Table des matières

[1. Introduction 3](#_Toc60394493)

[1.1 Context 3](#_Toc60394494)

[1.1.1 Usual approach 4](#_Toc60394495)

[1.1.2 New approach 4](#_Toc60394496)

[1.2 CRDT 5](#_Toc60394497)

[1.2.1 Principles 5](#_Toc60394498)

[1.2.2 Advantages 6](#_Toc60394499)

[1.3 Lasp 7](#_Toc60394500)

[1.3.1 Lasp libraries 7](#_Toc60394501)

[1.3.2 ORSWOT 8](#_Toc60394502)

[1.4 Goals and contributions 11](#_Toc60394503)

[1.4.1 Improved API 11](#_Toc60394504)

[1.4.2 Measurements 12](#_Toc60394505)

[1.5 Summary and structure 13](#_Toc60394506)

[2. Contributions 14](#_Toc60394507)

[2.1 Measurement tools 14](#_Toc60394508)

[2.1.1 Principle 14](#_Toc60394509)

[2.1.2 API 16](#_Toc60394510)

[2.2 Adaptation tools 20](#_Toc60394511)

[2.2.1 Principle 20](#_Toc60394512)

[2.2.2 API 20](#_Toc60394513)

[2.3 Scripts 22](#_Toc60394514)

[2.3.1 Static measurement scripts 22](#_Toc60394515)

[2.3.2 Dynamic measurement scripts 24](#_Toc60394516)

[2.3.3 Quality-of-life scripts 24](#_Toc60394517)

[2.4 Lasp additions 25](#_Toc60394518)

[2.4.1 Memory leak 26](#_Toc60394519)

[2.4.2 Readme improvement 27](#_Toc60394520)

[3. Measures and results 28](#_Toc60394521)

[3.1 Parameters 28](#_Toc60394522)

[3.2 Initial measures 28](#_Toc60394523)

[3.2.1 Number of Nodes 28](#_Toc60394524)

[3.2.2 Nodes distance 28](#_Toc60394525)

[3.2.3 CRDT size 28](#_Toc60394526)

[3.2.4 CRDT operation 28](#_Toc60394527)

[3.2.5 Partition 28](#_Toc60394528)

[3.2.6 Update speed 28](#_Toc60394529)

[3.3 Adapted Lasp 28](#_Toc60394530)

[3.4 Analysis 29](#_Toc60394531)

[4. Conclusion 31](#_Toc60394532)

[4.1 Summary 31](#_Toc60394533)

[4.2 Analysis conclusion 31](#_Toc60394534)

[4.3 Future work 31](#_Toc60394535)

[4.4 Personal opinion 31](#_Toc60394536)

[4.5 Methodology 31](#_Toc60394537)

[6. Bibliography 32](#_Toc60394538)

# 1. Introduction

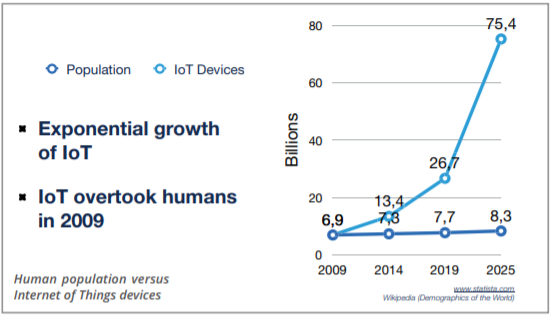
This chapter presents the general content and context of this manuscript, describing what is Lasp, what are CRDTs, what are their innovative aspects and why they are so useful. The goals of this master thesis and its main structure will also be briefly introduced.

## 1.1 Context

In today world, large-scale distributed applications are more and more common. These applications, to work correctly on multiple devices must share distributed variables, in other words, values that can be accessed and modified consistently from any node of the system. These variables may then be used by the application for thousands of different possible usages. A good example is the case of IoT small devices with captors and sensors collecting information such as temperature, light, pressure...

The way to handle these distributed variables is generally hidden to the end-user but can represent an important part of the application implementation requiring for the developer to consider consistency and distribution. This master thesis will focus on the way to handle these distributed variables considering a particularly innovative approach that was introduced around 2011 and of which the Ecole Polytechnique de Louvain research team has developed an experimental version called Lasp.

To illustrate the need in the domain of distributed applications and thus distributed variables, let’s simply show the growth of distributed applications with the case of Internet of Things (IoT) devices1.



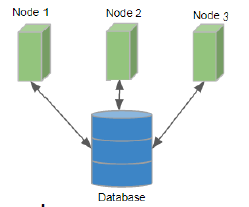
There is no doubt, the explosive growth in that domain deserves our attention and the seek for new innovations to handle it.

### 1.1.1 Usual approach

The most common way to handle distributed variables is to centralize them with a database. This means every node will connect to the database to access the variables. This is generally handled with an API for the developer to avoid overthinking on technical problems such as causality and consistency. These databases usually allow some interesting features such as atomic operations and log history but actually require some (hidden) heavy algorithms.

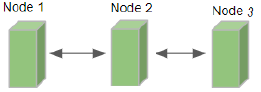
Furthermore, this kind of distributed structure usually relies on redundancy with replicated databases in multiple data-centers to achieve high scalability, adding more complexity to handle causality and operation order consistency between replicas, introducing Consensus algorithms.

Finally, since strong Consistency conflicts with Availability and Partition-tolerance (CAP theorem1), these systems have to choose between CP (strong Consistency and Partition tolerance but low Availability), AP (high Availability and Partition tolerance but weak Consistency) and CA (strong Consistency and Availability and no Partition tolerance). While a good part of the mainstream distributed applications goes for the AP model with a loss of Consistency, no ideal solution exists.



### 1.1.2 New approach

A totally different approach is to rely on peer-to-peer instead of the usual structure with databases. This means no database servers running heavy algorithms is required, instead the distributed variables are handled via messages exchanges between nodes. This new alternative relies on an innovative way to represent the distributed variables. As opposed to the usual approach where distributed variables are generally just values registered and updated in a specific database, variables will be represented as a specific data-structure called Conflict-free Replicated Data Types (CRDTs2). It is the key concept that will be detailed below to understand this new approach along with all its advantages.



## 1.2 CRDT

CRDT is for Conflict-free Replicated Data Type. The main idea is that it is an abstract data type

with an interface designed for replication on multiple nodes and satisfying the following properties3:

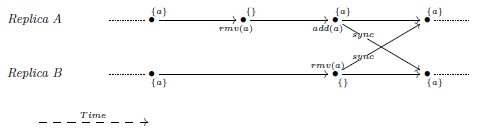
1. Any replica can be modified without requiring any coordination with any other replica.
2. Two replicas receiving the same set of updates reach the same deterministic state guaranteeing state convergence.

Even if this approach might look surprising at first sight (since it does not involve recording the distributed variable state in a specific place such as a database, nor require any consensus algorithm), this new way to represent distributed variables introduced in 2011 is already used by some big compagnies such as Riot Games, TomTom, Bet365, SoundCloud and some others4.

### 1.2.1 Principles

Convergence is the key-concept to understand CRDT principle. To clarify this concept, let’s illustrate it with a very general example considering a single distributed variable:

1. Every node has a local state representing the distributed variable. This local state is a data-structure than contains values and metadata, it is called a CRDT. The specific structure is not relevant here since it depends on the type of CRDT. In other words, the specific structure is not the same if the nodes share a variable representing a counter, a set of elements, a Boolean…
2. A node can adapt its local state to modify the variable without requiring any coordination with other nodes. For example, if the variable represents a set, it can add an element in it. When doing such, it will modify the values in the data-structure (CRDT) as well as the metadata.
3. From time to time, the nodes will send their local state to their peers. In other words, they will send their own version of the CRDT to their peers. When receiving such a message, the node will merge the received state with its own local state. The way this merge is implemented is very important since it is this specific operation that will guarantee the system convergence. Indeed, the merge uses the metadata to determine how to merge the two versions in a deterministic way representing the most causally recent modifications. This allows the most recent modifications to propagate from peer-to-peer to the entire system and eventually reach a consistent state on every node.

Here is an extremely basic example5 with an add-wins set (if concurrent add and remove occur, the add wins).

As mentioned above, the key-concept here is the convergence. It is the fact that, automatically, due to the CRDT metadata and the merge implementation, all the nodes will eventually reach the same consistent state.

### 1.2.2 Advantages

The incredible part is that the convergence described above is automatic, deterministic, independent of the received messages order (scheduler) and does not require any consensus algorithm other than simple metadata comparison. In other words, based on messages received from its peers, the node will determine how to update its local state, efficiently handling the distributed variable without requiring heavy algorithms or database. Cherry on the cake, it also makes it automatic to handle partitions.

* **Automatic**: The synchronization is pretty simple and straight forwards since the nodes send their local state regularly and automatically update their states based on peers messages.
* **Deterministic**: A set of received messages will always update the local state in the same way, resulting in the same final state.
* **Independent of the message order**: The merge operation will compare the received metadata with the local metadata to determine how to update the local state. When receiving, for example, a recent message followed by an old message, the node will update its local state based on the recent message and will just ignore the older message since its metadata are older than its own updated metadata. In other words, the message order has no impact since the merge operation will follow causality handled by metadata and not the receiving message order. Furthermore, since the implementation is state-based (the messages represents a state, not an operation), potentially lost messages are not a problem either since the most recent message represents the most recent state and does not require previous messages to be correctly interpreted.
* **No consensus required**: Simple metadata comparison within the merge operation allows the receiver node to easily determine how to update its state. No database server is required, consensus algorithm either.
* **Partition-tolerant:** The previous properties, especially the fact that message order and lost messages do not impact converge, allow to easily handle partition-tolerance. Indeed, when a node is temporarily unreachable, it will continue to work with its own state which might be temporarily inconsistent with other nodes. Then, when the partition is resolved, it will receive state messages from other nodes and directly update its local state to represent the most recent version.

Let’s consider that strong Consistency is good for the ease of programming but requires heavy synchronization algorithms. At the opposite, we can consider that weaker Consistency is harder to use for the application developer (he is not sure every node has the same value for a distributed variable) but requires less synchronization algorithms. With these two basic principles in mind, we would logically want a consistency model as strong as possible while running with a synchronization algorithm as light (weak) as possible. Here, CRDT new way to handle distributed variables comes in with a very efficient model allowing strong eventual Consistency with a weak synchronization algorithm (even called “sync free”, which was the name of the initial project leading to Lasp development6).  
 Strong eventual Consistency (SEC) is achievable to the fact every node receiving the same set of updates (in any order) have equivalent state and the fact every update will be eventually delivered to every node due to peer-to-peer communications. It is in fact even stronger than that since nodes do not require to receive the exact same set of updates, some previous updates may not be received that it will not perturb the system as long as recent messages eventually deliver.

In regard of the CAP theorem, CRDT model allows strong eventual Consistency7 with high Availability and Partition tolerance which is probably the best compromise from the CAP theorem yet while not even requiring any heavy algorithm (no consensus required!).

No doubt the good features and properties described above together with the excellent CAP theorem compromise are the reasons why CRDT usage is growing quickly8 and has been adopted by some big compagnies as previously mentioned.



## 1.3 Lasp

Lasp is an experimental implementation of CRDTs developed by the Ecole Polytechnique de Louvain (EPL) research team and initiated in 2013 with the impetus of two European projects; SyncFree9 in 2013 then LightKone10 in 2017. More precisely, it takes the form of a group of Erlang libraries acting together to offer a programming framework based on CRDT. There entire project can be found on their official github repositories: <https://github.com/lasp-lang/lasp>.

### 1.3.1 Lasp libraries

The libraries offer everything to handle different types of distributed variables (different kind of CRDTs are implemented such as counter, set, Boolean, map…) including the communication part, distribution and easy-to-use API to update or query on CRDTs. The particularity of Lasp compared to other alternatives to handle CRDTs is the tools it offers to manipulate and compose on CRDTS. Indeed, CRDTs are very handy to easily handle distributed variables but they require caution when using their outputs to compute or compose data. More precisely, the CRDT itself composed of values and metadata will reflect the known most recent version of itself but this is not especially the case for the values we got from querying the CRDT previously. In other words, if a developer queries the value of a CRDT then computes something based on this value, he got the most recent value from the CRDT and his computation is momentarily true. But any moment later, his computation might be wrong since the CRDT got updated but the value he got previously from it was not updated unless he queries again the CRDT and starts his computation again. Lasp is specifically designed to address these issues and to allow easy computation and even composition on CRDTs without requiring the developer to handle these problems himself11. The idea is to consider the CRDT as an input stream and to ouput a stream of values always leading to the known most recent value. With this approach, the developer has tools to compute things based on a CRDT while his computations will be automatically updated with the CRDT. Thus Lasp offers a complete API to facilitate always updated unions, intersection, maps,… on CRDTs. This particular aspect is very convenient but will not be discussed in this master thesis since it will mainly focus on the distribution and communication part without detailing in depth the end developer aspects.

### 1.3.2 ORSWOT

Since Lasp offers multiple different CRDTs with their own representation and metadata, selecting one particular CRDT was a good starting point to have a reference to understand CRDTs principle and practical implementation while being able to measure its performances. The CRDT that was mainly used for this work is the ORSWOT. It is a relatively recent CRDT that offers some good properties since it represents a set where nodes can add or remove elements allowing it to represent basically anything even if it might not be optimized for every kind of elements. For example, it could represent a set containing only an integer where nodes only operation would be to increment the integer by one. This example is possible with an ORSWOT while it could be better optimized using a CRDT specifically designed for counter. The fact the ORSWOT allows many possible usages was a good starting argument then comes the fact it is relatively well optimized for general usage. Indeed, compared to its predecessor, the well-known ORSET (Observe Remove Set), it addressed and resolved many little issues.

The ORSET, which is the previous version and is still used by many CRDT programs, represented, as for the ORSWOT, a set where nodes could add or remove elements. The problem was the fact when an element was removed, a reference to that removed item was still present in the CRDT (reminder: the CRDT is implemented as a data-structure containing values and metadata). This introduced tombstones for every removed element which could translate into significative memory leak and network usage on the long run if elements were frequently removed and replaced by others.

Thus, the ORSWOT is the general selected CRDT for this master thesis. As a little remark, the generic name ORSWOT comes from orset without tombstones due to the fact, as just explained, it addressed the tombstone issue from the generic orset. As a note, the rest of this document might use the name “awset” instead of ORSWOT, it is simply the name used for the ORSWOT inside Lasp specific implementation. The name awset itself is a reference to the fact the implementation had to make a choice for the way to handle a concurrent add and remove of the same element to achieve determinism. As suggested in the name, in this specific scenario, add wins.

Finally, since it is the CRDT used for this work as a core use case, let’s go a little bit more in depth about its implementation then let’s illustrate with an example. A very good presentation from Bet36512 shows the ORSWOT principle, which is why my explanation will re-use some of their own examples.

The data-structure is as follow:

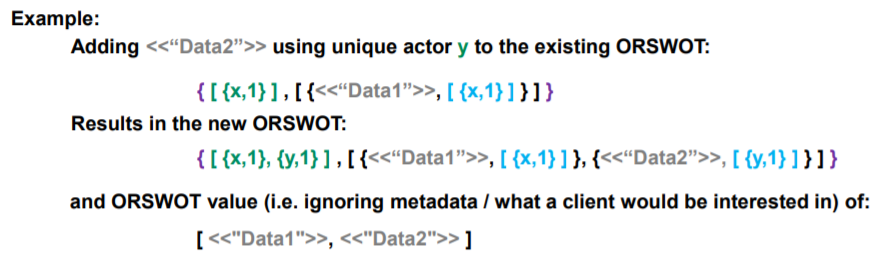
* It starts with a version vector. It is a set of tuples of size 2 (pairs).
  + Each pair consists of a unique actor name (unique identifier for a replica) and a counter. It is represented in green on the example image.
* Then comes the entries. It is a set of tuples of size 2 (pairs).
  + Each pair consists of an element (data such as visible from a client when querying the CRDT) and a Dots set represented in blue.
    - This Dot set himself is composed of tuples of size 2 (pairs) and generally contains only one. The Dot set contains multiple pairs only if multiple concurrent adds for the same element are merged (two nodes concurrently added the same element).
      * Each pair is composed of a unique actor name and a counter.



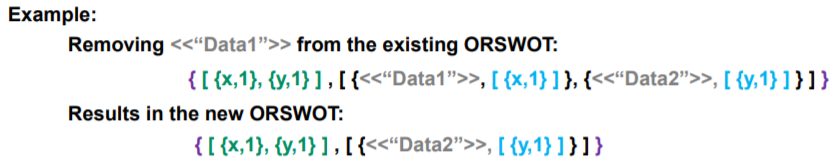
The way to handle the metadata is the basis for understanding the functioning.

* When a node adds an element:  
  The version counter is updated, incrementing by 1 (or setting to 1 if not currently present) the pair for that unique actor.

A pair is added in the entries with the added element and the updated pair {UniqueActorName, Counter} as its Dots. If a pair already existed for that element, it is replaced by the new one.



* When a node removes an element:  
  The version counter is not modified.  
  The pair containing that element is simply removed from the entries (without tombstone).

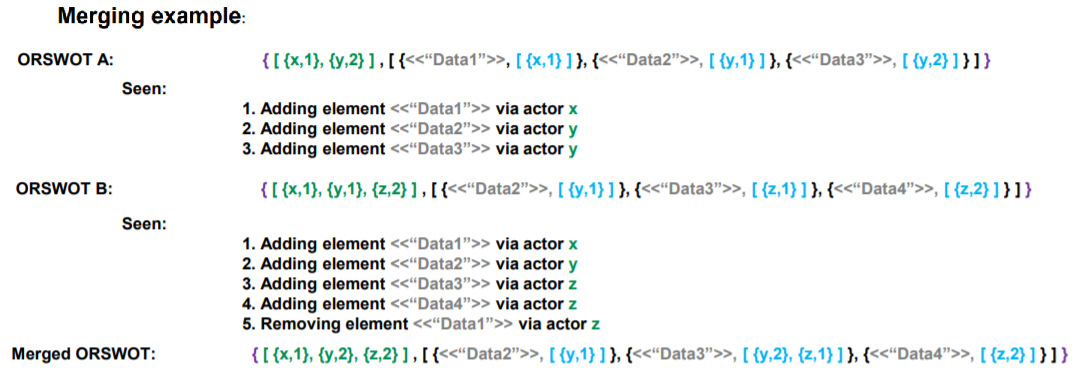


* When a node merge two states:
  + The version counter is merged, taking only the higher counter for every unique actor (as for usual vector clocks).
  + For common elements (elements present in both versions):  
    Common pair inside the Dots are preserved (same unique actor name and counter).  
    Dots pair present only in Replica A is preserved only if its counter is higher than the counter for this unique actor name in Replica B version clock.   
    Dots pair present only in Replica B is preserved only if its counter is higher than the counter for this unique actor name in Replica A version clock.

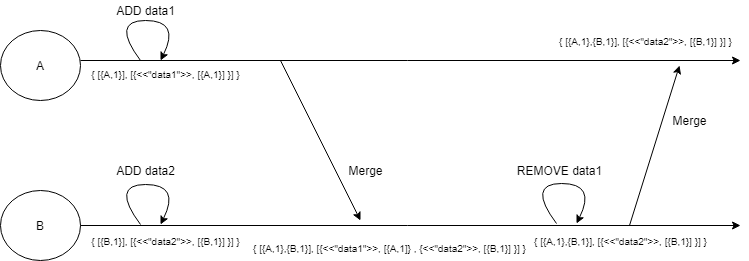
At this point, if there is still a Dots pair for an element (it went through the filter), this element and its Dots are preserved.

In other words, the Dots pair is preserved only if it’s the most recent known information.

* + For non-common elements (elements that were present only in one of the two versions):  
    Dots pair present for an element in replica A are preserved only if its counter is higher than the counter for this unique actor name in replica B version clock.  
    Dots pair present for an element in replica B are preserved only if its counter is higher than the counter for this unique actor name in replica A version clock.  
    At this point, if there is still a Dots pair for an element (it went through the filter), this element and its Dots are preserved.



CRDTs implemented in Lasp are state-based, meaning the peer-to-peer messages simply contain the CRDT state (and no operation as opposed to operation-based). This means the three operations; adding, removing and merging are everything that is needed to implement and understand the ORSWOT (awset in Lasp).

Let’s close this point with a small visual example resuming adding, removing and merging. To mention that this schematic is just one scheduler example while any other scheduler would give the same results since CRDT are convergent and deterministic whatever the received message order:

## 1.4 Goals and contributions

The previous points mainly discussed the state of the art referring to already published articles and explanations. Reading reports and documentation about CRDTs is a good starting point but insufficient to fulfil the objectives of this Master thesis. Let’s discuss the concrete goals that were pursued in this work.

### 1.4.1 Improved API

A first remark we can make about Lasp is that its documentation is very limited. There are some information and a little documentation available at <https://lasp-lang.readme.io/docs> but it is limited to a few examples some of which are not up to date, resulting being incorrect due to some API changes. It is a shame because when diving into Lasp code, it is very wide and offer many features that are documented nowhere. In the current state, it is purely an experimental tool where developers not initiated into Lasp would have difficulties to set up their settings to use it properly. As visible in the short documentation, it is not difficult to start a local example and to share a CRDT between nodes but there is no directly available information such as how to parametrize it properly or how to measure its performance.

In this optic, a first goal was to improve the API in a particular direction related to the convergence principle described in section 1.2.1. Indeed, convergence is a fantastic feature allowing every node to eventually end up with a consistent state without requiring any heavy synchronization algorithm but how much time does it take for a specific cluster of nodes to converge? There was, up to now, no directly available tool to easily know such information from an end developer perspective. This is an important issue for practical usage since an application would probably not work as intended on a cluster than converges in 10sec instead of 1sec for example and the developer had no tool to easily detect that. Therefore, a first objective was to develop a measurement tool that could be easily incorporated in Lasp to measure convergence time together with network utilisation.

Once this first task implemented, another objective was to add a tool to modify the convergence time. In other words, once possible to measure the convergence time it would be useful to offer tools to easily modify it for example to make a cluster converge faster. This is thus also a part of this work objectives.

### 1.4.2 Measurements

Once the new tools implemented, using them to measure different cases is the second main objective. How is Lasp awset CRDT performing in practice? How is the convergence time influenced by the various parameters of real usages? Is Lasp implementation really meeting all the suggested CRDT features such as partition-tolerance? What is the minimum achievable value for convergence time and how does it affect the network usage? All these are real questions that deserve reflexion for the future.

A first approach that will be described later in the work is to measure convergence time for different scenarios. In this optic, here is a list of parameters that might potentially influence the convergence time:

* + Cluster size (number of nodes)
  + Geographical distance between nodes
  + Nodes heterogeneity (different hardware, architecture, CPU…)
  + Nodes workload (nodes might be busy with other heavy processes)
  + Nodes crashing
  + Nodes under partition
  + Type of CRDT (orset, orswot, counter, map, boolean…)
  + CRDT size (number of elements in a set for example)
  + CRDT operation (e.g. adding an element might converge faster than removing one)
  + Number of parallel CRDTs (e.g. a cluster sharing high number of different CRDTs at the same time might consume more CPU and network bandwidth slowing down the system)
  + CRDT value update speed (Nodes might want to update the value in a CRDT extremely frequently)
  + Network available bandwidth
  + Network speed
  + Network packet loss rate

While many of these aspects are interesting and could have real impact on performances, the context of this master thesis pushed the experimental work to be limited to only some of these parameters. From the above list, here are the selected parameters:

* Cluster size
* Geographical distance (at a small scale)
* Nodes under partition
* CRDT size
* CRDT operation
* CRDT value update speed

These parameters were selected mainly for being focused on the CRDT principle itself (where some other parameters were more focused on the network or nodes CPU workload aspects) while being relatively practical to test and measure within the limited duration of this master thesis.

Finally, one last important aspect of this work is to analyse the measures, to explain the results and to find Lasp limitations. For example, it might be impossible to push a cluster of 5 nodes to converge faster than within 50ms. Or it might be impossible for a cluster convergence to catchup with a CRDT which is updated every 10ms introducing a “never-really-converged” permanent state where nodes are always few updates late compared to the updating source and never catchup. These limit cases are the last point that will be discussed in this work.

## 1.5 Summary and structure

This work will be based on Lasp implementation of CRDT, using the orswot (named awset in Lasp) as use case. The contributions to Lasp will be presented in the next chapter (chapter2) including the new developed tools and a few improvements that were proposed to enrich Lasp or make it a bit more user-friendly. The scripts used for this work will also be briefly presented. Following, in chapter 3, the measurement and results will be presented and analysed. Finally, chapter 4 concludes with some summary on observations, some future work propositions, a personal opinion on Lasp and the general methodology followed during this work.

# 2. Contributions

The technical contributions are mainly two tools, one to measure convergence time and network usage while the other is about modifying the convergence time. Apart from that, many little useful scripts were implemented to help test and measure different cases. The entire work can be found at the public github respository: <https://github.com/darkyne/LaspDivergenceVisualization>.  
The different readme files there explain the entire structure with the different directories and files but is mainly written in French. Mainly instructions on how to run the scripts, how to correctly run the tools and what restriction they require are described on the github repository of this work.

## 2.1 Measurement tools

This tool is designed to allow the end developer to easily get information about a cluster convergence time and network usage (number of messages per second). It was developed as a few methods inside the erlang module lasp\_convergence\_measure which was created for that purpose. It is functional in the sense it does, as intended, give the end user useful information about its cluster convergence time but it does not exactly fulfil the ideal task of directly measuring the convergence time of a specific CRDT shared on a cluster. This is due to the approach that is to mimic a CRDT and to measure its convergence time on the cluster instead of directly measuring an already shared CRDT under use. That said, it offers some useful and relatively precise information to the end user to know how fast a cluster converge as well as how many messages per second are exchanged on the cluster.

### 2.1.1 Principle

The idea is to allow the nodes to launch a small background process which will be tasked to continuously do measurements on the cluster. Since it would be difficult to measure a specific CRDT convergence time already under use on the cluster without modifying it or impacting its performance, another approach was adopted. A specific awset will be shared on the cluster and will be used from every node for measurement. The implemented solution consists of few simple steps:

* Every node launches a background process (for example on boot)
* A leader election is run on the cluster
* The leader puts an element (signal) on a specific CRDT (awset)
* Other nodes detect the element and answer with a timestamp
* The leader waits for all the answers and compute convergence time

These measurements are designed to be automatically run in continuous on an under-use cluster thus it must be reset and run again ever few seconds to allow always recent information available which was achievable through a time parametrized loop. This is the general principle, but some details require more explanation.

A first idea was to allow every node to be the source of the measurement signal to allow convergence time measurement from any source on the cluster. This is probably a good idea if we are only interested in measurements and precision. But since the tool is designed to be run easily in background on a real under-use cluster, it was preferable to only have one node initiating and orchestrating the measurement where other nodes simply answer. This allows smaller impact on the performance (nodes workload and network usage) while still giving some general information about the cluster convergence time.

The leader election step allows to automatically chose a leader to orchestrate the continuous measurements. Indeed, the leader has to put an element (acting like a signal) on the awset, wait for every other nodes answers, compute information (convergence time, round-trip duration, total messages/second) and make the measurement system clean again for the next measurement (reset) before next measurement loop. Finally, it is also responsible for making the recent measures available from every other node.

The case of partition or crashes during the continuous measurements is also easily handled by timeouts and the leader election step. Indeed, if a basic node gets partitioned or crashes, the leader will simply timeout waiting for its answer and will not take that node into account for the current measure. If the leader itself gets partitioned or crashed during measurement, the measurement loop will fail and have no influence while the most recent measurements are still available anyway. At next measurement loop, a new leader is simply elected via leader election and the continuous measurement continues. If a node joins the cluster (or resolved partition) during measurement, it will simply be considered at the next measurement round, as long as that node runs the continuous-measurement process, if it does not it will simply not be taken into account for measurements. This allows the measurements to continue when partitions, crashes and joining occurs on the cluster while being compatible with nodes who are running the continuous-measurement tool or not (simply ignored).

The reset part was also important since it makes sure everything is clean and the measurement signal is removed from every node point of view (from their own local state of the awset) before initiating the next measurement round.

While the principle has been explained, the fully commented code can be found in the lasp\_convergence\_measure erlang module with the launchContinuousMeasurement function. For more specific and technical details, the exact measurement steps are the following, using a total of 2 little (maximum number of elements is equal to number of nodes) awsets for measurement:

* The leader detects how many reachable nodes are on the cluster.
* The leader puts an element in a specific awset A (acting like a measurement signal).
* Nodes detects the presence of that element on A and put their {Id, TimeStamp and number of messages received par second since last measurement} on a specific awset B.
* Leader detects that every other node answered on B with their information (number of answers equal number of reachable nodes).
* Leader computes convergence time based on time between measurement signal setting and TimeStamps, round-trip time based on duration between measurement signal (on A) and all answers gathering (on B) and sum the number of received messages per sec on every node to have global cluster information.
* Leader removes the measurement signal (from A).
* Other nodes detect the signal was removed (from A) and remove their own information (from B) then start waiting for new measurement round.
* Leader detects every other node removed their information (from B) and can start next measurement round (based on the period between every round).

### 2.1.2 API

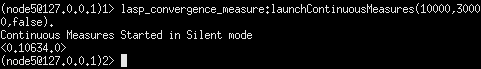
The API to use the measurement tool is straight forward. There is a function that must be called on every node to initiate the measurement process with some specific arguments (it can be launched at node booting for example) then a few getters to get information about the cluster whenever needed. As a reminder, the goal is to make the information easy to access while not impacting the cluster too much since it is designed to run on an under-use cluster. This is also the reason why, as explained above, it does not directly measure a specific CRDT shared on the cluster but gives

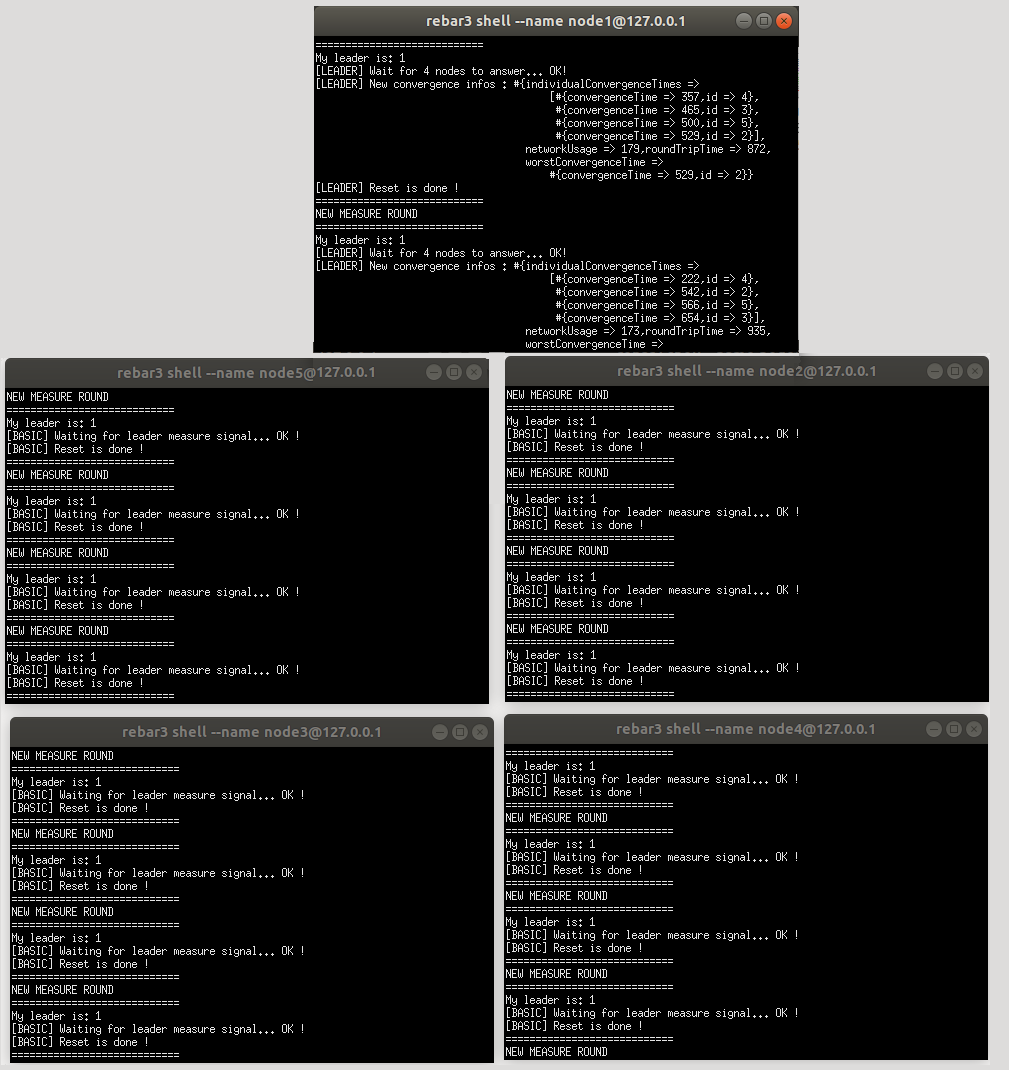
Important information:  
All the functions are accessible via lasp\_convergence\_measure module.  
An important assumption was made to facilitate some parts of the code such as the leader election:  
Every node in the cluster must follow the name “nodeX@IpAddress” where IpAddress can be localhost or any Ip address and X can be any unique integer (X will be considered as unique node Id).

Here is the complete API with some little examples on a local cluster of 5 nodes:

|  |  |
| --- | --- |
| Function Name :  **launchContinuousMeasures** | Starts a process to manage continuous measurements in background. |
| Argument 1 : MeasurementPeriod | The number of ms between each measurement round. This acts as a minimum period and as the real round period in general cases. But since a round might take more time in case of slow convergence or timeouts, the round period may raise up to 2 times Timeouts on worst case. |
| Argument 2 : TimeOut | Maximum waiting duration in ms before considering a node timed out. This allows the measurements not to block indefinitely if some nodes crash. |
| Argument 3 :  Debug | Boolean to allow or disallow the process to print information in a terminal. Should be set to false for real usage, set to true only for debugging or demonstration. |
| Output | Returns the process Pid. |

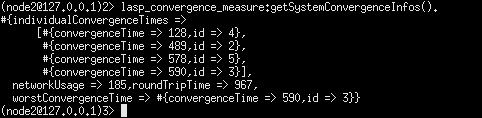
Example usage, Debug=false :



Example usage, Debug=true :

|  |  |
| --- | --- |
| Function Name :  **getSystemConvergenceInfos** | Queries to get the general information about the cluster most recent measurements. |
| Output | Returns a map containing the last available measurements. Time values are expressed in ms and network usage in term of messages/sec on the entire cluster. |

Example usage :



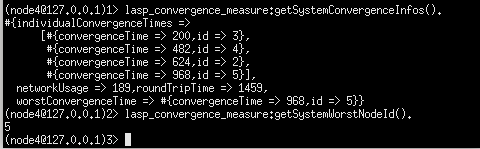
|  |  |
| --- | --- |
| Function Name :  **getSystemConvergenceTime** | Queries to get the convergence time from the cluster most recent measurements. |
| Output | Returns the cluster convergence time (in ms) lastly measured. As a reminder, convergence time is considered as being the time between an element put on an awset and the moment when every node has the same consistent state with that element in their local states. |

Example usage :



|  |  |
| --- | --- |
| Function Name :  **getSystemWorstNodeId** | Queries the information from the cluster most recent measurements to get the Id from the slowest node to converge (making the global convergence slower). |
| Output | Returns the Id from the slowest Node to converge. As a reminder, the nodes are supposed to be named nodeX@IpAddress where X is an unique integer considered as node Id. |

Example usage :



|  |  |
| --- | --- |
| Function Name :  **getSystemRoundTrip** | Queries the information from the cluster most recent measurements to get the round-trip time. |
| Output | Returns the round-trip time (ms) from the most recent measurement information. As a reminder, the round-trip timed is considered as the time between leader signal setting and the moment when leader detects answer from all the other nodes. In other words, it is the slowest round-trip on the cluster. |

Example usage :



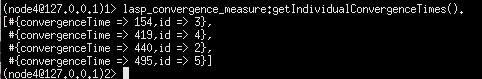
|  |  |
| --- | --- |
| Function Name :  **getSystemNetworkUsage** | Queries the information from the cluster most recent measurements to get the network usage information. |
| Output | Returns the number of messages per sec delivered on the cluster. It is computed as the sum of all the nodes received messages per sec. |

Example usage :



|  |  |
| --- | --- |
| Function Name :  **getIndividualConvergenceTimes** | Queries the information from the cluster most recent measurements to get the convergence times for every nodes. |
| Output | Returns a map containing the convergence times (ms) for every node. Convergence time is considered here as the time for that node to converge based on the source (get the same element in its local state). |

Example usage :



|  |  |
| --- | --- |
| Function Name: **getConvergenceTime** | Queries the information from the cluster most recent measurements to get the convergence time for a specific node. |
| Argument1: Id | A valid node Id. As a reminder, the nodes are supposed to be named nodeX@IpAddress where X is an unique integer considered as node Id. |
| Output | Returns the convergence time (in ms) for that specific node. |

Example usage :



## 2.2 Adaptation tools

With the first measurements, the fact the convergence was very slow quickly became apparent. Indeed, as will be presented in the chapter 4 (results and analysis), Lasp as directly cloned from the official repository is actually very slow. Measures and inspection quickly highlighted the cause of this slowness. As explained in chapter 1 (introduction), nodes sharing a CRDT (awset in our case) must send their local state from time to time to allow the system to converge. The default version of Lasp without more parameterized was making the nodes send their local state on a regular time interval which was hardcoded in a configuration file. Since the original value for that interval was 10000ms, it was making the system very slow to converge (around 10 seconds). Thus, a very simple but useful idea was to make that time interval modifiable via an API.

### 2.2.1 Principle

Instead of taking a configuration file hardcoded value for the time interval triggering the state send, there will be a default value and a setter function to modify that value. Let’s call this value the state\_interval for simplification. This state\_interval will be shared to all nodes so that every node in the cluster send their states on same intervals. The node who asks for the system to modify its state\_interval (aka the node at the origin of the modification) will directly change its state\_interval while some other nodes may take a bit more time to detect the modification on that parameter but will eventually adapt. The exact implementation for now, while not perfect but functional, is to use a one-element CRDT to share that parameter on all the nodes guaranteeing (due to CRDT properties) that eventually every node adopts the same state\_interval.

### 2.2.2 API

The API to adapt the cluster convergence time via modifying the time interval between states sending (state\_interval) is very straight forward with a setter and a getter. The setter directly modifies the state\_interval for the current node but may take up to one convergence time (generally around one state\_interval time) for all the nodes to adopt that same state\_interval. Let’s illustrate the principle with a little example:

If a cluster was running with a state\_interval of 1000ms (and thus a convergence time mean around 1 sec) and a node wants the cluster to converge faster, modifying the state\_interval to 100ms, it may take approximately 1000 ms before adopting that same sending speed on the entire cluster. This is due to the fact the cluster must converge on the state\_interval value using previous parameters before adopting the new value for that parameter.

|  |  |
| --- | --- |
| Function Name: **setStateInterval** | Modify the state\_interval which defines the time interval between each state sending from the nodes. |
| Argument1: newStateInterval | The new desired state\_interval value in term of ms. |
| Output | No output. Simply modify the value which may take up to one convergence time for the entire cluster to adopt after which the cluster is faster or slower to converge based on the entered value. |

Example usage :



|  |  |
| --- | --- |
| Function Name :  **getStateInterval** | Gets the currently used state\_interval which defines the time between each state sending. |
| Output | Returns the state\_interval in terms of ms. |

Example usage :



An interesting approach allowed with the entire API developed in this work is to allow the end-developer to call, for example, getSystemConvergenceInfos() to get information on its cluster. If he finds the cluster too slow to converge, he can call getStateInterval() to see at what rate do the nodes send their states and can modify it with setStateInterval(newStateInterval) to make, for example, the cluster converge faster. Then he can call getSytemConvergenceInfos() again to see if the cluster converges fast enough and can also see the impact on the network via the number of messages exchanged per second on the cluster. While very easy to use, it already allows great convergence visualization and notifiable modifications from the end-developer which is exactly one of this master thesis goals.

TODO: put a screenshot of an example: ContinuousMeasureSilent: j’attends un peu, j’affiche les infos, je modifie le state\_interval, j’attends un peu et je reaffiche les infos. Ce serait un excellent exemple d’utilisation de mon API ! Ou alors peut-petre mettre ça dans la conclusion pour montrer que mon truc est bien.

## 2.3 Scripts

Beside the new API developed, many scripts were written to allow easily testing the tools, making new measurements, creating a cluster with particular parameters and so on. These scripts, for the majority of them, are available within the mylasp/lasp/Memoire/MyScripts folder on this master thesis github: <https://github.com/darkyne/LaspDivergenceVisualization> where the readme files already describe them and explain how to launch them. All the scripts available on the github repository are designed to be easily runnable directly from any clone from the repository as long as Lasp (erlang 9 or later) requirements are met with as little parameters to modify as possible. By simply modifying mylasp/lasp/Memoire/AppsToLaunch/IpAddress.txt to enter [node1@127.0.0.1](mailto:node1@127.0.0.1) inside the txt file, the scripts should be correctly running with cluster running nodes locally. For details, this allows the scripts to know the tester wants nodes running locally with names starting from node1 (incrementing, node2, node3…).

### 2.3.1 Static measurement scripts

These scripts are designed in a very simple fashion simply starting a cluster of nodes then when the cluster is created (number of reachable nodes from every node is consistent with the cluster desired size), it realises one single operation (potentially one operation on every node) and measures how much time it takes for the cluster to converge on the result (using timestamps). While not dynamic and unusable on a real under-use cluster (which was the case for the tools from points 2.1 and 2.2), these scripts allow some measurements on clusters with specific parameters and specific CRDT (for example, defined initial number of elements inside the CRDT) where the previously described tools only mimic a generic CRDT on an under-use cluster to do general measurements. The difference from previous developed tool is very important since here a cluster is specifically created with the purpose of the measurement and then killed to start a new measurement round. Also, the measurements are analysed afterward and not dynamically on the run. The measurements are written in files on every measurement iteration then the script reads all the files to gather information and compute statistics.  
While relatively basic in their principle and not extremely precise (in term of precise times which may be shifted by different nodes unix times), these scripts allow to give a good overview and intuition of the variations due to the different parameters (such as number of nodes, number of elements in the cluster…).

These scripts include:

* LaunchSet1.sh: It launches a cluster of 5 nodes. The cluster shares an initially empty awset where nodes will each add elements and wait for convergence. Each node adds 10 unique elements (all at once) on the CRDT and wait to detect the final 50 elements (since there are 5 nodes and each adds 10 elements). The script runs this experiment 50 times (iterations) with the same parameters then switch to another version with other parameters. The different versions are:

- Nodes puts 10 elements (all at once) and wait to detect 50 elements.

- Nodes puts 100 elements (all at once) and wait to detect 500 elements.

- Nodes puts 1000 elements (all at once) and wait to detect 5000 elements.

- Nodes puts 5000 elements (all at once) and wait to detect 25000 elements.

-Nodes put elements then join cluster (as if every node added their elements on local state while under partition then resolved partition).

-Nodes join cluster then put elements (no partition simulation at all).

Resulting in a total of 8 different experimentations (the 10, 100, 1000 and 5000 elements versions are run each both with and without joining the cluster beforehand) and 50 iterations on each. The measurements cover convergence time, CRDT state size in memory and number of messages exchanged per second. After all the iterations on different versions, the script starts reading all the output files to compute mean, median and standart deviation and writes this in result file. All the detailed information such as where a written the outputs files, where is written the result file at the end of the script etc… are available and described in depth within the readme files. As a remark, be warned, if you want to launch the script, that according to the parameters, it may take multiple hours to run the entire script since it runs many iterations on cluster that may require multiple seconds to converge.

* LaunchSet2.sh: This uses the exact same principle and structure as the previous one, again with a cluster of 5 nodes but runs the following experimentations:

- Awset starts with 50 elements, every node removes 10 elements and wait until it is empty.  
- Awset starts with 500 elements, every node removes 100 elements and wait until it is empty.  
- Awset starts with 5000 elements, every node removes 1000 elements and wait until it is empty.  
-Two versions of each case is run, joining before or after removing elements.

* LaunchSet3.sh: This is exactly similar to LaunchSet1.sh with nodes adding 10,100,1000 elements but on a cluster of 10 nodes, thus nodes wait to detect respectively 100, 1000 and 10000 elements.
* LaunchSet4.sh: This is exactly similar to LaunchSet2.sh with nodes removing 10,100,1000 elements but on a cluster of 10 nodes, thus starting with CRDT of respectively 100, 1000 and 10000 elements and waiting to detect it is empty.
* LaunchSet5.sh: This is exactly similar to LaunchSet1.sh with nodes adding 10 or 100 elements but on a cluster of 20 nodes, thus nodes wait to detect respectively 200 and 2000 elements.
* LaunchSet6.sh: This is exactly similar to LaunchSet2.sh with nodes removing 10 or 100 elements but on a cluster of 20 nodes, thus starting with a CRDT of respectively 200 and 2000 elements and waiting to detect it is empty.

These scripts are all based on the same structure and allow to easily launch a full session of measurements testing different parameters values with relatively high iteration in an automated way allowing for it to conveniently run during hours, for example at night.

### 2.3.2 Dynamic measurement scripts

Previously described scripts were only measuring convergence time for a single-fire operation to reach every node. This model does not allow to measure dynamic scenarios where a CRDT value is constantly updated by an active source. To allow this new scenario, a totally different approach was adopted. Nodes update continuously the CRDT value (adding or removing element for example on a regular time basic, e.g. few ms) and write to a file the operation done together with their own local version of the CRDT very frequently along with timestamps. In other words, there is actually no measurement during the running itself, just information dump to files. All the measurement is done afterward with another script that reads all the output files and, for every added/removed element from a node, checks when that element is present in all the other nodes states. Since every element addition/removal and every CRDT state is accompanied by timestamps, the analyse script can compute convergence time afterwards by analysing all the output files. Again, more detailed information such as the repertories where output files are written are presented inside the different readme files on the github repository.

* LaunchSet7.sh: This script launches a cluster of 5 nodes that will share an awset. Every node will remove an unique element and add another one (the CRDT number of elements eventually stays the same) on a regular time basis while outputting to files the added element, removed element and current awset local state. The iteration is not, as previously done, handled by killing the cluster and restarting it again for next iteration but by letting it run longer to allow more elements addition/removal and thus more measurements. Here are the update speed tested:  
  - One element is added/removed every 0.5 sec (2/sec).

- One element is added/removed every 0.25 sec (4/sec).

- One element is added/removed every 0.05 sec (20/sec).

- One element is added/removed every 0.01 sec (100/sec).

Remark: These values are CRDT update speed per node, in other words the 100/sec version on a 5 nodes cluster is equivalent to a 500 CRDT updates per second on the cluster.

* LaunchSet8.sh: This script is exactly similar to LaunchSet7.sh but on a cluster of 10 nodes, reaching a maximum update speed of 1000 updates per second on the cluster.

### 2.3.3 Quality-of-life scripts

Some other scripts were developed for convenient little purposes not directly related to some specific measurements but useful mainly for local measurements or testing. While very simple, they can be a first step to launch some nodes locally on a computer to manually test little scenarios, test the new developed tools or simply get accustomed to the lasp API. As previously mentioned, it might be required to write your IP address or 127.0.0.1 (localhost) in a file designed for that purpose. More details are available within the github repository readme files.

These little handy scripts include:

* LaunchBasicNode.sh that simply launches a local node that does not nothing special but can be used for little manual tests.
* LaunchBasicCluter5.sh that simply launches a cluster of 5 nodes that does nothing automatically but allow manual tests on the cluster. Also present in version LaunchBasicCluster10.sh for a 10 nodes cluster.
* Clean\_measures.sh that simply removes every trace from previous measures. This is normally automatically run when starting a new measurement script. While handy, this means if you want to run the measurements scripts, you must save the outputs or result files in a remote folder to avoid deleting them.
* Recompile.sh that simply recompiles the entire lasp code including CRDT types, partisan (communication layer), lasp core modules, etc. This should not be used unless you want to modify lasp code and test your modifications.
* LaunchLeaderElection5.sh is a simple script that launches a cluster of 5 nodes that tests the leader election protocol that is used for the continuous measurement tool. Basically, it is just a debug tool to check that the leader election works correctly. It is also available in 10 nodes version with LaunchLeaderElection10.sh. If this does not work smoothly, verify that you entered your IpAddress (or 127.0.0.1) in the related file (see readme files).
* LaunchContinuousMeasurementSilent5.sh launches a cluster of 5 nodes that run continuous measurement in background. In other words, it is a real example of the developed tool that runs on a cluster that you can use for tests, modify convergence time and check the impact on measurement in a dynamic way. See section 2.1 and 2.2 for more details on the API to use. This is also available in 10 nodes version with LaunchContinuousMeasurementSilent10.sh.
* LaunchContinuousMeasurementTalkative5.sh launches a cluster of 5 nodes that run continuous measurement under debug mode. While not practical due to all the prints, it is very interesting to analyse to understand in real time the measurement tool principle.

## 2.4 Lasp additions

Since Lasp is not (yet) a general public commonly used tool, it might be a bit complicated to approach at first sight mainly due to the very limited documentation. The scope of what it allows is actually very wide and much bigger than what is discussed in the little official documentation available at <https://lasp-lang.readme.io/docs>. Discovering all the implemented modules, the available functions and already implemented tools was a good surprise, but any experimental system necessarily has its few flaws that have not yet been explored. Therefore, I faced from time to time little issues that I tried to address within my work. Among these, some were because of my own mistakes or misunderstandings while a few were simply because of apparently undiscovered bug.

### 2.4.1 Memory leak

One strange thing that showed up while running continuous measures was the fact convergence time had a slow trend to become bigger (slower) with elapsed time. The fact letting a little cluster run locally for a certain time (around 30 minutes) was making my computer totally crash due to non-available memory was the trigger for my doubts. Firstly, thinking the problem was because of one of my code, it quickly became obvious it was not the case. Indeed, a simple local cluster of 5 nodes, initiated exactly as detailed in the official instruction (official Lasp github or documentation) on a clear just cloned repository (clone from official lasp github) was causing the same issue when running for too long. This was causing my computer to crash after 25-30 minutes even if the cluster was not doing anything. For example, a cluster was initiated, shared an awset with one single element then did not update anything and still ended up crashing.

Since it was obvious there was there a real issue, I wrote a little script to measure process memory size while the nodes were running. A particular process (lasp\_ets\_backend\_storage) was growing indefinitely. After checking in detail what this process was doing, it was found that it was updating the awset state in local memory (via ets table). The issue there was the fact at every iteration, the stored record was growing. Indeed each time it received a state, it stored the new local state while adding a node reference inside the record, slowly making the record becoming huge while it did not store any new useful information since it was literally recording the same node reference again and again (exact redundancy of a same reference hundred times in a single record being stored). While relatively hidden initially, this problem became very visible after modifying the state\_interval with the tool described in section 2.2. Indeed, since the nodes received states much more often while still storing increasing size records, the process size was growing much faster. For example, when putting the state\_interval to 100ms instead of the initial 10000ms (which was the initial default value hardcoded in Lasp), my computer was crashing after only a few minutes (2-3 minutes).

All these strange behaviours and the fact purely redundant information were stored pushed me to believe it was purely a bug in Lasp implementation or at least something was badly configured in the most recent github version. This is why I opened an issue talking about this: <https://github.com/lasp-lang/lasp/issues/310>. While still haven’t got any answer about it, I tried a little solution which seems to have done the trick: when lasp\_ets\_backend\_storage tries to store a record, it checks it to avoid redundant information inside it.

This simple modification allowed the process memory size to stay smaller than 0Mb where previously starting at 0 Mb it escaladed quickly to hundreds Mb. This was also translated into convergence measurements being more constant where previously it was progressively slowing down (obviously due to the process eating all the memory).

While haven’t received any feedback on this from Lasp developer yet, it is obvious storing an increasing size record containing purely redundant information (which are more and more redundant on each iteration) seems strange or at least very non-optimal.

### 2.4.2 Readme improvement

One initial issue I had when starting to work with Lasp was about clustering remote nodes together. Indeed, running a cluster of local nodes on a single machine was very easy but making remote nodes communicate together was initially not working correctly. This was due to different factors, including the fact the server I was running nodes on (INGI virtual machine) was initially not configured to allow entering connections but also due to a lack of setup instruction concerning Lasp.

Indeed, after correctly configuring firewalls, nodes were still unable to correctly initiate communications due to another security issue related to Lasp this time. Indeed, nodes were rejecting communications because it was required to share an erlang cookie beforehand on every remote machine. This was actually not mentioned anywhere in Lasp documentation or github readme files.

Since it made me lost a bit of time stupidly, I thought it would be a good idea to add this little detail in the readme file where instructions were detailed to launch a cluster. Here is the related pull request: <https://github.com/lasp-lang/lasp/pull/311>.

# 3. Measures and results

This chapter will present the measurements and results together with some analysis. As detailed in chapter 1, this work uses the Lasp awset CRDT (named orswot in general literature) as use-case and measures impact of various parameters on the convergence time of a cluster together with the network usage (in term of messages per sec). As a reminder, the awset is a CRDT that acts like a set where nodes can add or remove elements allowing a wide range of usages, for details on orswot/awset please refer to section 1.3.2.

## 3.1 Parameters

The parameters that are modified to see the impact on convergence time are:

* Cluster size
* Geographical distance (at a small scale)
* Nodes under partition
* CRDT size
* CRDT operation
* CRDT value update speed

The adaptation tool described in section 2.2 is also tested by measuring its impact on convergence time and network usage. In other words, the time interval between node states sending acts like a 7th parameter being tested.

## 3.2 Measures

The number of iterations for every experiment is always 50. The majority of the measurements were executed with the scripts described in section 2.3. While all the measurements are not present on this work github page, some examples are (in folder mylasp/lasp/Memoire/Saved\_measures). More details are available there within readme files to explain how to run these scripts again by yourself.

### 3.2.1 Number of Nodes

Here are the results for experimentation with a cluster where nodes put 10 elements each and wait for convergence while measuring convergence time and network usage. The nodes are running under a same wireless network (2 computers running multiple nodes each) with a state\_interval (refer to section 2.2 if you need information on state\_interval) of 10000ms which was the default value in Lasp (as given from Lasp official github).

TODO: Graphe nodes add 10 elements valeurs pour 5,10,20 nodes

We can see that the convergence time increases with the number of nodes which is a relatively expected behaviour…

Just to make sure this measurement is really related to the number of nodes (and not the fact that the number of elements in the cluster is also logically increasing with the number of nodes), let’s verify if it gives the same behaviour if nodes adds 100 elements .

TODO: Graphe nodes add 100 elements valeurs pour 5,10,20 nodes

We can see, even if values are not exactly the same, that the same behaviour came back.

Finally let’s check that this is also the case for elements removal instead of addition:

TODO: Graphe nodes remove 10 elements valeurs pour 5,10,20 nodes

TODO: Graphe nodes remove 100 elements valeurs pour 5,10,20 nodes

### 3.2.2 Nodes distance

### 3.2.3 CRDT size

### 3.2.4 CRDT operation

### 3.2.5 Partition

### 3.2.6 Update speed

## 3.3 Adapted Lasp

Expliquer que les memes mesures ont été réalisées à nouveau mais avec la version adaptée de Lasp sur laquelle j’ai pu modifier le convergence time. Par exemple, on a pris par défaut un interval de 100ms plutôt que l’initial 10000ms.  
Expliquer que toutes les mesures décrites plus haut ont également été réalisées avec un interval plus court.

Expliquer que dans cette partie tous les résultats vont être dévoilés.  
Toutes les mesures ont été réalisées sur base du Lasp initial (tel que fournis sur le github officiel) puis également sur une version pour lequel le state\_interval a été réduit à 100ms.  
Ensuite une analyse sera faite.

## 3.4 Analysis

Résumer ce qu’on a appris sur base de tous les graphes.  
Revenir sur le fait que globalement, l’API pour modifier le convergence time fonctionne.  
Essayer d’expliquer ce qui pourrait (faire des hypothèses) expliquer telle ou telle comportement (pourquoi c’est plus lent quand il y a plus d’éléments par exemple ?).

# 4. Conclusion

## 4.1 Summary

Rappeler donc ce qu’est initiallement Lasp. Ce que j’ai apporté à Lasp. Ce que j’ai mesuré et découvert.

## 4.2 Analysis conclusion

Expliquer ce que les résultats semblent dire de Lasp. Plot un graphe montrant la limite de ce que Lasp peut faire (udpate speed en verticale, convergence time en horizontal).  
Expliquer qu’avec l’API il est possible de configurer Lasp selon les besoins si on veut une convergence plus rapide (au risque de surcharger un peu le réseau) ou pas etc…  
Expliquer les différentes sections sur ce graphe.

## 4.3 Future work

## 4.4 Personal opinion

## 4.5 Methodology

Expliquer ce que je pense personnellement de Lasp. Est-ce que je l’utiliserais personnellement. Expliquer que je trouve cela encore assez expérimental (pour le cas de Lasp) bien que prometteur car certaines grandes entreprises utilisent déjà du CRDT, etc…  
Expliquer comment j’ai travaillé, chaque semaine, avec mes réunions hebdomadaires avec Peter Van Roy etc.

# 6. Bibliography

Nouveau1 : article de Peter Van Roy pour la figure IoT vs humanity

1. CAP theorem (trouver une bonne source)

2. CRDT

3. document : crdts\_overview

4. source montrant que Rio Games, TomTom, etc utilisent des CRDTs (info obtenue initiallement via slides).

5. Example CRDT addwins, document crdts-overviews

6. Projet SyncFree (leur site web)

7. Article qui dit que Lasp permet SEC, Availability et Partiton tolerance (bon compromis du CAP theorem).

8. Article qui dit à un moment que les CRDT sont devenus mainstream.

9. SyncFree

10. LighKone

11. Article qui dit que Lasp est conçu pour utiliser un Stream in / Stream out pour faciliter les opérations.

12. présentation de BET365 sur ORSWOT