École polytechnique de Louvain

Visualizing and controlling convergence in Lasp program execution

Authors : **Gregory** CREUPELANDT
Supervisors : **Peter Van Roy**
Readers : **Etienne Riviere, Igor Zavalyshyn**
Academic year 2020–2021
Master [120] in Computer Science and Engineering

Abstract

Lasp is an experimental tool designed to handle distributed variables and based on a principle of data convergence: consistency between replicas of a distributed data structure is achieved by using specially designed data structures, named CRDTs (Conflict-free Replicated Data Types). CRDTs have the property that replicas are consistent when they receive the same updates in any order. This allows an efficient implementation in which updates are simply disseminated between replicas, achieving data convergence without requiring usually heavy consensus algorithm. This innovative approach offers many benefits such as a good scalability and an incredibly powerful compromise from the CAP theorem thanks to that new concept of convergence. This master thesis objectives are to improve Lasp by adding new tools to it and to increase knowledge on Lasp deep functioning and limits by analysing its behaviours and performances regarding multiple parameters. To achieve this, a dynamic tool to measure and modify convergence was developed with a clear API and measurements were run and analysed to effectively put Lasp under test.

Acknowledgements

First and foremost, I would like to thank my advisor, Professor Peter Van Roy, for his advices and methodology, his constant interest in my work, his great availability and his always positive attitude.

I also would like to thank Christopher Meiklejohn, the main Lasp developer, who noticed an issue I faced with Lasp during my work and quickly found the fix.

Thanks also to Vanessa Maons, secretary of the INGI department who always answered to my administrative questions and she alone knows how many questions I ask.

Finally, I wanted to thank my family for their daily support.

Table of contents

Table of figures

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 Christopher Meiklejohn 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) devices[1].
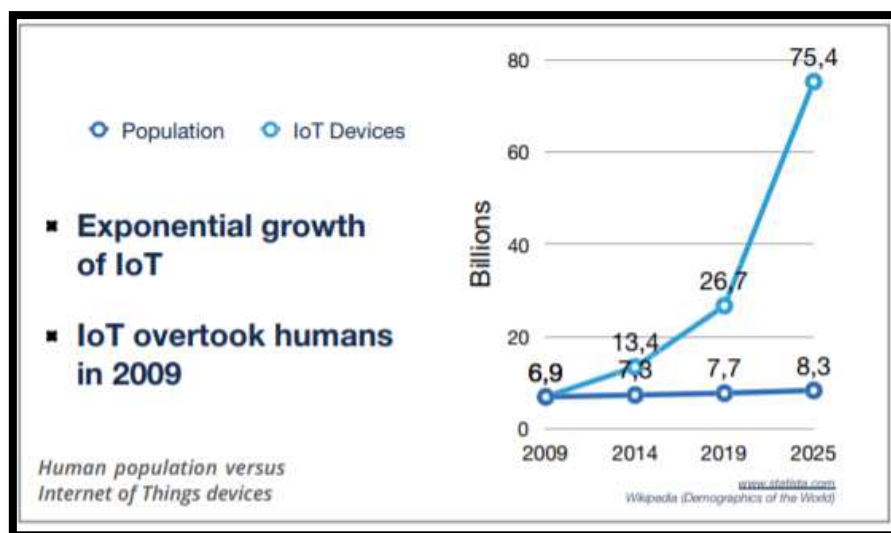
Figure 1: IoT devices vs human population

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 Traditional 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. This Consensus algorithm is a key element for these systems to work correctly but is heavy and complicated to implement correctly. Some improvements are slowly achieved with that approach such as using Raft[2] (which is a lighter implementation of consensus) instead of Paxos[3] (which was used a lot but represented some real challenge to implement). But anyway, starting from a complicated approach and trying hard to improve it… Wouldn't it be more beneficial to consider a new approach?

Finally, since strong Consistency conflicts with Availability and Partition-tolerance (CAP theorem[4]), 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 with that usual approach.
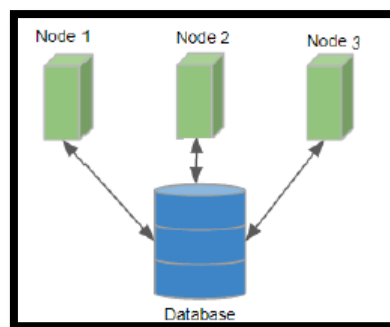
Figure 2: Traditional approach very simple schematic

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 (CRDTs[5]). It is the key concept that will be detailed below to understand this new approach along with all its advantages.
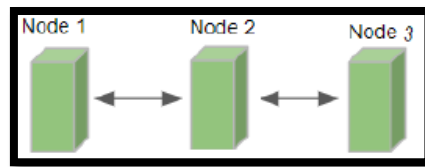
Figure 3: Peer-to-peer approach very simple schematic

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 properties[6]:

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 others[7].

The adoption of CRDTs was quite rapid since it took less than 10 years.

- It was formally defined at an academic level around 2011 by Marc Shapiro, Nuno Preguiça, Carlos Baquero and Marek Zawirski[10].
- Some implementations were developed, notably Basho Riak that was initially developed in 2009 and improved in 2014 to incorporate CRDTs with Riak 2.0 by Russel Brown, Sean Cribbs, Christopher Meiklejohn and Sam Elliott[20].
- With the improvements added to CRDTs, such as the ORSWOT introduction (see section 1.3.2 for details), they are now more and more used in industrial companies.

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 example[6] with an add-wins set (if concurrent add and remove occur, the add wins).
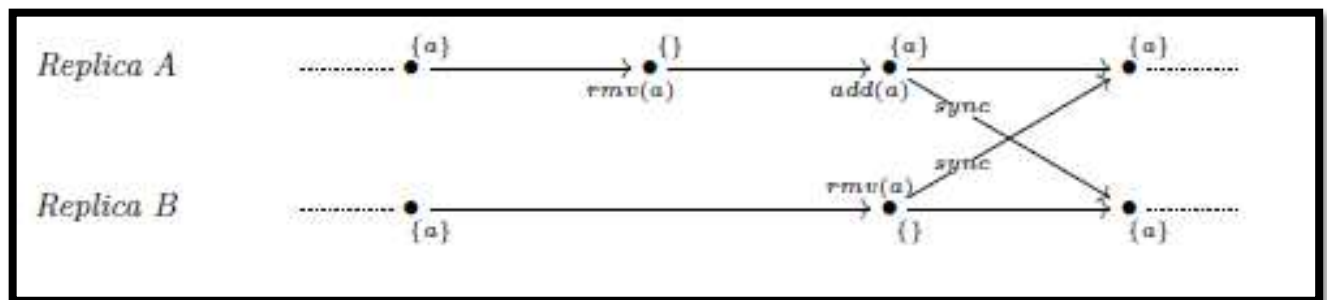
Figure 4: CRDT add-min set schematic example

As mentioned above, the key-concept here is the convergence. It is the fact that, automatically, due to the CRDT metadata, the merge implementation and the peer-to-peer messages that are eventually delivered to every node, 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 only 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 partition tolerance.

- **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 used in this work 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[8]. 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[9] leading to Lasp development).

Strong eventual Consistency (SEC[10]) is achievable due 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 Consistency 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 quickly[11] and has been adopted by some big compagnies as previously mentioned.

1.3 Lasp

Lasp[12] is an experimental implementation of CRDTs developed by Christopher Meiklejohn in the context of the SyncFree[9] and LightKone[13] European research projects. 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.

Figure 5: Lasp official logo

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 (Partisan[14]), 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[15] on CRDTs without requiring the developer to handle these problems himself. 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 these 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 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 (Observed Remove Set[16]), it addressed and resolved many little issues.

The ORSET, which is the previous version and is still used by some 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 Bet365[17] 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 pair. 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.
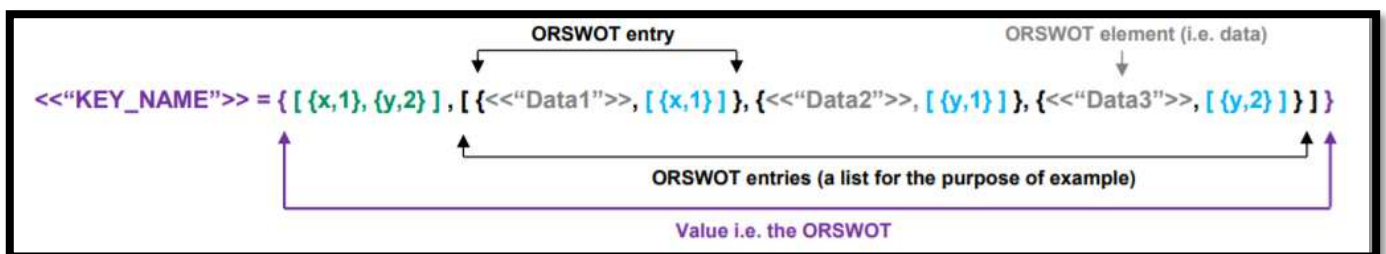
Figure 6: ORSWOT general structure example

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 counter 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.

Example:
    Adding <<"Data2">> using unique actor y to the existing ORSWOT:

$$\{\,[\,\{x,1\}\,]\,,\,[\,\{<<\text{"Data1"}>>,\,[\,\{x,1\}\,]\,\}\,]\,\}$$

Results in the new ORSWOT:

$$\{\,[\,\{x,1\},\,\{y,1\}\,]\,,\,[\,\{<<\text{"Data1"}>>,\,[\,\{x,1\}\,]\,\},\,\{<<\text{"Data2"}>>,\,[\,\{y,1\}\,]\,\}\,]\,\}$$

and ORSWOT value (i.e. ignoring metadata / what a client would be interested in) of:

$$[\,<<\text{"Data1"}>>,\,<<\text{"Data2"}>>\,]$$

Figure 7: Orswot element addition example

- 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).

Example:
    Removing <<"Data1">> from the existing ORSWOT:

$$\{\,[\,\{x,1\},\,\{y,1\}\,]\,,\,[\,\{<<\text{"Data1"}>>,\,[\,\{x,1\}\,]\,\},\,\{<<\text{"Data2"}>>,\,[\,\{y,1\}\,]\,\}\,]\,\}$$

Results in the new ORSWOT:

$$\{\,[\,\{x,1\},\,\{y,1\}\,]\,,\,[\,\{<<\text{"Data2"}>>,\,[\,\{y,1\}\,]\,\}\,]\,\}$$

Figure 8: Orswot element removal example

- 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.
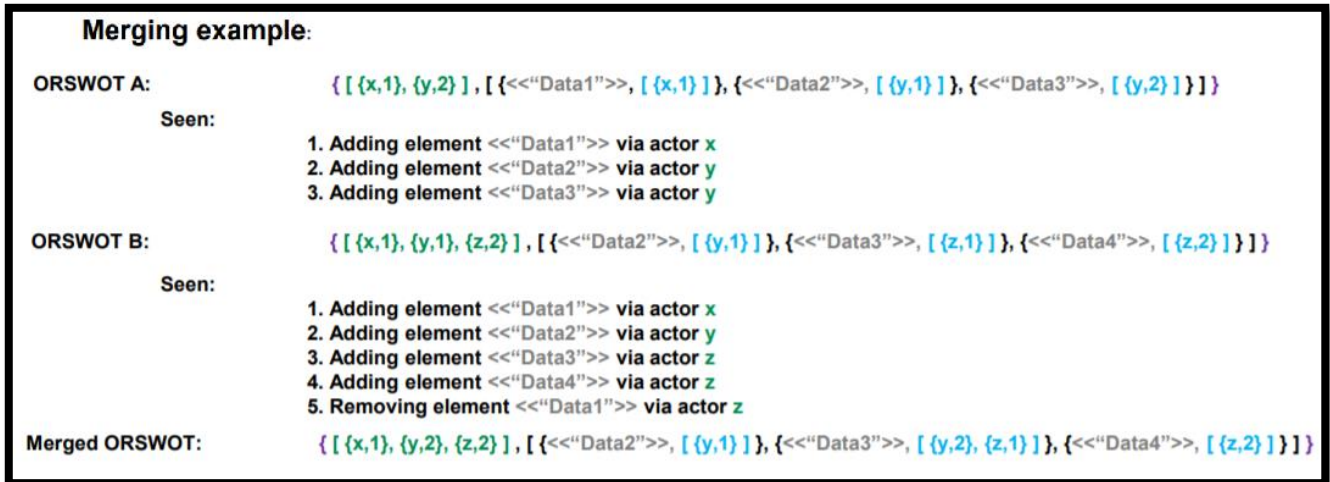
Figure 9: Orswot states merging example

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). One optimisation named delta-state exists but will not be discussed here.

Let's close these explanations 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:
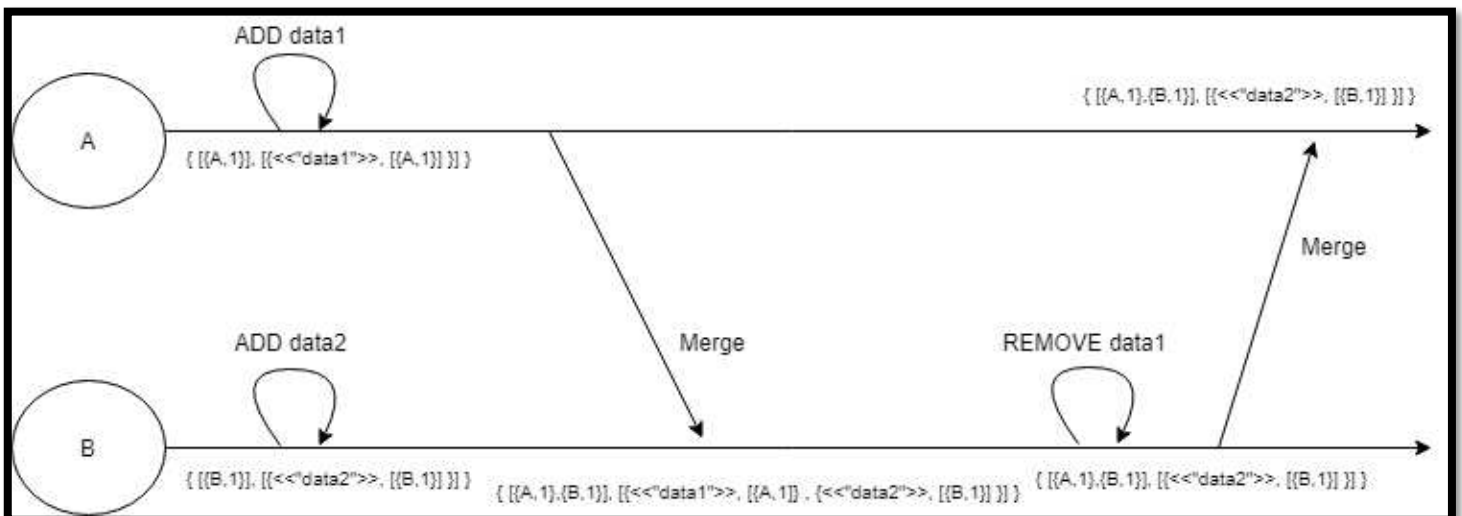
Figure 10: Orswot timeline example

The Orswot, considered as use-case for this entire work, was a great improvement to CRDT potential usages by removing the need for garbage collector for the vast majority of cases. It is one of the important factors that allowed the CRDTs to grow quickly in industries which is the reason we considered important to explain its principle in this introduction.

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 Developed tools and 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 into Lasp to measure convergence time together with network utilisation on the fly.

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 an important part of this work objectives.

1.4.2 Measurements and results

Measuring different cases, testing the newly developed tools and looking at the impact of different parameters 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.

An important task that will be developed later in the work is to measure convergence time and network usage 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 (unable to receive/send messages on the cluster)
- Type of CRDT (orset, orswot, counter, map, boolean…)
- CRDT content (number of elements in a set for example)
- CRDT operation (e.g. removing an element might converge faster than adding 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)
- Continuous updates interval (Nodes might want to update the content of a CRDT extremely frequently)
- CRDT state sending interval (how often nodes send their local state to peers)
- 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 (number of nodes)
- Geographical distance (between nodes)
- Nodes under partition (nodes unable to communicate with the cluster)
- CRDT content (number of elements in the set)
- CRDT operation (element addition or removal)
- Continuous updates interval (how often do node modify the CRDT content)
- State sending interval (How often nodes send their local state to peers)

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 measurements, to explain the results and ideally to better understand Lasp or even to find its 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 content 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 core use case. The contributions to Lasp will be presented in the next chapter (chapter 2) 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 measurements and results will be presented and analysed. Finally, chapter 4 concludes with some summary about the developed tools, the observations, some future work propositions, a personal opinion on Lasp and the general methodology followed during this work.

2. Developed tools and API

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

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 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 (I advise to launch it on boot)
- A leader election is run on the cluster
- The leader puts an element (acting like a measurement signal) on a specific CRDT (awset)
- Other nodes detect the element and answer with notably 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 on the github page of this work within 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 in awset A and put their {Id, TimeStamp and number of messages received per second since last measurement} on a specific awset B.
- Leader detects that every other node answered via awset 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 (I advise to launch it on node booting) 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 an under use CRDT shared on the cluster but mimics it.

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 desired number of ms between each measurement round. Actually, this will not be the real period between each measurement since the code will adapt the period (add an offset) to allow better precision (see section 3.8 for more explanation) while trying to be close to that desired period between each measurement round.
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 :


```
(node5@127.0.0.1)1> lasp_convergence_measure:launchContinuousMeasures(10000,3000
0,false).
Continuous Measures Started in Silent mode
<0.10634.0>
(node5@127.0.0.1)2>
```

Figure 11: launchContinuousMeasures debug=false real example

Example usage, Debug=true :

Figure 12: launchContinuousMeasures debug=true real example

While not practical for real usage due to all the prints, this allows to really well understand and visualize in real time the measurement tool principle. Indeed, we can clearly see which node acts as a leader, the fact it correctly detected the number of nodes reachable, when it puts a measurement signal and

the fact it waits for the answers. Other nodes answering is also visible then the leader prints the results and start resetting the measurements for next round. While understandable on this images, it can be very interesting to launch this in practice to see the real-time prints.

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 :

```
(node2@127.0.0.1)2> lasp_convergence_measure:getSystemConvergenceInfos().
#{individualConvergenceTimes =>
      [#{convergenceTime => 128,id => 4},
       #{convergenceTime => 489,id => 2},
       #{convergenceTime => 578,id => 5},
       #{convergenceTime => 590,id => 3}],
  networkUsage => 185,roundTripTime => 967,
  worstConvergenceTime => #{convergenceTime => 590,id => 3}}
(node2@127.0.0.1)3>
```

Figure 13: getSystemConvergenceInfos real example

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 :

```
(node4@127.0.0.1)1> lasp_convergence_measure:getSystemConvergenceTime().
634
(node4@127.0.0.1)2>
```

Figure 14: getSystemConvergenceTime real example

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 :

Figure 15: getSystemWorstNodeId real example

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 :

Figure 16: getSystemRoundTrip real example

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 :

```
(node3@127.0.0.1)5> lasp_convergence_measure:getSystemNetworkUsage().
193
(node3@127.0.0.1)6>
```

Figure 17: getSystemNetworkUsage real example

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 :

```
(node4@127.0.0.1)1> lasp_convergence_measure:getIndividualConvergenceTimes().
[#{convergenceTime => 154,id => 3},
 #{convergenceTime => 419,id => 4},
 #{convergenceTime => 440,id => 2},
 #{convergenceTime => 495,id => 5}]
(node4@127.0.0.1)2>
```

Figure 18: getIndividualConvergenceTimes real example

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 :

Figure 19: getConvergenceTime real example

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 3 (measures and results), Lasp as directly cloned from the official repository is actually very slow. Measurements 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 parameterization 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 10000 ms, 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 on the fly via an API.

2.2.1 Principle

Instead of taking a configuration file hardcoded value for the time interval triggering the state sending, 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 time 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 it is guaranteed they will eventually adapt. The exact implementation 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. The fact CRDT guarantees strong eventually consistency is the main factor that allowed to easily implement this task by putting the state_interval inside its own CRDT which directly allows to update the state_interval in a globally consistent way on the entire cluster.

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) 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 10000ms (and thus a convergence time mean around 10 sec) and a node wants the cluster to converge faster, modifying the state_interval to 1000ms, it may take approximately 10000 ms before adopting that same state sending interval 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 :

```
(node4@127.0.0.1)1> lasp_convergence_measure:setStateInterval(100).
state sending interval set to: 100 ms.
ok
(node4@127.0.0.1)2>
```

Figure 20: setStateInterval real example

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 :

```
(node3@127.0.0.1)1> lasp_convergence_measure:getStateInterval().
100
(node3@127.0.0.1)2>
```

Figure 21: getStateInterval real example

An interesting approach allowed with this newly developed API 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 not extremely precise, it is very easy to use and allows convergence visualization and notifiable modifications from the end-developer which is exactly one of this master thesis goals.

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 19 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 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…). Here comes a descriptions of the different script.

2.3.1 Static measurement scripts

These scripts are designed in a very simple fashion 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 awset…).

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 awset 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 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 files. All the detailed information such as where a written the outputs files, where is written the result files at the end of the script etc… are available and described in depth within the readme files on the github page of this work. 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 the cluster 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 (e.g. awset content) is constantly updated by an active source. To allow this new scenario, a totally different approach was adopted. These new scripts launch a cluster where 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 dumped 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 in the files, 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 number of elements eventually stays the same in the CRDT) on a regular time basis while outputting to files the added element, removed element, timestamp 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 intervals 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 interval on each 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 case 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 local 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 previous 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 node1@IpAddress (with your IP address 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 it visually to understand in real time the measurement tool principle.

- Analyse_static.sh is a script used to analyse measurements from a static experiment (such as experiments launched by scripts described in section 2.3.1). The outputs files are normally analysed automatically at the end of the experimentation script but if you stopped that experimentation script before it ended and want to analyse the already outputted files, you can use this. Refer to the readme file on the github page of this work for more details.

- Analyse_dynamic.sh does the same as Analyse_static.sh but for dynamic experimentations (their outputs are different) which are launched by LaunchSet7.sh script.

2.4 Lasp corrections

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 (which is not up to date). 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 worries. 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 doing next to nothing. 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 (every state_interval, see section 2.2 for details on this value), 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 master version. This is why I opened an issue talking about this: https://github.com/lasp-lang/lasp/issues/310.

After receiving feedback from Lasp main developer, Christopher Meiklejohn, it was found that Lasp code was updated without updating the documentation accordingly. Actually, the documentation from https://lasp-lang.readme.io/docs and also the readme file from the official github were showing example usages with wrong argument types. While it could sound like a very little detail, it was actually hard to detect since the system was behaving normally with the readme examples on the short run but only showed its issues when running longer (or with shorter state_interval).

After acknowledgement of the issue from the main developer, who quickly found the solution, he modified himself the readme file together with some other codes files that were using wrong type of argument. For more details, the issue was from the update method that is used to update a CRDT content. Indeed, the implementation was modified with time and the last argument that represents the source of the update was expected to be a binary while it was not the case within the readme example nor within some other files causing real issues when running Lasp on the long run. More details with the two following links: https://github.com/lasp-lang/lasp/pull/312 and https://github.com/lasp-lang/lasp/pull/313.

When changing the concerned argument to be a binary, the problem directly disappeared. Indeed, the stored records stopped increasing in size, the process memory size stopped growing and the convergence times measuring on a cluster running longer stopped increasing.

While it may not be a big element, the issue made me lost some significant time initially not understanding where the problem came from. I hope this clarification may help future users or future students that may work with Lasp on their master thesis.

2.4.2 Readme improvement

One initial issue I had when starting to work with Lasp (first weeks) 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 which is the reason why I did not immediately find the solution.

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. Measurements 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 on the entire cluster). 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). It is important to note that the absolute values that will be presented are not of high importance while the variations according to the parameters are what we are interested in. Indeed, the above measurements have very big standard deviations which is inherent to Lasp utilisation as it will be highlighted in section 3.8 which details why. This means, the results should not be used in the case of benchmarks by any case but simply for behaviour analysis.

3.1 Parameters

The parameters that are modified to see the impact on convergence time (time between the moment an element is modified on the awset CRDT and the moment other nodes detected that modification) and network usage (in term of number of messages per second on the cluster) will be detailed further in each respective section but here is a summary of the measurements that will be presented:

- Cluster size :
  Measurements will be run on clusters with different number of nodes.
    - 5 nodes
    - 10 nodes
    - 20 nodes

- Geographical distance:
  Measurements will be run on clusters with different setups.
    - Nodes locally on the same computer
    - Nodes on multiple computers on the same wireless network
    - Nodes on multiple computers on remote networks

- Nodes under partition:
  Measurements will be run on clusters where nodes are temporarily unable to communicate with other nodes.
    - Nodes are under partition when updating their local state then resolve partition to see the impact on the cluster
    - One node is under partition while other nodes converge on a value then the partition is resolved to see the impact on the previously partitioned node

- CRDT content:
  Measurement will be run on a cluster where nodes will add or remove a specific number of elements (each node will add a same number of elements, doing symmetrical load), making the awset CRDT to eventually contain a predefined number of elements.
    - 50 elements
    - 500 elements
    - 5000 elements

- CRDT operation:
  Measurements will be run on a cluster where nodes do different operations (each node will do the same type of operation, doing symmetrical load).
    - Nodes add elements
    - Nodes remove elements

- Continuous updates interval:
  Measurements will be run on a cluster where nodes want to modify the content of the awset CRDT at various intervals (again, every node doing the same operation at same intervals, doing symmetrical load).
    - One element modification every 0.5 second
    - One element modification every 0.25 second
    - One element modification every 0.05 second

- State sending interval :
  Measurements will be run on a cluster where nodes send their state to peers on various time intervals.
    - Nodes send their state every 10000 ms
    - Nodes send their state every 5000 ms
    - Nodes send their state every 1000ms
    - Nodes send their state every 500 ms
    - Nodes send their state every 100 ms
    - Nodes send their state every 50 ms
    - Nodes send their state every 10 ms

The number of iterations for every experimentation is always 50. Every experimentation other than the continuous updates interval (section 3.8) are using single-fire operation, meaning nodes will do an operation then wait for convergence while doing nothing (using scripts from section 2.3.1). Only the experimentation on continuous updates interval (section 3.8) will use a different approach (using scripts from section 2.3.2). The ORSWOT CRDT that was described in section 1.3.2 corresponds to the awset implemented in Lasp that is used as use-case for all the measurements. It is important to node an improved version has been developed, using delta-states instead of full-states where only the modifications to the local state is sent to peers. While this optimization may be interesting to analyse, the awset used as use-case for this work does not use delta-state optimization. This can be translated

in the fact the measurements for high number of elements in the awset content may reflect worst case compared to the potential optimization.

The results presented on graphs in this paper will show mean and standard deviation while other raw measurements are available on the github webpage of this work (in folder mylasp/lasp/Memoire/Saved_measures). More details are available there within readme files notably to explain how to run the measurements scripts again by yourself locally if you wish to.

Remark 1:
While the 6 first parameters being tests (section 3.3 to 3.8) are elements that could be seen as constraints that Lasp should handle to correctly run a real application (which impose a number of nodes, a probably short time interval between content updates, potential high geographical distance between nodes, etc...), the last parameter, the state sending interval, can be seen as a parameter where we have much more control.

Indeed, while it was hardcoded in Lasp configuration file, the state_interval can now be modified with the tool developed in section 2.2 and could, ideally, be directly adapted by Lasp on the fly in the future to allow automatic adaptation (see future works section 4.4). This means, its impact on convergence time is particularly interesting.

Remark 2:
It is important to understand well the difference between the continuous update interval and the state sending interval:

The continuous update interval is related to the rate at which the application running Lasp on a node wants to modify the CRDT content. For example, each node may want to modify the awset CRDT content (add an element for example) every second, which will be considered as a 1 second continuous updates interval (the experimentation makes every node act the same way with symmetrical load).

The state sending interval is related to the rate at which the node sends its awset CRDT local state to peers. For example, even if an application tries to update an awset content every second, the node may be configured to send its state to peers only once every 10 second, which translates as a 10 second state sending interval (aka state_interval).
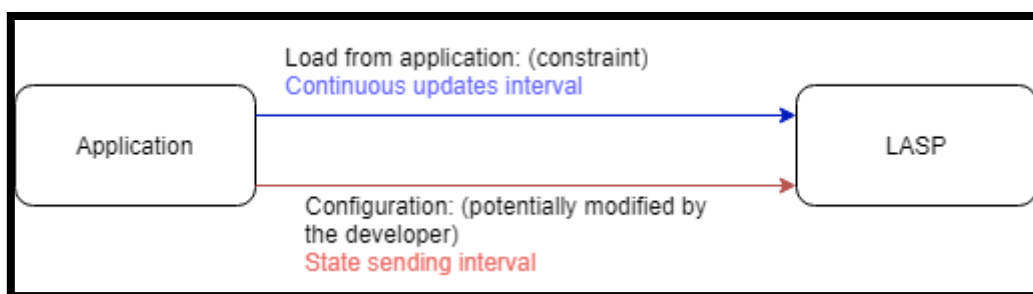
Figure 22: Difference between Continuous updates interval and State sending interval

3.2 Orchestration of experiments

Prior to the different measurements and results, it may be interesting to mention one difficult task that was implemented for some of the following measurements was the orchestration part. Indeed, some experimentations were run locally on a single computer which allow some easy orchestration, but some other experimentations were running on multiple devices and required more complex orchestration.

The orchestration used for such cases where multiple nodes on multiple devices had to start the measurements at the same time is the following:

- Create the cluster (some nodes on local computer, some nodes on UCLouvain ingi servers).
- Run the leader election protocol that was initially implemented for measurement tool (section 2.1.1).
- The leader is in charge for the global start of measurements. Leader puts a timestamp in a specific CRDT (awset of one element specifically declared for that purpose). This timestamp corresponds to a future unix time (actually leader unix time + 30000ms) and acts as the real starting moment for the experimentation.
- Nodes gets the leader timestamp and loop checking their unix time every 1 ms. When detecting it reached the timestamp value, node starts experimentation.
- Nodes do an operation each (example: adding 10 elements on the awset CRDT).
- Nodes measure time and received messages while waiting for convergence (the final CRDT state is predefined. For example, 5 nodes adding 10 elements each will wait to detect a final state with 50 elements).
- Nodes detected convergence and send a signal (on another specific CRDT) to let other nodes know it finished.
- Nodes detect every node finished and write measurements to output file.
- Everything is shut down and start again for new measurement iteration.

It is obvious this orchestration method is not perfect, indeed it has three flaws, it assumes the start time value will converge on the cluster within a specific period (30 seconds, which was a deliberately big value for such small clusters), it assumes the different nodes have the same exact unix time which is not always perfectly true (little shifts are possible) and finally it makes the nodes loop every 1ms until starting the experiment which adds a measurement error up to 1ms. But still, while not perfect, the described orchestration was precise enough to make the nodes start measurements at the (relatively) same time especially when comparing to measures absolute values which are of the magnitude order of seconds not ms anyway.

Now that the orchestration part was detailed, let's present the results.

3.3 Number of nodes

This section will present the measurements details and results together with an analysis looking at the convergence time as a function of the number of nodes in the cluster. A conclusion proposal about that parameter will end the section together will small analysis remarks.

Below 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 cluster with a state_interval (refer to section 2.2 if you need information on state_interval) of 10000 ms which was the default value in Lasp (as given from Lasp official github) and was used for the vast majority of the measurements below. The cluster is always run based on 3 computers, the exact setup is the following:

- 5 nodes: 2 computers running 2 nodes each and one last slower (raspberry pi) with 1 node.
- 10 nodes: 2 computers running 4 nodes each and one last slower (raspberry pi) with 2 nodes.
- 20 nodes: 2 computers running 9 nodes each and one last slower (raspberry pi) with 2 nodes.

One of the computers and the raspberry Pi are on the same wireless network while the last computer is on a remote network (server from UCLouvain ingi).

Here are the measured results with the different cluster size (number of nodes):

Figure 23: Graphs convergence time and network usage for different cluster size (nodes adding 10 elements each, state_interval of 10000ms)

Analysis:

Clearly, we see the convergence time (defined as the time between an element addition and the moment other nodes detected it) is increasing with the number of nodes. Same goes for the number of messages per second on the cluster. When looking closer to the convergence time and consider it as a function only of the number of nodes, the results seems to be approximated by:

$$Convergence\ time\ \approx 3850 * \sqrt{cluster\ size}\ \text{(ms)}$$

While it is obvious the absolute value here has no importance since it is mainly related to the experimentation protocol and precision, the fact it seems to grow as a square root is promising.

When looking closer to the number of messages per second on the cluster and consider it as a function only of the number of nodes, the results seems to be approximated by:

$$Number\ of\ messages\ per \sec\ (on\ the\ entire\ cluster)\ \approx 0.47 * (cluster\ size)^2$$

Again, the absolute value here has no importance, but the fact it seems to grow as a square function of the number of nodes is interesting. It is important to note that the results represent the number of messages per second on the entire cluster, if we divide by the number of nodes, we can get an approximation of the network usage per node. This value then looks much more linear with the number of nodes in the cluster. It is also important to note this experimentation involved one unique operation performed by each node, which means if we look at the network usage (number of messages per second) per node and analyse it as a function of the number of operations on the cluster, we get a constant (approximately 0.47). Based on these results, the network load per node per operation tend to be approximated by a constant which sounds promising.

One little drawback from these first results is the fact the experimentation protocol while trying to only modify the number of nodes also modified another parameter. Indeed, where 5 nodes adding 10 elements each would wait to detect 50 elements, a cluster of 10 nodes where nodes add 10 elements each would wait to detect 100 elements. And 20 nodes would wait to detect 200 elements. This means the awset content, while not being the parameter under test for this experiment, is a changing parameter that might impact the results. To verify the results observed were mainly related to the number of nodes and not to the CRDT content, same measurements were run again but with nodes adding 100 elements each instead of 10 to verify the behaviour is the same.

Here are the results for the same experiment but where nodes add 100 elements each instead of 10:
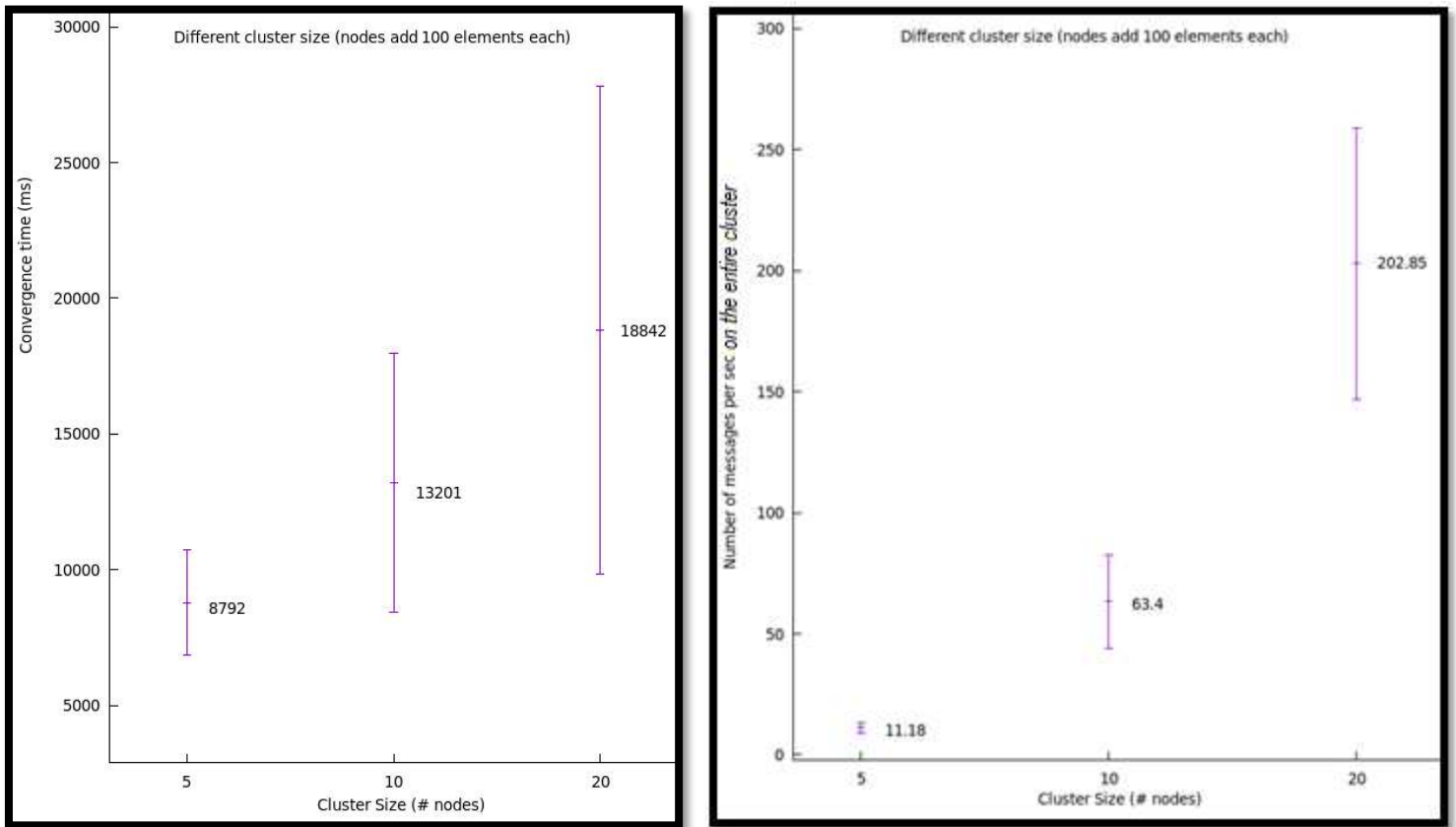
Figure 24: Graphs convergence time and network usage for different cluster size (nodes adding 100 elements each, state_interval of 10000ms)

Again, with these new measurements, the same behaviour shows up where the convergence time seems to grow approximately as a square root function of the number of nodes and the number of messages per second as a square function. This confirms the results show mainly the impact due to the number of nodes and not the impact due to the CRDT content.

Finally, before concluding with this parameter and adding some little remarks, let's see if the removal operation is impacted the same way by the number of nodes in the cluster. Indeed, all the presented measurements were using elements additions, but elements removals might not be impacted the same way by the cluster size.

Here are the results for the same experiments but with elements removals instead of additions:
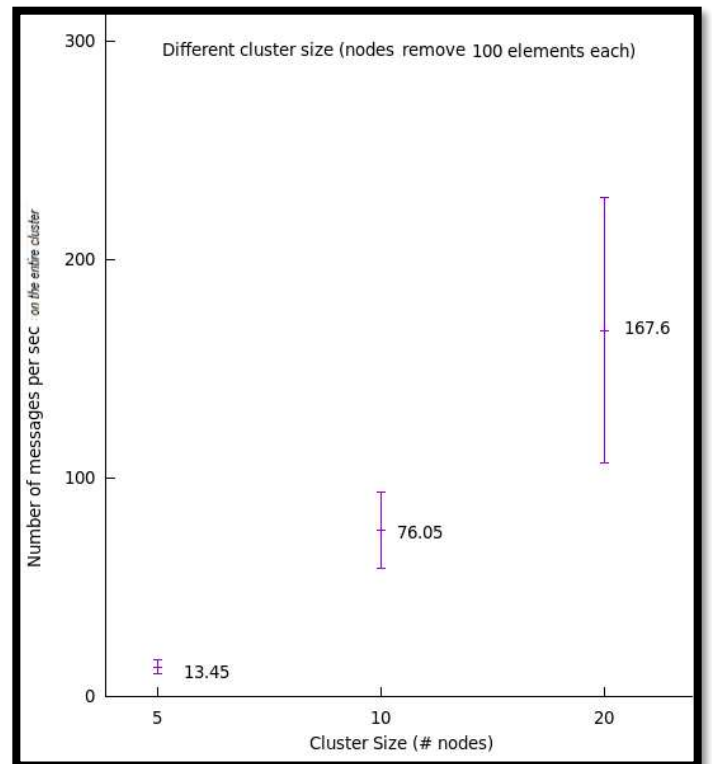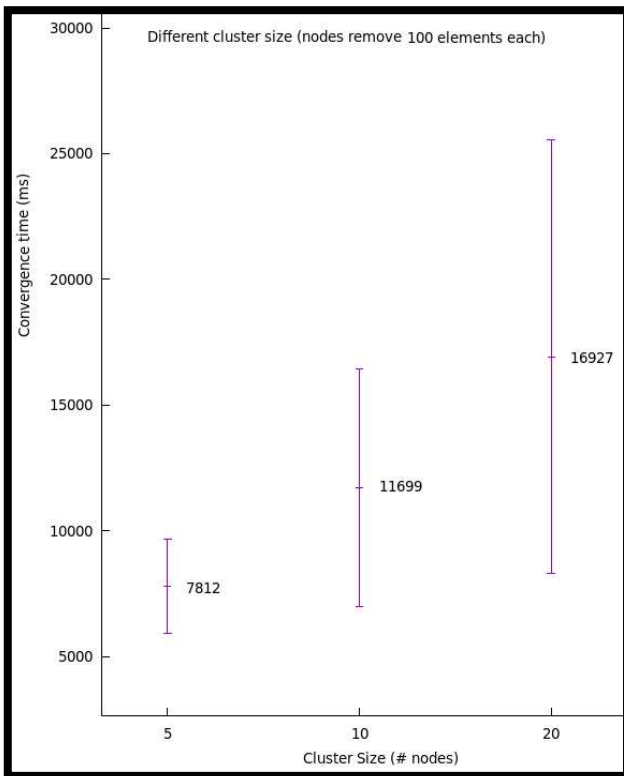
Figure 25: Graphs convergence time and network usage for different cluster size (nodes removing 10 and 100 elements each, state_interval of 10000ms)

Clearly, while the absolute values are slightly different, we see that the element removals operation convergence time and number of messages per second are impacted by the number of nodes the same way as element additions were. Showing the previous analysis is relevant for element additions but also for element removals.

Conclusion:

The presented results seem to be consistent and give a strong intuition the convergence time could be approximated as a square root function of the number of nodes. Regarding the network usage, the number of messages on the entire cluster could be approximated as a square function of the number of nodes but more importantly, the network workload (number of messages per second) per node per operation could be approximated by a constant. These conclusions push the idea Lasp is well scalable with regard to the number of nodes.

Remark 1:
Looking at the results graphs while having in mind the state_interval value (which is 10000ms), allows to notice the convergence time is of same order of magnitude as the state_interval which sounds logical.

Remark 2:
Not unlimited computer resources were available for this work. This means it was not possible in the context of this master thesis, when running a cluster of 20 nodes, for example, to have 20 independent devices which would run a single node each. Instead, a limited number of computers were running the nodes as detailed in the first paragraph of section 3.3. This obviously is not ideal since while modifying the number of nodes (intended modified parameter) it also modified the workload on hosts (since the computers were running more nodes). The ideal case to measure the variation, for example, between a 5 nodes, 10 nodes and a 20 nodes cluster would be to have up to 20 independent same devices with the possibility to use only 5 of them, 10 of them or the 20 of them (thus always one node per device) which would probably give more reliable and precise results. In regard of this aspect, it is important to note the experimentation here necessarily worsens the results by making them a bit worse for bigger clusters. This means the convergence time for a real 20 nodes cluster would probably be better than what was measured in the previous graphs.

3.4 Nodes distance

After checking the impact of the number of nodes, another important parameter in regard to the nodes is the distance between them. To measure this, three different nodes setup where tested:

- Only local nodes (on a single computer)
- Nodes under a same wireless network (small distance)
- Remote nodes (nodes on local computer and nodes on a remote network which is around 20 km distance)

Let's see the results with a cluster of 5 nodes, each putting 10 elements in an awset and measuring convergence time and network usage (in term of number of messages per second on the entire cluster as previously).
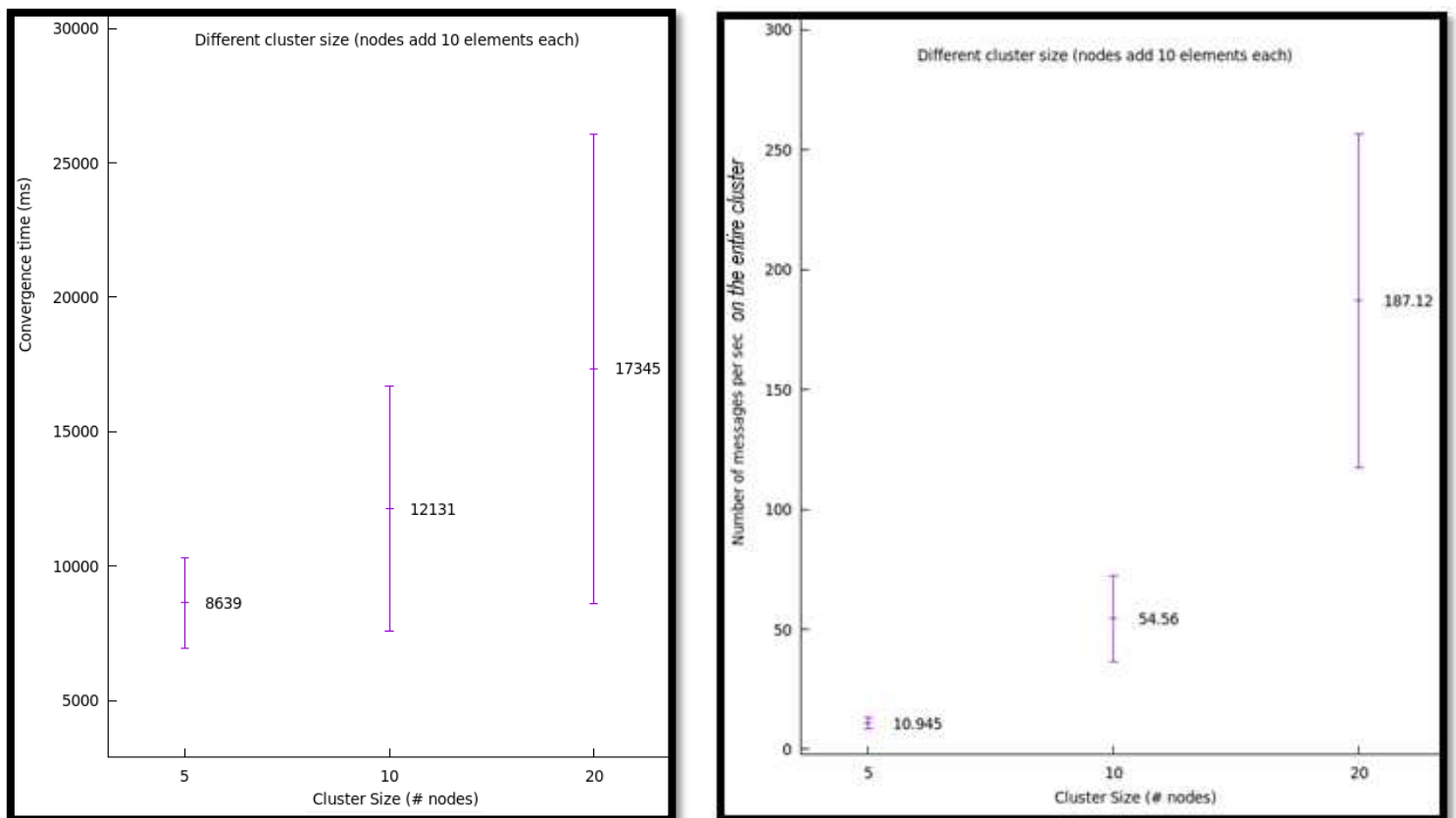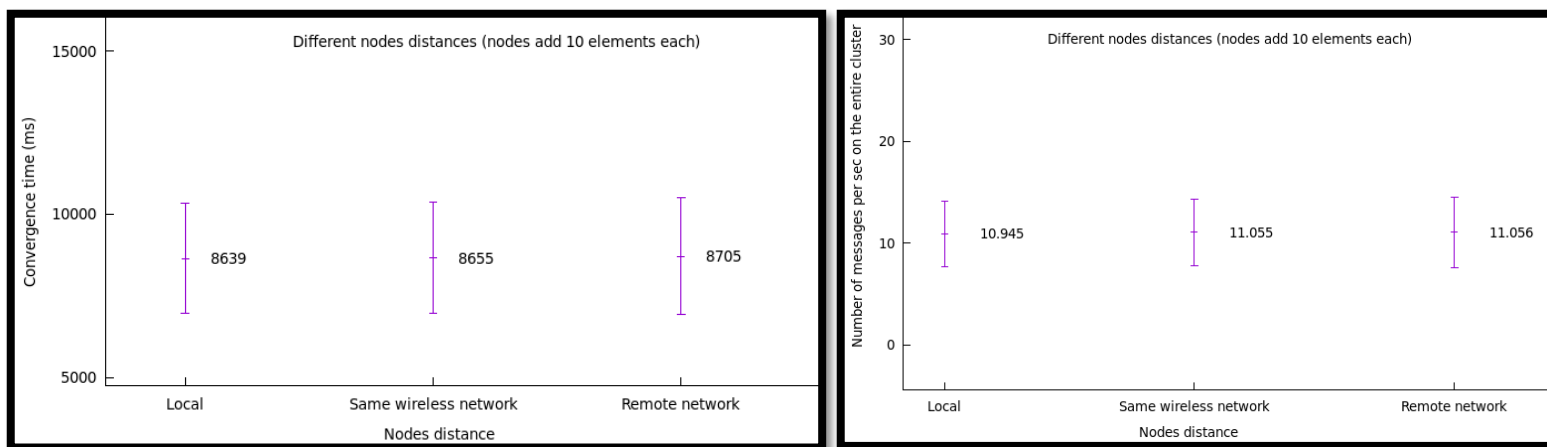

Figure 26: Graphs convergence time and network usage for different nodes distance (nodes adding 10 elements each, state_interval of 10000ms)

Analysis:

With a cluster of 5 nodes, we can see the distance between the nodes does not have a big impact. At least for a reasonable distance (from locally to around 20km distance) the impact is very little. The communication between remote nodes using IP addressing seems to be very fast which is not a surprise since 20 km is not a big distance at all on the internet scale (convergence time being unimpacted is probably not truly the case anymore when distance gets around thousands of kilometres).

Conclusion:

Being at the convergence time or at the number of messages sent per second, the results seems to be extremely close together being nearly independent of the nodes distance. It seems that only the physical limitations related to distance may have some impact for long distance. In other words, the impact seems to be proportional to the ipv6 ping duration between localisations which is a few ms for small distances such as 20 km. As a conclusion, Lasp should be very little impacted by geographical distance between nodes unless this one becomes extremely big.

Remark:

As for previous experimentations (section 3.3), some other measurements were run with nodes adding more elements or removing them. While showed in the case of the first parameter (number of nodes) to present the entire experimental approach to the reader, all the measurements will not be shown for every single parameter since they generally represent the same results with slightly different absolute value. These raw measurements are still accessible to motived readers within the github repository of this work.

3.5 CRDT content

While precedent experimentations made nodes put elements on a CRDT to measure the convergence time on that CRDT, the number of elements put by the nodes in itself was never the parameter being tested. In other words, some previous measurements checked if a cluster where nodes add 10 elements or 100 elements each were impacted the same way by the number of nodes in the cluster or the distance between them. Now, let's try to not modify other parameters and focus on the CRDT content itself via the number of elements in the CRDT.

As a remind about the experimental procedure, every node puts elements on the awset CRDT (single-fire operation as opposed to continuous updates) and wait to detect convergence. Let's see the results for awset growing up to 50 elements (5 nodes which add 10 elements each), 500 elements (5 nodes which add 100 elements each) and 5000 elements (5 nodes which add 1000 elements each).

Here are the measured results:
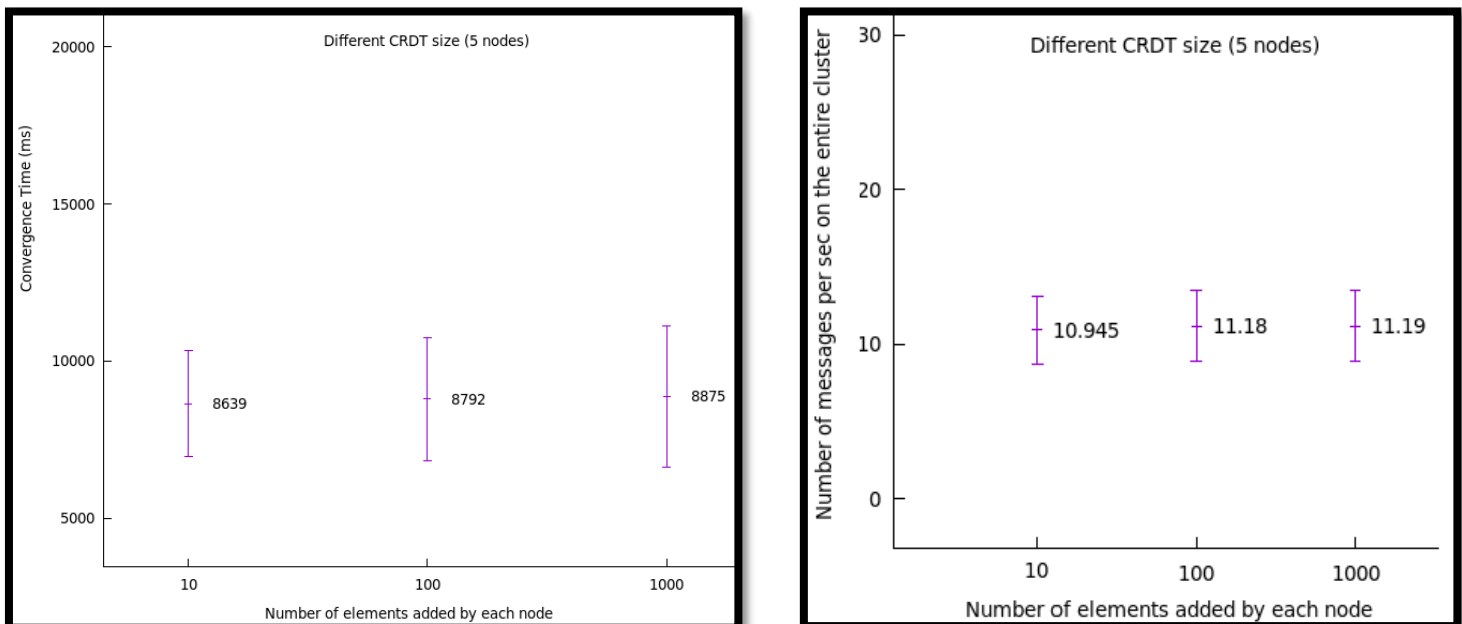
Figure 27: Graphs convergence time and network usage for different CRDT content (5 nodes adding elements 10, 100 or 1000 elements each, state_interval of 10000ms)

Analysis:

It looks like the convergence time for an awset CRDT containing a total of 50, 500 or 5000 elements is hardly impacted by its content size (number of elements in the set). Indeed, the convergence time seems to be very slightly longer when the number of elements is bigger, but the variation is very little. Since an awset CRDT with more elements means bigger messages exchanges (not especially more messages or more sending rounds but bigger messages), it is not especially a surprise that it hardly impacts the convergence or number of messages per second. Still, the fact the convergence time is very slightly longer with more elements might be a sign the nodes simply take a very little bit more time to process information (decode messages, compute merges, etc…).

Let's verify this also applies to elements removals. To do this, measurements were run to measure the impact with removals of 10,100 or 1000 elements by each node (on a 5 nodes cluster with an awset CRDT initially containing 50, 500 or 5000 elements and ending empty).
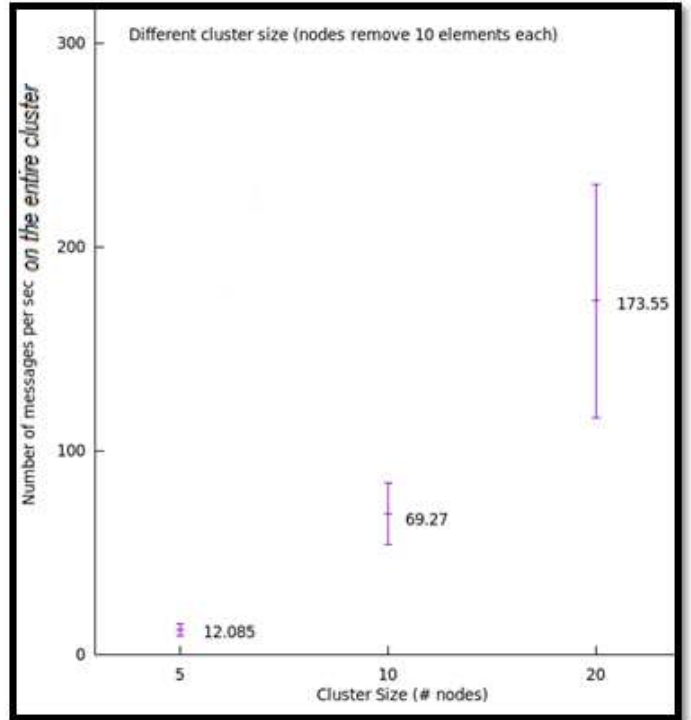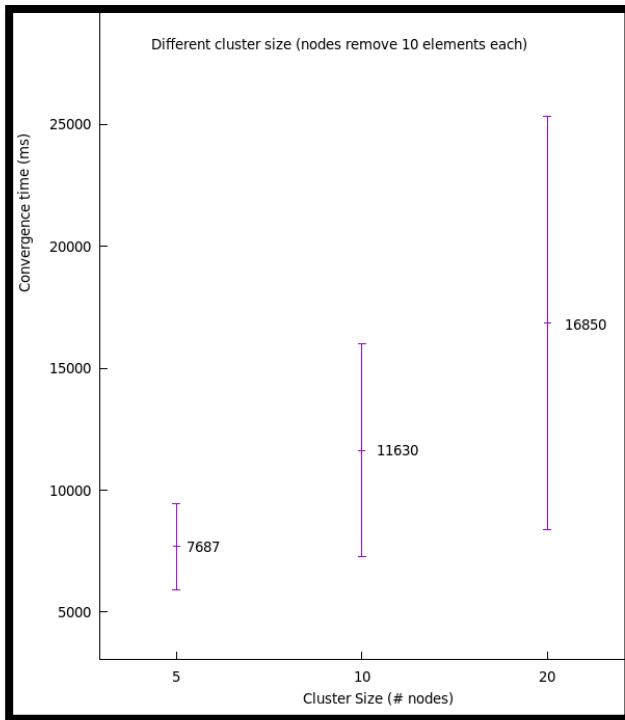
 Here are the results:
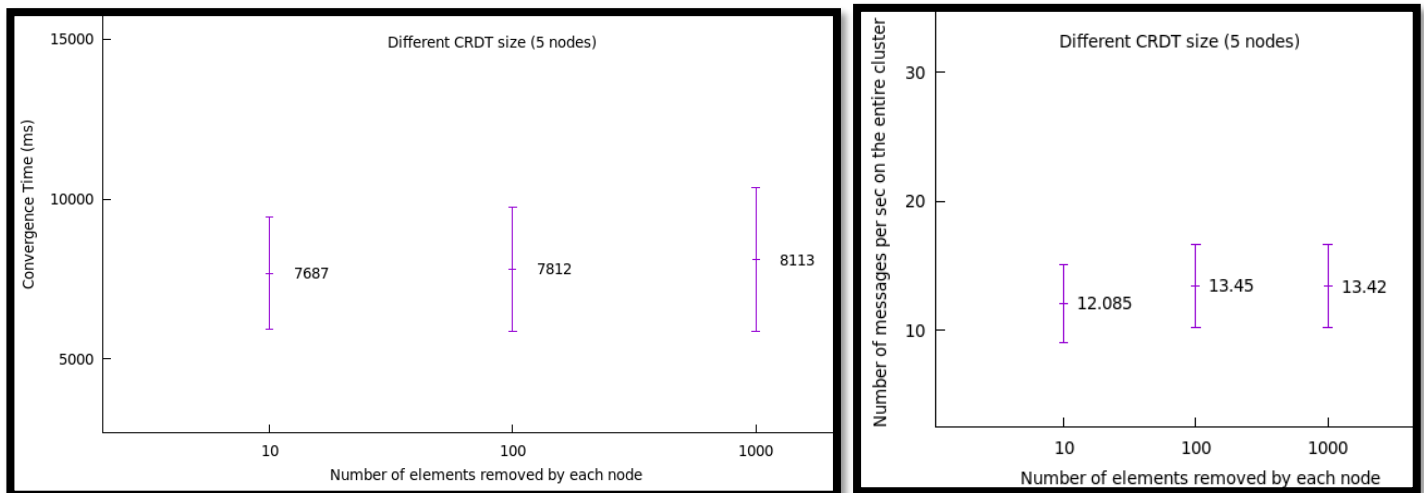
Figure 28: Graphs convergence time and network usage for different CRDT content (5 nodes removing 10, 100 or 1000 elements each, state_interval of 10000ms)

Clearly, the same behaviour shows up with convergence time being slightly impacted by the number of elements, potentially due to computations. The number of messages per second also looks relatively constant (no real significative increase) with the number of elements.

Conclusion:

The number of elements in the awset does not have a big impact on the convergence time for that awset nor on the number of messages exchanged per second on the cluster. While a little influence exists, it is quite little compared to the impact of other parameters (such as number of nodes).

Remark 1:
Few other measurements were run in regard to that parameter notably with a cluster of 10 nodes instead of 5. Since the results are very similar with only a bit higher absolute values (due to the cluster size), they were not shown here but these raw results are on the github repository of this work.

Remark 2:
Measurements were only run with awset containing a limited number of elements and showed relatively little impact, but this does not give any security concerning scalability with very high number of elements (CRDT content being very big). What would happen with an awset containing millions of elements? This was not measured but should probably not be used for that purpose anyway.

Remark 3:
While the results only mentioned the number of elements in the awset CRDT, it might be interesting to mention these elements were integers. This was mainly because of practical ease but could be tested with other types of elements which size may be greater than integers.

3.6 CRDT operation

Previous measurements sometimes showed cases of elements addition or elements removals to see how these operations were impacted by some other parameters (number of nodes, number of elements…). Attentive readers may have noticed, while it was not directly the item under measurement yet, that elements removal seems to be faster than elements addition. When going back to section 1.3.2 which explains the ORSWOT principle (as a reminder the awset is an implementation of the ORSWOT CRDT), the fact removals are faster than additions does not sound illogical. Now let's see exactly what it is by directly comparing elements addition and removals.

Here are the results for a 5 nodes cluster adding or removing elements:
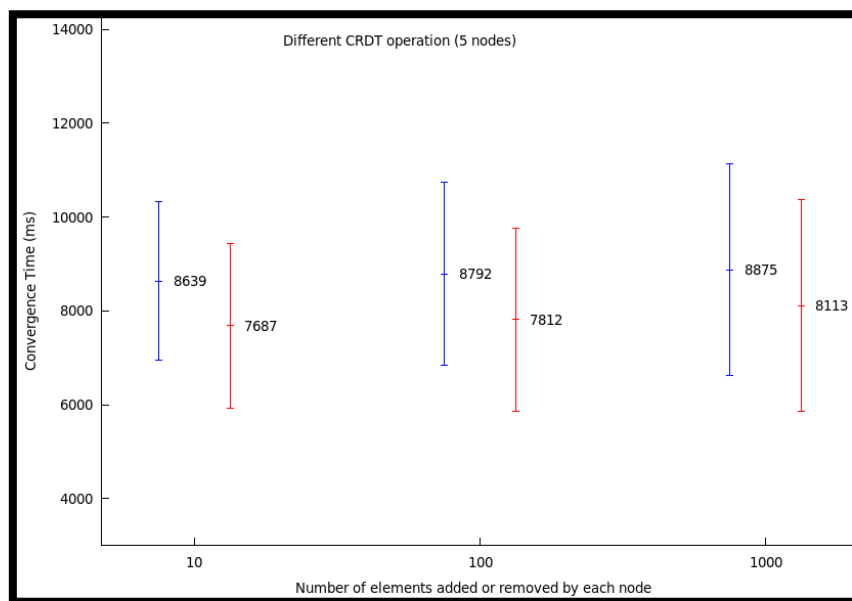
Figure 29: Graph convergence time for different CRDT operation (5 nodes, state_interval of 10000ms, nodes add/remove 10, 100 and 1000 elements each, blue is addition, red is removal)
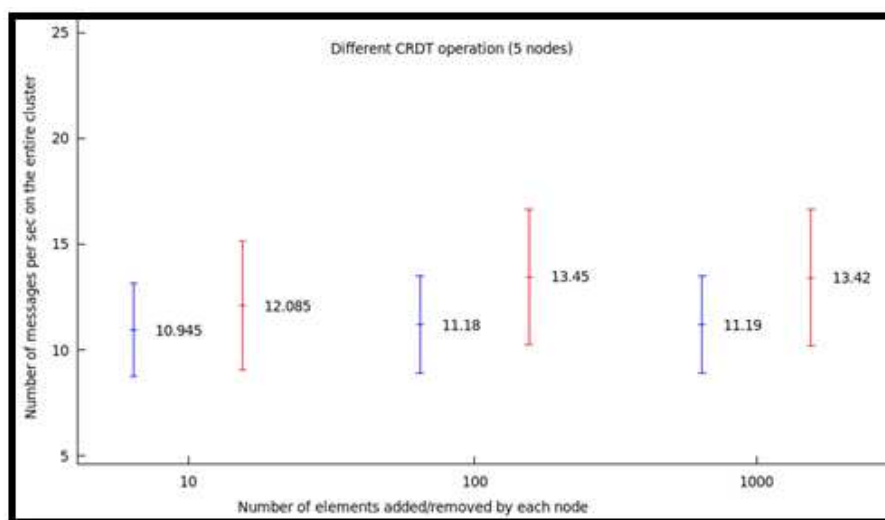
Figure 30: Network usage for different CRDT operation (5 nodes, state_interval of 10000ms, nodes add/remove 10, 100 and 1000 elements each, blue is addition, red is removal)

Analysis:

We can notice convergence time tends to be a bit longer when the operations performed by nodes on the CRDT are elements additions compared to elements removals. As a reminder, experiment where 5 nodes add 10 elements each ends up with a CRDT of 50 elements, same goes for experiment where 5 nodes add 1000 elements each, ending up with a CRDT of 5000 elements. About removing, it's the exact opposite scenario where the awset CRDT initially have elements in its content which are removed by nodes, for example experiment where 5 nodes remove 1000 elements each starts with a CRDT of 5000 elements and ends up with 0 elements. When comparing for a same maximum CRDT content, we see that removals are faster than additions by a quite significative margin.

While it was possible to have this intuition from previous graphs, it now appears clearly. The fact is the intuition removal would be faster because the operation in itself seems to be simpler compared to additions is quite true. Indeed, the removal operation simply removes an element from the awset without requiring modifying version clock or checking the dot set related to that element (see section 1.3.2 for details). The problem is such operations are supposed to be extremely fast and cannot explain by themselves the 1-2 second variation shown on the graph.

Same goes for the network utilization where it looks like element removals seem to send more messages per second compared to addition. Again, no convincing explanation was found yet to explain this behaviour. While the experimental method is probably not perfect nor precise at the 1 ms margin, such differences being for convergence time or for network usage show clearly something strange or at least unexpected in my opinion.

As a verification, to see if the same behaviour shows up, here are the graphs for the same experimentations with 10 nodes:
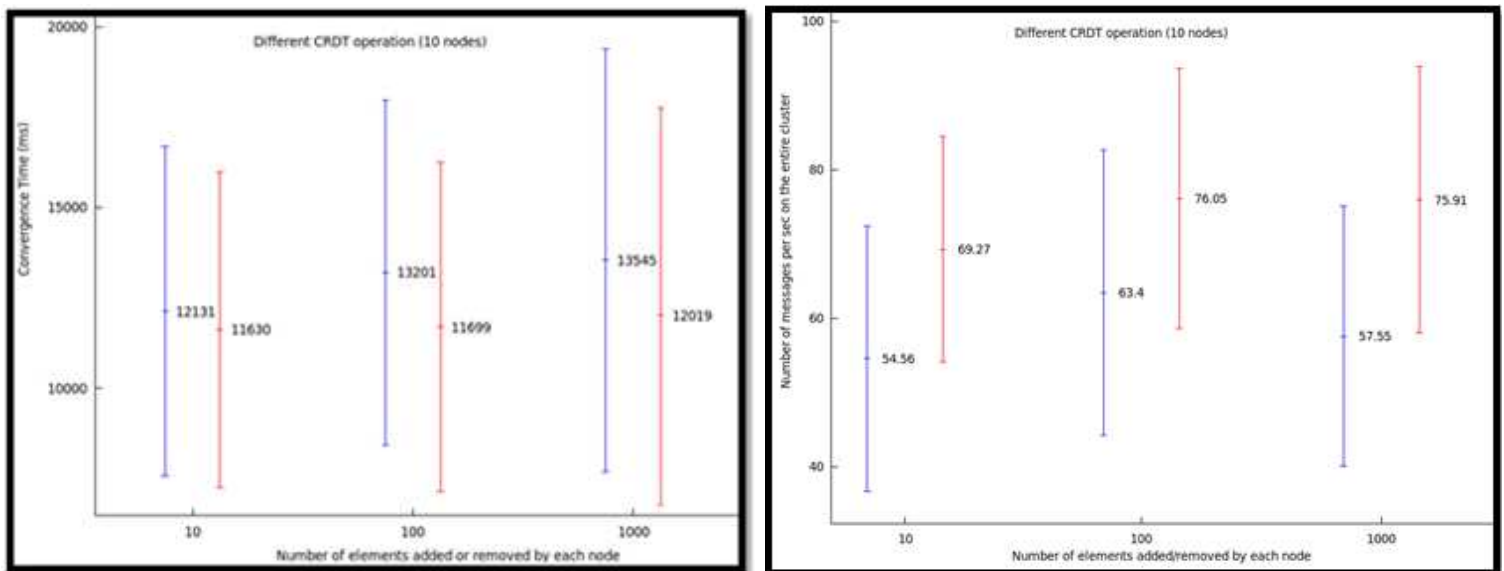
Figure 31: Graphs convergence time and network usage for different CRDT operation (10 nodes, nodes add/remove 10, 100 and 1000 elements each, state_interval is 10000ms, blue is addition, red is removal)

While the absolute values are not the same because the cluster is now of 10 nodes, the difference between additions and removals is similar to previously shown. Removals tend to be faster while sending more messages per second. The difference, on the network usage graph is even more marked

than before. While up to now, no specific explanation was found and validated to entirely justify this observed behaviour (simpler local operations does not seem enough), here are some details on the experimental approach that was used.

- Elements addition:
    - Nodes start and cluster together (orchestration see section 3.2)
    - Nodes add X elements each and start measuring (time and messages)
    - Nodes detect the awset contains X*(number of nodes) and stop measurement
    - Nodes detect every node have finished (end of experiment orchestration signal)
    - Nodes output their measurements to a file
    - Nodes are shut down
    - The full experiment starts again for next iteration (new cluster created)

- Elements removal:
    - Nodes starts and cluster together (orchestration see section 3.2)
    - Nodes have an already full awset containing X*(number of nodes) elements. These elements, present in the initial awset, are the same on every node and are considered as added by a single (fake) source, allowing every node to start with the exact same CRDT local state (same values and same metadata). This step, while different than previous experiment (element addition) is less than 10 ms.
    - Nodes remove X elements each and start measuring (time and messages)
    - Nodes detect the awset contains 0 elements and stop measurement
    - Nodes detect every node have finished (end of experiment orchestration signal)
    - Nodes output their measurements to a file
    - Nodes are shut down
    - The full experiment starts again for next iteration (new cluster created)

Conclusion:

For this specific parameter under test (CRDT operation), the experimentations show some interesting results, elements addition convergence time being about 10% longer than elements removals on measurements. For network usage, it seems like elements removals send about 20% more messages than elements additions. No precise explanation has been found yet to explain such behaviour, nor in theoretic elements nor in the experimental approach.

Remark:
If any reader comes with an idea to explore why such difference in the results, please do not hesitate to post about it by opening an issue on the github repository of this work or directly via email (gregory.creupelandt@student.uclouvain.be).

3.7 Partition

CRDTs are supposed to support partition tolerance in a very clean way, allowing any partitioned node to quickly catch up with the cluster when the partition is resolved. This is because that node, when partition resolved, will receive states from other nodes which reflect their more recent state and since message order and even message losses are not an issue at all, it will directly catch up. Since a little example generally speaks better than a long explanation, let's consider a node A which is temporarily disconnected and does not receive messages 1,2 and 3. When resolving partition (connect back to the cluster or re-join it), A will receive message 4 from a peer and will directly catch up by merging this newly received state. Indeed, since message 4 is likely to represent a more recent state than A state, A merge operation will consider message 4 metadata representing a recent state and will "accept" it.

This is theoretically well handled, let's see if it's the case in practice. Firstly, I wanted to check if there was an impact on the cluster if a node was under partition while updating its local state then resolved partition. In other words, does the updates done while under partition are directly taken in account (send to peers) when a node resolves its partition? This was verified by simply making nodes leave the cluster (making them unreachable and not able to reach anyone either), do updates on awset then join back the cluster (as if the partition was resolved). The idea is to start to measure convergence time on the cluster when the under-partition nodes resolve the partition to see if they directly send their states on partition resolved or not.

Here are the results for this experimentation:
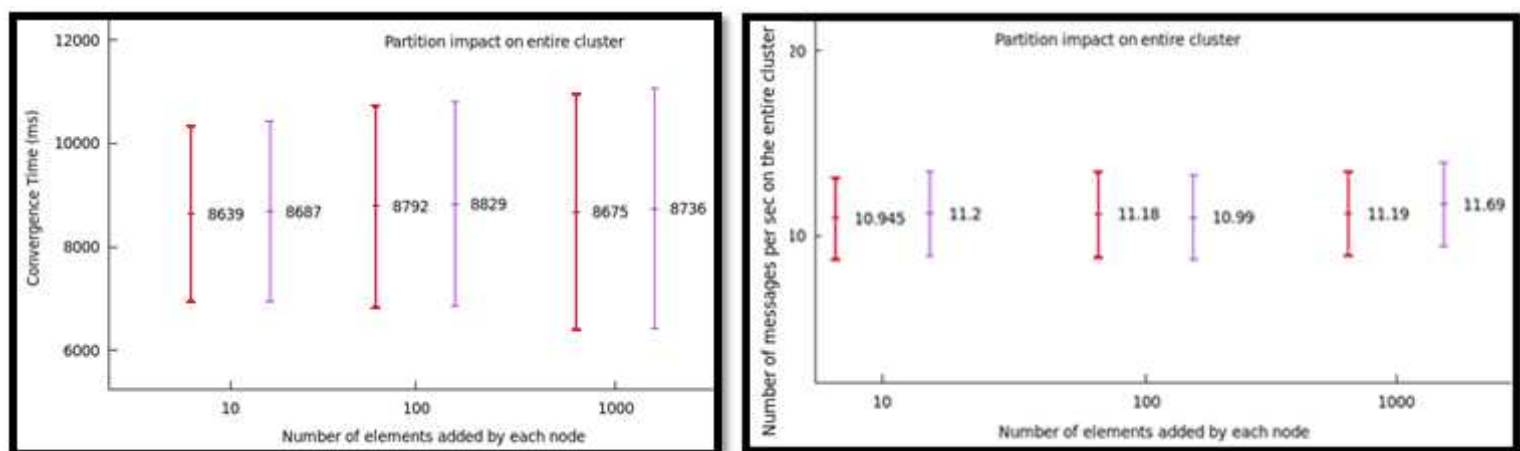
Figure 32: Graphs impact of partition on convergence time and network usage (on a 5 nodes cluster, special case: all the nodes do update their awset while under partition then resolve partition, state_interval is 10000ms, red is usual scenario, purple is for partition scenario)

Analysis:

Again, as for previous measurements, it was tested with nodes adding 10 elements, 100 elements or 1000 elements each. It is a bit of an extreme case since all the nodes are under partition when adding elements then the measurement starts when the partitions are resolved. While being not a very realistic scenario, the goal is simply to check that the updates that were done while under partition are correctly sent when the partition are resolved. As we can see, the normal scenario where

measurement starts when nodes update and the new partition scenario where measurement starts at partitions resolve give the same results (or at least extremely similar results). This shows that updates under partition are correctly taken into account for the global cluster convergence when the partitions are resolved with same performances as if the partitioned nodes were updating the awset at partition resolution moment.

While interesting, showing the cluster convergence is not impacted by nodes being temporarily under partition (in the sense that when nodes partitions are resolved, their updates are taken into account), it does not give any information about partition impact itself on a single node. Let's now see how a node previously under partition catches-up with the cluster. This was done by using a cluster were nodes add elements in an awset (again 10,100 or 1000 elements), before having the time to converge (for this following measurements, actually 1 second after the elements addition), one node gets partitioned (during 30 seconds) while the rest of the cluster converges. In other words, that partitioned node got temporarily inconsistent and divergent with the cluster state (in the sense it will never converge with the cluster if the partition is not resolved). Then the partition is resolved on that node and it starts measuring how much time it needs to converge (catch-up) with the cluster.

Here are the results for this experimentation:
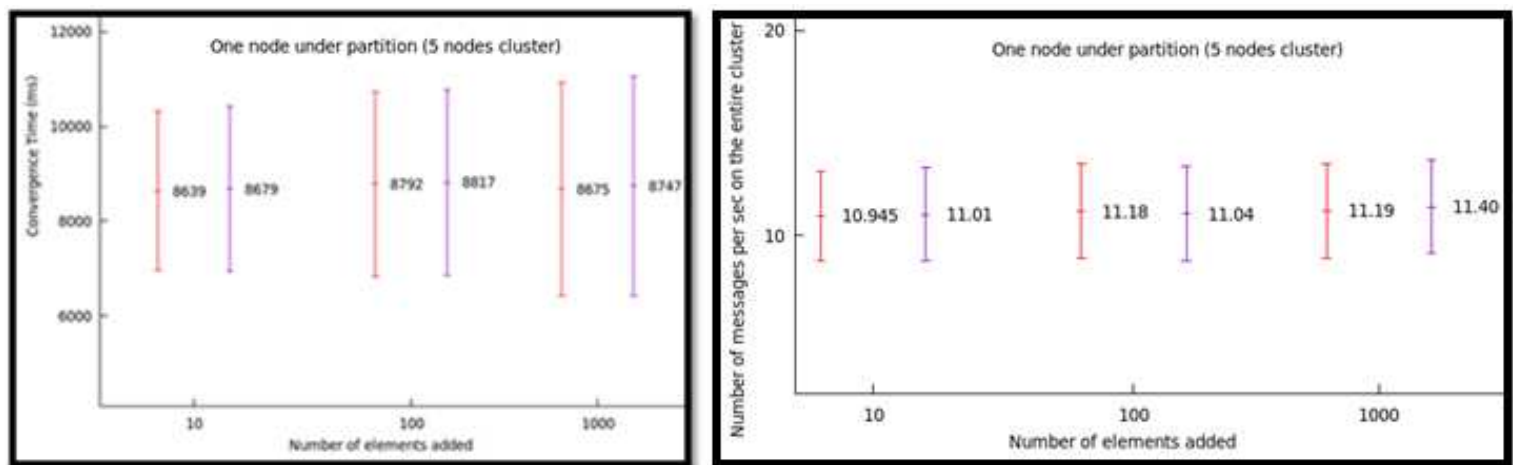
Figure 33: Graphs convergence time and network usage for a partitioned node (5 nodes cluster, nodes add 10, 100 or 1000 elements each, one node is partitioned for 30 seconds then resolves partition and catches-up with the cluster, state_interval is 10000ms, red is usual scenario, purple is partitioned node catching-up)

We clearly see that there is no big difference between both scenarios on the graphs, being at convergence time or number of messages per second. This means the partitioned node when resolving partition catches up with the cluster with the usual convergence time as if the updates were done (on other nodes) at the partition resolution, probably catching-up at next state received from a peer.

Conclusion:

From measurements, it seems that nodes updating their local state while being under partition will send their state to peers at next state sending interval if the partitioned is resolved by then. Node previously under partition, when resolving partition, will also receive peers state and converge with usual convergence time. As a conclusion, the partition tolerant property is nicely handled by Lasp, taking in consideration updates from nodes that were under partition and also making nodes that were under partition catch-up with the cluster with the usual convergence time.

3.8 Continuous updates interval

All the previous measurements were from a static approach. Indeed, nodes did some update then started measuring while doing nothing other than waiting for convergence. This approach while easy to implement in practice for measurements does not cover many real usage cases where a CRDT value is updated continuously. Indeed, many applications may want to update a CRDT value every second for example. While the previous measurements could give clues to understand this new scenario, a new experimentation set of measurements with this new approach is obviously a benefit.

The continuous approach, as briefly described in section 2.3.2, is to run a cluster of nodes that continuously update the awset content while outputting their current state on a very regular time basis to allow afterward analysis.

Here are some details on the experimental protocol:

Let's start with a cluster of 5 nodes sharing an aswet. Every node has a range of 10000 unique elements (values) on which they will loop adding and removing an element at a specific interval. For the exact experimental implementation, node 1 has the range 0-9999, node 2 has 10000-19999,… At start, node 1, for example, will have initially values from 5000 to 9999 inside its local state (CRDT content), it will add element 0 and remove element 5000 then at next round it will add element 1 and remove element 5001,… running cycle on its own range of values. Every node does the exact same thing allowing the shared CRDT on the 5 nodes cluster to constantly contain 25000 elements (every node has a 10000 elements range where 5000 are present at once) while these elements are constantly changing.

By modifying the interval at which nodes loop on their values, it directly modifies the parameter named as continuous updates interval. By printing on regular time basis to files the nodes local states together with timestamps and information on the removed and added element, it is possible to analyse the files after the experimentation to measure the time it required for the nodes to receive other nodes updates (elements).

Since there are a lot of measurements which can become incommodious to analyse all at once, only the measured convergence time for nodes to detect elements added by node 1 will be showed on a graph. For the results below, elements are added/removed at the interval of 1 element every 0.5 second, the cluster is composed of 5 nodes each doing updates at that same rate (but we only consider the elements added by node 1 to be more readable on the graph). The fact to only represent measurements for elements added by node 1 does not impact the results since the experimentation is symmetrical in regard to every node.

Here are the results:
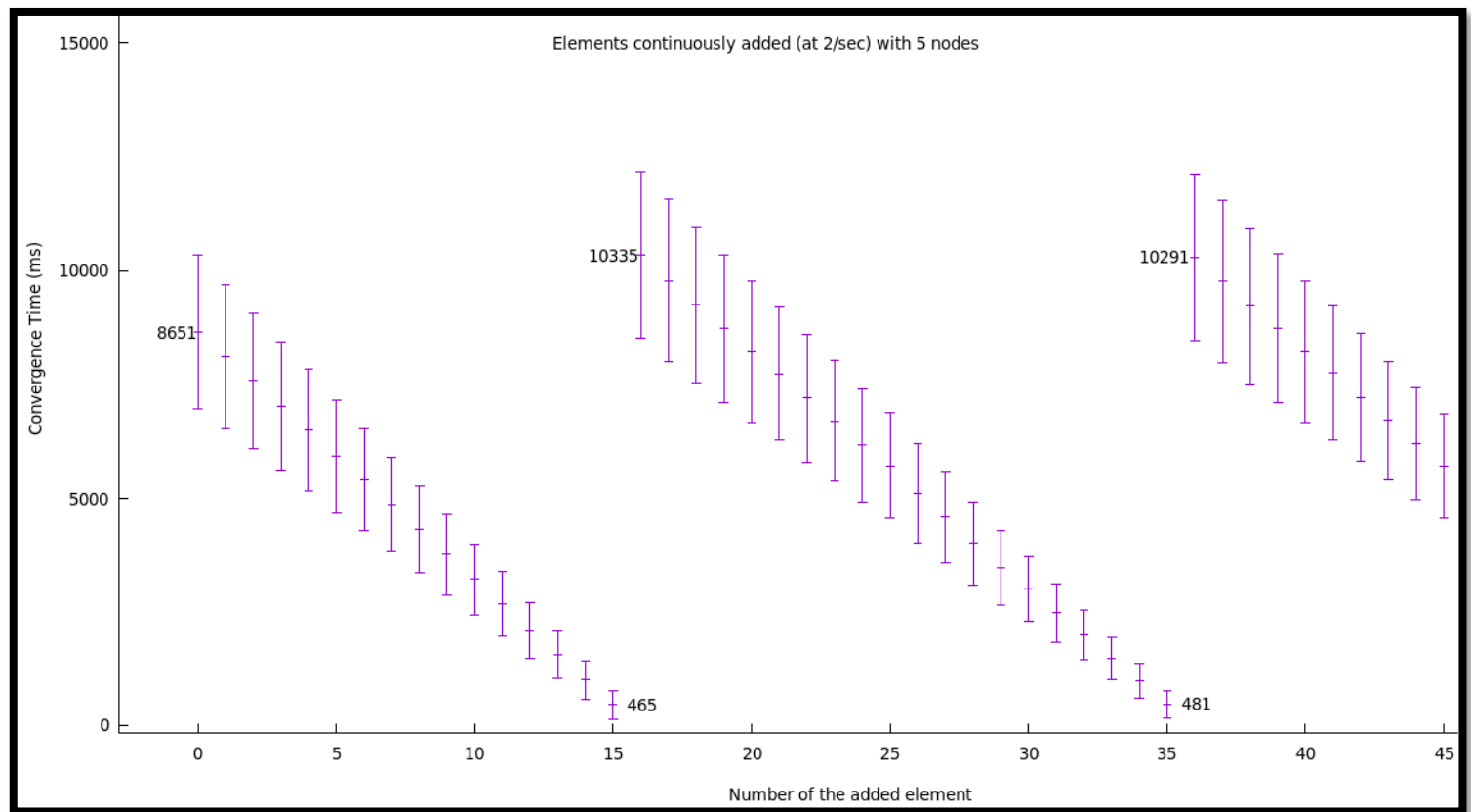
Figure 34: Graph convergence time for continuous updates at one element update every 0.5 second (5 nodes cluster, only elements added by node 1 are shown, number of the element corresponds to its value and its order. For example, element 5 had value 5 and was the 5th element added by node 1. State_interval is 10000ms)

Analysis:

There is a very clear behaviour showing up. As we can see, the first element that was added when starting the measurements took 8.65 second to be detected by other nodes. Next elements took gradually less time to be detected until resetting back to longer convergence times. As a reminder, the convergence time is measured here as the time between the element addition (by node 1 for this graph) and the moment when other nodes detected that element. The first element added is detected after 8.65 seconds which is an already seen value for previous measurements and is not surprising. Then the second element (added 0.5 second later) takes arounds 0.5 second less time (around 8.15 seconds) to be detected by other nodes, element 3 takes again 0.5 second less time to be detected and so on… Element 15 is detected very quickly (465 ms) then the next element (16) gets extremely long (slightly more than 10 seconds) to be detected, starting again the decreasing convergence times.

This can be easily explained by looking at Lasp principle. Since nodes (here we focus on node 1) send their states every 10 seconds (default state_interal is 10000ms), every update done during that time interval simply affects local state, then on timer trigger, the state is sent. This means only one specific state (the most recent one) is sent by the node every 10 seconds covering all the intermediary local states. The fact the first element takes around 8.65 second is related to the fact the experiment must start at a precise same moment on every node which is not especially at the start of a new state sending round (state_interval). Indeed, there is a little orchestration protocol to start the experiment which

delays the starting for a synchronized start on all the nodes. The fact the next element is detected 0.5 second faster is simply related to the fact it was added 0.5 second closer to the next state sending round, and same goes for the next elements. Element 15 was added just before the state sending was triggered (which occurs every 10000ms) and thus was very quickly detected by other nodes. Then element 16 was added to the local state just after the state sending triggered and was thus sent at the next round which was around 10 seconds later. It is also interesting to note that element 16 was really added near after the start of the new state sending interval and thus took slightly more than 10 seconds to converge on other nodes as opposed to initial element which was delayed because of orchestration and was thus detected faster (in comparison to the moment it was added). It is also interesting to note that the second round starting with addition of element 16 gathered all the elements until element 35 which represents the 20 elements that were added between the 10 second state sending interval, indeed 20 elements added at one element every 0.5 second corresponds to the 10 seconds interval between each state sending (state_interval).

This means two important things which were already guessable but are now highlighted clearly:

- Updating the awset elements at a smaller interval (higher rate) than the state_interval will result in a cluster that only receive some of the source states. Indeed, if a node acts like a source continuously modifying elements in the awset, only its most recent state will be sent at every state_interval period hiding its intermediary states. As a conceptual example, let's imagine the awset is used a bit like a counter (which is not optimal but represents nicely the principle) where a node adds (+1) to the awset single element every 1 second (continuous update interval of 1000 ms). If the state_interval is 10000ms, the other nodes will, for example, only detect the element 10 then the element 20, 10 seconds later then element 30… The intermediary values such as 1,2,3… which were just temporary steps on the source node will be hidden to the cluster. If we consider convergence as the fact to eventually reach a consistent state on the entire cluster, we could say that the previously mentioned example will never converge. Indeed, the cluster will always receive values which are not up to date with the source node (which already updated to next value) and thus there will be no moment where the state on every node (including souce node) would be consistent. That being said, the cluster will never become totally divergent, it will simply get a constant delay being always late compared to the source state. To achieve a real consistent state on every single node including the continuously updating source node, it would at least require the state_interval to be shorter than the continuous updates interval. While not especially a sufficient condition, it is a required one. Otherwise, by definition, the cluster will always be late compared to the source and thus the state will only be consistent on every node but the source one.

- The previous measurements (3.3 to 3.7) had to start the experiment and measuring convergence time at the same moment on all the nodes together which was not especially at the start of a new state sending interval. This means, as clearly visible on the graphs, some big standard deviations are measured since the moment when an update is done (element added for example) has an impact on its measured convergence time. Indeed, as clearly visible on the graph above, if an element is added just before a new state sending round, it will converge very quickly on other nodes but if an element is added just after a state sending, it will take much longer to be detected by other nodes. Therefore, it was important to have the less possible variations regarding to the moment when updates were done for previous measurements. It is also the reason why, as clearly mentioned in section 3, the measurements are relevant only if we look at the variations according to parameters not if we only look at the absolute values. Indeed, while modifying some parameters, the experimental protocol

tried to avoid modifying the delay between node booting and start of experiment to avoid the exact phenomenon described above which, otherwise, would have resulted in totally inconsistent measurements. These measurements and discovering also involved adapting a little bit the continuous measurement tool described in section 2.1. Indeed, it initially measured some very changing convergence times from round to round which is not the case anymore since it now adapt the period at which it does measurements to be an integer number of state_intervals that way it does not shift in the state_interval between iterations.

Same experimentations were done with faster value update speed, 4 updates per second, 20 updates per second and 100 updates per second as described in section 2.3.2. The fact is, they only represent the same results as the graph above but with more elements inside each round with measured convergence time decrease between following elements being accordingly smaller before resetting to full long convergence time.

Basically, to have a quick overview, the different cases can be resumed as these summaries (which as simply overview figures):
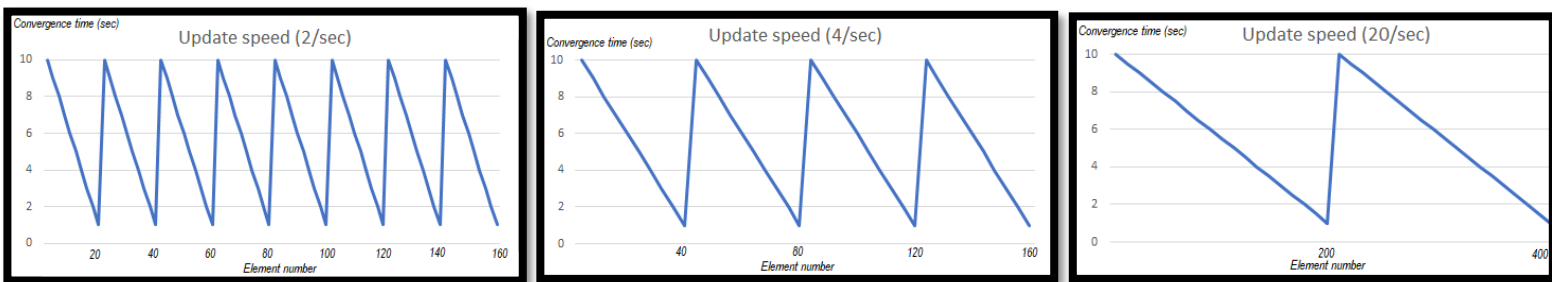
Figure 35: Overviews convergence time for continuous updates at different speeds (5 nodes, state_interval of 10000ms)

The only difference is the number of updates (elements addition) that are done within a state sending round, it is very straight forward, since the first case with one element addition every 0.5 second allows to add 20 elements within 10 seconds (which is the state_interval), pack of 20 elements will be detected at once, then the 20 next elements at the next round, and so on… For the second case since the value update speed is higher (on element added every 0.25 second), 40 elements are added during one interval (10 second) and thus pack of 40 elements are detected at once. Same goes for the last one where 200 updates are done within a sending interval.

Conclusion:

As a conclusion for that parameter, the value update speed is not really a parameter that affects the convergence time, it is more of an element that, if we want perfect convergence, should be limited by the state sending interval (aka state_interval). Indeed, if a programmer wants a cluster to be convergent with perfect consistent and thus every single state from any source to converge on all the nodes without hiding any temporary state, he must at least limit his continuous update interval to be bigger than the state_interval, by definition.

In other words, if the continuous updates interval is smaller than the state sending interval, the cluster will always be late compared to the continuous updating source and will never catch-up, meaning the state will never be consistent on all the nodes (since the source already updated to next value). But

the cluster will not be really divergent either since it will simply have a relatively constant delay compared to the source, always being a few updates late.

Remark:
While not showing any impact on the convergence time, the presented measures in this section have the merit to extremely well represent Lasp principle and implementation in practice. Indeed, such graphs, compared to previous ones, are the first to show how the mechanism works in practice where previous graphs only allowed some intuitions.

3.9 State sending interval

While all the previous measurements presented in sections 3 were running on relatively slow convergent clusters, it is now possible to make the convergence much faster by modifying the interval at which the states are send. Indeed, the developed tool described in section 2.2 does exactly that by modifying the state_interval on the fly while the cluster is already running. While it was not possible to start again all the previous measurements to see the impact of the different parameters on a faster cluster (for example nodes sending their states every 100ms instead of every 10000ms) mainly due to time restrictions for this work, it will be interesting to, at least, check for the impact of the state_interval value on the convergence time together with the network usage.

The intuition, after all the previous analysis, is that reducing the state_interval will decrease the convergence time (combined with analyse of section 3.8 conclusion, it would thus allow a smaller continuous updates interval while still being totally convergent) at the cost of an increased number of messages per second exchanged on the cluster. Let's see exactly how it affects the cluster.

The results are directly measured via the exact same scripts (section 2.3.1) as for some of the previous measurements but with a state_interval modified to different values from 10ms to 10000ms (which is the default Lasp value on the Lasp official github).

Here are the results for a cluster of 5 nodes each adding 10 elements (all at once) and waiting for convergence. Measurements were also run with nodes adding more elements (100 and 1000 elements each) but since the results were very similar (section 3.5 highlighted the fact the variation according to the awset content is very little), they are not shown here. More raw results are available on the github webpage of this work.

Here are the results:
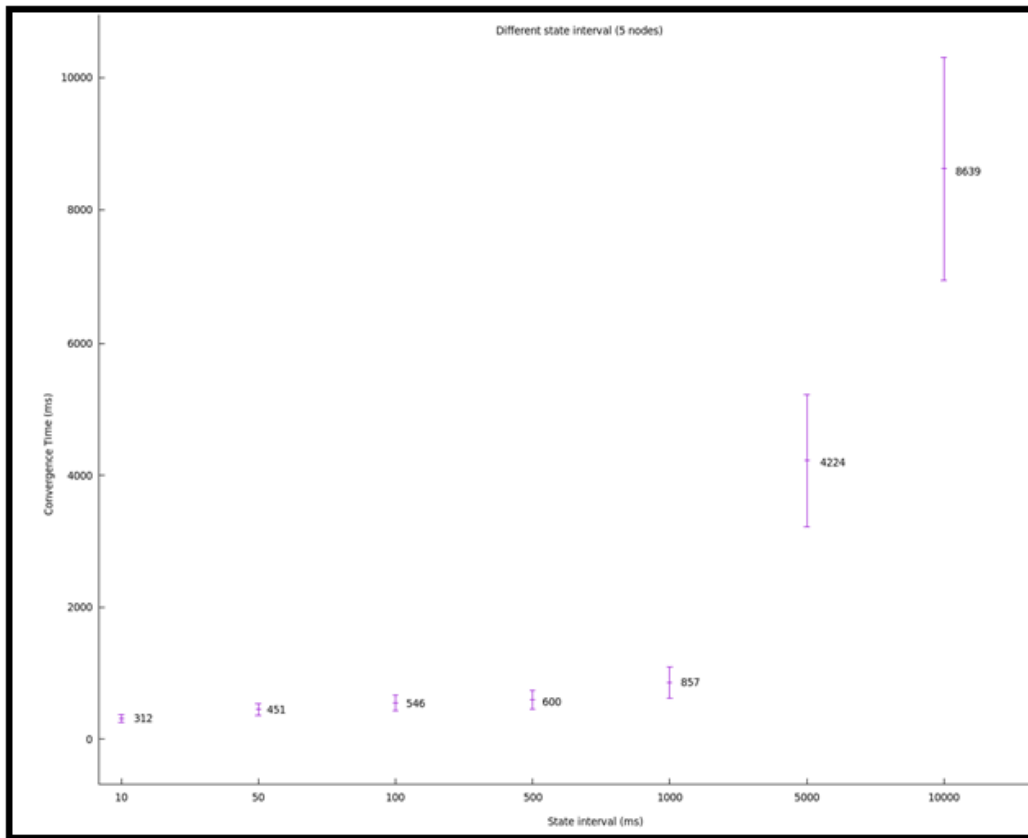
Figure 36: Graph convergence time for different state_intervals (5 nodes adding 10 elements each)
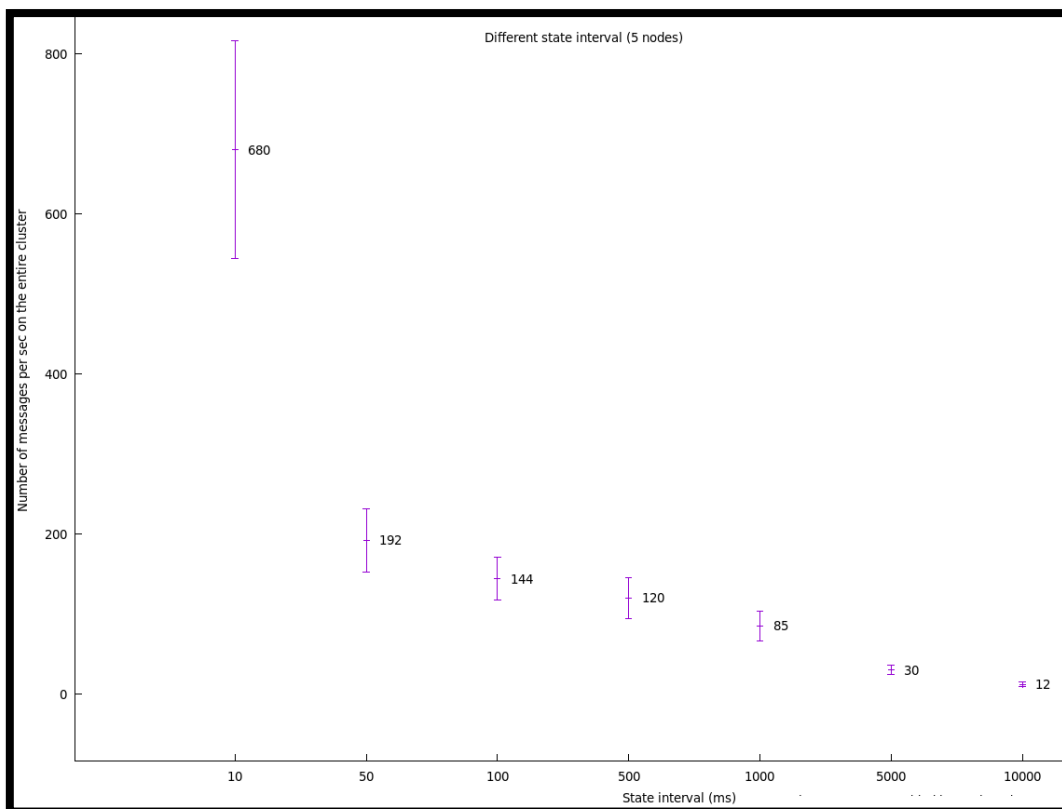
Figure 37: Graph Network usage for different state_intervals (5 nodes, adding 10 elements each)

Analysis:

We clearly see, as expected, that the convergence time is bigger for bigger state_interval which is a totally logical behaviour. On the other hand, if we try to express the convergence time as a function of the state_interval, we can notice it looks directly proportional for big state_intervals but not linear at all for state_interval values smaller than 500 ms. Indeed, we can notice a kind of limit to the minimal convergence time in the sense that even if the state_interval gets much smaller, the convergence time does not decrease as much. This is likely due to two logical factors:

- The operations such as messages decode, state merging etc on nodes are not instantaneous. Indeed, even in a theoretical case where states were sent infinitely often, some computations still need to be done, introducing a hard-minimum convergence time. There might also be some guards that add waiting times inside Lasp code to protect from network overuse. When the state_interval gets smaller, these slowdowns get more and more important in proportion (where they were probably negligible for big state_interval values).

- Sending states more often means the network get heavier due to messages. It also means nodes receive much more messages to treat which can affect performances.

Talking about this last point, let's get a look at the impact of the state_interval value on the number of messages per second on the cluster. As expected, we clearly notice that the smaller the state_interval, the bigger the number of messages per second goes. If we try to express the number of messages per second per node (dividing the number of the network graph by 5) as a function of the state_interval we can notice something interesting. Indeed, the expected behaviour would be to have something near:

Intuition: $number\ of\ messages\ per\ second\ per\ node\ \approx \frac{1000}{state\_interval}$

The interesting fact is, if we try to express it like that, we obtain a result with a relatively constant shift:

Measurements: $number\ of\ messages\ per\ second\ per\ node\ \approx \frac{1000}{state\_interval} + 20$

While not extremely precise, this gives some good intuition the state messages are indeed sent at the rate of one every state_interval but some other messages seems to be also sent while relatively independent of the state_interval which could probably be other usage messages such as logging and keep alive messages to detect crashs.

Conclusion:

The convergence time seems to be directly proportional to the state_interval when this parameter is higher than 500 ms. When decreasing to smaller values of state_interval, the convergence time seems to stop following a linear function even reaching what seems to be a limit around 300 ms convergence time. In regard to the number of messages per second, the values seems to be consistent with the intuition that the number of messages is inversely proportional to the state_interval value but with an important shift that looks relatively constant around 20 messages per second per node which may be related to other purpose messages that do not involve CRDT state sending.

Remark 1:

When combining figures 36 and 37 together, we can notice going with a state_interval smaller than 500 ms does not especially translates into a much smaller convergence time while it could mean a much bigger network usage especially if going with a state_interval smaller than 50 ms. Following the need, it could be useful to go with smaller state_interval than 500 ms but going smaller than 50 ms does not seem to be efficient, at least when looking at these measurements.

Remark 2:

When combining recent figures (36 and 37) with section 3.8, we could conclude an application running on 5 nodes that requires total convergence would have difficulties to run using Lasp awset if it requires a continuous updates interval smaller than 300 ms. Indeed, it seems to be impossible for the CRDT to convergence on a 5 nodes cluster faster than within 300 ms which would mean the source node would have already updated to next value before reaching consistency on all the nodes. This is quite annoying for real usages but only reflects this work measurements which may not have discovered all the Lasp possible parametrizations that could potentially allow better performances.

Remark 3:

Finally, as a little informal observation, we can notice some likely consistency between figure 34 from section 3.8 (sawtooth graph) and figure 36. Indeed, figure 34 could already give the intuition a kind of minimal convergence time around 300 ms would exist as a limit even if the state_interval goes very small. We can notice on that graph (figure 34) that element 15 took 465 ms to converge on other nodes. If we (falsely) considered the convergence to be instantaneous on state sending, it would mean element 15 was added 465 ms before state sending. This would mean element 16 was added 35 ms after state sending (because elements were added on a 0.5 second interval). If the convergence was instantaneous at state sending, it would mean element 16 would have converged in 9965 ms but it did converge in 10335 ms introducing the intuition an instantaneous convergence on state sending is a faulty theory and an approximative 300 ms minimum delay on convergence is plausible which was highlighted in section 3.9.

4. Conclusion
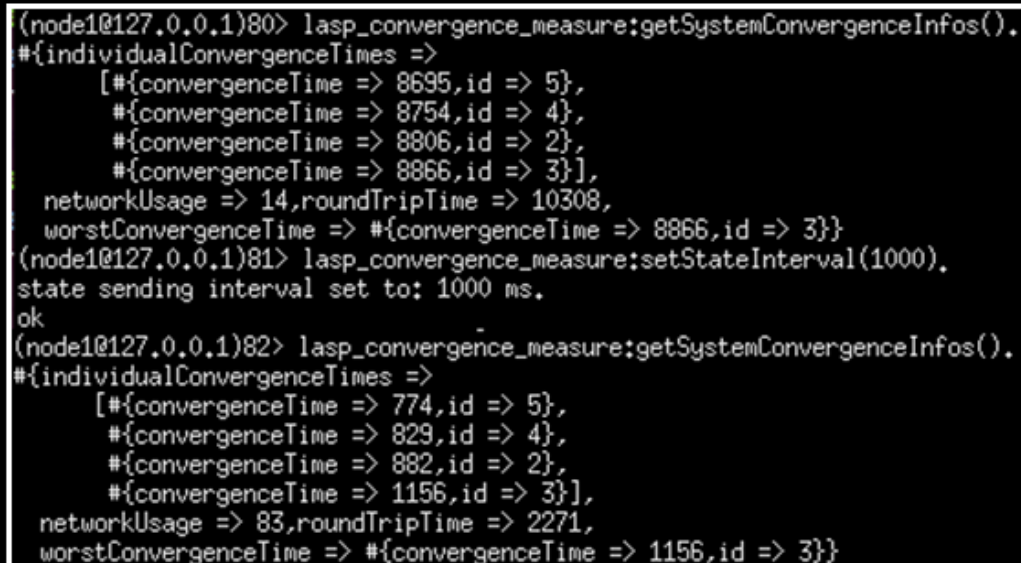
4.1 Summary

We have been able to use Lasp, an experimental tool developed by Christopher Meiklejohn, to handle distributed variables in the form of a CRDT, precisely named awset (Orswot in general CRDT literature). Measurements were done to measure the impact of various parameters and to verify Lasp fulfils the CRDT features. Tools were developed to measure and modify convergence on the fly. Finally, some little bugs or imprecision were found in Lasp with the acknowledgement and correction of the Lasp main developer himself.

4.2 Developed tools conclusion

The developed tools allow convergence visualization and modification with a clear and easy-to-use API. While not extremely precise (they should not be used in the case of benchmark for example), they allow the end-developer to have a good overview of his cluster performances. As a reminder, the measurement tool does not directly measure a specific CRDT already under-usage on a cluster but mimic it by doing measurement on a little awset CRDT on the under-usage cluster. While this means it will not consider the real CRDT under-use (with its number of elements), it was shown the number of elements in the CRDT (section 3.5) is not of a big impact compared to the cluster size (number of nodes: section 3.3) or the state sending period (state_awset: section 3.9) which are parameters taken in consideration by the measurement tool. Furthermore, the results measured by this continuous tool were compared with results measured by the static measurement scripts and show similar results.

As a conclusion, while not perfect and allowing some future improvements, the developed tools allow the end developer to visualize and control the convergence. A very simple but speaking example is the following:

```
(node1@127.0.0.1)80> lasp_convergence_measure:getSystemConvergenceInfos().
#{individualConvergenceTimes =>
      [#{convergenceTime => 8695,id => 5},
       #{convergenceTime => 8754,id => 4},
       #{convergenceTime => 8806,id => 2},
       #{convergenceTime => 8866,id => 3}],
  networkUsage => 14,roundTripTime => 10308,
  worstConvergenceTime => #{convergenceTime => 8866,id => 3}}
(node1@127.0.0.1)81> lasp_convergence_measure:setStateInterval(1000).
state sending interval set to: 1000 ms.
ok
(node1@127.0.0.1)82> lasp_convergence_measure:getSystemConvergenceInfos().
#{individualConvergenceTimes =>
      [#{convergenceTime => 774,id => 5},
       #{convergenceTime => 829,id => 4},
       #{convergenceTime => 882,id => 2},
       #{convergenceTime => 1156,id => 3}],
  networkUsage => 83,roundTripTime => 2271,
  worstConvergenceTime => #{convergenceTime => 1156,id => 3}}
```

Figure 38: Developed tools real usage example

As visible, the developer can easily get information about his cluster performance (via background continuous measurements), modify the period between each state sending and see the impact on the cluster by asking again for the last measurements.

One last remaining drawback is the fact the measurements are dependent of the moment they are run (see section 3.8 for details). Indeed, the continuous measurement would give consistent measurements since it is orchestrated by a leader that run measurements rounds with a time loop corresponding to an integer number of state_intervals. But, even if consistent, these measurements could be shifted compared to reality by the fact the measurements are always run, for example, at the beginning or at the end of the state sending time interval. To improve this tool, it would be better to go with an even more continuous approach more in line with section 3.8 way of measuring to have a better representation and select the longer convergence times which is a better representation of the real worst case.

4.3 Results analysis conclusion

Based on the various measurements that were presented in section 3, we can conclude multiple things. First of all, Lasp does allow convergence as intented, allowing every node to eventually have the same CRDT state which is the basis of CRDT principle.

Secondly, we saw the time required for a cluster to converge can be impacted from various parameters the more impactful seems to be the number of nodes and the state sending period (aka state_interval), same goes for the number of messages exchanged on the cluster.

- Number of nodes:
  - Convergence time: It seems to be proportional to the square root of the number of nodes.
  - Number of messages per second per node per operation: It seems to be a constant.
- Nodes distance:
  - Convergence time: Impact seems to be extremely little.
  - Number of messages per second on the cluster: Impact seems to be extremely little.
- CRDT content:
  - Convergence time: Impact seems to be extremely little.
  - Number of messages per second on the cluster: Impact seems to be extremely little.
- CRDT operation:
  - Convergence time: Elements additions seem to be 10% longer to converge than removals
  - Number of messages on the entire cluster: Elements removals seem to send 20 % more messages than elements addition.
- Partition:
  - Convergence time: Nodes under partition seem to catch up with the cluster directly on next round of states received from peers.
  - Number of messages per second on the cluster: No real impact, in other words, when resolution is resolved, the previously under partition node does not especially send or received more messages than any other node.
- Continuous updates interval:
  - Convergence time: No impact on the convergence time but to achieve a real convergence (eventually reaching a consistent state on every node including a continuously updating source), the continuous updates interval must be bigger than the state sending interval.

- o Number of messages per second on the cluster: No impact at all.
- State sending interval:
  - o Convergence time: It seems to be directly proportional to the state_interval when bigger than 500ms. When going to smaller values, it seems to meet a limit around 300 ms.
  - o Number of messages per second per node: It seems to be relatively inversely proportional of the state_interval but shifted by a relatively constant value probably not related to the awset state sendings.

When regarding the two most significative parameters, while the state sending period impact is very logical, easily measurable and controllable, the case of the cluster size (number of nodes) is a bit more difficult to analyse with certainty. Indeed, the fact the convergence time would increase with the number of nodes could sound logical since more nodes could mean more state sending rounds required to reach every node (peer-to-peer communication may not reach all the cluster on first round for example). But this impact, while logical, could be accentuated by the experimental protocol which involved running more nodes on devices and thus increased workload. While we believe this specific measure might not be precise enough due to the multiple parameters modifications at once, it could act as a maximum impact meaning the convergence time increase with number of nodes would probably be smaller than the results measured.

While other parameters, such as the number of elements in the CRDT (CRDT content) or geographical distance between nodes, globally did not show any very impactful variations on the convergence time or number of messages, they allowed to verify the good functioning of Lasp such as verifying CRDT properties or partition tolerance. The continuous updates interval parameter results and graphs, in particular, allowed to better understand how Lasp was working by clearly showing the updates being sent together acting like a single state at every state sending round (indeed Lasp uses state-based CRDT). On the other hand, CRDT operation showed unexpected variation between elements additions and removals being on the convergence time or the number of messages exchanged. While clues are conceivable such as the operation local complexity, I believe it is not enough to explain such variations (up to 10% variation) and should deserve more exploration in the future to better understand what is causing this being in Lasp implementation or in this work experimental protocol.

Finally, when looking closer to the state sending period (state_interval) and its impact, it seemed Lasp could be easily parametrized to allow convergence time of the order of 200-300 ms while it would be harder to achieve much higher speed. Still, this work is not complete in itself and should be taken as a brick in a bigger construction which goal is to fully understand Lasp and find its real limits.

4.4 Future work

As just mentioned, while this work offered new tools and interesting overviews on Lasp properties and performances according to various parameters, it should be taken as a step towards bigger goals such as fully understanding Lasp possibilities and pushing it towards greater world-wide recognition. With this optic in mind, a lot of work is still awaiting completion.

Such interesting works include :

- **Measure the impact of other parameters on Lasp**: Indeed, section 1.4.2 introduced about fifteen parameters that might influence Lasp performances. Only seven of them were analysed in this work, leaving the other parameters with no knowledge on them yet. There is no doubt analysing these parameters will be beneficial to the future. Even more parameters than mentioned in section 1.4.2 can be found and were never discussed here while extremely interesting (for example: check if the delta-state optimization offers the same behaviours).

- **Verify this work measurements with another approach**: The measurements done during this work, as mentioned in previous section, are interesting in their variations according to the parameters but must not be taken for precise measurements on their absolute values. Indeed, as explained in section 3.8, the measurements themselves could be affected by the moment they were started in regard to the state sending interval. While minimizing timing variations between iterations, these measurements still show some important standard deviations that might be smaller if measured with another experimental approach. Indeed, multiple possibilities exist to measure something, and this work simply show one way to measure it with its advantages and drawbacks. Finding another approach to measure the influences of the described parameters could give other results especially on their absolute values but should give same overviews on the parameters impacts. This is something that would be very interesting and could give more value both for the future work in question and for this one. It would also be the occasion to dig in some of the flaws from this master thesis work such as the not extremely rigorous experimental protocol for the cluster size measurements (section 3.3) that could be improved (with more devices) and the lack of clear explanation for the unexpected variation between elements additions and removals (section 3.6). Also, the fact Lasp was corrected (see section 2.4.1) at the last minute did not allow all the measurements to be run again with the correction which could be done correctly as a future work. Hopefully it did not impact the results by a lot since they were run on very short runs but still, corrected measurements would be a good acknowledgement.

- **Expand this work to all the Lasp CRDTs**: This work clearly focused on the awset (ORSWOT) CRDT that is provided in Lasp but it could be interesting to expand it to allow measurements on all the different kind of CRDTs that are available in Lasp. While it would require some work to adapt the measurements tools which were designed with the awset in mind, it should not be too difficult and could give interesting results such as verifying if all the different CRDTs are impacted the same way by the different parameters or if, for example, some of them are more suitable for a certain type of cluster, etc.

- **Compare Lasp with more conventional systems:** This work directly focused on Lasp trying to analyse it and understand its deep elements but did not have the opportunity to compare Lasp to other existing systems, mainly due to time limitations. It would be interesting to see how different approaches handle a similar scenario with totally different solutions and implementations. While extremely interesting, it could require to measure many different elements (such as CPU usage, process memory size, ability to run on slower devices…) which

were not especially taken into consideration during this work that only focused on measuring convergence time and network usage.

- **Improve the newly developed tools:** While working correctly with measuring awset performances, it is obvious this tool could be improved in many ways. Since the principle of this tool is to mimic a CRDT on a real cluster for measurement, it could be beneficial to add more parametrization to it to allow to better represent the real use case. For example, it could be improved by allowing measuring on different kind of CRDTs (awmap, orset, gcounter…), not only awsets which should not be too hard to implement. It could also allow more parameters such as the number of elements that would generally be present in the CRDT or the type of operations that are generally performed. This would allow the measurement tool to better mimic the real use-case to allow more precise information. Also, while interesting for overview, the mechanism in itself could be improved. Indeed, it starts measurement round based on a time interval that is automatically assigned as an integer number of state_interval to limit variations due to timings. While this approach has its advantages (relative precision while allowing easy implementation), it could be improved as mentioned in section 4.2.

- **Develop an automatic tool to dynamically adapt convergence:** While the developed tool is a nice first step and allow overview and controlling on the convergence it is not fully automated in the sense it still requires the end developer to adapt the state sending interval. A possible improvement would be, after improving the tools (which is detailed in previous point just above), to combine measurement and adaptation tools together. This would allow, for example, the measurement to be automatically analysed and the state_interval to be dynamically adjusted based on measurements. This would require the end-developer to only enter a desired convergence time and maximum number of messages per second (to limit network usage) and would automatically try to constantly reach these performances. While simple in its principle, it would require reflexion on many little details and would require the already developed tools to be greatly improved with that optic in mind (such as adding inertia to the measurement rounds to avoid sudden variations that could be caused by timing).

- **Improve Lasp documentation:** While very wide in its possibilities, Lasp clearly lacks documentation. It allows many usages and implement many tools and functions that are documented nowhere. The only way to even know they exist is to dive into the code and to seek by yourself. While the principle of CRDT itself is easy to understand, it is relatively hard to understand how exactly it is implemented in Lasp due to this lack of documentation. This could be greatly improved by the developers themselves or by people willing to dive into Lasp and to make it easier to use for future works.

4.5 Personal opinion

Personally, I found CRDT principle extremely clever and was amazed by its simplicity compared to heavy algorithms like consensus. However, I had difficulties to familiarize myself with Lasp mainly due to its lack of documentation. While it was easy to do the first steps, create a first cluster and do little tests with the few little given examples (even if they were actually incorrect since the implementation was updated without the documentation), it was harder to understand how the backend implementation worked.

The measurements, while looking coherent and showing some logical results are also not as precise as I would have wanted mainly due to the timing variations highlighted in section 3.8. Indeed, the moment when an update is done (for example adding an element in the awset) has an impact on its measured convergence. This was not the initially expected behaviour I had in mind when starting this master thesis which required many measurement protocols to be reviewed to verify every iteration was not modifying its starting point moment (compared to the node booting moment), at least to allow comparison between measurements. This required many measurements do be started again. Same goes for the Lasp correction (see section 2.4.1) that was discovered during measurements due to the involved memory leak and was acknowledged by Lasp main developer pretty late making it impossible to start all the measurements with the corrected version. These are basically just examples about the fact Lasp is a great tool but requires more documentation to be recognized at its true value.

Lastly, while nothing up to now proved me I am wrong in my measurements about the state sending interval results (section 3.9), I feel uncomfortable when seeing it can hardly converge faster than within 300 ms on a 5 nodes cluster even if reducing the state_interval much smaller. I am pretty sure Lasp could go faster if better parametrized, but no information yet is available nowhere to explain how exactly to parametrize it correctly. The small documentation mentioning such things is not up to date which means it requires the end developer to dive into the configuration files and codes to understand what the different values are used for and to try things by himself. No doubt, in my own opinion, documentation is the main drawback about using Lasp but hope is it will be greatly improved in the future. That being said, I am not used to work with experimental tools which is probably why I was used to find more documentation.

4.6 Methodology

This master thesis was a bit of a timing challenge in the sense it was done within less than 4 months which is quite short for this kind of work. However, by starting quickly by doing little programs, scripts and basically playing with Lasp, it was possible to quickly start developing measurements tools and work on contributions. The topic being assigned to me only on the 21 september 2020, I had to extremely quickly take in touch with Professor Peter Van Roy to start working. The documentations he provided me, the advices and multiple interesting questions he mentioned allowed me to quickly better understand the topic and the potential goals of this work even within the first week of the semester.

Then started our extremely enriching meetings every week where we exchanged on my work done and his impressions, advices and encouragements. Every week work, improvement and meetings summaries are available in the methodology section on the github page of this work.

My personal decision was to firstly understand a bit better Lasp then to develop the continuous measurement tool to gain better knowledge then finally to start all the measurements mainly using scripts while continuously discovering new potential improvements and going back and forward between the two main goals (developed tools and measurement results) to improve them.

The last weeks were a bit more complicated since the dead line was coming closer and some measurements were still not completed partially due to the issue described in section 2.4.1 that was causing measurements on small state_interval to crash pretty quickly due to the memory leak. Anyway, I ended up able to do the most important measurements at last minute and write down this report, that I hope, will fulfil the readers and especially Professor Peter Van Roy curiosity.

5. Bibliography

[1] Van Roy, P. *Building the Future of Edge Computing with LightKone*, Open Access Government, 2020.

[2] Ongaro, D. Ousterhout, J. *In Search of an Understandable Consensus Algorithm*, Stanford University, 2014.

[3] LAMPORT, L. *Paxos made simple*. ACM SIGACT News 32, 2001.

[4] Gilbert, S. Lynch, N. 2002. *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services*. SIGACT News 33, 2002.

[5] Preguiça, N. Baquero, C. Shapiro, M. *Conflict-free Replicated Data Types*, DI, FCT, Universidade NOVA de Lisboa and NOVA LINCS, HASLab / INESC TEC & Universidade do Minho, Sorbonne-Universite & Inria, 2018.

[6] Preguiça, N. *Conflict-free Replicated Data Types: An Overview*, NOVA LINCS & DI, FCT, Universidade NOVA de Lisboa, 2018.

[7] Sypytkowski, B. *An introduction to state-based CRDTs*, Software Dev blog, 2018.

[8] Meiklejohn, C. Van Roy, P. Bieniusa, A. *Distributed Programming with Weak Synchronization Models: Introduction to CRDTs, Lasp, and Antidote*, KTHx: ID2203.2x, edx, 2016.

[9] SyncFree. *Large-Scale Computation without Synchronisation*, European Union Seventh Framework Programme, 2013-2016. https://syncfree.lip6.fr/

[10] Shapiro, M. Preguiça, N. Baquero, C. Zawirski M. *Conflict-free Replicated Data Types: Rapport de recherche*, INRIA & LIP6, Universidade Nova de Lisboa, Universidade do Minho, 2011.

[11] Martyanov, D. *CRDTs in production*, Qcon, 2018.

[12] Meiklejohn, C. Enes, V. Sloughter, T. Lasp official github repository, https://github.com/lasp-lang/lasp, 2014-2021.

[13] LightKone. *Lightweight Computation for Networks at the Edge*, European Union Horizon 2020, 2017-2019. http://lightkone.eu/

[14] Meiklejohn, C. Miller H, Alvaro, P. *Partisan: scaling the distributed actor runtimes, Annual Technical Conference USENIX ATC, 2019, 63-76*

[15] Meiklejohn, C. Enes, V. Yoo, J. Baquero, C. Van Roy, P. Bieniusa, A. *Practical evaluation of the Lasp programming model at large scale*, Université catholique de Louvain, Universidade do Minho, University of Oxford, Technische Universität Kaiserslautern, 2017.

[16] Shapiro, M. Preguiça, N. Baquero, C. Zawirski, M. *Convergent and commutative replicated data types*, University of Crete, universidade Nova de Lisboa, Universidade do Minho, INRIA & UMPC, 2011.

[17] Owen, M. *Using Erlang, Riak and the Orswot CRDT at bet365 for scalability and performance*, Erlang User Conference, 2015.

[18] Meiklejohn, C. Van Roy, P. *Lasp: A Language for Distributed, Coordination-Free Programming*, International Symposium on Principles and Practice of Declarative Programming, 2015.

[19] Ali Shoker, Paulo Sergio Almeida, Carlos Baquero, Annette Bieniusa, Roger Pueyo Centelles, Pedro Ákos Costa, Dimitrios Vasilas, Vitor Enes, Carla Ferreira, Pedro Fouto, Felix Freitag, Bradley King, Igor Kopestenski, Giorgos Kostopoulos, João Leitão, Adam Lindberg, Albert van der Linde, Sreeja Nair, Nuno Preguiça, Mennan Selimi, Marc Shapiro, Peer Stritzinger, Ilyas Toumlilt, Peter Van Roy, Georges Younes, Igor Zavalyshyn, and Peter Zeller. *LightKone Reference Architecture (LiRA) White Paper version 0.9*, The LightKone Consortium, Dec. 2019.

[20] Brown, R. Sean, C. Meiklejohn, C. Elliott, S. *Riak DT map: a composable, convergent replicated dictionary*, First workshop on principles and practice of eventual consistency, 2014

6. Appendices

6.1 Entire code

The entire code that was produced for this work can be found on the public github repository:

https://github.com/darkyne/LaspDivergenceVisualization

It is important to note this repository includes the Lasp code that was, of course, not implemented by myself. The code produced for this work is mainly the lasp_convergence_measure.erl file that I created to bring together all my tools in a single erlang module.

While the majority of the experimentations and measurements elements are implemented within the lasp_convergence_measure module, some scripts were also used to launch the nodes that will call the erlang newly implemented module and functions. These scripts are within the mylasp/lasp/Memoire directory that regroups the majority of the relevant elements such as scripts and measurements.

The readme file on the github page details the content of every relevant directory.

Examples representing a good part of the coding work will be presented below.

6.2 Example of generic output files for static measurements

Many experiments were run with many iterations involving many nodes that each wrote output files. This means the total number of output files is quite heavy, but some generic examples can be shown here. For example, all the static measurements were outputting files such as (from each node):

```
1   Launching 5 nodes at the same time, each generating a total of 10 elements (while under partition = false)
2   then start timer and wait until all the elements converged
3
4   ============
5   Parameters :
6   ============
7   Node Id:
8   1
9   Node to join cluster:
10  node1@192.168.1.39
11  Generating elements under partition:
12  false
13  Type of CRDT:
14  state_awset
15  Number of nodes:
16  5
17  Number of elements generated by each node:
18  10
19  Sending all at once:
20  true
21  Theoretical time (ms) between each element sending, for each node, valuable only if not sending all at once:
22  10
23  Time (ms) required (for this node) to generate and send all the elements:
24  43
25  Initial size of the CRDT (in term of words) :
26  69
27  Total final number of elements in the CRDT:
28  50
29  Final size of the CRDT (in term of words) :
30  149
31  Ellapsed time (ms) after elements generation to get the total 50 elements:
32  9227
33  Number of message received before convergence :
34  22
35  Number of message received per second :
36  2.384
37  ====================================================================================
38  ====================================================================================
```

Output file Node1_1604449840320.txt can be found in mylasp/lasp/Memoire/Saved_measures/measures_data/Static/5Nodes/Saved_1/Exp1 folder with many other output files.

These output files are then used by the analyse_script to compute means and standard deviations creating results files such as:

```
1   Results for Exp1
2   convergence mean time: 8639,272
3   convergence time standart deviation: 1685,2846094579
4   convergence time median: 8131
5   initial size of the CRDT (with 10 elements): 69
6   final size of the CRDT (with 50 elements): 149
7   mean number of messages/sec : 2.189
8   number of messages/sec standart deviation: .508
9   number of messages/sec median: 2.264
```

Result file Exp1.txt can be found in mylasp/lasp/Memoire/Saved_measures/measures_data/Static/5Nodes/Saved_1/Results folder with other results files.

6.3 Example of generic output files for continuous measurements

Since the continuous approach measured convergence in a totally different way (afterwards, based on output files), the output files had to be totally different. Here is an example:

```
67763   5750,15520,25290,28931,35060,38701,48471,19962,8505,18275,28045,37815,47585,
67764   36128,45898,7619,17389,5932,15702,25472,35242,48653,29914,5046,18457,28227,
67765   37997,47767,46080,7801,17571,27341,37111,15884,25654,35424,45194,39866,49636,
67766   5228,38179,47949,9670,19440,7983,17753,27523,37293,47063,35606,45376,7097,
67767   16867,49818,5410,15180,48131,9852,19622,29392,39162,17935,27705,37475,47245,
67768   7279,17049,26819,36589,15362,25132,6393,10034,19804,29574,39344,49114,37657,
67769   47427,9148,18918,7461,17231,27001,36771,46541,35084,6575,16345,19986,29756,
67770   39526,49296,9330,19100,28870,38640,17413,27183,36953,46723,8444,6757,16527,
67771   26297,36067,39708,49478,5871,9512,19282,29052,38822,48592,37135,46905,8626,
67772   18396,6939,16709,26479,36249,46019,6053,15823,19464,25593,29234,39004,48774,
67773   8808,18578,28348,38118,26661,36431,46201,7922,6235,16005,25775,35545,39186,
67774   48956,5349,8990,18760,28530,38300,48070,36613,46383,8104,17874,27644,6417,
67775   16187,25957,35727,45497,5531,15301,25071,28712,38482,48252,9973].
67776
67777   NEXT
67778   TIME 1605186238183
67779   ADDED ELEMENT 45
67780   REMOVED ELEMENT 5045
67781   CRDT Size 25007
67782   CONTENT :
67783   [0,6129,9770,19540,29310,39080,48850,37393,47163,8884,18654,7197,16967,26737,
67784    36507,46277,6311,16081,19722,25851,29492,39262,49032,9066,18836,28606,38376,
67785    26919,36689,46459,8180,6493,16263,26033,35803,39444,49214,5607,9248,19018,
67786    28788,38558,48328,36871,46641,8362,18132,27902,6675,16445,26215,35985,45755,
67787    5789,15559,19200,25329,28970,38740,48510,8544,18314,28084,37854,26397,36167,
67788    45937,7658,5971,15741,25511,35281,38922,45051,48692,5085,8726,18496,28266,
67789    38036,47806,46119,7840,17610,27380,6153,15923,25693,35463,45233,39905,5267,
67790    28448,38218,47988,9709,8022,17792,27562,37332,47102,25875,35645,45415,7136,
67791    49857,5449,15219,38400,48170,9891,19661,29431,8204,17974,27744,37514,47284,
67792    45597,7318,17088,26858,15401,25171,19843,29613,39383,27926,37696,47466,9187,
67793    7500,17270,27040,36810,46580,25353,35123,6614,20025,29795,39565,49335,47648,
67794    9369,19139,28909,7682,17452,27222,36992,46762,45075,6796,16566,26336,29977,
67795    39747,49517,9551,19321,29091,38861,48631,27404,37174,46944,8665,6978,16748,
67796    26518,36288,46058,49699,6092,9733,15862,19503,29273,39043,48813,47126,8847,
67797    18617,28387,16930,26700,36470,46240,6274,16044,25814,29455,39225,48995,9029,
67798    18799,28569,38339,48109,26882,36652,46422,8143,17913,6456,16226,25996,35766,
67799    45536,49177,5570,15340,18981,28751,38521,48291,46604,8325,18095,27865,16408,
```

Output file Node1.txt can be found in mylasp/lasp/Memoire/Saved_measures/measures_data/Dynamic/5Nodes/First_set7/Exp1 folder with other output files. This is just an example of the file content since it actually contains about a hundred thousand line.

This kind of output file is then analysed by a script than run through all the output files to look for a specific element, when detecting it, it compares the moment it was added and the moment it was detected (each node outputs one different file thus one element added in a file is detected in another file together with timestamps to compute the convergence time required). This process is relatively heavy since the files are huge and each element added in a file has to be found in all the other files to compute convergence times in regard to each node for each added element… As much saying it was long.

Here is an example of result file computed by such scripts:

```
19   Node4.txt for element 4 convergence time:4284
20   Node5.txt for element 4 convergence time:3726
21   Node2.txt for element 5 convergence time:4399
22   Node3.txt for element 5 convergence time:3922
23   Node4.txt for element 5 convergence time:3763
24   Node5.txt for element 5 convergence time:3205
25   Node2.txt for element 6 convergence time:3938
26   Node3.txt for element 6 convergence time:3461
27   Node4.txt for element 6 convergence time:3302
28   Node5.txt for element 6 convergence time:2744
29   Node2.txt for element 7 convergence time:3413
30   Node3.txt for element 7 convergence time:2936
31   Node4.txt for element 7 convergence time:2777
32   Node5.txt for element 7 convergence time:2219
33   Node2.txt for element 8 convergence time:2920
34   Node3.txt for element 8 convergence time:2443
35   Node4.txt for element 8 convergence time:2284
36   Node5.txt for element 8 convergence time:1726
37   Node2.txt for element 9 convergence time:2329
38   Node3.txt for element 9 convergence time:1852
39   Node4.txt for element 9 convergence time:1693
40   Node5.txt for element 9 convergence time:1135
41   Node2.txt for element 10 convergence time:1932
42   Node3.txt for element 10 convergence time:1455
43   Node4.txt for element 10 convergence time:1296
44   Node5.txt for element 10 convergence time:738
45   Node2.txt for element 11 convergence time:14321
46   Node3.txt for element 11 convergence time:14171
47   Node4.txt for element 11 convergence time:13733
48   Node5.txt for element 11 convergence time:13224
49   Node2.txt for element 12 convergence time:12848
50   Node3.txt for element 12 convergence time:12698
51   Node4.txt for element 12 convergence time:12260
52   Node5.txt for element 12 convergence time:11751
53   Node2.txt for element 13 convergence time:12194
54   Node3.txt for element 13 convergence time:12044
55   Node4.txt for element 13 convergence time:11606
56   Node5.txt for element 13 convergence time:11097
57   Node2.txt for element 14 convergence time:11532
58   Node3.txt for element 14 convergence time:11382
59   Node4.txt for element 14 convergence time:10944
60   Node5.txt for element 14 convergence time:10435
```

Result file Exp1.txt can be found in mylasp/lasp/Memoire/Saved_measures/measures_data/Dynamic/5Nodes/First_set7/Exp1 folder with other output files. This is just an example of the content since these files can contain a high number of lines. As a remark, we can already notice the sawtooth behaviour.

6.4 Measurement script example

Many scripts were used for measurement. They are pretty basic since they simply launch multiple terminals via xterm and launch a node in each. Then the nodes tasks are not written in the script bash file but in the convergence_measure erlang module that offer all the functions needed. The script task itself is mainly to put some files at correct places, launch the nodes, wait a predefined long direction, kill all the related process and start again for next iteration. When all the iterations are done for every modified parameter (for example 6 parameter values with 50 iterations each means 300 experimentations to run), the scripts read the outputs files to compute results.

The fact, many of these scripts are the sames just modifying few elements such as values or files to move (these files can contain the node application with the few method calls).

Here is a script example:

```bash
#!/bin/bash

#Set1 5nodes, 6experiments, 50iterations, 60sec duration

#This Script launchs 5nodes that will add elements and measure convergence time,
#CRDT memory size and number of messages received
#You may need to modify IpAddress, put it to 127.0.0.1 for local tests.s

decalage=0 #decalage=pi_booting_time - pc_booting_time = 8.32
initialNode=1
localNodes=5
iterations=50
experiments=6 #number of different parameter values
duration=60 #number of seconds for each iteration

#IpAddress is set via Memoire/AppsToLaunch/IpAddress file

#ipAdress will be taken from config file
#For local teta:
#ipAddress=127.0.0.1

#on PC:
#ipAddress=192.168.1.35

#on Pi:
#ipAddress=192.168.1.62

#on Inai:
#ipAddress=130.104.221.134


#Path to get the correct app to launch
sourcePath=Memoire/AppsToLaunch/Set1/measure
sourcePath2=/lasp_app.erl
destinationPath=src

#==================================================================================
#==================================================================================
#============================ REMOVE PREVIOUS MEASURES ============================
#==================================================================================
#==================================================================================
cd ../.. #Go to main lasp directory

echo "deleting previous measures"
rm Memoire/Mesures/Exp1/*.txt
rm Memoire/Mesures/Exp2/*.txt
rm Memoire/Mesures/Exp3/*.txt
rm Memoire/Mesures/Exp4/*.txt
rm Memoire/Mesures/Exp5/*.txt
rm Memoire/Mesures/Exp6/*.txt
rm Memoire/Mesures/Exp7/*.txt
rm Memoire/Mesures/Exp8/*.txt
rm Memoire/Mesures/Exp9/*.txt
rm Memoire/Mesures/Exp10/*.txt

rm Memoire/Mesures/Network/*.txt
```

```
57
58    ☐#========================================================================
59      #========================================================================
60      #========================= LAUNCH EXPERIMENTS==============================
61      #========================================================================
62    └#========================================================================
63
64     Node=$(sed -n 1p Memoire/AppsToLaunch/IpAddress.txt) #get Ip from config file
65     Ip=$(cut -d "@" -f2- <<< "$Node")
66
67
68     echo "LAUNCHING THE NEW EXPERIMENTS SET"
69     sleep $decalage #Initial shift related to raspberry pi slow nodes booting. Generally not required
70
71
72     for k in $(seq 1 1 "$experiments")  #number of different expriments
73     do
74       #Load next experiment
75       sleep 1
76       cp $sourcePath$k$sourcePath2 $destinationPath
77       sleep 1
78
79       for j in $(seq 1 1 "$iterations") #number of iterations
80       do
81
82         for i in $(seq "$initialNode" 1 "$localNodes") #number of local nodes
83         do
84           xterm -hold -e "rebar3 shell --name node$i@$Ip" &
85         done
86         sleep $duration
87         killall xterm
88         sleep 1
89         rm Memoire/Mesures/Network/*.*
90       done
91
92       echo "experiment$k finished"
93
94     done
95
96
97    ☐#========================================================================
98      #========================================================================
99      #========================= ANALYSE OUTPUTS ==============================
100     #========================================================================
101   └#========================================================================
102    echo "ANALYSING THE OUTPUTS"
103
104    cd Memoire/Mesures
105
106    mkdir Results
107
108    for d in Exp*/; #for every experiment in the set
109    do
110      echo "Analysing ${d%/*}"
111      echo "Results for ${d%/*}" > Results/${d%/*}.txt
112
113      #1LOAD DATA
114      cd ${d%/*}
115      declare -a convergence_array
116      declare -a message_array
117      MEGA=1000
118      index=0
119      for file in *.txt
120      do
121        localvar1=$(sed -n 33p $file)
122        localvar2=$(sed -n 37p $file)
123        localvar2=$(bc -l <<<"${localvar2}*${MEGA}")
124        localvar2=${localvar2%.*}
125        convergence_array[$index]=$localvar1
```

```
126      message_array[$index]=$localvar2
127      index=$index+1
128    done
129
130    initialSize=$(sed -n 27p $file)
131    finalSize=$(sed -n 31p $file)
132
133    initialNumberOfElements=$(sed -n 21p $file)
134    finalNumberOfElements=$(sed -n 29p $file)
135
136    cd ..
137    #DATA ARE LOADED in convergence_array,
138
139
140
141    #CONVERGENCE_TIME
142    printf "%s\n" "${convergence_array[@]}" > Results/temp${d%/*}.txt
143    convergence_mean=$(datamash -W mean 1 < Results/temp${d%/*}.txt)
144    convergence_ecart=$(datamash -W sstdev 1 < Results/temp${d%/*}.txt)
145    convergence_median=$(datamash -W median 1 < Results/temp${d%/*}.txt)
146    rm Results/temp*.*
147
148    #MESSAGES/SEC
149    div=1000
150    printf "%s\n" "${message_array[@]}" > Results/temp${d%/*}.txt
151    message_mean=$(datamash -W mean 1 < Results/temp${d%/*}.txt)
152    message_mean=${message_mean%,*}
153    message_mean=$(bc <<< "scale=3;$message_mean/$div")
154
155    message_ecart=$(datamash -W sstdev 1 < Results/temp${d%/*}.txt)
156    message_ecart=${message_ecart%,*}
157    message_ecart=$(bc <<< "scale=3;$message_ecart/$div")
158
159    message_median=$(datamash -W median 1 < Results/temp${d%/*}.txt)
160    message_median=${message_median%,*}
161    message_median=$(bc <<< "scale=3;$message_median/$div")
162
163    rm Results/temp*.*
164
165
166    echo "convergence mean time is $convergence_mean"
167    echo "convergence mean time: $convergence_mean " >> Results/${d%/*}.txt
168    echo "convergence time standart deviation: $convergence_ecart" >> Results/${d%/*}.txt
169    echo "convergence time median: $convergence_median" >> Results/${d%/*}.txt
170
171    echo "initial size of the CRDT (with $initialNumberOfElements elements): $initialSize" >> Results/${d%/*}.txt
172    echo "final size of the CRDT (with $finalNumberOfElements elements): $finalSize" >> Results/${d%/*}.txt
173
174    echo "mean number of messages/sec : $message_mean " >> Results/${d%/*}.txt
175    echo "number of messages/sec standart deviation: $message_ecart" >> Results/${d%/*}.txt
176    echo "number of messages/sec median: $message_median" >> Results/${d%/*}.txt
177
178    echo "=================="
179
180  done
181  #put back to main lasp directory
182  cd ..
183  cd ..
```

This is the kind of scripts that outputs the files presented in appendice 2.

Basically, it does not so many things, especially regarding Lasp since the code is not in the script itself but in the application being launched by nodes. These applications are coded beforehand and are move to the right directory by the script. Here below is an example of app:

```
21  -module(lasp_app).
22
23  -behaviour(application).
24
25  -include("lasp.hrl").
26
27  %% Application callbacks
28  -export([start/2, stop/1]).
29
30
31  %% ================================================================
32  %% Application callbacks
33  %% ================================================================
34
35
36  start(_StartType, _StartArgs) ->
37      case lasp_sup:start_link() of
38          {ok, Pid} ->
39          {ok, ReadIpAddress}=file:read_file("Memoire/AppsToLaunch/IpAddress.txt"),
40          IpAddress=list_to_atom( unicode:characters_to_list(string:trim(ReadIpAddress,trailing, "\n")) ),
41          lasp_convergence_measure:launchExperimentAdding(1, IpAddress, <<"set1">>, 5, true, 10, 10, false),
42          {ok, Pid};
43
44          {error, Reason} ->
45              {error, Reason}
46      end.
47
48  %% @doc Stop the lasp application.
49  stop(_State) ->
50      ok.
```

Pretty simply, the node boots, it read the ip config file (to know on which node and which ip to join) then starts an experimentation via lasp_convergence_measure:launchExperimentAdding(…).

This kind of functions that start a full experimentation are available in the lasp_convergence_measure module.

6.5 Lasp_convergence_measure module

This is where the majority of the code work was gathered together. It regroups a multitude of functions that have different usages and were used for the various measurements presented in this work.

```
1
2   -module(lasp_convergence_measure).
3   -author("Gregory Creupelandt <gregory.creupelandt@student.uclouvain.be>").
4
5   -export([launchExperimentAdding/8,
6              launchExperimentRemoving/8,
7          simpleAddition/0,
8          messageReceived/0,
9          created/0,
10         launchExperimentDynamic/4,
11         launchContinuousMeasures/3,
12         readThresholdMaxDuration/3,
13         readThresholdMaxDurationSilent/3,
14         getSystemConvergenceInfos/0,
15         getSystemConvergenceTime/0,
16         getSystemRoundTrip/0,
17         getSystemWorstNodeId/0,
18         getSystemNetworkUsage/0,
19         getIndividualConvergenceTimes/0,
20         getConvergenceTime/1,
21         addition_awset_time/0,
```

```erlang
22        addition_orset_time/0,
23        addition_gcount_time/0,
24        continuousMeasurementLoop/4,
25        getStateInterval/0,
26        getInternalStateInterval/1,
27        setStateInterval/1,
28        partitionHelper/0
29            ]).
30
31
32
33 %% ================================================================
34 %% launchExperimentAdding:
35 %% ================================================================
36
37
38 %Launch an "adding" experiment for the current node
39 % IN:
40 %Specify a number for the experiment (used to write the result file)
41 %Specify the node to join to create the cluster
42 %Specify a CRDT_ID (format as <<"setX">>)
43 %Specify the Total Number of Nodes taking part of the experiment
44 %Specify the Sending Speed (as the number of ms between each send, considering one node), 0 means the fastest possible
45 %Specify the Number of Values each node will have to generate and send on the CRDT
46 %Specify (with a boolean) if you want the node to join the cluster then generate and send values. Or rather generate
47 %and send values on the isolated CRDT (as if it was under partition) then join.
48 %Specify (with a boolean) if you want all the values to be added at once or gradually via All_At_Once. If set to false,
49 % it will use SendingSpeed to send gradually the values.
50 % OUT:
51 %The node will generate the number of values and send them, trying to achieve the specified Sending Speed.
52 %It will join the cluster before or after sending the values based on GeneratingUnderPartition
53 %The time required to generate and send the values, the time required after the generation to reach convergence and all
54 % the paramters are written to a file
55 %The file name will be in the folder /lasp/Memoire/Mesures with the name Exp+ExperimentNumber+_Node+NodeId
56
57 launchExperimentAdding(ExperimentNumber, NodeToJoin, CRDT_Id, TotalNumberOfNodes, All_At_Once, SendingSpeed,
58  NumberOfValues, GeneratingUnderPartition) ->
59   %-------------------------------------
60   %Little setup
61   %-------------------------------------
62   ExperimentStartTime = erlang:system_time(1000),
63   timer:sleep(1000), %start with a little sleep to allow NodeToJoin to be booted in case it is a bit slower than this node
64   Id = list_to_integer( lists:nth(2,string:split(lists:nth(1,string:split(atom_to_list(erlang:node()),"@")), "e")) ),
65   NetworkPath="Memoire/Mesures/Network/Node"++integer_to_list(Id)++".txt",
66   NetworkPath2="Memoire/Mesures/Network/Node"++integer_to_list(Id)++"_Final.txt",
67   file:write_file(NetworkPath,"NEW \n",[append]),
68   CRDT_Type = state_awset,
69   CRDT_Type_String = "state_awset",
70   io:format("EXPERIMENT ~p ", [ExperimentNumber]),
71   io:format("Node ~p (Adding elements) ~n", [Id]),
72   Threshold = (TotalNumberOfNodes*NumberOfValues),
73
74   %-------------------------------------
75   %Join cluster before adding values
76   %-------------------------------------
77   case {GeneratingUnderPartition, erlang:node()==NodeToJoin} of
78   {false, false} ->
79     lasp_peer_service:join(NodeToJoin),
80     io:format("Clustering is done ~n");
81   {true, false} ->
82     ok;
83   {_,true} ->
84     lasp:declare({CRDT_Id, CRDT_Type}, CRDT_Type)
85   end,
86
87   %-------------------------------------
88   %Generate Values
89   %-------------------------------------
90   StartSending = erlang:system_time(1000),
91   lasp:declare({CRDT_Id, CRDT_Type}, CRDT_Type),
92   generateValues(Id, CRDT_Id, CRDT_Type, NumberOfValues, SendingSpeed, All_At_Once),
93   Initial_CRDT_Size = erts_debug:flat_size(  lasp:query({CRDT_Id, CRDT_Type})  ),
94   io:format("Initial CRDT size: ~p (words) ~n", [Initial_CRDT_Size]),
95   EndSending = erlang:system_time(1000),
96   SendingTime = EndSending - StartSending,
97   %-------------------------------------
98   %Or Join cluster after adding values
99   %-------------------------------------
100   case {GeneratingUnderPartition, erlang:node()==NodeToJoin} of
101   {true, false} ->
102     lasp_peer_service:join(NodeToJoin),
103     io:format("Clustering is done ~n");
104   {false, false} ->
105     ok;
106   {_,true} -> ok
107   end,
108
109   %-------------------------------------
110   %Start Timer for convergence
111   %-------------------------------------
112   io:format("Waiting for convergence... ~n"),
113   InitialTimer = erlang:system_time(1000),
114   lasp:read({CRDT_Id, CRDT_Type}, {cardinality, Threshold}),
115   ConvergedTimer = erlang:system_time(1000),
116   file:copy(NetworkPath, NetworkPath2),
117
118   Final_CRDT_Size = erts_debug:flat_size(  lasp:query({CRDT_Id, CRDT_Type})  ),
119   io:format("Final CRDT size: ~p (words) ~n", [Final_CRDT_Size]),
120   %-------------------------------------
121   %End Timer for convergence
122   %-------------------------------------
```

```erlang
    %--------------------------------------
    %Write to file
    %--------------------------------------
    EllapsedTime = ConvergedTimer - InitialTimer,
    io:format("Correct number of elements in the set reached! (~p elements) ~n", [Threshold]),
    io:format("Convergence took ~p milliseconds ~n", [EllapsedTime]),
    io:format("Waiting experiment to finish on every node..."),
    lasp:update({<<"done">>, state_awset}, {add, Id}, self()), %Mark that I finished
    lasp:read({<<"done">>, state_awset}, {cardinality, TotalNumberOfNodes}), %Wait everyone finished
    io:format("OK! Writting output file ~n"),
    ExperimentEndTime = erlang:system_time(1000),
    TotalExperimentTime = ExperimentEndTime - ExperimentStartTime,
    io:format("Total experiment duration : ~p ~n", [TotalExperimentTime]),
    generateFileAdd(ExperimentNumber, Id, NodeToJoin, CRDT_Type_String, TotalNumberOfNodes, SendingSpeed, NumberOfValues,
    Threshold, EllapsedTime, GeneratingUnderPartition, SendingTime, All_At_Once, Initial_CRDT_Size, Final_CRDT_Size),
    io:format("Done. ~n"),
    io:format("~n"),
    file:delete(NetworkPath),
    timer:sleep(15000), %wait before stopping.
    partisan_peer_service:stop().




%% ==================================================================
%% Helpers for LaunchExperimentAdding
%% ==================================================================


%Generate the file to save measurements
generateFileAdd (ExperimentNumber, Id, NodeToJoin, CRDT_Type_String, TotalNumberOfNodes, SendingSpeed, NumberOfValues,
 Threshold, EllapsedTime, GeneratingUnderPartition, SendingTime, All_At_Once,Initial_CRDT_Size, Final_CRDT_Size) ->
    UniqueValue = integer_to_list(erlang:system_time(1000)),
    Path = "Memoire/Mesures/Exp"++integer_to_list(ExperimentNumber)++"/Node"++integer_to_list(Id)++"_"++UniqueValue++".txt",
    {ok, File} = file:open(Path, [write]),
    case (GeneratingUnderPartition) of
    true ->
      Partition = "true";
    false ->
      Partition = "false"
    end,
    ExperimentName = "Launching " ++integer_to_list(TotalNumberOfNodes) ++ " nodes at the same time, each generating a total of "
                ++integer_to_list(NumberOfValues)++  " elements (while under partition = " ++ Partition ++ ") "
                ++" then start timer and wait until all the elements converged",
    io:format(File, "~s~n", [ExperimentName]),
    io:format(File, "~s~n", [""]),
    io:format(File, "~s~n", ["============="]),
    io:format(File, "~s~n", ["Parameters :"]),
    io:format(File, "~s~n", ["============="]),
    io:format(File, "~s~n", ["Node Id:"]),
    io:format(File, "~s~n", [integer_to_list(Id)]), %Line7
    io:format(File, "~s~n", ["Node to join cluster: "]),
    io:format(File, "~s~n", [NodeToJoin]), %Line9
    io:format(File, "~s~n", ["Generating elements under partition: "]),
    io:format(File, "~s~n", [GeneratingUnderPartition]), %Line11
    io:format(File, "~s~n", ["Type of CRDT: "]),
    io:format(File, "~s~n", [CRDT_Type_String]), %Line13
    io:format(File, "~s~n", ["Number of nodes: "]),
    io:format(File, "~s~n", [integer_to_list(TotalNumberOfNodes)]), %Line15
    io:format(File, "~s~n", ["Number of elements generated by each node: "]),
    io:format(File, "~s~n", [integer_to_list(NumberOfValues)]), %Line17
    io:format(File, "~s~n", ["Sending all at once: "]),
    io:format(File, "~s~n", [All_At_Once]),
    io:format(File, "~s~n", ["Initial number of elements before adding:"]),
    io:format(File, "~s~n", ["0"]),
    io:format(File, "~s~n", ["Theoretical time (ms) between each element sending, for each node, valuable only if not sending all at once: "]),
    io:format(File, "~s~n", [integer_to_list(SendingSpeed)]),
    io:format(File, "~s~n", ["Time (ms) required (for this node) to generate and send all the elements: "]),
    io:format(File, "~s~n", [integer_to_list(SendingTime)]),
    io:format(File, "~s~n", ["Initial size of the CRDT (in term of words) : "]),
    io:format(File, "~s~n", [integer_to_list(Initial_CRDT_Size)]),
    io:format(File, "~s~n", ["Total final number of elements in the CRDT: "]),
    io:format(File, "~s~n", [integer_to_list(Threshold)]),
    io:format(File, "~s~n", ["Final size of the CRDT (in term of words) : "]),
    io:format(File, "~s~n", [integer_to_list(Final_CRDT_Size)]),
    io:format(File, "~s~n", ["Ellapsed time (ms) after elements generation to get the total " ++ integer_to_list(Threshold) ++ " elements:"]),
    io:format(File, "~s~n", [integer_to_list(EllapsedTime)]),


    %Network measure file
    NetworkPath2="Memoire/Mesures/Network/Node"++integer_to_list(Id)++"_Final.txt",
    ReceivedMessages = count_line(NetworkPath2)-1, %-1 because of the initial "NEW"
    io:format("number of received messages ~p ~n", [ReceivedMessages]),
    io:format(File, "~s~n", ["Number of message received before convergence : "]),
    io:format(File, "~s~n", [integer_to_list(ReceivedMessages)]),
    io:format(File, "~s~n", ["Number of message received per second : "]),
    Messages_speed = io_lib:format("~.3f",[((ReceivedMessages*1000)/EllapsedTime)]),
    io:format(File, "~s~n", [Messages_speed]),
    io:format(File, "~s~n", ["=================================================================================="]),
    io:format(File, "~s~n", ["=================================================================================="]),
    io:format(File, "~s~n", [""]),
    file:close(File),
    file:delete(NetworkPath2).
```

```erlang
220
221
222
223    %%Generate values for crdt
224    %%Limit is the number of values to generate
225    %% Maxvalue is the maximum value acceptable
226    %% Period is the time in ms between each value generation
227    %% Example: generateValues (10, 100, 1000) will generate a total of 10 values (numbers between 0 and 100) at speed of 1/sec
228
229    generateValues(Id, CRDT_Id, CRDT_Type, NumberOfValues, Period, All_At_Once) ->
230      io:format("Generating ~p values ~n", [NumberOfValues]),
231
232      Values = lists:seq( (Id-1)*NumberOfValues , (Id*NumberOfValues)-1 ),
233      case (All_At_Once) of
234      false ->
235        io:format("Adding the ~p generated elements, ", [NumberOfValues]),
236        io:format("at speed of one every ~p milliseconds ~n", [Period]);
237      true ->
238        io:format("Adding the ~p generated elements, ", [NumberOfValues]),
239        io:format("all at once ~n")
240      end,
241      generateValues_helper2(CRDT_Id, Values, Period, 1, NumberOfValues, CRDT_Type, All_At_Once).
242      %start counter at 2 to avoid sending the initial 0
243
244
245
246    %Helper2 to generate values (used to send them)
247    generateValues_helper2(CRDT_Id, ValuesArray, PeriodMs, Counter, NumberOfValues, CRDT_Type, All_At_Once) ->
248      case (All_At_Once) of
249      true ->
250        lasp:update({CRDT_Id, CRDT_Type}, {add_all, ValuesArray}, self());
251      false ->
252        if (Counter =< NumberOfValues) ->
253          CurrentValue = lists:nth(Counter, ValuesArray),
254          lasp:update({CRDT_Id, CRDT_Type}, {add, CurrentValue}, self()),
255          timer:sleep(PeriodMs),
256          generateValues_helper2(CRDT_Id, ValuesArray, PeriodMs, Counter+1, NumberOfValues, CRDT_Type, All_At_Once);
257        true ->
258          io:format("Finished adding values to CRDT ~n")
259        end
260      end.
261
262
263
264    %% ===================================================================
265    %% launchExperimentRemoving:
266    %% ===================================================================
267
268    launchExperimentRemoving(ExperimentNumber, NodeToJoin, CRDT_Id, TotalNumberOfNodes, All_At_Once, RemovingSpeed,
269    NumberOfValues, RemovingUnderPartition) ->
270      %----------------------------------------
271      %Little setup (putting initial elements)
272      %----------------------------------------
273      ExperimentStartTime = erlang:system_time(1000),
274      timer:sleep(1000), %start with a little sleep to allow NodeToJoin to be booted in case it is a bit slower than this node
275      Id = list_to_integer( lists:nth(2,string:split(lists:nth(1,string:split(atom_to_list(erlang:node()),"@")), "e")) ),
276      NetworkPath="Memoire/Mesures/Network/Node"++integer_to_list(Id)++".txt",
277      NetworkPath2="Memoire/Mesures/Network/Node"++integer_to_list(Id)++"_Final.txt",
278      file:write_file(NetworkPath,"NEW \n",[append]),
279      CRDT_Type = state_awset,
280      CRDT_Type_String = "state_awset",
281      case (Id) of
282      1 -> lasp:declare({CRDT_Id, CRDT_Type}, CRDT_Type);
283      _ -> ok
284      end,
285      timer:sleep(1000),
286      io:format("EXPERIMENT ~p ", [ExperimentNumber]),
287      io:format("Node ~p (Removing elements) ~n", [Id]),
288      Threshold = 0,
289      ValuesArray = lists:seq(0, ((TotalNumberOfNodes*NumberOfValues)-1) ),
290      % Valeurs de 0 à XX9 dont les indices vont de 1 à 1XX
291      lasp:update({CRDT_Id, CRDT_Type}, {add_all, ValuesArray}, <<04760>>),
292      %They consider all the initial values are from the same actor, that way they start with exactly the same CRDT
293      io:format("Initial ~p values are set ~n", [TotalNumberOfNodes*NumberOfValues]),
294      Initial_CRDT_Size = erts_debug:flat_size( lasp:query({CRDT_Id, CRDT_Type}) ),
295      io:format("Initial CRDT size: ~p (words) ~n", [Initial_CRDT_Size]),
296
297      case {RemovingUnderPartition, erlang:node()==NodeToJoin} of
298      {false, false} ->
299        lasp_peer_service:join(NodeToJoin),
300        io:format("Clustering is done ~n");
301      {true, false} ->
302        ok;
303      {_,true} -> ok
304      end,
305
306      %----------------------------------------
307      %Removing values
308      %----------------------------------------
309      RemovingTime = removeValues(Id, CRDT_Id, CRDT_Type, NumberOfValues, All_At_Once, RemovingSpeed),
310      io:format("Removing values took ~p ms ~n", [RemovingTime]),
311      %----------------------------------------
312      %Or Join cluster after removing values
313      %----------------------------------------
314      case {RemovingUnderPartition, erlang:node()==NodeToJoin} of
315      {true, false} ->
316        lasp_peer_service:join(NodeToJoin),
317        io:format("Clustering is done ~n");
318      {false, false} ->
319        ok;
320      {_,true} -> ok
321      end,
322
323      %----------------------------------------
324      %Start Timer for convergence
325      %----------------------------------------
```

```erlang
        io:format("Waiting for convergence... ~n"),
        InitialTimer = erlang:system_time(1000),
        lasp:read({CRDT_Id, CRDT_Type}, {cardinality, -1}), % argument -1 corresponds to detect it is empty
        %Ajouter une fonction dans state_awset pour cardinalityG et cardinalityS (bigger et smaller)
        EndTimer = erlang:system_time(1000),
        ConvergedTime = EndTimer - InitialTimer,
        file:copy(NetworkPath, NetworkPath2),
        io:format("Correct number of elements in the set reached! (~p elements) ~n", [Threshold]),
        Final_CRDT_Size = erts_debug:flat_size( lasp:query({CRDT_Id, CRDT_Type}) ),
        io:format("Final CRDT size: ~p (words) ~n", [Final_CRDT_Size]),
        io:format("Convergence took ~p ms ~n", [ConvergedTime]),
        %---------------------------------------
        %End Timer for convergence
        %---------------------------------------

        %---------------------------------------
        %Write to file
        %---------------------------------------
        io:format("Waiting experiment to finish on every node..."),
        lasp:update({<<"done">>, state_awset}, {add, Id}, self()), %Mark that I finished
        lasp:read({<<"done">>, state_awset}, {cardinality, TotalNumberOfNodes}), %Wait everyone finished
        io:format("OK! Writting output file ~n"),
        ExperimentEndTime = erlang:system_time(1000),
        TotalExperimentTime = ExperimentEndTime - ExperimentStartTime,
        io:format("Total experiment duration : ~p ~n", [TotalExperimentTime]),
        generateFileRmv (ExperimentNumber, Id, NodeToJoin, CRDT_Type_String, TotalNumberOfNodes, RemovingSpeed,
        NumberOfValues, Threshold, ConvergedTime, RemovingUnderPartition, RemovingTime, All_At_Once, Initial_CRDT_Size, Final_CRDT_Size),
        io:format("Done. ~n"),
        io:format("~n"),
        file:delete(NetworkPath),
        timer:sleep(15000), %wait before stopping.
        partisan_peer_service:stop().


%% ==============================================================
%% Helpers for LaunchExperimentRemoving
%% ==============================================================

removeValues(Id, CRDT_Id, CRDT_Type, NumberOfValues, All_At_Once, RemovingSpeed) ->
    ValuesToRemove = lists:seq( (Id-1)*NumberOfValues , (Id*NumberOfValues)-1 ),
    io:format("Removing unique ~p elements, ",[NumberOfValues]),
    StartRemovingTime = erlang:system_time(1000),
    case (All_At_Once) of
    true ->
        io:format("all at once ~n"),
        lasp:update({CRDT_Id, CRDT_Type}, {rmv_all, ValuesToRemove}, self());
    false ->
        io:format("at speed of one every ~p ms ~n", [RemovingSpeed]),
        removeValues_helper(CRDT_Id, CRDT_Type, NumberOfValues, ValuesToRemove, RemovingSpeed, 1)
    end,
    EndRemovingTime = erlang:system_time(1000),
    RemovingTime = EndRemovingTime - StartRemovingTime,
    io:format("Removing elements is done ~n"),
    RemovingTime.


removeValues_helper(CRDT_Id, CRDT_Type, NumberOfValues ,ValuesToRemove, RemovingSpeed, Counter) ->
    if (Counter =< NumberOfValues) ->
        CurrentValue = lists:nth(Counter, ValuesToRemove),
        lasp:update({CRDT_Id, CRDT_Type}, {rmv, CurrentValue}, self()),
        timer:sleep(RemovingSpeed),
        removeValues_helper(CRDT_Id, CRDT_Type, NumberOfValues ,ValuesToRemove, RemovingSpeed, Counter+1);
    true ->
        io:format("Finished removing values to CRDT ~n")
    end.


generateFileRmv (ExperimentNumber, Id, NodeToJoin, CRDT_Type_String, TotalNumberOfNodes, RemovingSpeed,
 NumberOfValues, Threshold, EllapsedTime, RemovingUnderPartition, RemovingTime, All_At_Once, Initial_CRDT_Size, Final_CRDT_Size) ->

    UniqueValue = integer_to_list(erlang:system_time(1000)),
    Path = "Memoire/Mesures/Exp"++integer_to_list(ExperimentNumber)++"/Node"++integer_to_list(Id)++"_"++UniqueValue++".txt",
    {ok, File} = file:open(Path, [write]),

    case (RemovingUnderPartition) of
    true ->
        Partition = "true";
    false ->
        Partition = "false"
    end,
    ExperimentName = "Launching " ++integer_to_list(TotalNumberOfNodes)
                    ++ " nodes at the same time with a common filled CRDT, each removing a total of "
                    ++integer_to_list(NumberOfValues)++  " elements (while under partition = " ++ Partition ++ ") "
                    ++" then start timer and wait until it converged",
    io:format(File, "~s~n", [ExperimentName]),
    io:format(File, "~s~n", [""]),
    io:format(File, "~s~n", ["============"]),
    io:format(File, "~s~n", ["Parameters :"]),
    io:format(File, "~s~n", ["============"]),
    io:format(File, "~s~n", ["Node Id:"]),
    io:format(File, "~s~n", [integer_to_list(Id)]), %Line7
    io:format(File, "~s~n", ["Node to join cluster: "]),
    io:format(File, "~s~n", [NodeToJoin]), %Line9
    io:format(File, "~s~n", ["Removing elements under partition: "]),
```

```erlang
421        io:format(File, "~s~n", [RemovingUnderPartition]), %Line11
422        io:format(File, "~s~n", ["Type of CRDT: "]),
423        io:format(File, "~s~n", [CRDT_Type_String]), %Line13
424        io:format(File, "~s~n", ["Number of nodes: "]),
425        io:format(File, "~s~n", [integer_to_list(TotalNumberOfNodes)]), %Line15
426        io:format(File, "~s~n", ["Number of elements removed by each node: "]),
427        io:format(File, "~s~n", [integer_to_list(NumberOfValues)]), %Line17
428        io:format(File, "~s~n", ["Removing all at once: "]),
429        io:format(File, "~s~n", [All_At_Once]),
430        io:format(File, "~s~n", ["Initial number of elements before removal: "]),
431        io:format(File, "~s~n", [integer_to_list(TotalNumberOfNodes*NumberOfValues)]),
432        io:format(File, "~s~n", ["Theoretical time (ms) between each element removing, for each node, valuable only if not removing all at once: "]),
433        io:format(File, "~s~n", [integer_to_list(RemovingSpeed)]),
434        io:format(File, "~s~n", ["Time (ms) required (for this node) to remove all its assigned elements: "]),
435        io:format(File, "~s~n", [integer_to_list(RemovingTime)]),
436        io:format(File, "~s~n", ["Initial size of the CRDT (in term of words) : "]),
437        io:format(File, "~s~n", [integer_to_list(Initial_CRDT_Size)]),
438        io:format(File, "~s~n", ["Total final number of elements in the CRDT: "]),
439        io:format(File, "~s~n", [integer_to_list(Threshold)]),
440        io:format(File, "~s~n", ["Final size of the CRDT (in term of words) when 0 elements left : "]),
441        io:format(File, "~s~n", [integer_to_list(Final_CRDT_Size)]),
442        io:format(File, "~s~n", ["Ellapsed time (ms) after elements removal to get the total " ++ integer_to_list(Threshold) ++ " elements:"]),
443        io:format(File, "~s~n", [integer_to_list(EllapsedTime)]),
444
445        %Network measure file
446        NetworkPath2="Memoire/Mesures/Network/Node"++integer_to_list(Id)++"_Final.txt",
447        ReceivedMessages = count_line(NetworkPath2)-1, %-1 because of the initial "NEW"
448        io:format("number of received messages ~p ~n", [ReceivedMessages]),
449        io:format(File, "~s~n", ["Number of message received before convergence : "]),
450        io:format(File, "~s~n", [integer_to_list(ReceivedMessages)]),
451        io:format(File, "~s~n", ["Number of message received per second : "]),
452        Messages_speed = io_lib:format("~.3f",[((ReceivedMessages*1000)/EllapsedTime)]),
453        io:format(File, "~s~n", [Messages_speed]),
454        io:format(File, "~s~n", ["================================================================================================="]),
455        io:format(File, "~s~n", ["================================================================================================="]),
456        io:format(File, "~s~n", [""]),
457        file:close(File),
458        file:delete(NetworkPath2).
459
460
461
462    %% ==============================================================
463    %% launchExperimentDynamic:
464    %% ==============================================================
465
466    launchExperimentDynamic(ExperimentNumber, NodeToJoin, CRDT_Id, SendingPeriod) ->
467        io:format("Experiment ~p ~n",[ExperimentNumber]),
468        timer:sleep(1000), %start with a little sleep to allow NodeToJoin to be booted in case it is a bit slower than this node
469        Id = list_to_integer( lists:nth(2,string:split(lists:nth(1,string:split(atom_to_list(erlang:node()),"@")), "e")) ),
470        lasp_peer_service:join(NodeToJoin),
471        MyRange = lists:seq( (Id-1)*10000 , (Id*10000)-1 ), %0-9999, 10000-19999, 20000-29999,... Each node has its own range of 1000
472        StartingElements= lists:seq( ((Id-1)*10000)+5000 , (Id*10000)-1 ),
473        lasp:update({CRDT_Id, state_awset}, {add_all, StartingElements}, self() ), %CRDT already contains 5000-9999, 15000-19999, 25000-29999...
474        Path = "Memoire/Mesures/Exp"++integer_to_list(ExperimentNumber)++"/Node"++integer_to_list(Id)++".txt",
475        {ok, File} = file:open(Path, [write]),
476        ExperimentName = "Launching nodes that will add/remove elements at a specific speed. See info file for more details.",
477        io:format(File, "~s~n", [ExperimentName]),
478
479        %Start=erlang:system_time(1000),
480        startLoop(MyRange, 0, CRDT_Id, SendingPeriod, Path).
481
482
483
484
485    %% ==============================================================
486    %% Helpers for launchExperimentDynamic:
487    %% ==============================================================
488
489    startLoop(Range,AddIndex, CRDT_Id, SendingPeriod, Path) ->
490        StartOperation=erlang:system_time(1000),
491        RemoveIndex=(AddIndex+5000) rem 10000,
492        ElementToAdd=lists:nth(AddIndex+1, Range), %Start with NextIndex=1 -> it had 500-999 and we add 0
493        ElementToRemove=lists:nth(RemoveIndex+1,Range), %Starting example: it had 500-999, we add 0 and remove 500.
494        io:format("Added element: ~p ~n", [ElementToAdd]),
495        io:format("Removed element: ~p ~n", [ElementToRemove]),
496        io:format("~n"),
497        lasp:update({CRDT_Id, state_awset}, {add, ElementToAdd}, self()),
498        lasp:update({CRDT_Id, state_awset}, {rmv, ElementToRemove}, self()),
499        {ok, CurrentContent} = lasp:query({CRDT_Id, state_awset}),
500        ReadableCurrentContent = sets:to_list(CurrentContent),
501        CurrentTime = erlang:system_time(1000),
502        Size = length(ReadableCurrentContent),
503        file:write_file(Path,"\n",[append]),
504        file:write_file(Path,"NEXT \n",[append]),
505        file:write_file(Path,"TIME " ++ integer_to_list(CurrentTime)++"\n",[append]),
506        file:write_file(Path,"ADDED ELEMENT " ++ integer_to_list(ElementToAdd)++"\n",[append]),
507        file:write_file(Path,"REMOVED ELEMENT " ++ integer_to_list(ElementToRemove)++"\n",[append]),
508        file:write_file(Path,"CRDT Size " ++ integer_to_list(Size)++"\n",[append]),
509        file:write_file(Path,"CONTENT : \n",[append]),
510        file:write_file(Path,io_lib:format("~p.~n", [ReadableCurrentContent]),[append]),
511        EndOperation=erlang:system_time(1000),
512        OperationDuration=EndOperation-StartOperation,
513        timer:sleep(max(SendingPeriod-OperationDuration, 0)),
514        startLoop(Range, ((AddIndex+1) rem 10000), CRDT_Id, SendingPeriod, Path).
515
516
517
518    %% ==============================================================
519    %% launchContinuousMeasures
520    %% ==============================================================
521
522    launchContinuousMeasures(MeasurePeriod, TimeOut, Debug) ->
523    %MeasurePeriod should be at minimum state_interal*4
524
525        Default_interval = 10000,
526        State_interval = lasp_convergence_measure:getInternalStateInterval(Default_interval),
527        BestInterval = 4*State_interval,
528
529        case (MeasurePeriod < BestInterval) of
530          true ->
```

```erlang
        RealInterval = BestInterval; % if measurePeriod too small, put to ideal interval
      false ->
        Reminder = MeasurePeriod rem BestInterval,
        Offset = State_interval - Reminder,
        RealInterval = MeasurePeriod + Offset
        % if measurePeriod bigger than ideal interval, aff an offset to reach an integer number of state_interval
    end,

    case Debug of
    true ->
      io:format("Continuous Measures Started in Talkative mode ~n");
    false ->
      io:format("Continuous Measures Started in Silent mode ~n")
    end,

    Id=list_to_integer( lists:nth(2,string:split(lists:nth(1,string:split(atom_to_list(erlang:node()),"@")), "e")) ),
    NetworkPath="Memoire/Mesures/Network/Node"++integer_to_list(Id)++".txt",
    file:delete(NetworkPath),
    Self = self(),
        _Pid = spawn (fun() ->
            continuousMeasurementLoop(Id, RealInterval, TimeOut, Debug),
            Self ! {self(), ok} end),
    _Pid.


continuousMeasurementLoop(Id,MeasurePeriod, TimeOut, Debug) ->
    LoopStartTime = erlang:system_time(1000),
    {LeaderId, Cluster_size}=lasp_leader_election:checkLeader(20000), %MeasurePeriod must be at least 4*realConvergenceTime
    case Debug of
    true ->
      io:format("============================= ~n"),
      io:format("NEW MEASURE ROUND ~n"),
      io:format("========================= ~n"),
      io:format("My leader is: ~p ~n", [LeaderId]);
    false ->
      ok
    end,

    case {LeaderId==Id, Cluster_size>0} of
    {true,true} -> %I am LEADER

      %Leader starts by removing TimeStamps in case previous leader crashed and did not remove them.
      {ok , RawPreviousTimeStamps} = lasp:query({<<"basic_task">>, state_awset}),
      PreviousTimeStamps = sets:to_list(RawPreviousTimeStamps),
      lasp:update({<<"basic_task">>, state_awset}, {rmv_all, PreviousTimeStamps}, self()),
      %Remove in case some node crashed and did not remove its TimeStamp

      %Measure Phase
      StartTime=erlang:system_time(1000),
      lasp:update({<<"leader_task">>, state_awset}, {add, Id}, self()), %I add my Id
      case Debug of
      true ->
        io:format("[LEADER] Wait for ~p nodes to answer... ", [Cluster_size]);
      false ->
        ok
      end,
      readThresholdMaxDuration({<<"basic_task">>, state_awset}, {cardinality, Cluster_size},TimeOut),
      %I wait everyone answered. Skip after TimeOut ms waiting
      case Debug of
      true ->
        io:format("OK! ~n");
      false ->
        ok
      end,

      %Compute Phase
      RoundTripTime = erlang:system_time(1000)-StartTime,
      {ok, TimeStampSet}=lasp:query({<<"basic_task">>, state_awset}),
      {WorstConvergenceTime, IndividualConvergenceTimes, RatioMsg} = computeWorstConvergenceTime(TimeStampSet, StartTime),
      NewConvergenceInfos = #{roundTripTime => RoundTripTime, worstConvergenceTime => WorstConvergenceTime,
      individualConvergenceTimes => IndividualConvergenceTimes, networkUsage => RatioMsg},
      PreviousConvergenceTime=getSystemConvergenceInfos(),
      lasp:update({<<"system_convergence">>, state_awset}, {rmv, PreviousConvergenceTime}, self()),
      lasp:update({<<"system_convergence">>, state_awset}, {add, NewConvergenceInfos}, self()),
      case Debug of
      true ->
        io:format("[LEADER] New convergence infos : ~p ~n", [NewConvergenceInfos]);
      false ->
        ok
      end,
      %Reset Phase
      timer:sleep(1000),
      {ok, PotentialLeadersId}=lasp:query({<<"leader_task">>, state_awset}), %Useful to remove potential previous leader Id if he crashed.
      ListPotentialLeadersId=sets:to_list(PotentialLeadersId), %Thus we remove ListOfPotentialLeadersId and not simply my own Id.

      lasp:update({<<"leader_task">>, state_awset}, {rmv_all, ListPotentialLeadersId}, self()), %I remove my Id (reset for next measure)
      readThresholdMaxDuration({<<"basic_task">>, state_awset}, {cardinality, -1}, TimeOut),
      %I wait everyone removed its TimeStamp (reset for next measure). Skip after TimeOut msec waiting

      {ok , RawPreviousTimeStamps2} = lasp:query({<<"basic_task">>, state_awset}),
      PreviousTimeStamps2 = sets:to_list(RawPreviousTimeStamps2),
      lasp:update({<<"basic_task">>, state_awset}, {rmv_all, PreviousTimeStamps2}, self()),
      %Remove in case some node crashed and did not remove its TimeStamp
      case Debug of
      true ->
        io:format("[LEADER] Reset is done ! ~n");
      false ->
```

```erlang
                    ok
                end,

            LoopEndTime=erlang:system_time(1000),
            LoopDuration=LoopEndTime-LoopStartTime,
            timer:sleep(max(MeasurePeriod-LoopDuration, 0));



        {false,true} ->%I am NOT-LEADER

            %Measure Phase
            case Debug of
            true ->
                io:format("[BASIC] Waiting for leader measure signal... ");
            false ->
                ok
            end,
            StartWaiting = erlang:system_time(1000),
            %io:format("put strings to try to figure out when read is timing out"),
            readThresholdMaxDuration({<<"leader_task">>, state_awset}, {cardinality, 1},TimeOut), %I wait to detect leader added his Id
            case Debug of
            true ->
                io:format("OK ! ~n");
                %Timer = erlang:system_time(1000),
                %io:format("at ~p", [Timer]); % TO REMOVE
            false ->
                ok
            end,
            TimeStamp=erlang:system_time(1000),
            WaitedTime = TimeStamp - StartWaiting,
            Basic_MessageRatio = computeNetworkUsage(Id, WaitedTime),
            %lasp:update({<<"basic_task">>, state_awset}, {add, {Id, TimeStamp}}, self()), %I add my {Id, TimeStamp}
            lasp:update({<<"basic_task">>, state_awset}, {add, {Id, TimeStamp, Basic_MessageRatio}}, self()), %WHY THIS LINE MAKES IT TIME OUT??
            %Reset Phase

            readThresholdMaxDuration({<<"leader_task">>, state_awset}, {cardinality, -1},TimeOut), %I wait to detect leader removed his Id (reset for next measure)
            case Debug of
            true ->
                io:format("[BASIC] Reset is done ! ~n");
            false ->
                ok
            end,
            lasp:update({<<"basic_task">>, state_awset}, {rmv, {Id, TimeStamp, Basic_MessageRatio}}, self()); %I remove my {Id, TimeStamp} (reset for next measure)
        {_,false} -> %Cluster is empty, I am alone
            case Debug of
            true ->
                io:format("I am alone (or just booted) ~n");
            false ->
                ok
            end,
            timer:sleep(500), %Will check for new leader every 500ms
            ok
    end,
    continuousMeasurementLoop(Id, MeasurePeriod, TimeOut, Debug).


computeWorstConvergenceTime(TimeStampSet, StartTime) ->
    TimeStampList=sets:to_list(TimeStampSet),
    case length(TimeStampList) of
    0 ->
        WorstConvergenceTime=nan,
        OrderedConvergenceTimes = [],
        MessagesRatio=nan;
    _ ->
        ConvergenceTimes = lists:map(fun({A,B,_}) -> #{id => A, convergenceTime => B-StartTime} end, TimeStampList),
        OrderedConvergenceTimes = lists:sort( fun(A,B) -> maps:get(convergenceTime, A) =< maps:get(convergenceTime, B) end, ConvergenceTimes),
        WorstConvergenceTime = lists:nth(length(OrderedConvergenceTimes), OrderedConvergenceTimes),
        MessagesNumbers = lists:map(fun({_,_,C}) -> C end, TimeStampList),
        Sum = lists:sum(MessagesNumbers),
        MessagesRatio = Sum
    end,
    {WorstConvergenceTime, OrderedConvergenceTimes, MessagesRatio}.




getSystemConvergenceInfos() ->
    {ok, RawConvergenceInfo} = lasp:query({<<"system_convergence">>, state_awset}),
    ConvergenceInfoList=sets:to_list(RawConvergenceInfo),
    case length(ConvergenceInfoList) of
    1 ->
        ConvergenceInfo = lists:nth(1, ConvergenceInfoList);
    _ ->
        ConvergenceInfo = nomeasure
    end,
    ConvergenceInfo.

getSystemConvergenceTime() ->
    {ok, RawInfos} = lasp:query({<<"system_convergence">>, state_awset}),
    InfosList = sets:to_list(RawInfos),
    case length(InfosList) of
    1 ->
        Infos = lists:nth(1, InfosList),
        ConvergenceTime = maps:get(convergenceTime , maps:get(worstConvergenceTime, Infos));
    _ ->
        ConvergenceTime = noValue
    end,
    ConvergenceTime.


getSystemWorstNodeId() ->
    {ok, RawInfos} = lasp:query({<<"system_convergence">>, state_awset}),
    InfosList = sets:to_list(RawInfos),
    case length(InfosList) of
    1 ->
```

```erlang
736        Infos = lists:nth(1, InfosList),
737        Node = maps:get(id ,maps:get(worstConvergenceTime, Infos));
738      _ ->
739        Node = noValue
740    end,
741    Node.
742
743
744  getSystemRoundTrip() ->
745    {ok, RawInfos} = lasp:query({<<"system_convergence">>, state_awset}),
746    InfosList = sets:to_list(RawInfos),
747    case length(InfosList) of
748    1 ->
749        Infos = lists:nth(1, InfosList),
750        RoundTrip = maps:get(roundTripTime, Infos);
751      _ ->
752        RoundTrip = noValue
753    end,
754    RoundTrip.
755
756  getIndividualConvergenceTimes() ->
757    {ok, RawInfos} = lasp:query({<<"system_convergence">>, state_awset}),
758    InfosList = sets:to_list(RawInfos),
759    case length(InfosList) of
760    1 ->
761        Infos = lists:nth(1, InfosList),
762        IndividualTimes = maps:get(individualConvergenceTimes, Infos);
763      _ ->
764        IndividualTimes = noValue
765    end,
766    IndividualTimes.
767
768  getSystemNetworkUsage() ->
769    {ok, RawInfos} = lasp:query({<<"system_convergence">>, state_awset}),
770    InfosList = sets:to_list(RawInfos),
771    case length(InfosList) of
772    1 ->
773        Infos = lists:nth(1, InfosList),
774        NetworkUsage = maps:get(networkUsage, Infos);
775      _ ->
776        NetworkUsage = noValue
777    end,
778    NetworkUsage.
779
780  getConvergenceTime(Id) -> %VERIFIER QUON NA PAS UN TRUC VIDE
781    {LeaderId, _ }=lasp_leader_election:checkLeader(20000),
782    case (Id==LeaderId) of
783    true ->
784        MyConvergenceTime = getSystemConvergenceTime(); %I am leader, there is no specific measure for myself. I take the worst case instead.
785    false ->
786        IndividualConvergenceTimes = getIndividualConvergenceTimes(),
787        case IndividualConvergenceTimes of
788        noValue ->
789          MyConvergenceTime = noValue;
790        _ ->
791          MyIndividualConvergenceInfoList = lists:filter(fun(A) -> maps:get(id, A)==Id end, IndividualConvergenceTimes),
792          case length(MyIndividualConvergenceInfoList) of
793          0 ->
794            MyConvergenceTime = noValue;
795          _ ->
796            MyIndividualConvergenceInfo = lists:nth(1, MyIndividualConvergenceInfoList),
797            MyConvergenceTime = maps:get(convergenceTime, MyIndividualConvergenceInfo)
798          end
799        end
800    end,
801    MyConvergenceTime.
802
803
804
805
806  readThresholdMaxDuration(CRDT_Id, Threshold, MaxDuration) ->
807    Self = self(),
808    _Pid = spawn (fun() ->
809            lasp:read(CRDT_Id, Threshold),
810            Self ! {self(), ok} end),
811    receive
812      {_PidSpawned, ok} -> ok
813    after
814      MaxDuration ->
815        io:format("time out! ~n"),
816        erlang:exit(_Pid, kill)
817    end.
818
819  readThresholdMaxDurationSilent(CRDT_Id, Threshold, MaxDuration) ->
820    Self = self(),
821    _Pid = spawn (fun() ->
822            lasp:read(CRDT_Id, Threshold),
823            Self ! {self(), ok} end),
824    receive
825      {_PidSpawned, ok} -> ok
826    after
827      MaxDuration ->
828        erlang:exit(_Pid, kill)
829    end.
830
831  computeNetworkUsage(Id, Period) ->
832    NetworkPath="Memoire/Mesures/Network/Node"++integer_to_list(Id)++".txt",
833    ReceivedMessages = count_line(NetworkPath),
834    MessagesPerSec = round( (ReceivedMessages*1000) / Period),
835    file:delete(NetworkPath),
836    MessagesPerSec.
837
838  setStateInterval(NewInterval) ->
839    {ok, Interval_query} = lasp:query({<<"state_interval">>, state_awset}),
```

```erlang
    Interval_list = sets:to_list(Interval_query),
    lasp:update({<<"state_interval">>, state_awset}, {rmv_all, Interval_list}, self()),
    lasp:update({<<"state_interval">>, state_awset}, {add, NewInterval}, self()),
    io:format("state sending interval set to: ~p ms. ~n", [NewInterval]).

getInternalStateInterval(Default_interval) ->
    case (readThresholdMaxDurationSilent({<<"state_interval">>, state_awset}, {cardinality, 1},1)) of
    %ReadwithMaxDuration of 1ms for booting to select default value instead
    ok -> Interval = getStateIntervalHelper(Default_interval);
    true -> Interval = Default_interval
    end,
    Interval.

getStateInterval() ->
    {ok, Interval_query} = lasp:query({<<"state_interval">>, state_awset}),
    Interval_list = sets:to_list(Interval_query),
    case length(Interval_list) of
    1 ->
      Interval = lists:nth(1, Interval_list);
    _ ->
      Interval = default
    end,
    Interval.

getStateIntervalHelper(Default_interval) ->

    {ok, Interval_query} = lasp:query({<<"state_interval">>, state_awset}),
    Interval_list = sets:to_list(Interval_query),
    case length(Interval_list) of
    1 ->
      Interval = lists:nth(1, Interval_list);
    _ ->
      Interval=Default_interval
    end,
    Interval.


%% =================================================================
%% Other small tests:
%% =================================================================

addition_awset_time() ->
    lasp:declare({<<"special_awset_measure">>, state_awset}, state_awset),
    Start = erlang:system_time(1000),
    fprof:trace(start),
    lasp:update({<<"special_awset_measure">>, state_awset}, {add , 5}, self()),
    fprof:trace(stop),
    EndAdd = erlang:system_time(1000),
    Duration = EndAdd - Start,
    io:format("Computation duration to add a simple little element to an empty awset: ~p ms. ~n", [Duration]).


addition_orset_time() ->
    lasp:declare({<<"special_orset_measure">>, state_orset}, state_orset),
    Start = erlang:system_time(1000),
    lasp:update({<<"special_orset_measure">>, state_orset}, {add , 5}, self()),
    Duration = erlang:system_time(1000) - Start,
    io:format("Computation duration to add a simple little element to an empty orset: ~p ms. ~n", [Duration]).

addition_gcount_time() ->
    lasp:declare({"<<special_gcounter_measure>>", state_gcounter}, state_gcounter),
    Start = erlang:system_time(1000),
    lasp:update({"<<special_gcounter_measure>>", state_gcounter}, increment, self()),
    Duration = erlang:system_time(1000) - Start,
    io:format("Computation duration to increment a starting gcounter: ~p ms. ~n", [Duration]).


simpleAddition() ->

    case (erlang:node()=='node1@127.0.0.1') of
    false ->
      lasp_peer_service:join('node1@127.0.0.1'),
      io:format("Clustering is done ~n");
    true ->
      lasp:declare({<<"set99">>, state_awset}, state_awset)
    end,
    Id = list_to_integer( lists:nth(2,string:split(lists:nth(1,string:split(atom_to_list(erlang:node()),"@")), "e")) ),
    Values = lists:seq(1000*Id, (1000*Id)+1000),
    lasp:update({<<"set99">>, state_awset}, {add_all, Values}, self()),
    io:format("I added 1000 elements ~n"),
    lasp:read({<<"set99">>, state_awset}, {cardinality, 5000}).


partitionHelper() ->

timer:sleep(5000),
lasp_peer_service:join('node1@127.0.0.1'),
StartTime=erlang:system_time(1000),
lasp:read({<<"partition">>, state_awset}, {cardinality, 1}),
Duration=erlang:system_time(1000) - StartTime,
io:format("Duration to catchup with the cluster: ~p ~n", [Duration]).


%This is called by the partisan when the node receives a message!
%I must make it able to modify a global variable (related to this file)
%That way I will be available to print information about it in the output file.
messageReceived() ->
    Id = list_to_integer( lists:nth(2,string:split(lists:nth(1,string:split(atom_to_list(erlang:node()),"@")), "e")) ),
    Path="Memoire/Mesures/Network/Node"++integer_to_list(Id)++".txt",
```

```
941    Phrase = "Message received \n",
942    file:write_file(Path,Phrase,[append]).
943
944
945  created() ->
946    io:format("PARTISAN CREATED ! ~n").
947
948
949  count_line(Filename) ->
950      case file:open(Filename, [read]) of
951          {ok, IoDevice} ->
952              Count = count_line(IoDevice, 0),
953              file:close(IoDevice),
954              Count;
955          {error, Reason} ->
956              io:format("~s open error  reason:~s~n", [Filename, Reason]),
957              ng
958      end.
959
960  count_line(IoDevice, Count) ->
961      case file:read_line(IoDevice) of
962          {ok, _} -> count_line(IoDevice, Count+1);
963          eof     -> Count
964      end.
965
966  wait_threshold(UnixTimeThreshold) ->
967    Current = erlang:system_time(1000),
968    case (Current < UnixTimeThreshold) of
969      true -> wait_threshold(UnixTimeThreshold);
970      false -> ok
971    end.
```

And that concludes the biggest coding part of this work.

6.6 Lasp_state_based_synchronization_backend

One little modification was required directly within Lasp source code. Indeed, the lasp_state_based_synchronization_backend erlang file was implementing the module responsible for

launching the state sending after every state_interval wait. I had to allow the state_interval there to be taken from the local state (replica) of the state_interval specific CRDT instead of using the hard-coded value. Also, when first node boots, no state_interval value is set in a CRDT so the boot must be working correctly within the state_interval being set on a CRDT yet. This was implemented by checking for state_awset value within local replica with a time limitation of 1ms meaning if it does not directly find a value, it uses the default value.

Only lines 326 and 341 were modified, see below:

```
320  %%%===============================================================
321  %%% Internal functions
322  %%%===============================================================
323
324  %% @private
325  schedule_state_synchronization() ->
326    Default_interval=10000,
327      ShouldSync = true
328              andalso (
329                ?SYNC_BACKEND:peer_to_peer_mode()
330                orelse
331                (
332                ?SYNC_BACKEND:client_server_mode()
333                andalso
334                not (?SYNC_BACKEND:i_am_server() andalso ?SYNC_BACKEND:reactive_server())
335                )
336              ),
337
338      case ShouldSync of
339          true ->
340              %Interval = lasp_config:get(state_interval, 10000), %This is the original Lasp code
341              Interval = lasp_convergence_measure:getInternalStateInterval(Default_interval),
342
343              ObjectFilterFun = fun(_, _) -> true end,
344              case lasp_config:get(jitter, false) of
```