

Table des matières

1. INTRODUCTION.....	4
1.1 Context	4
1.1.1 Usual approach	5
1.1.2 New approach.....	5
1.2 CRDT	6
1.2.1 Principles.....	6
1.2.2 Advantages	7
1.3 Lasp	8
1.3.1 Lasp libraries	9
1.3.2 ORSWOT	9
1.4 Goals and contributions	13
1.4.1 Developed tools and API.....	13
1.4.2 Measurements and results	13
1.5 Summary and structure	15
2. DEVELOPED TOOLS AND API.....	16
2.1 Measurement tools	16
2.1.1 Principle	16
2.1.2 API.....	18
2.2 Adaptation tools	23
2.2.1 Principle	23
2.2.2 API	23
2.3 Scripts	25
2.3.1 Static measurement scripts	25
2.3.2 Dynamic measurement scripts	27
2.3.3 Quality-of-life scripts	28
2.4 Lasp corrections	29
2.4.1 Memory leak.....	29
2.4.2 Readme improvement	30
3. MEASURES AND RESULTS.....	32
3.1 Parameters	32
3.2 Number of Nodes	33
3.3 Nodes distance	37
3.3 CRDT content	38

3.4 CRDT operation	40
3.5 Partition	43
3.6 Value update speed	45
3.7 State sending period.....	49
4. CONCLUSION	53
4.1 Summary	53
4.2 Developed tools conclusion	53
4.3 Results analysis conclusion	54
4.4 Future work.....	55
4.5 Personal opinion	57
4.6 Methodology.....	58
5. BIBLIOGRAPHY	59

Table of figures

Figure 1 : IoT devices vs human population.....	p4
Figure 2 : Usual approach very simple schematic.....	p5
Figure 3 : Peer-to-peer approach very simple schematic.....	p6
Figure 4 : CRDT add-min set schematic example.....	p7
Figure 5 : <u>Lasp</u> official logo.....	p8
Figure 6 : ORSWOT general structure example.....	p10
Figure 7 : <u>Orswot</u> element addition example.....	p11
Figure 8 : <u>Orswot</u> element removal example.....	p11
Figure 9 : <u>Orswot</u> states merging example.....	p12
Figure 10 : <u>Orswot</u> timeline example.....	p12
Figure 11 : <u>launchContinuousMeasures</u> debug=false real example.....	p18
Figure 12 : <u>launchContinuousMeasures</u> debug=true real example.....	p19
Figure 13 : <u>getSystemConvergenceInfos</u> real example.....	p20
Figure 14 : <u>getSystemConvergenceTime</u> real example.....	p20
Figure 15 : <u>getSystemWorstNodeId</u> real example.....	p21
Figure 16 : <u>getSystemRoundTrip</u> real example.....	p21
Figure 17 : <u>getSystemNetworkUsage</u> real example.....	p22
Figure 18 : <u>getIndividualConvergenceTimes</u> real example.....	p22
Figure 19 : <u>getConvergenceTime</u> real example.....	p23
Figure 20 : <u>setStateInterval</u> real example.....	p24
Figure 21 : <u>getInterval</u> real example.....	p24
Figure 22 : Graphs convergence time and network usage for different cluster size (nodes adding 10 elements each).....	p34
Figure 23 : Graphs convergence time and network usage for different cluster size (nodes adding 100 elements each).....	p35
Figure 24 : Graphs convergence time and network usage for different cluster size (nodes removing 10,100 elements).....	p36
Figure 25 : Graphs convergence time and network usage for different nodes distance (nodes adding 10 elements each).....	p37
Figure 26 : Graphs convergence time and network usage for different CRDT content (5 nodes adding elements).....	p39
Figure 27 : Graphs convergence time and network usage for different CRDT content (5 nodes removing elements).....	p39
Figure 28 : Graph convergence time for different CRDT operation (5 nodes).....	p40
Figure 29 : Graph network usage for different CRDT operation (5 nodes).....	p41
Figure 30 : Graphs convergence time and network usage for different CRDT operation (10 nodes).....	p42
Figure 31 : Graphs Impact of partition on convergence time and network usage (on a 5 nodes cluster).....	p44
Figure 32 : Graphs convergence time and network usage for a partitioned node (5 nodes cluster).....	p45
Figure 33 : Graph convergence time for continuous updates at 2 updates/sec (5 nodes cluster).....	p46
Figure 34 : Overviews convergence time for continuous updates at different speeds (5 nodes).....	p48
Figure 35 : Graph convergence time for different <u>state intervals</u> (5 nodes).....	p50
Figure 36 : Graph Network usage for different <u>state intervals</u> (5 nodes).....	p51
Figure 37 : Developed tools real usage example.....	p53

1. Introduction

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

1.1 Context

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

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

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

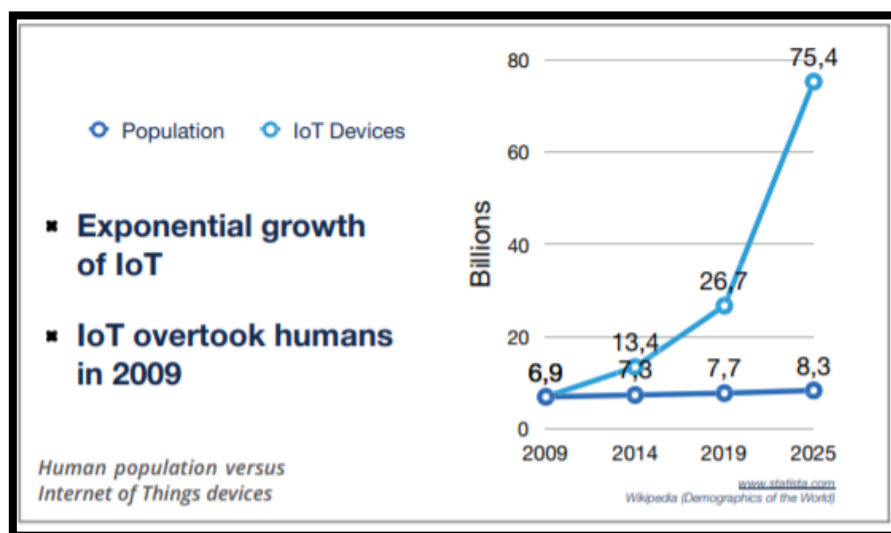


Figure 1: IoT devices vs human population

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 Usual approach

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

Furthermore, this kind of distributed structure usually relies on redundancy with replicated databases in multiple data-centers to achieve high scalability, adding more complexity to handle causality and operation order consistency between replicas, introducing Consensus algorithms. 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 Raft² (which is a lighter implementation of consensus) instead of Paxos³ (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 theorem⁴), 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.

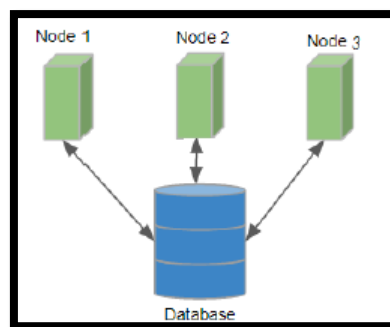


Figure 2: Usual approach very simple schematic

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 (CRDTs⁵). It is the key concept that will be detailed below to understand this new approach along with all its advantages.

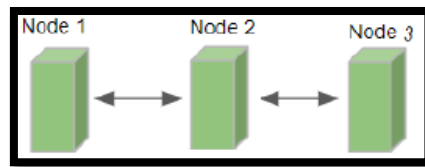


Figure 3: Peer-to-peer approach very simple schematic

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 properties⁶:

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 companies such as Riot Games, TomTom, Bet365, SoundCloud and some others⁷.

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 example⁶ with an add-wins set (if concurrent add and remove occur, the add wins).

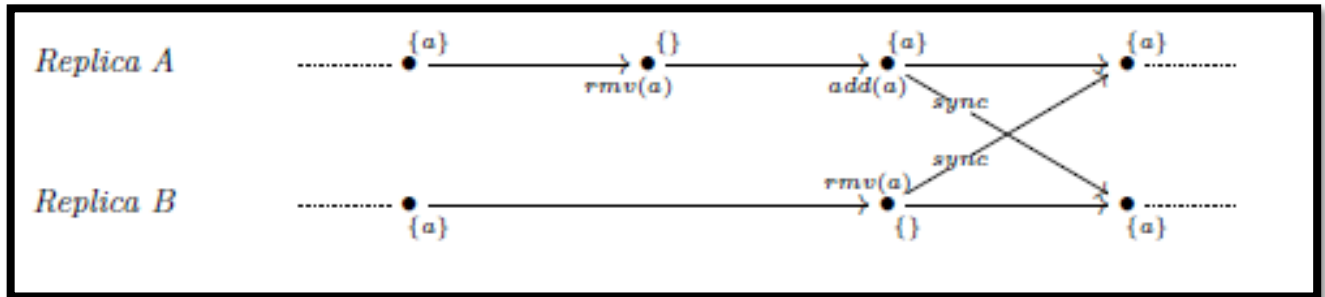


Figure 4: CRDT add-min set schematic example

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 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 algorithms⁸. 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 project⁹ leading to Lasp development).

Strong eventual Consistency (SEC¹⁰) is achievable to the fact every node receiving the same set of updates (in any order) have equivalent state and the fact every update will be eventually delivered to every node due to peer-to-peer communications. It is in fact even stronger than that since nodes do not require to receive the exact same set of updates, some previous updates may not be received that it will not perturb the system as long as recent messages eventually deliver.

In regard of the CAP theorem, CRDT model allows strong eventual 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 quickly¹¹ and has been adopted by some big companies as previously mentioned.

1.3 Lasp

Lasp¹² is an experimental implementation of CRDTs developed by the Ecole Polytechnique de Louvain (EPL) research team and initiated in 2013 with the impetus of two European projects; SyncFree⁹ in 2013 then LightKone¹³ in 2017. More precisely, it takes the form of a group of Erlang libraries acting together to offer a programming framework based on CRDT. There entire project can be found on their official github repositories: <https://github.com/lasp-lang/lasp>.



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 (Partisan¹⁴), 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 composition¹⁵ on CRDTs without requiring the developer to handle these problems himself. The idea is to consider the CRDT as an input stream and to output a stream of values always leading to the known most recent value. With this approach, the developer has tools to compute things based on a CRDT while his computations will be automatically updated with the CRDT. Thus Lasp offers a complete API to facilitate always updated unions, intersection, maps,... on CRDTs. This particular aspect is very convenient but will not be discussed in this master thesis since it will mainly focus on the distribution and communication part without detailing in depth the end developer aspects.

1.3.2 ORSWOT

Since Lasp offers multiple different CRDTs with their own representation and metadata, selecting one particular CRDT was a good starting point to have a reference to understand CRDTs principle and practical implementation while being able to measure its performances. The CRDT that was mainly used for this work is the ORSWOT. It is a relatively recent CRDT that offers some good properties since it represents a set where nodes can add or remove elements allowing it to represent basically anything even if it might not be optimized for every kind of elements. For example, it could represent a set containing only an integer where nodes only operation would be to increment the integer by one. This example is possible with an ORSWOT while it could be better optimized using a CRDT specifically designed for counter. The fact the ORSWOT allows many possible usages was a good starting argument then comes the fact it is relatively well optimized for general usage. Indeed, compared to its predecessor, the well-known ORSET (Observe Remove Set¹⁶), 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 Bet365¹⁷ 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.

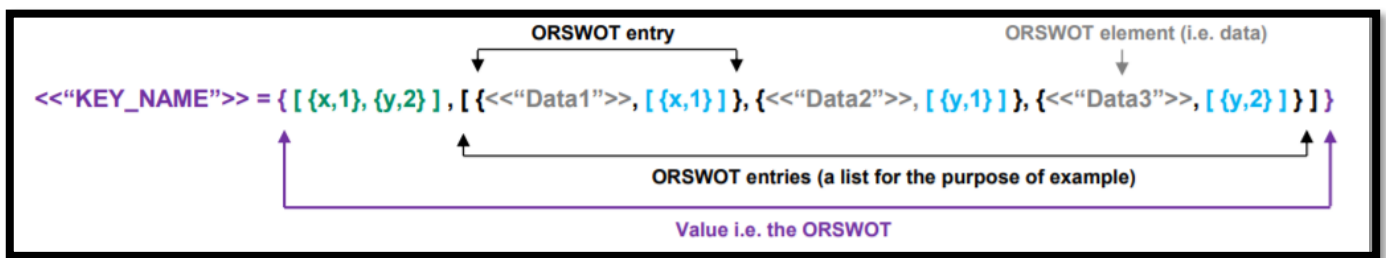


Figure 6: ORSWOT general structure example

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}], [{<<"Data1">>, [{x,1}]}] }`

Results in the new ORSWOT:

`{ [{x,1}, {y,1}], [{<<"Data1">>, [{x,1}] }, {<<"Data2">>, [{y,1}]}] }`

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

`[<<"Data1">>, <<"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}], [{<<"Data1">>, [{x,1}] }, {<<"Data2">>, [{y,1}]}] }`

Results in the new ORSWOT:

`{ [{x,1}, {y,1}], [{<<"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.

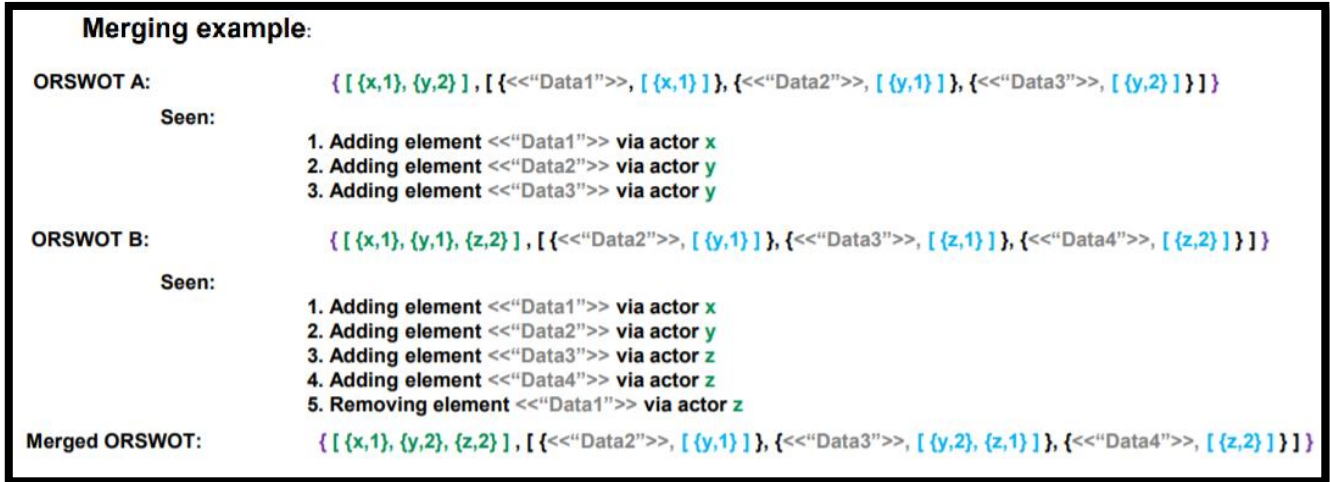


Figure 9: Orswot states merging example

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).

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

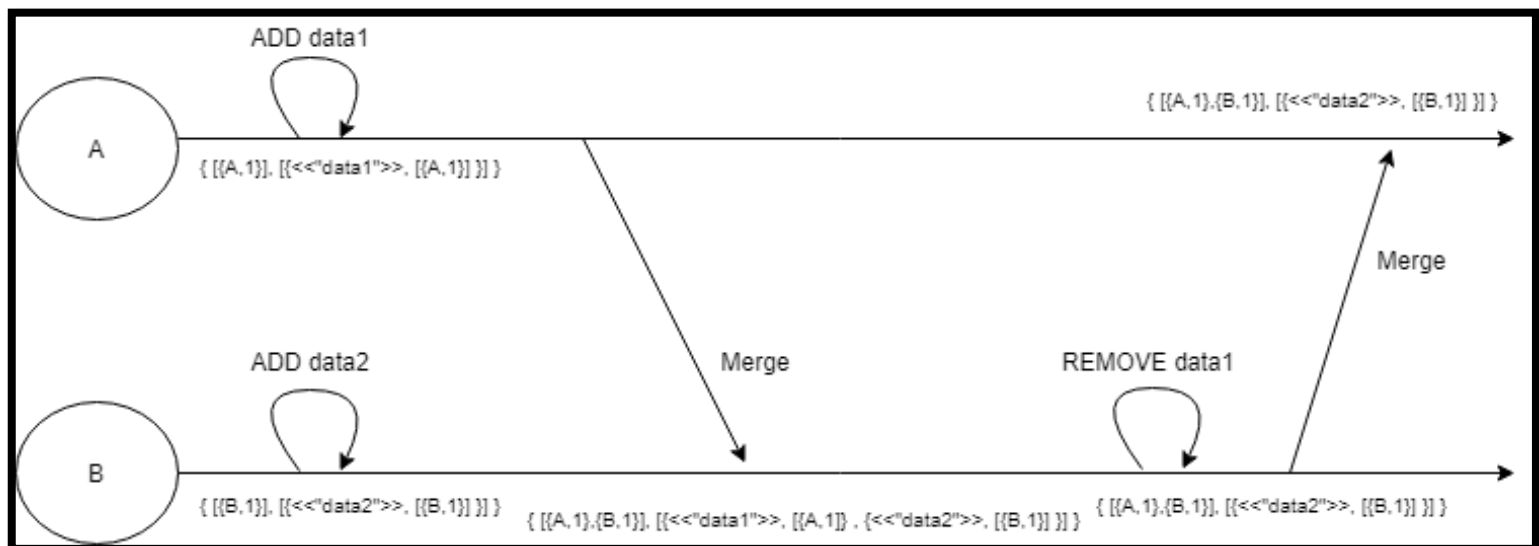


Figure 10: Orswot timeline example

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)
- CRDT value update speed (Nodes might want to update the content of a CRDT extremely frequently)
- CRDT state sending period (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
- Geographical distance (at a small scale)
- Nodes under partition
- CRDT content (number of elements in the set)
- CRDT operation (element addition or removal)
- CRDT value update speed (how often do node modify the CRDT content)
- State sending period (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 measures, 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 (chapter2) including the new developed tools and a few improvements that were proposed to enrich Lasp or make it a bit more user-friendly. The scripts used for this work will also be briefly presented. Following, in chapter 3, the measurement and results will be presented and analysed. Finally, chapter 4 concludes with some summary about the develop 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 measures 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 on the github repository of this work.

2.1 Measurement tools

This tool is designed to allow the end developer to easily get information about a cluster convergence time and network usage (number of messages per second). It was developed as a few methods inside the erlang module `lasp_convergence_measure` which was created for that purpose. It is functional in the sense it does, as intended, give the end user useful information about its cluster convergence time but it does not exactly fulfil the ideal task of directly measuring the convergence time of a specific CRDT shared on a cluster. This is due to the approach that is to mimic a CRDT and to measure its convergence time on the cluster instead of directly measuring an already shared CRDT under use. That said, it offers some useful 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 to allow better precision (see section 3.7 for more details) 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 :

```
rebar3 shell --name node1@127.0.0.1
=====
My leader is: 1
[LEADER] Wait for 4 nodes to answer... OK!
[LEADER] New convergence infos : *(individualConvergenceTimes =>
    *(convergenceTime => 357,id => 4),
    *(convergenceTime => 465,id => 3),
    *(convergenceTime => 500,id => 5),
    *(convergenceTime => 529,id => 2)),
    networkUsage => 179,roundTripTime => 872,
    worstConvergenceTime =>
    *(convergenceTime => 529,id => 2))
[LEADER] Reset is done !
=====
NEW MEASURE ROUND
=====
My leader is: 1
[LEADER] Wait for 4 nodes to answer... OK!
[LEADER] New convergence infos : *(individualConvergenceTimes =>
    *(convergenceTime => 222,id => 4),
    *(convergenceTime => 542,id => 2),
    *(convergenceTime => 566,id => 5),
    *(convergenceTime => 654,id => 3)),
    networkUsage => 173,roundTripTime => 935,
    worstConvergenceTime =>

rebar3 shell --name node5@127.0.0.1
=====
NEW MEASURE ROUND
=====
My leader is: 1
[BASIC] Waiting for leader measure signal... OK !
[BASIC] Reset is done !
=====
NEW MEASURE ROUND
=====
My leader is: 1
[BASIC] Waiting for leader measure signal... OK !
[BASIC] Reset is done !
=====
NEW MEASURE ROUND
=====
My leader is: 1
[BASIC] Waiting for leader measure signal... OK !
[BASIC] Reset is done !
=====
NEW MEASURE ROUND
=====
My leader is: 1
[BASIC] Waiting for leader measure signal... OK !
[BASIC] Reset is done !
=====

rebar3 shell --name node2@127.0.0.1
=====
NEW MEASURE ROUND
=====
My leader is: 1
[BASIC] Waiting for leader measure signal... OK !
[BASIC] Reset is done !
=====
NEW MEASURE ROUND
=====
My leader is: 1
[BASIC] Waiting for leader measure signal... OK !
[BASIC] Reset is done !
=====
NEW MEASURE ROUND
=====
My leader is: 1
[BASIC] Waiting for leader measure signal... OK !
[BASIC] Reset is done !
=====
NEW MEASURE ROUND
=====
My leader is: 1
[BASIC] Waiting for leader measure signal... OK !
[BASIC] Reset is done !
=====

rebar3 shell --name node3@127.0.0.1
=====
NEW MEASURE ROUND
=====
My leader is: 1
[BASIC] Waiting for leader measure signal... OK !
[BASIC] Reset is done !
=====
NEW MEASURE ROUND
=====
My leader is: 1
[BASIC] Waiting for leader measure signal... OK !
[BASIC] Reset is done !
=====
NEW MEASURE ROUND
=====
My leader is: 1
[BASIC] Waiting for leader measure signal... OK !
[BASIC] Reset is done !
=====
NEW MEASURE ROUND
=====
My leader is: 1
[BASIC] Waiting for leader measure signal... OK !
[BASIC] Reset is done !
=====
NEW MEASURE ROUND
=====
My leader is: 1
[BASIC] Waiting for leader measure signal... OK !
[BASIC] Reset is done !
=====

rebar3 shell --name node4@127.0.0.1
=====
My leader is: 1
[BASIC] Waiting for leader measure signal... OK !
[BASIC] Reset is done !
=====
NEW MEASURE ROUND
=====
My leader is: 1
[BASIC] Waiting for leader measure signal... OK !
[BASIC] Reset is done !
=====
NEW MEASURE ROUND
=====
My leader is: 1
[BASIC] Waiting for leader measure signal... OK !
[BASIC] Reset is done !
=====
NEW MEASURE ROUND
=====
My leader is: 1
[BASIC] Waiting for leader measure signal... OK !
[BASIC] Reset is done !
=====
NEW MEASURE ROUND
=====
My leader is: 1
[BASIC] Waiting for leader measure signal... OK !
[BASIC] Reset is done !
=====
```

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 :

```
(node4@127.0.0.1)1> lasp_convergence_measure:getSystemConvergenceInfos().
#{individualConvergenceTimes =>
  [{convergenceTime => 200,id => 3},
   {convergenceTime => 482,id => 4},
   {convergenceTime => 624,id => 2},
   {convergenceTime => 968,id => 5}],
 networkUsage => 189,roundTripTime => 1459,
 worstConvergenceTime => #{convergenceTime => 968,id => 5}}
(node4@127.0.0.1)2> lasp_convergence_measure:getSystemWorstNodeId().
5
(node4@127.0.0.1)3> █
```

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 time 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 :

```
(node2@127.0.0.1)1> lasp_convergence_measure:getSystemRoundTrip().
565
(node2@127.0.0.1)2> █
```

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 :

```
(node5@127.0.0.1)1> lasp_convergence_measure:getConvergenceTime(3).  
508  
(node5@127.0.0.1)2> █
```

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. Measures and inspection quickly highlighted the cause of this slowness. As explained in chapter 1 (introduction), nodes sharing a CRDT (awset in our case) must send their local state from time to time to allow the system to converge. The default version of Lasp without more 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 will eventually adapt. The exact implementation for now, while not perfect but functional, is to use a one-element CRDT to share that parameter on all the nodes guaranteeing (due to CRDT properties) that eventually every node adopts the same `state_interval`.

2.2.2 API

The API to adapt the cluster convergence time via modifying the time interval between states sending (`state_interval`) is very straight forward with a setter and a getter. The setter directly modifies the `state_interval` for the current node but may take up to one convergence time (generally around

one state_interval time) for all the nodes to adopt that same state_interval. Let's illustrate the principle with a little example:

If a cluster was running with a state_interval of 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 sending speed on the entire cluster. This is due to the fact the cluster must converge on the state_interval value using previous parameters before adopting the new value for that parameter.

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

Example usage :

```
(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 `getSystemConvergenceInfos()` 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 repertoires 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 speed tested:
 - One element is added/removed every 0.5 sec (2/sec).
 - One element is added/removed every 0.25 sec (4/sec).
 - One element is added/removed every 0.05 sec (20/sec).
 - One element is added/removed every 0.01 sec (100/sec).

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

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

2.3.3 Quality-of-life scripts

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

These little handy scripts include:

- `LaunchBasicNode.sh` that simply launches a local node that does not nothing special but can be used for little manual tests.
- `LaunchBasicCluter5.sh` that simply launches a cluster of 5 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, the stored record was growing. Indeed each time it received a state, it stored the new local state while adding a node reference inside the record, slowly making the record becoming huge while it did not store any new useful information since it was literally recording the same node reference again and again (exact redundancy of a same reference hundred times in a single record being stored). While relatively hidden initially, this problem became very visible after modifying the `state_interval` with the tool described in

section 2.2. Indeed, since the nodes received states much more often while still storing increasing size records, the process size was growing much faster. For example, when putting the `state_interval` to 100ms instead of the initial 10000ms (which was the initial default value hardcoded in Lasp), my computer was crashing after only a few minutes (2-3 minutes).

All these strange behaviours and the fact purely redundant information were stored pushed me to believe it was purely a bug in Lasp implementation or at least something was badly configured in the most recent github version. This is why I opened an issue talking about this: <https://github.com/lasp-lang/lasp/issues/310>.

After receiving feedback from Lasp main developer, Cristopher 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. Measures and results

This chapter will present the measurements and results together with some analysis. As detailed in chapter 1, this work uses the Lasp awset CRDT (named orswot in general literature) as use-case and measures impact of various parameters on the convergence time of a cluster together with the network usage (in term of messages per sec). As a reminder, the awset is a CRDT that acts like a set where nodes can add or remove elements allowing a wide range of usages, for details on orswot/awset please refer to section 1.3.2. 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 3.2.6 which detail 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 and network usage (in term of number of messages per second) are:

- Cluster size (number of nodes)
- Geographical distance (at a small scale)
- Nodes under partition (nodes unable to communicate with the cluster)
- CRDT content (number of elements in the awset)
- CRDT operation (adding or removing elements)
- CRDT value update speed (the rate at which nodes want to modify the awset content)

The adaptation tool described in section 2.2 is also tested by measuring its impact on convergence time and network usage. In other words, the time interval between node states sending (aka `state_interval`) acts like a 7th parameter being tested. It is considered apart since it is not really a constrained parameter defined by the cluster specification itself but more of a editable parameter that can be adapted according to the other parameters.

The number of iterations for every experiment is always 50. 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 if you wish to.

3.2 Number of Nodes

The measurements details and results together with an analysis proposal focusing on the number of nodes in the measured cluster will be presented in this section ending with a very small conclusion about that parameter.

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

Prior to the results, it may be interesting to mention one difficult task for this set of experimentations was the orchestration part since nodes are on multiple devices. Indeed, it was not as simple as some other experiments where it was easily possible to use a local instant signal to launch an experimentation on local nodes (with a script for example). Here, the orchestration, while not complicated, is a little bit more important. The principle used is to start a cluster with local nodes and remote nodes (from UCLouvain Ingi servers), to detect that every node is running (predefined cluster size) and to run the little leader election protocol which was already implemented for the continuous measurement tool (described in section 2.1). The leader will be in charge for the global start of the experimentation orchestration. The idea is the following: the leader knows every node is ready (they are reachable), it puts a timestamp in a CRDT to share it with the cluster. This timestamp is a future unix time (actually it was `current_time + 30000ms`) that represents the global start of experiment. Nodes get that time value and loop every 1ms on their own current time (ms) to check if it's time to start, whenever its own unix time reaches the threshold, the node stops looping and starts the experimentation (for example adding elements on a specific CRDT). 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 a small cluster), it assumes the different nodes have the same exact unix time which is not always perfectly true (very little shift 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.

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

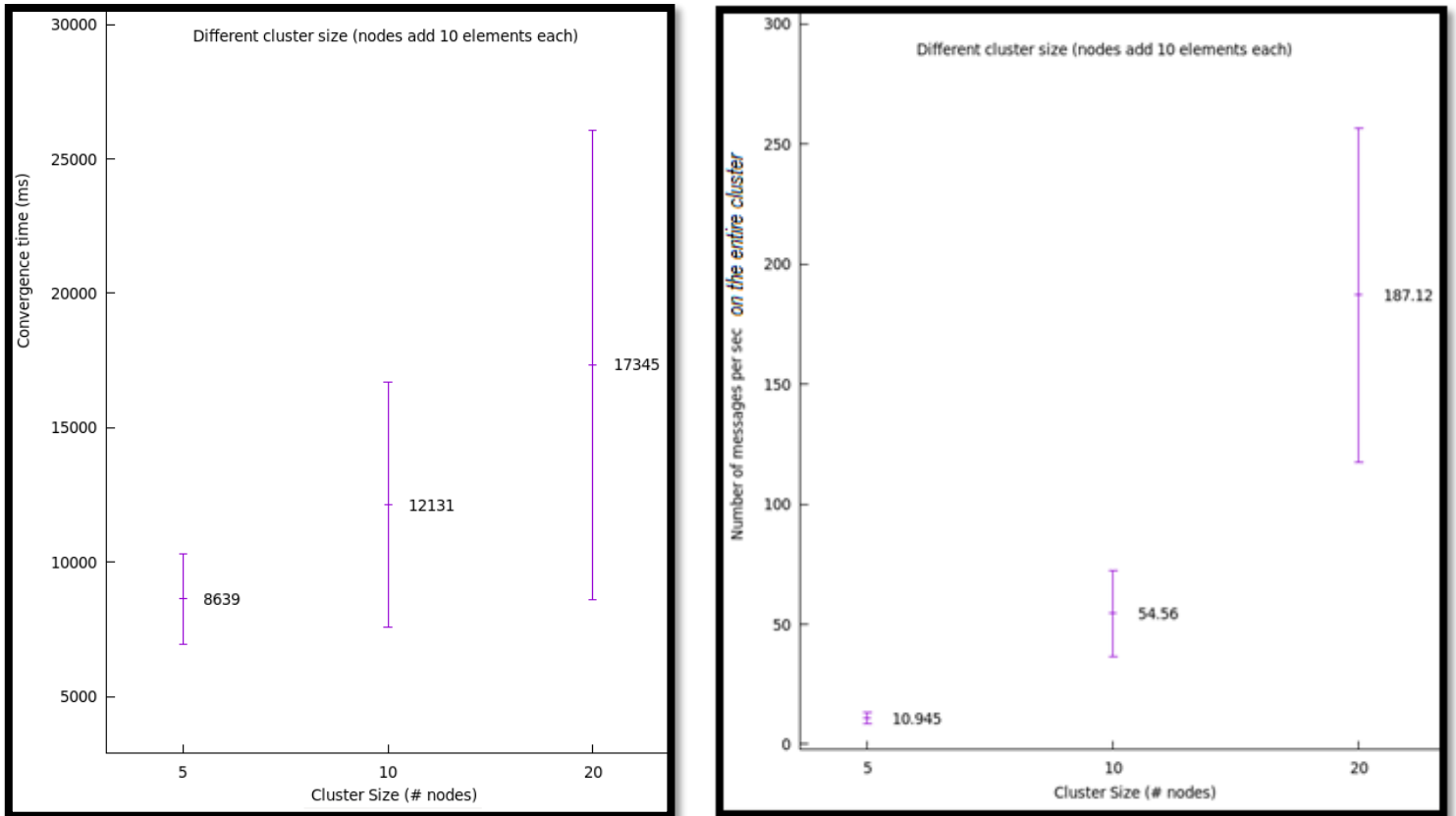


Figure 22: Graphs convergence time and network usage for different cluster size (nodes adding 10 elements each)

At first sight, these convergence time values might look very big (order of magnitude of 10 seconds) but this is due to the `state_interval` Lasp default value that is 10 seconds. The fact, the convergence time is of same magnitude as the `state_interval` is already a quite logical sign about the measurement correctness. Still, these measurements show a clear impact of the number of nodes in the cluster. Indeed, we can see that the convergence time increases with the number of nodes which, although annoying, is a relatively expected behaviour (the other way would have been extremely strange!).

Analysing these results while having in head the `state_interval` allows an interesting approach where we can easily compare the two parameters and guess what is happening. Indeed, we can see that for 5 nodes, the convergence time is smaller than the `state_interval`. In other words, since the nodes send their states on a regular basis (not especially at the moment the node updates its state), we can imagine the first state send from every node reached every node at first round allowing some relatively fast convergence (in regard to the `state_interval`). When the number of nodes increases, the convergence time exceeds the `state_interval` value which is a sign all the nodes were probably not directly reached by first states sending round. It is also interesting to note that convergence time does not especially match an integer of `state_interval`(s) since nodes first sending is not especially after a whole `state_interval` period, it depends when the state was update in regards to that time interval (close to new round or not). It is also interesting to note that some recent Lasp improvements (for example on different branches than the Master one) are trying to address these elements by, for example, allowing the nodes to directly send their state when updating it (via `propagate_on_update` boolean).

When looking at the number of messages per seconds on the entire cluster, as expected, we can see that the more nodes the more messages are sent. Still an important remark that is not visible on the graph is the fact all the messages are not directly related to the CRDT state. Indeed, Lasp uses communications between nodes for many other purposes such as crash detection and alive messages or even debug reasons, representing a significative part of the messages. One other very important remark is that the graph represents the number of messages per second on the entire cluster. In other words, if 5 nodes send 10 messages per second, the number of messages per second on the entire cluster is 50. If 20 nodes send 10 messages per second, the number of messages per second on the cluster is 200. While both number of messages per second per node and number of messages par second on the entire cluster are interesting data, they do not show the same way on a graph. The fact the number of messages per second increases with the number of nodes is a certainty but this growth, when looking at number of messages per second per node (simply divide the value on graph by the number of nodes) does not, obviously, grow as a square relation. This is an important notion to have in mind to not misunderstand the graph.

One other aspect to consider is the fact the presented measurement, as explained above and detailed in section 2.3.1 (describing the script used) test different cluster size but involve an increasing number of elements in the CRDT (awset). Indeed, where 5 nodes adding 10 elements each wait to detect 50 elements, 10 nodes adding 10 elements each will wait to detect 100 elements. This means the CRDT itself (reminder: it is a data-structure) is increasing in size because of its content. To be sure the variations in the results are mainly related to the number of nodes (and not the number of elements in the CRDT), the same experiment was run again with other number of elements added by nodes

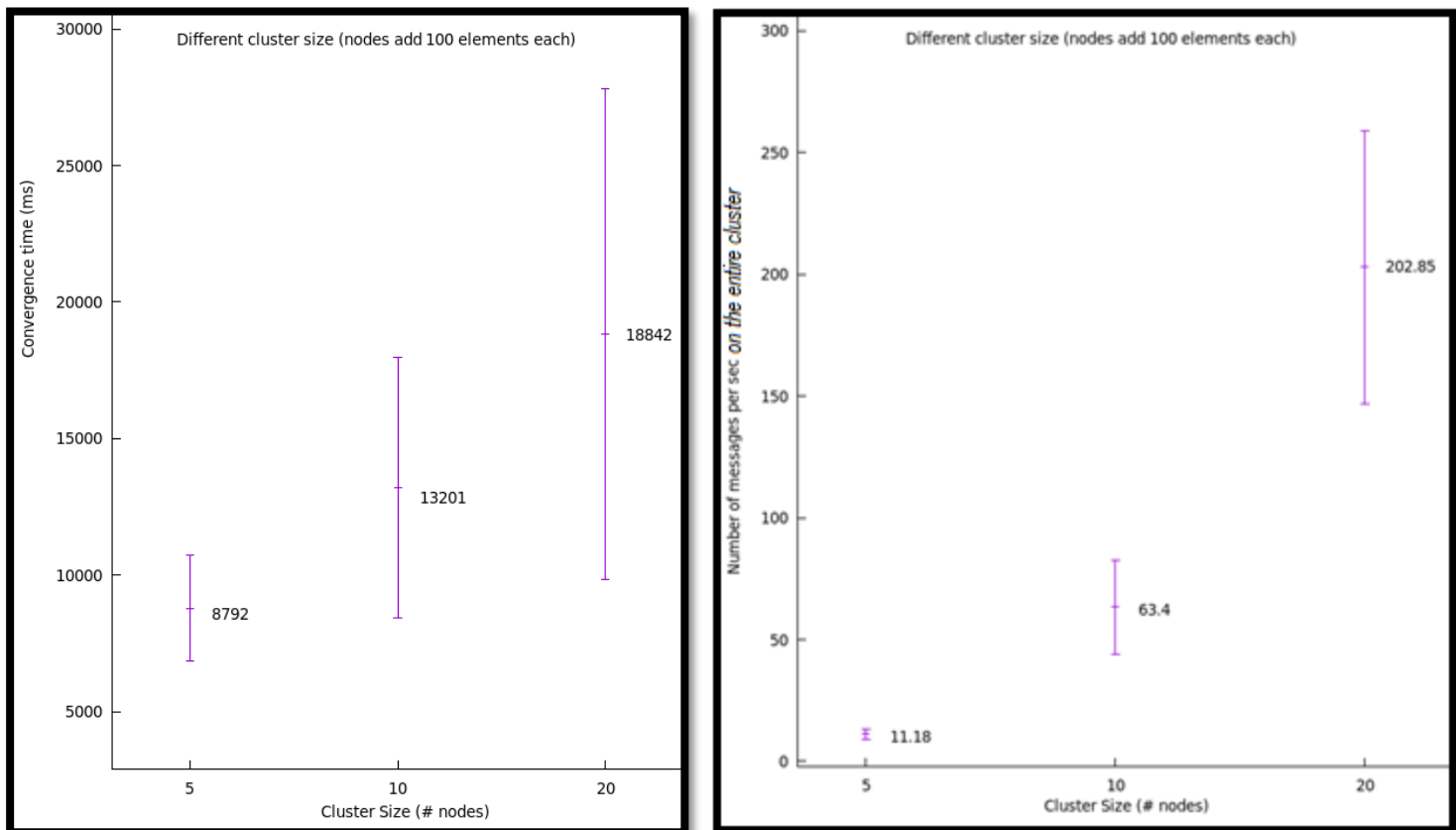


Figure 23: Graphs convergence time and network usage for different cluster size (nodes adding 100 elements each)

Clearly, with nodes adding 100 elements instead of 10, the exact same behaviour shows up, confirming the results were reflecting the modification in number of nodes and not the slight variation in the CRDT size (content). Still, the finest readers might remark the convergence time is very slightly higher on this new graph but the impact of the CRDT content on the convergence will be discussed in section 3.2.3.

Another question that can arouse curiosity is to know if this impact on the convergence due to the number of nodes is the same for element addition and removal. When the nodes do not add elements but remove them, do we observe the same behaviour in regard to the number of nodes? In other words, is element removal impacted the same way by the cluster size? Here are the same graphs as above but for elements removal instead of addition.

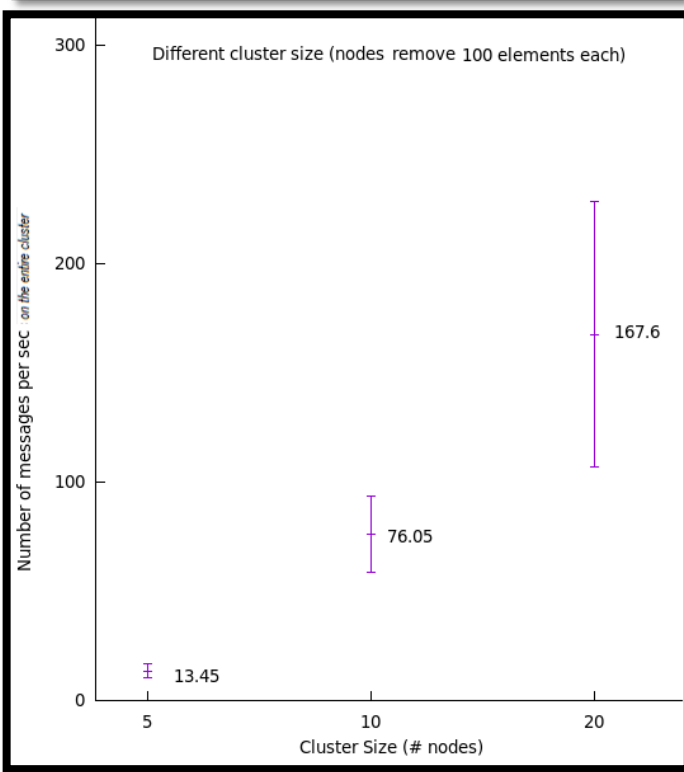
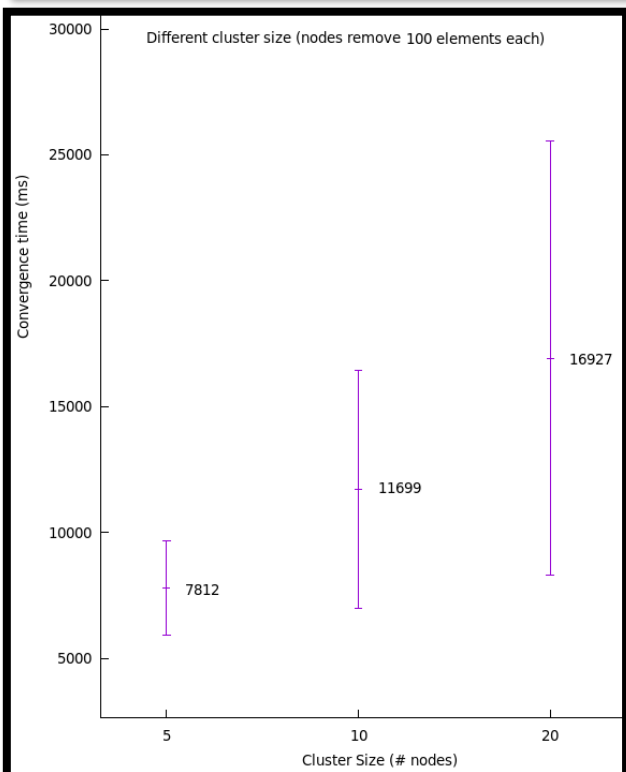
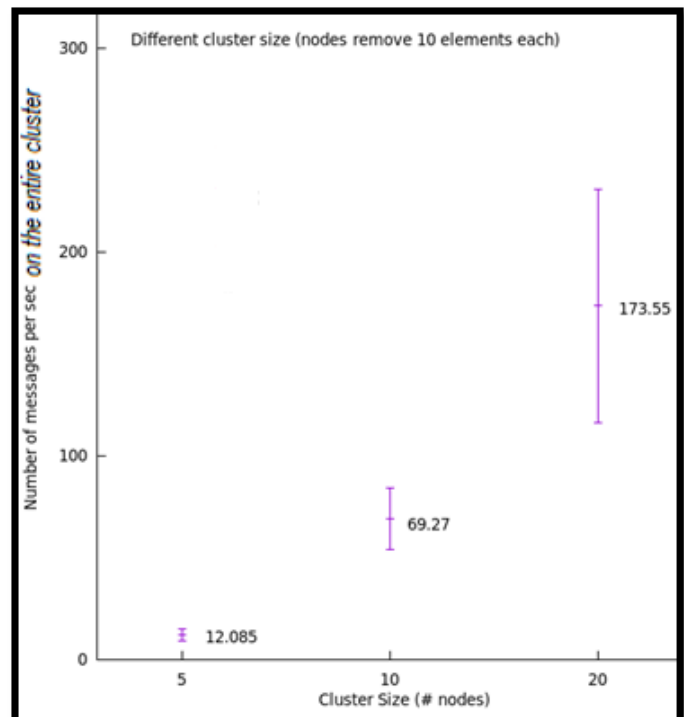
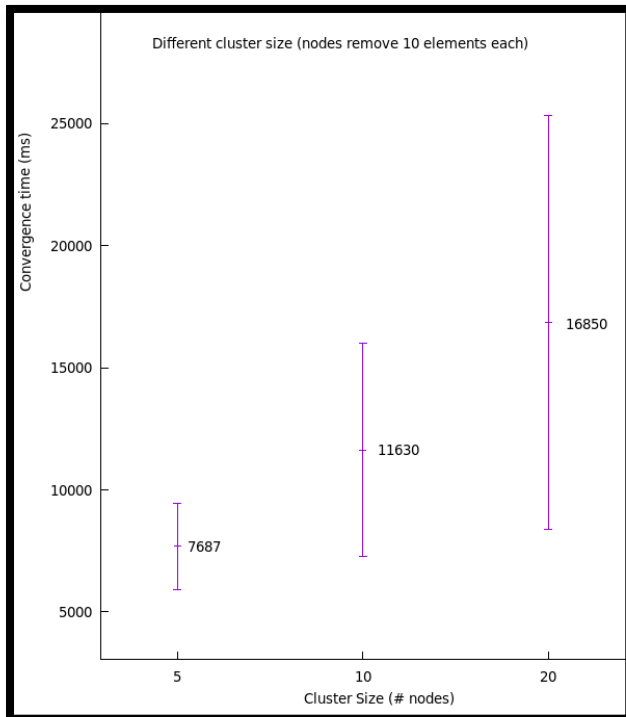


Figure 24: Graphs convergence time and network usage for different cluster size (nodes removing 10 and 100 elements each)

Clearly, we see that the element removal operation convergence time is impacted by the number of nodes the same way as element addition was. This concludes the experiments on the number of nodes, showing the impact of the cluster size and the fact this impact is the same whatever the content of the CRDT (number of elements) is or the type of operation (addition or removal of elements) being used.

A last important remark on this first set of experimentations is the fact 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 machines 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 3.2. This obviously is not ideal at all 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 and a 20 nodes cluster would be to have up to 20 independent devices with the possibility to use only 5 of them or the 20 of them (thus always one node per device) which would probably give more reliable results. In regard of this aspect, it is important to note the experimentation here necessarily worsens the results by making them worse for bigger clusters. This means the convergence time for a real 20 nodes cluster would be better than what was measured in the previous graphs which is reassuring for Lasp future.

3.3 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 locally and nodes on a remote network which is around 20 km distance)

As for the previous experiment, since nodes are running on multiple devices, at first step was to implement the orchestration protocol for the measurements which is the same as for the previous point (3.2). 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).

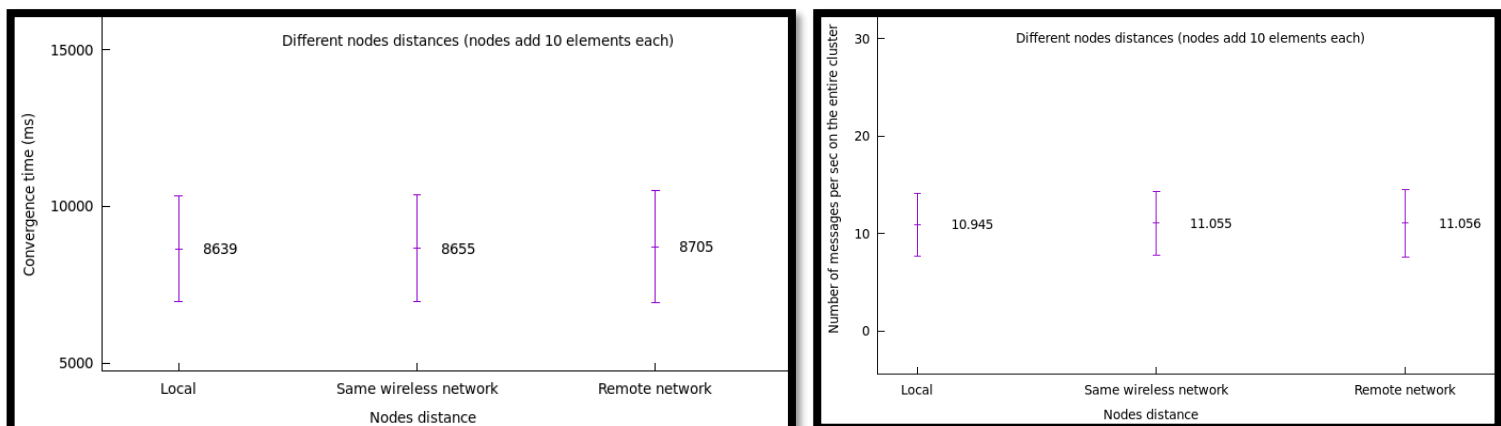


Figure 25: Graphs convergence time and network usage for different nodes distance (nodes adding 10 elements each)

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 at an internet perspective.

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 (which is probably not truly the case anymore when distance gets around thousands of kilometres).

Again, for a more rigorous experimental approach, this experiment was run with nodes adding 10 and 100 elements then with nodes removing elements instead of adding them. While showed in the case of the first parameter (number of nodes) to present the entire experimental approach to the reader, I decided not to show all the little measures graphs since they represent globally the same result with extremely little variation. More measurements and results are available within the saved_measures folder of the github page of this work.

Globally, the presented results tend to mean the element that slows down convergence is not from the communications delay itself since distance tends to have very little impact. This pushes the intuition the convergence time being so long must be from nodes waiting before to send their state to peers which is, for now, the most promising parameter but will be analysed later (section 3.7).

3.3 CRDT content

While precedent measures 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 measurements checked if a cluster where nodes add 10 elements or 100 elements were impacted the same way by, let's say, 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 (which was introduced in section 2.3.1), a cluster of nodes is created, in this case 5 nodes, every node puts a number of elements on the CRDT, let's say 10 elements, and wait for the cluster to converge (in this case, every node see a total of 50 elements in the CRDT). 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:

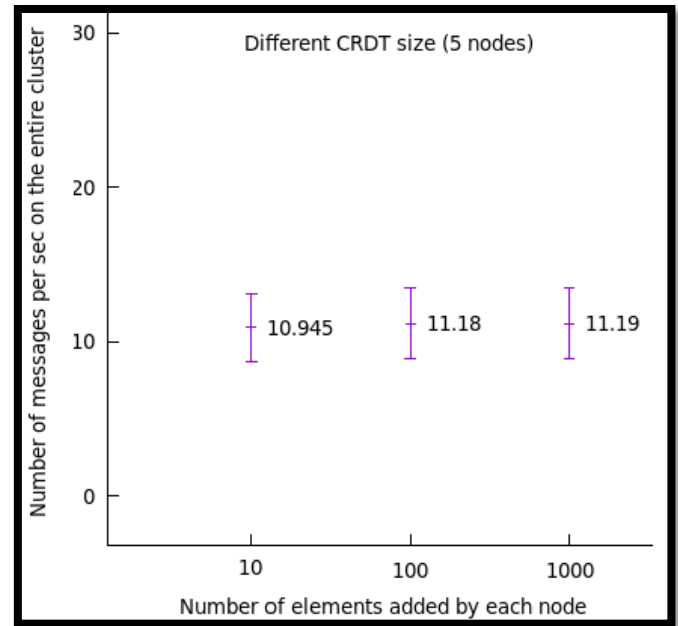
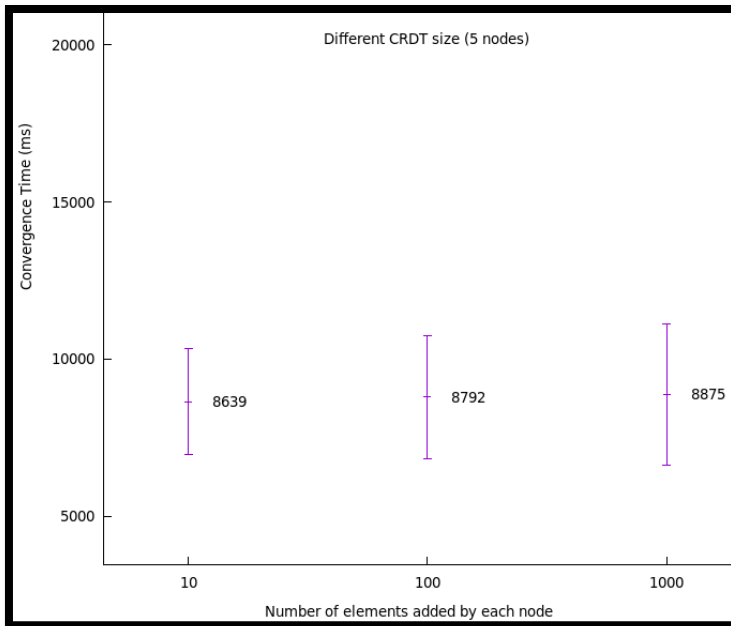


Figure 26: Graphs convergence time and network usage for different CRDT content (5 nodes adding elements)

From these results, 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). Indeed, the convergence time seems to be very slightly longer when the number of elements is bigger, but the variation is relatively little. Since a 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. Still, the fact it is very slightly longer with more elements can be a sign the nodes simply take a very little bit more time to process information (decode messages, compute merges, etc...).

On the other hand, when looking at message exchanges, the fact their quantity stays relatively constant sounds logical since, as mentioned, the messages will be bigger but not especially more numerous.

As for previous measurements, let's see if the measured impact is the same in the case of removals of 10, 100 or 1000 elements (on a 5 nodes cluster with a CRDT initially containing 50, 500 or 5000 elements).

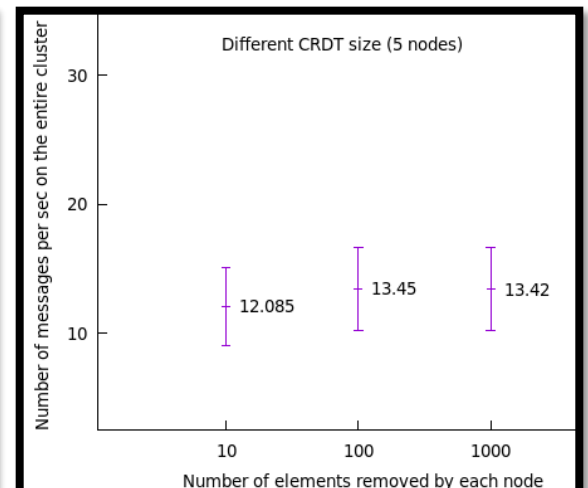
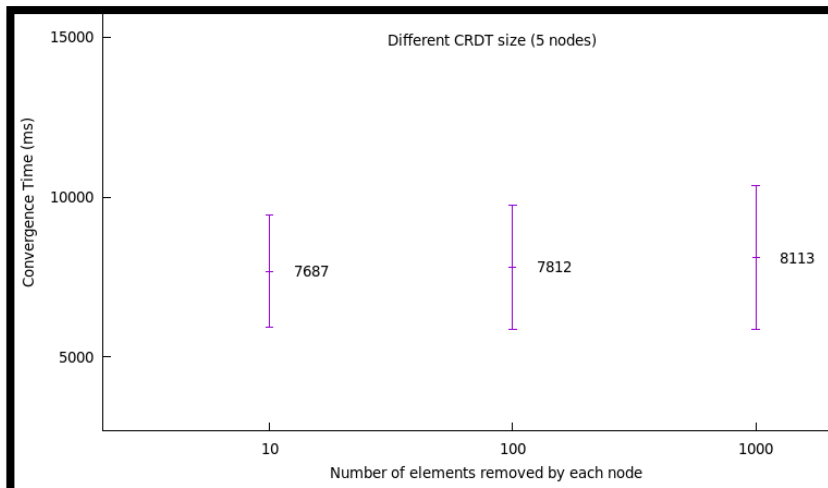


Figure 27: Graphs convergence time and network usage for different CRDT content (5 nodes removing elements)

When regarding these new graphs, we see globally the same profile with CRDT initially containing 50 elements, 500 or 5000 not having a big impact on the convergence time. Still, we see, as for additions, a small increase in convergence time with the number of elements. About the network usage, we see globally that the number of messages per second is not really impacted by the number of elements in the CRDT which seems logical as discussed above. We can conclude for this parameter that it has an impact on the convergence time (being for adding elements or removing them) but this impact is little. As a remark, which was already mentioned during previous measurements, we can notice elements removal is faster than elements addition but this will be highlighted in next section (3.4).

Few other measurements were made in regard to this parameter with clusters of 10 nodes, since they represent very similar graphs (the impact is the same while the values are a bit higher due to the cluster size). These results are available on the github webpage of this work.

3.4 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...). It shows up, while it was not directly the item under measurement, 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:

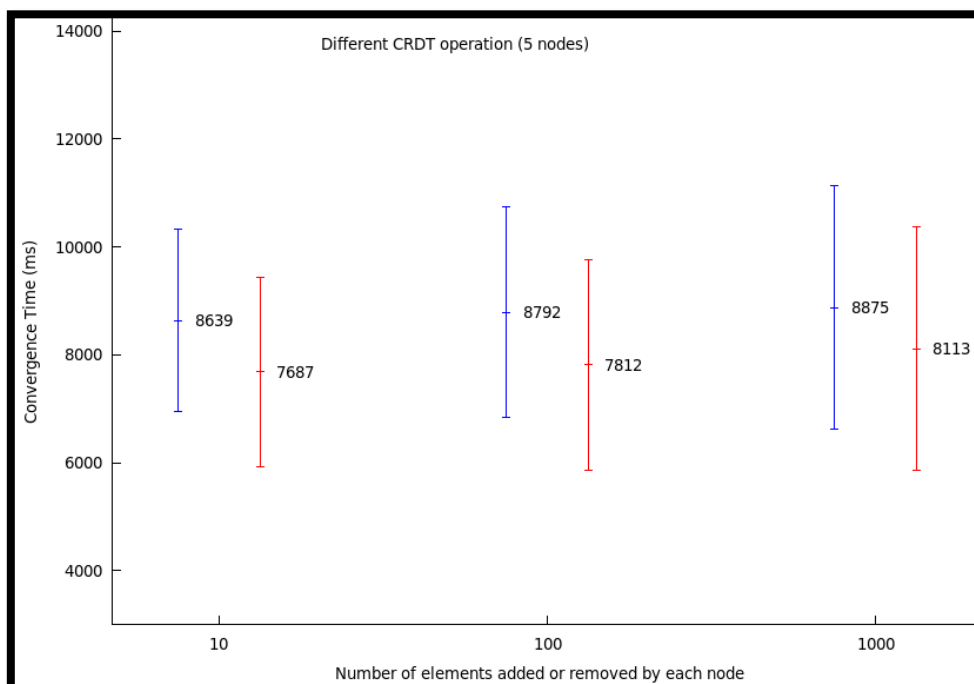


Figure 28: Graph convergence time for different CRDT operation (5 nodes)

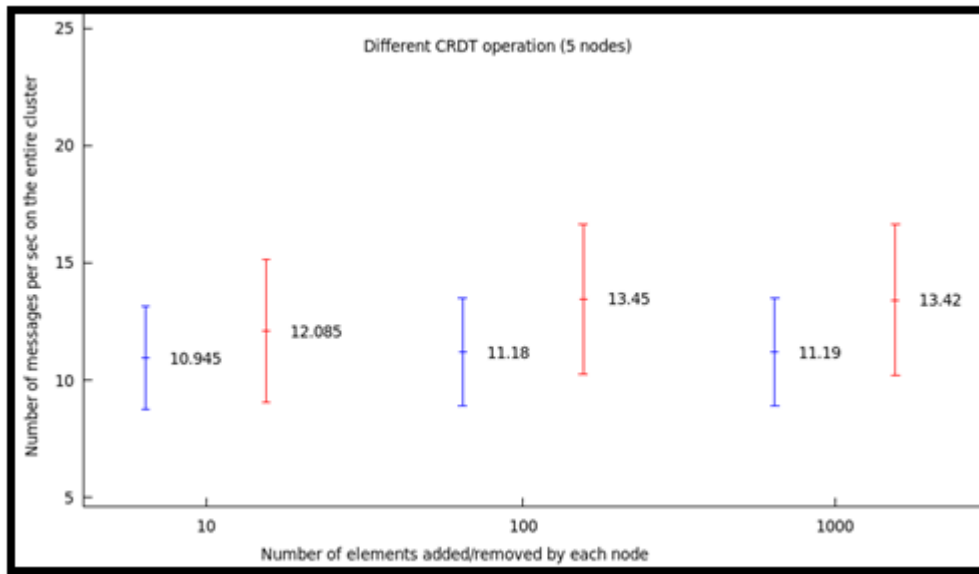


Figure 29: Graph network usage for different CRDT operation (5 nodes)

On these graphs, in blue are the elements addition and in red are the elements removals. Measures for adding/removing 10 elements, 100 elements and 1000 elements (by each node) are represented. We see that, as it was visible in previous section (3.3), convergence time tends to be a bit longer when the CRDT content increases (element addition). 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 CRDT initially have elements inside and 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 always faster than additions. While it was possible to have this intuition from previous graphs, it now appears clearly. The fact is the intuition it 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 CRDT without requiring modifying version clock or checking the dot set related to that element. For the merging operation (which is used by nodes at every new state received), it should not make any difference since additions and removals should trigger the same approach from the merge function ("non-common element", see section 1.3.2 for more details). So, it sounds like the convergence time difference explanation would only be about the operation on the data-structure in itself which is faster for removal. 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 (which are faster according to measurements) tend to send slightly more messages per second compared to addition. Again, I don't have any idea yet what is causing 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 to me.

As a verification, to see if the same behaviour shows up, here is the graphs for the same experimentation with 10 nodes:

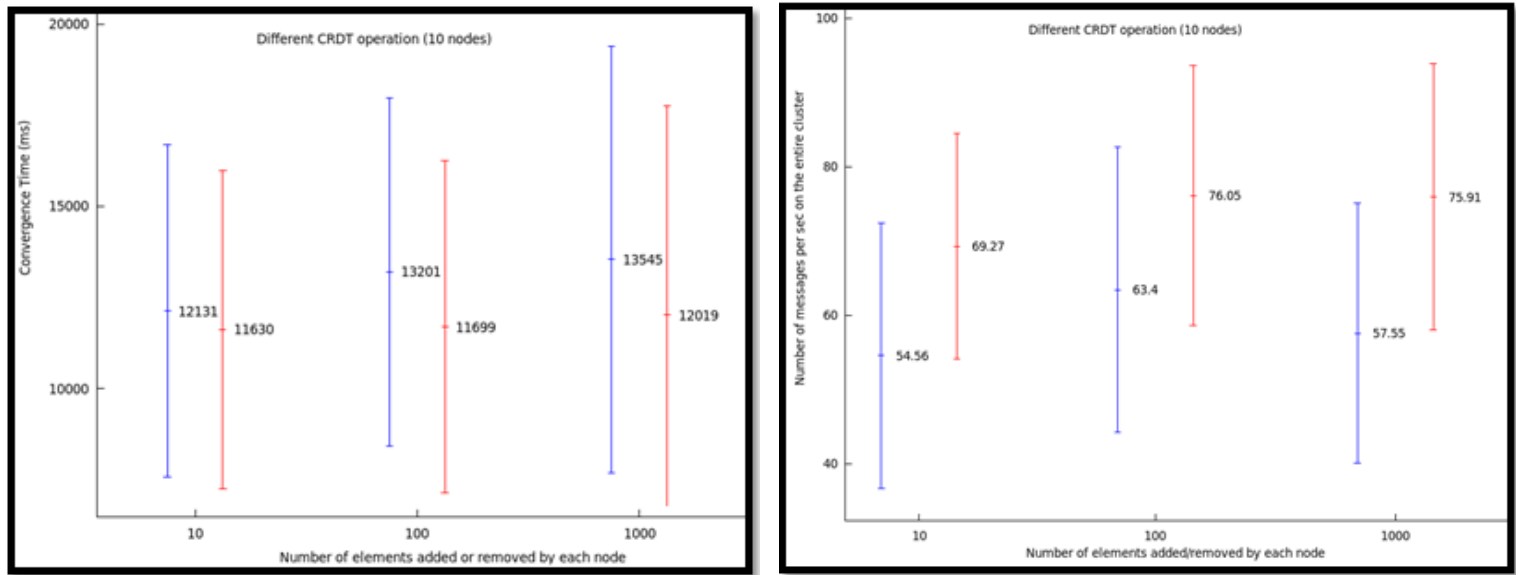


Figure 30: Graphs convergence time and network usage for different CRDT operation (10 nodes)

Again, in blue are additions and in red are removals. While values are not the same because the cluster is now of 10 nodes, the difference between additions and removals is exactly the same as 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 (local simpler operations does not seem enough), here is some details on the experimental approach that was used.

- Elements addition:
 - Nodes start and cluster together
 - Nodes have an empty awset
 - Nodes add X elements each and start measuring (time and messages)
 - Nodes detect the awset contains $X * (\text{number of nodes})$ 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
 - Nodes have an already full awset containing $X * (\text{number of nodes})$ elements. These elements, present in the initial awset, are the same on every replica (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)

As a reminder about the network measurement, nodes count the number of messages received while waiting for convergence then divide this value by the convergence time to have the number of messages per second. On the graphs, the represented value is the sum of all the nodes messages per second to represent the number of messages delivered per second on the entire cluster.

As a conclusion for this specific element under test (CRDT operation), while the measurements show some interesting results, no precise explanation have been found yet to explain such behaviour, nor in theoretic elements nor in the experimental approach. If any reader comes with an idea to explore why such difference in the results, please do not hesitate to post about it opening an issue on the github webpage of this work or directly via email (gregory.creupelandt@student.uclouvain.be).

3.5 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 it directly send their states on partition resolve or not.

Here are the results for this experimentation:

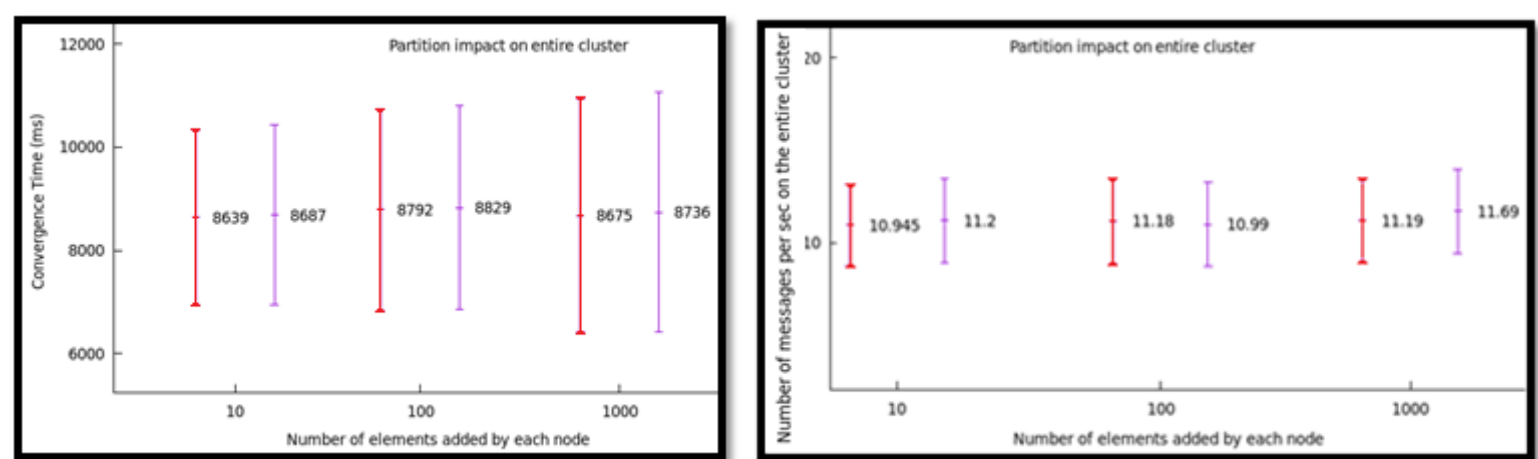


Figure 31: Graphs Impact of partition on convergence time and network usage (on a 5 nodes cluster)

In red is the normal scenario with no partition (previous measurements) and in purple is the new scenario measured where nodes are under partition when updating the awset then resolve partition. Again, as for previous measurements, it was tested with updates adding 10 elements, 100 elements or 1000 elements on a 5 nodes cluster. The measurement starts when the partitions are resolved. On this graph, it's an extreme case where all the nodes are initially under partition when updating their awset (local state) then resolve partition. 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 is resolved. As we can see, the normal scenario where measurement starts when nodes update and the new partition scenario where measurement starts at partitions resolves give the same 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 at normal speed), it does not give any information about partition impact on partitioned nodes themselves. Let's now see how nodes previously under partition catch-up with the cluster. This was done by using a cluster where 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 while the rest of the cluster is converging. 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 let's resolve the partition on that node and measure how much time it needs to converge (catch-up) with the cluster.

Here are the results for this experimentation:

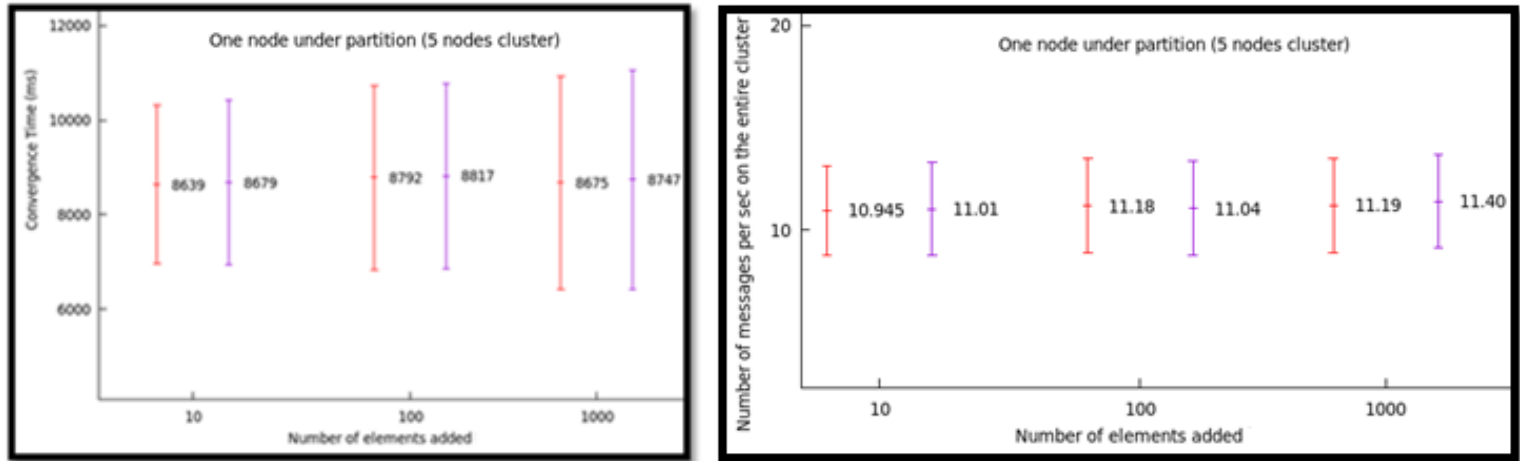


Figure 32: Graphs convergence time and network usage for a partitioned node (5 nodes cluster)

Again, in red is the normal scenario with no partition and in purple is the new scenario measuring the time required for the partitioned node to catch-up with the cluster state. We clearly see that there is no big difference between both scenarios being at convergence time or number of messages per second. As a reminder, for the normal scenario (purple), it measures the time for the nodes to converge starting the timer when nodes added elements. In the second scenario (purple), the nodes added their elements but one is partitioned and do not get the updates from peers, then the partition is resolved and the timer starts waiting for that node to get the same local state as the other nodes.

The fact we see no big difference between these two scenarios (figure 32) goes logically with the first two graphs (figure 31) showing that when nodes are under partition and do updates, their updates are sent to peers when the partition is resolved and also, the previously under-partition node gets updates from peers (at the normal usual speed).

While it could be conclude the partition tolerant property is nicely handled, some other measures were done in the same optic with a cluster of 10 nodes instead of 5 for acknowledgement. These results are available on the github webpage of this work.

3.6 Value update speed

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 1 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 new approach, as briefly described in section 2.3.2, is to run a cluster of nodes that continuously update the awset values 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 speed. 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 speed at which nodes loop on their values, it is easy to modify the value (content) update speed of the CRDT. 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 inconvenient to analyse all at once, let's just show on a graph the measured convergence time for other nodes to detect elements added by node 1. For the results below, elements are added/removed at the speed of 1 element every 0.5 second, the cluster is composed of 5 nodes each doing updates at that same speed (but we only consider the elements added by node 1 to be more readable). 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. The measurements for other nodes elements are available on the github webpage of this work (the readme files there can easily guide you there).

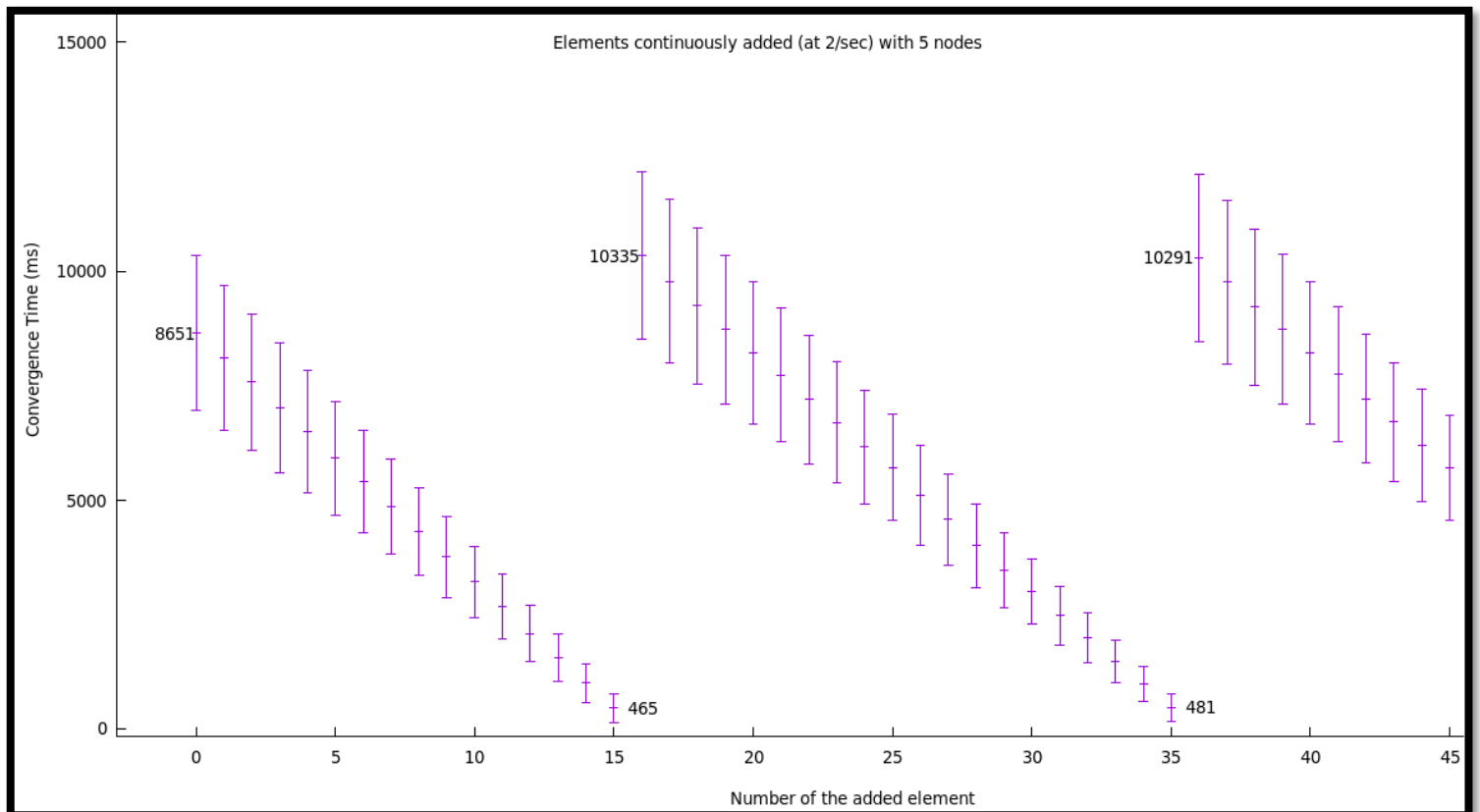


Figure 33: Graph convergence time for continuous updates at 2 updates/sec (5 nodes cluster)

There is a very clear behaviour showing up. As we can see, the first element that was added when starting the measurements (element 0) 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 around 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_interval` 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 truly sent near the start of the new 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 2 elements per 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 faster 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. If the `state_interval` is of 10000ms, the other nodes will 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 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 source node) would be consistent. 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 value update speed. 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 (3.2 to 3.5) had to start the experiment and measuring convergence time at a 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_interval` that way it does not shift in the `state_interval` during 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 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):

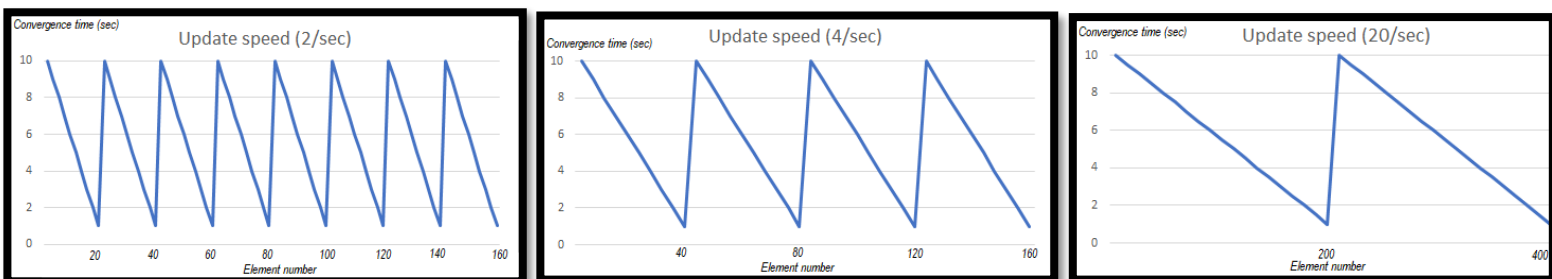


Figure 34: Overviews convergence time for continuous updates at different speeds (5 nodes)

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. It is important not to misunderstand these graphs, they do not show anything incredible or new at all, since X axis is for elements numbers and not time, they do not say sending intervals goes longer or anything like that, they are simply showing what was already discovered and described above.

As a conclusion for that parameter, the value update speed is not really a parameter that affects the convergence, it is more of an element that is limited by convergence speed. 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 update speed to the `state_interval`. Still, the presented measurements have the merit to, while not showing any impact, extremely well represent Lasp principle and implementation in practice. Indeed, such graphs, compared to previous ones, are the first to show how the communication works in practice where previous graphs only allowed some intuitions (such as the discussion in section 3.2).

3.7 State sending period

While all the previous measurements presented in sections 3.2 were running on relatively slow convergent clusters, it is now possible to make the convergence much faster by modifying the rate 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, 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 3.6, it would allow higher CRDT content update speed while still being perfectly convergent) while increasing the 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 `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 3.3 highlighted the fact the variation according to the CRDT content is very little), they are not shown here. More raw results are available on the github webpage of this work.

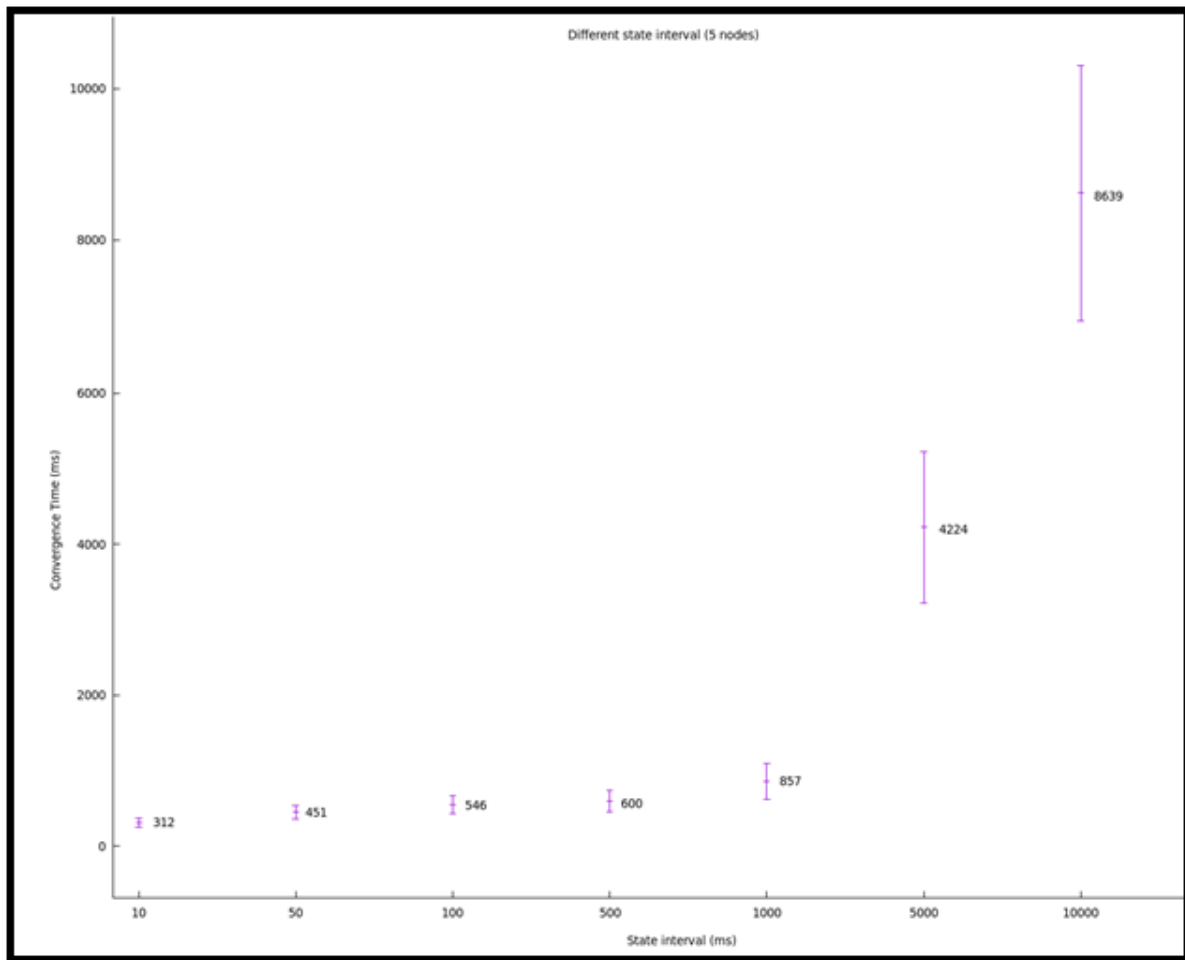


Figure 35: Graph convergence time for different state_intervals (5 nodes)

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, 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 very 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 the code to protect from network overuse.
- 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 see the impact of the state_interval value on the number of messages per second on the cluster:

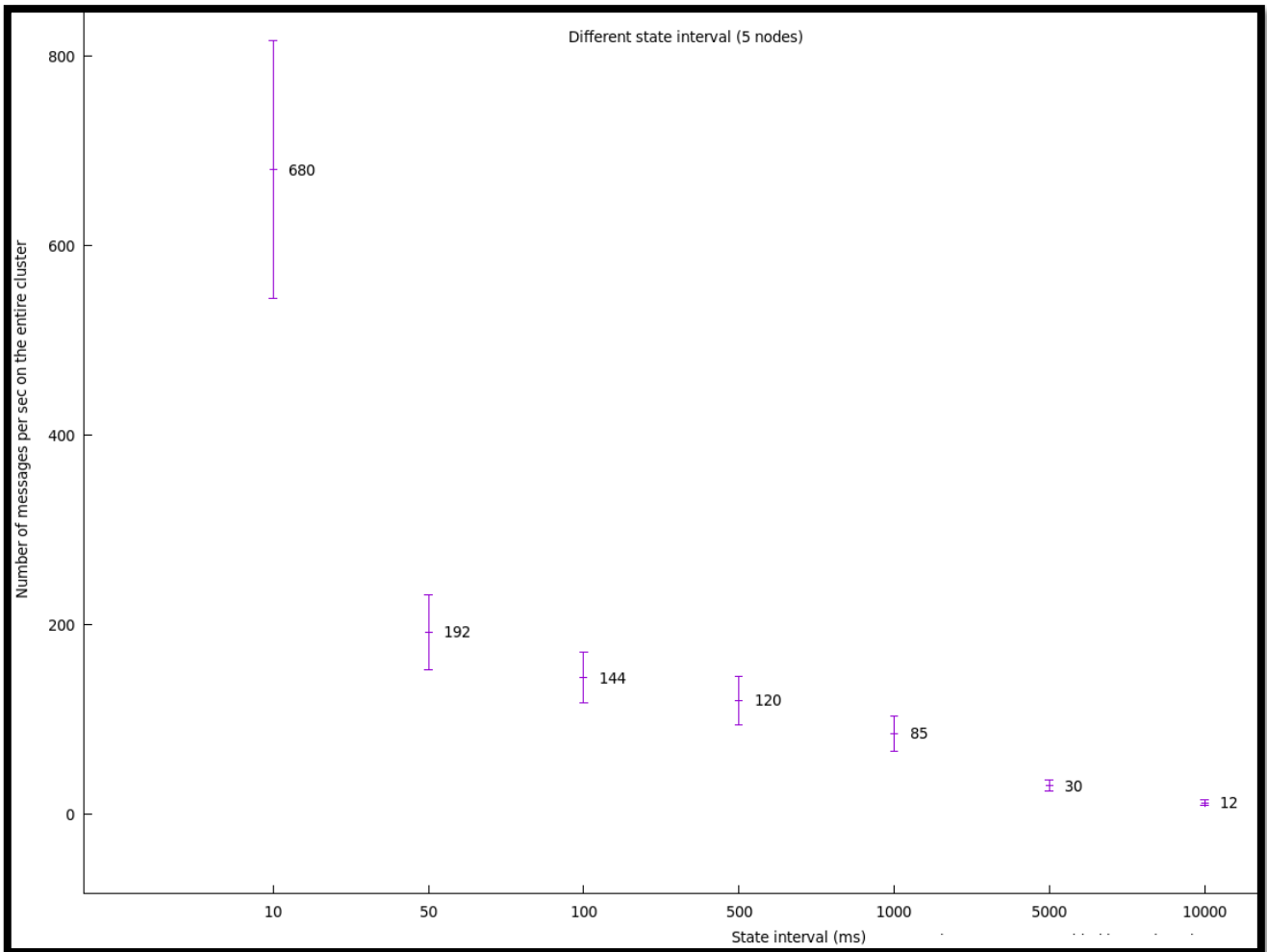


Figure 36: Graph Network usage for different state_intervals (5 nodes)

As expected, we clearly notice that the smaller the state_interval, the bigger the number of messages per second goes. As a reminder, the graphs (for all the network usage graphs) show the number of messages per second on the entire cluster. This means, the 680 (on average) messages per second from the 10 ms state_interval case corresponds to each node (it's a 5 nodes cluster) sending on average 136 messages per second. This number sounds rather plausible since a 10 ms state_interval would already mean about 100 messages per second per node then comes the messages which are not directly related to the awset state (such as logging messages and keep alive). On the other hand, the default 10000ms case obviously corresponds to what was previously measured for example in section 3.2 with 5 nodes. We can notice the number of messages per second is not perfectly linear with the state_interval value but follow the clear trend of decreasing when state_interval increases. This is probably due to non awset state related messages which may not be influenced the same way by the state_interval value.

As a side note, we can combine the results just discovered with previous point (3.6) and remark a state_interval around 500 ms allows a relatively low number of messages to be sent on the cluster while offering a relatively efficient convergence time of the same magnitude (around 500-600ms) where putting the state_interval lower would apparently not especially decrease the convergence time by a lot while increasing a lot the network usage. Based on this, as an intuition it sounds like an efficient

utilisation would be with an application requiring updating the CRDT value every 0.5 second. This can sound like poorly efficient but as a reminder this work was mainly focused on measuring variations according to parameters not especially absolute values which are hard to measure efficiently since, as highlighted in section 3.6, since these measures are influenced by the moment they are run. Also, it might be possible to configure Lasp more in depth to allow better performances which was not especially the main purpose of this work.

One last remark concerning these results combining with results from section 3.6 is to remark some likely consistency between them. Indeed, on figure 33 (sawtooth graph) we can notice element 15 took 465ms. If the convergence was instantaneous (or very small) on state sending trigger, it would mean element 15 addition was around 465 ms before state sending and element 16 addition was thus just 35ms after state sending (since there is 0.5 second between each element addition). With the 10000ms `state_interval`, if the convergence was instantaneous on state sending trigger, it would mean element 16 would have been detected in 10035 ms but it took 10335ms. While not extremely precise, this, from section 3.6, already tends to give the intuition the convergence time would hardly go smaller than around 300ms even if the `state_interval` was very small (such as 10ms). This intuition from section 3.6 was then confirmed with these last results.

4. Conclusion

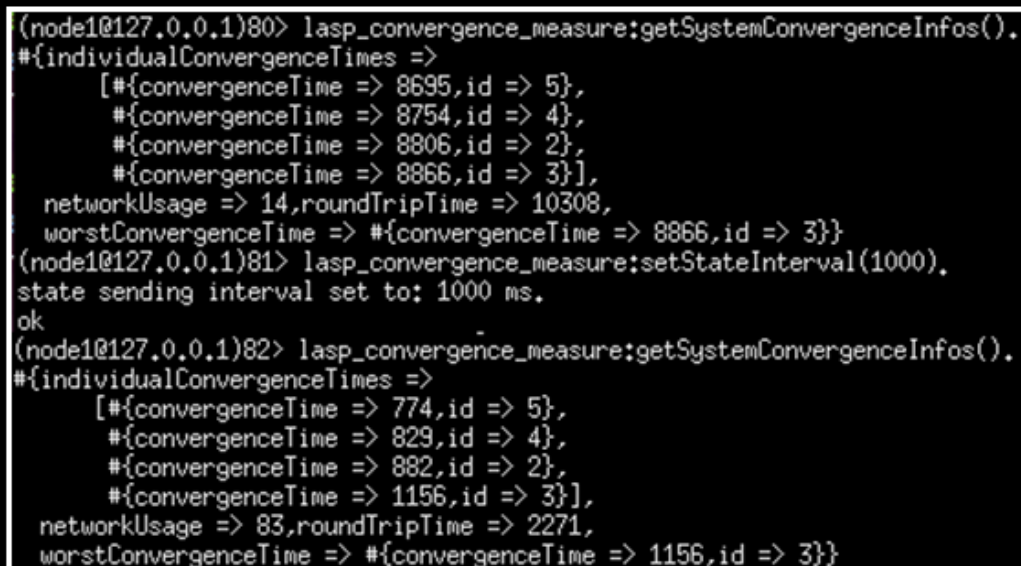
4.1 Summary

We have been able to use Lasp, an experimental tool developed by the the Ecole Polytechnique de Louvain research team, 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 and corrected with the acknowledgement 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 3.3) is not of a big impact compared to the cluster size (number of nodes: section 3.2) or the state sending period (state_awsset: section 3.7) 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 very 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 37: 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 3.6 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 inline with section 3.6 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 intended, 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. 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). 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 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 partition tolerance. The value update speed 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: modify the fanout which is the number of nodes contacted at each state sending interval).
- **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 3.7, 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 3.2) that could be improved (with more devices) and the lack of clear explanation for the unexpected variation between elements additions and removals. 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 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 would 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 (which 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 3.7. 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 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 period results (section 3.7), 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 3 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 understand the topic and the potential goals 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 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] Pregoça, N. Baquero, C. Shapiro, M. *Conflict-free Replicated Data Types*, DI, FCT, Universidade NOVA de Lisboa and NOVA LINC, HASLab / INESC TEC & Universidade do Minho, Sorbonne-Universite & Inria, 2018.
- [6] Pregoça, N. *Conflict-free Replicated Data Types: An Overview*, NOVA LINC & DI, FCT, Universidade NOVA de Lisboa, 2018.
- [7] Sypytkowski, B. *An introduction to state-based CRDTs*, Software Dev blog, 2018.
- [8] Van Roy, P. Meiklejohn, C. Bieniusa, A. *Distributed Programming with Weak Synchronization Models: Introduction to CRDTs, Lasp, and Antidote*, KTHx: ID2203.2x, edx, 2016.
- [9] Van Roy, P. Meiklejohn, C. *Lasp: A Language for Distributed, Coordination-Free Programming*, International Symposium on Principles and Practice of Declarative Programming, 2015.
- [10] Shapiro, M. Pregoça, 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. Slougher, T. *Lasp official github repository*, <https://github.com/lasp-lang/lasp>, 2014-2021.
- [13] 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 Kopestenskii, Giorgos Kostopoulos, João Leitão, Adam Lindberg, Albert van der Linde, Sreeja Nair, Nuno Pregoça, 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.
- [14] Meiklejohn, C. *Partisan: Enabling realm-world protocol evaluation*, Workshop on Advanced Tools, Programming Languages, and PLatforms for Implementing and Evaluating Algorithms for Distributed systems, 2018.
- [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. Pregoça, 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.