
S1 = achlys_lazy_stream_reducer:get_state(Pid).
```

As you can see, the interface is similar than the `achlys_stream_reducer`: The callback function has been removed. To retrieve the current state, the programmer will have to call the `get_state` function with the process id of the spawned worker.

The module works as follow: The main thread waits for a message. When the module receives a message, the module will retrieve the state of all input variables one after the other by using the read function of Lasp. By providing the atom `undefined`, the read function will return the current variable state immediately. Then, the module will check if the variable has been updated since the last call by comparing the vector clock with the cached one.

If the vector clocks are not equal, it means that the input variable has been updated. The module will then infer and operations that has been applied to the input variable and update the state by using the anonymous functions provided by the programmer. Finally, the module will update the cache: The module will update the variable state and the vector clock of the input variable that has been updated.

As we have seen, the resources of the GRiSP boards are limited. This module aims to limit the resource usage by updating the state and the cache of the node only when it is required.

E.3 Stochastic approach

E.3.1 Monte Carlo method

Let suppose that we want to estimate π. We know that the area S of circle can be calculated by using the formula $S = \pi r^2$ where r corresponds to the radius of the circle. If the radius is equal to 1, we can simplify the formula as follow: $S = \pi$. In other words, estimating π comes down to estimating the area of a circle having a radius of 1.

The Monte Carlo method can be used to estimate the area of a complex shape by sampling. The idea is to put a shape of an unknown area in another shape having a known area. Then, we can generate many points randomly over the entire encompassing area and we can estimate the unknown area by using the following formula:

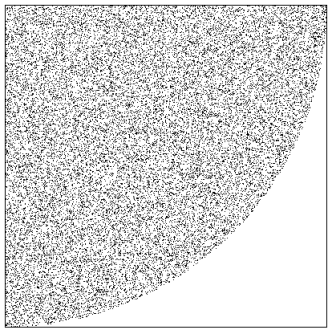
$$A_{estimated} \approx A_{total} \frac{m}{n}$$

where:

- $A_{estimated}$ corresponds to the unknown area.
- A_{total} corresponds to the area of the encompassing shape.
- m corresponds to the number of points inside the shape.
- n corresponds to the total number of points.

For the method to work, the shape having an unknown area should not be larger or go beyond the encompassing shape. If we produce many points, the point density should be similar all over the entire area of the encompassing shape. Otherwise, the estimation may be biased.

To estimate the surface of a circle of radius 1, we can estimate the surface of a quadrant and multiply the estimation by 4. Here, we can place a quadrant of radius 1 in a 1-by-1 square and use the Monte-Carlo method to estimate its surface. The idea is simple: we will select one of the four corners. Then, we will generate many points and compute the Euclidean distance between the points and the selected corner: If this distance is less than 1, we will increment one counter. If this distance is greater than 1, we will increment the other counter. Then, we can estimate the surface of the quadrant by considering the ratio of these two counters.



In this picture, the center of the circle corresponds to the top-left corner. Here, each point corresponds to a pixel. If the distance between the top-left corner and

the point is less than 1, the pixel will be colored in black. Otherwise, the pixel will remain in white. By generating many points, we can start seen the outline of the quadrant.

To estimate pi with Lasp, we created two grow-only counters: One counter will be used to count the number of points that are inside the quadrant (i.e: the inner counter) and one counter will be used to count the number of points that are outside the quadrant (i.e: the outer counter). The total number of points n can be obtained by summing these two counters.

Each node will generate a set of points. As we have a square of 1-by-1, the coordinates x and y can be generated by using uniform function of the rand module. This function will return a random float uniformly distributed between 0 and 1. Each time a node produces a new point, the node will increment one of the two counters mentioned above: If the point lies within the quadrant, the node increments the inner-counter. Otherwise, the node increments the outer-counter.

To get an estimation of π, we can simply retrieve the value of each counter with the query function of Lasp and use the following formula:

$$\pi = 4\frac{inner\,counter}{inner\,counter + outer\,counter}$$

Each node will take a sample independently and update the CRDT to consider the new sample. This operation does not require any synchronisation mechanism between the nodes. By combining their results, the nodes may get a better approximation. In this example, all nodes will converge to the right result.

The first disadvantage of this method is that we get an approximation of the solution in an empirical way: The only way to reduce the error is to increase the sample size. In some use cases, we cannot get new observations on demand. Having many observations is not always possible in practice.

The second disadvantage of this method is the obtained result is not always accurate: As the number of observations is finite, there is always a non-zero probability that the result we obtain is way off the value we try to estimate. The only way to mitigate this risk consist of increasing the number of observations.

E.3.2 Stochastic gradient descent

In machine learning, the learning process consists, most of the time, of finding the parameters of a model that minimize an error function. There exist several methods to find these parameters. The most used method is probably the gradient descent method.

$$x(t+1) = x(t) - \alpha \frac{\partial f}{\partial x}$$

where,

- α corresponds to the learning rate.
- x corresponds to a parameter of the model.
- f corresponds to the error function.

At each iteration, we have to compute the derivative of the error function with respect of every parameter of the model. Most of the time, the error function will be computed by comparing the predicted value ($\hat{y}$) and the observed value (y) of each observation of the train set. As a result, a single iteration can be computationally intensive if the number of parameters and the number of observations is large.

To avoid that, we can use a variant of the gradient descent called the stochastic gradient descent. The idea is to select randomly one or a few observations from the train set and to adapt the parameters of the model based on these observations. Using a few observations will lead to more stable results in fewer steps.

One of the main advantages of the stochastic gradient descent is that we can add new observations at run time without having to restart the algorithm. If we have converged to the optimal parameters, adding a new point will not have much influence on the parameters unless the new observation is an outlier and the error function is not robust to outliers.

With the gradient descent, the algorithm will update the parameters of the model by small adjustments. If it is no longer possible to reduce the error by slightly changing the parameters of the model, it means that the algorithm has converged to a minimum.

- If the error function is convex, it means that the error function will have a single minimum. Hence, the algorithm will converge to the global minimum. The parameters found will be the best possible ones. With such error function, we have the guarantee that the nodes will converge to the same result.

- If the error function is not convex, it means that the error function may have several minima. Hence, the algorithm will converge to a minimum but not necessary the global one. There may exists better parameters. With such error function, we do not have the guarantee that the nodes will converge to the same result.

There exist different strategies that can be used to escape from the local minima. For instance, we could decide to accept a worst solution. We could restart the program several times with different starting parameters and keep the best solution found. Unfortunately, none of these strategies gives the guarantee that the node will reach the global minimum.

E.3.3 Linear regression

The GRiSP boards can be used to collect data and execute the stochastic gradient descent to estimate the parameters of a model. With this algorithm, the boards can learn and make predictions even if the train set has not converged yet.

The error function is:

$$\begin{aligned}
f(m,p) &= \sum_i (y_i - (mx_i + p))^2 \\
&= \sum_i (y_i^2 - (2y_i(mx_i + p)) + (mx_i + p)^2) \\
&= \sum_i (y_i^2 - (2y_i(mx_i + p)) + (mx_i + p)^2) \\
&= \sum_i (y_i^2 - (2mx_iy_i + 2py_i) + (mx_i + p)^2) \\
&= \sum_i (y_i^2 - 2mx_iy_i - 2py_i + (mx_i)^2 + 2mpx_i + p^2) \\
&= \sum_i (y_i^2 - 2mx_iy_i - 2py_i + m^2x_i^2 + 2mpx_i + p^2)
\end{aligned}$$

Partial derivative with respect to m:

$$\begin{aligned}
\frac{\partial f}{\partial m} &= \sum_i (-2x_iy_i + 2mx_i^2 + 2px_i) \\
\frac{\partial f}{\partial m} &= \sum_i (2x_i(mx_i + p - y_i)
\end{aligned}$$

Partial derivative with respect to p:

$$\frac{\partial f}{\partial p} = \sum_i (-2y_i + 2mx_i + 2p)$$
$$\frac{\partial f}{\partial p} = \sum_i (2(mx_i + p - y_i)$$

We can update the intercept and the slope iteratively by using the following formula:

$$m_{(t+1)} = m_{(t)} - 2\alpha(\sum_i (x_i(m_{(t)}x_i + p_{(t)} - y_i)))$$
$$p_{(t+1)} = p_{(t)} - 2\alpha(\sum_i (m_{(t)}x_i + p_{(t)} - y_i))$$

With these formulas, the board can compute the parameters of the model by taking frequent samplings from a CRDT variable containing the input data. Here, the boards do not have to communicate with each other to fix the value of the parameters. Each board will compute them independently. If new data are added to the variable, the new data will be considered. If the CRDT variable has not converged yet, the board may converge towards different parameters as they do not have the same input data. But still, they will eventually converge towards the same parameters from the moment the CRDT variable converges.

Acronyms

API Application Programming Interfaces.

ARM Advanced RISC Machines.

BIF Build-In-Function.

CRDT Conflict-free Replicated Data Type.

DAG Directed Acyclic Graph.

DIP Dual In-line Package.

FPU Floating Point Unit.

GDPR General Data Protection Regulation.

GPIO General Purpose Inputs Outputs.

IOT Internet-Of-Things.

OTP Open Telecom Platform.

RTEMS Real Time Embedded Military System.

SPI Serial Peripheral Interface.

SRAM Static Random Access Memory.

UART Universal Asynchronous Receiver/Transmitter.

WLAN Wireless Local Area Network.

Glossary

Edge computing is a decentralized model that process and store data on the edge of the network. The edge computing does not rely on the cloud.

Extreme edge computing is edge computing where applications run on sensor boards.

Fog computing is an extension of cloud computing. A part of the data processing and storage is on the edge. The fog is connected to a cloud.

Hamiltonian path is path in a graph passing through all the vertices exactly once.

Traceable graph is a graph containing a Hamiltonian path.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**