Table des matières

1. Introduction

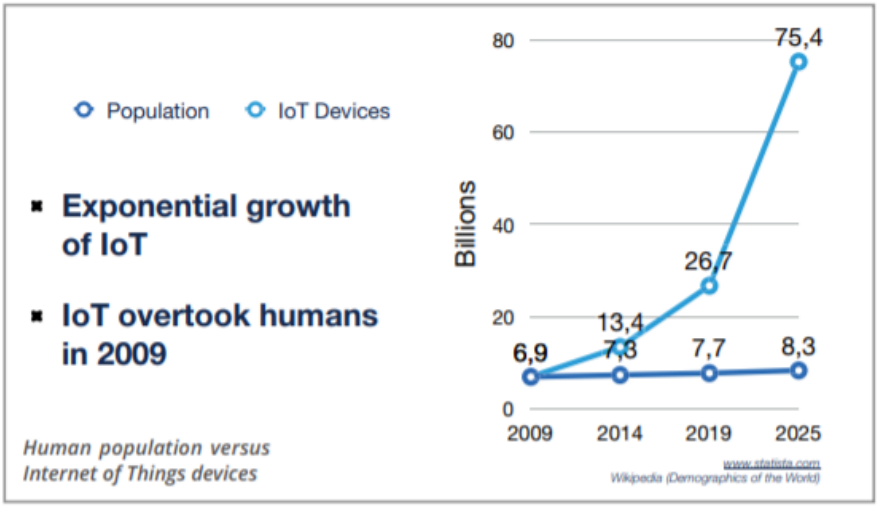
This chapter presents the general content and context of this manuscript, describing what is Lasp, what are CRDTs, what are their innovative aspects and why they are so useful. The goals of this master thesis and its main structure will also be briefly introduced.

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) devices[1].



There is no doubt, the explosive growth in that domain deserves our attention and the seek for new innovations to handle it. TODO: Put a bit more context/examples.

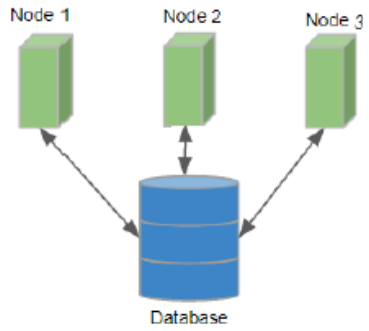
1.1.1 Usual approach

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

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

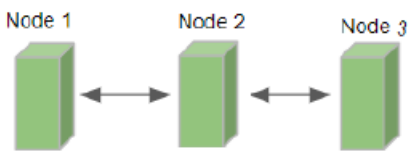
Finally, since strong Consistency conflicts with Availability and Partition-tolerance (CAP theorem[1]), these systems have to choose between CP (strong Consistency and Partition tolerance but low Availability), AP (high Availability and Partition tolerance but weak Consistency) and CA (strong Consistency 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.

TODO: put better representation (image) and give more references, probably mentioning consensus with Paxos.



1.1.2 New approach

A totally different approach is to rely on peer-to-peer instead of the usual structure with databases. This means no database servers running heavy algorithms is required, instead the distributed variables are handled via messages exchanges between nodes. This new alternative relies on an innovative way to represent the distributed variables. As opposed to the usual approach where distributed variables are generally just values registered and updated in a specific database, variables will be represented as a specific data-structure called Conflict-free Replicated Data Types (CRDTs[2]). It is the key concept that will be detailed below to understand this new approach along with all its advantages. TODO: put better representation (image) and put more references.



1.2 CRDT

CRDT is for Conflict-free Replicated Data Type. The main idea is that it is an abstract data type with an interface designed for replication on multiple nodes and satisfying the following properties[3]:

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

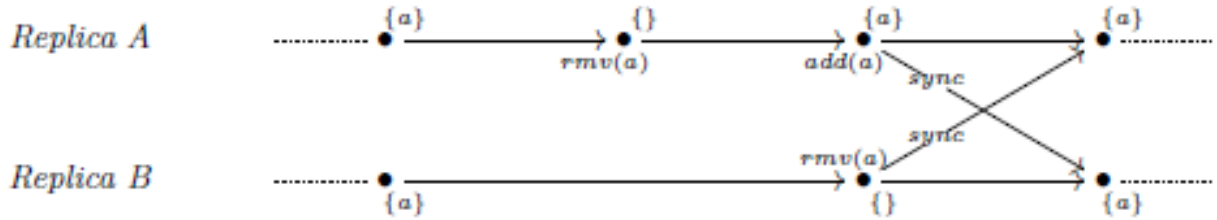
Even if this approach might look surprising at first sight (since it does not involve recording the distributed variable state in a specific place such as a database, nor require any consensus algorithm), this new way to represent distributed variables introduced in 2011 is already used by some big compagnies such as Riot Games, TomTom, Bet365, SoundCloud and some others[4].

1.2.1 Principles

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

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

Here is an extremely basic example[5] with an add-wins set (if concurrent add and remove occur, the add wins).



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

1.2.2 Advantages

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

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

state messages from other nodes and directly update its local state to represent the most recent version.

Let's consider that strong Consistency is good for the ease of programming but requires heavy synchronization algorithms. At the opposite, we can consider that weaker Consistency is harder to use for the application developer (he is not sure every node has the same value for a distributed variable) but requires less synchronization algorithms. With these two basic principles in mind, we would logically want a consistency model as strong as possible while running with a synchronization algorithm as light (weak) as possible. Here, CRDT new way to handle distributed variables comes in with a very efficient model allowing strong eventual Consistency with a weak synchronization algorithm (even called "sync free", which was the name of the initial project leading to Lasp development[6]). Strong eventual Consistency (SEC) is achievable to the fact every node receiving the same set of updates (in any order) have equivalent state and the fact every update will be eventually delivered to every node due to peer-to-peer communications. It is in fact even stronger than that since nodes do not require to receive the exact same set of updates, some previous updates may not be received that it will not perturb the system as long as recent messages eventually deliver.

In regard of the CAP theorem, CRDT model allows strong eventual Consistency[7] with high Availability and Partition tolerance which is probably the best compromise from the CAP theorem yet while not even requiring any heavy algorithm (no consensus required!).

No doubt the good features and properties described above together with the excellent CAP theorem compromise are the reasons why CRDT usage is growing quickly[8] and has been adopted by some big compagnies as previously mentioned. TODO: metre en avant à quell point c'est bien !

1.3 Lasp

Lasp is an experimental implementation of CRDTs developed by the Ecole Polytechnique de Louvain (EPL) research team and initiated in 2013 with the impetus of two European projects; SyncFree[9] in 2013 then LightKone[10] in 2017. More precisely, it takes the form of a group of Erlang libraries acting together to offer a programming framework based on CRDT. There entire project can be found on their official github repositories: https://github.com/lasp-lang/lasp.

1.3.1 Lasp libraries

The libraries offer everything to handle different types of distributed variables (different kind of CRDTs are implemented such as counter, set, Boolean, map…) including the communication part, distribution and easy-to-use API to update or query on CRDTs. The particularity of Lasp compared to other alternatives to handle CRDTs is the tools it offers to manipulate and compose on CRDTS. Indeed, CRDTs are very handy to easily handle distributed variables but they require caution when using their outputs to compute or compose data. More precisely, the CRDT itself composed of values and metadata will reflect the known most recent version of itself but this is not especially the case for the

values we got from querying the CRDT previously. In other words, if a developer queries the value of a CRDT then computes something based on this value, he got the most recent value from the CRDT and his computation is momentarily true. But any moment later, his computation might be wrong since the CRDT got updated but the value he got previously from it was not updated unless he queries again the CRDT and starts his computation again. Lasp is specifically designed to address these issues and to allow easy computation and even composition on CRDTs without requiring the developer to handle these problems himself[11]. The idea is to consider the CRDT as an input stream and to ouput a stream of values always leading to the known most recent value. With this approach, the developer has tools to compute things based on a CRDT while his computations will be automatically updated with the CRDT. Thus Lasp offers a complete API to facilitate always updated unions, intersection, maps,… on CRDTs. This particular aspect is very convenient but will not be discussed in this master thesis since it will mainly focus on the distribution and communication part without detailing in depth the end developer aspects.

1.3.2 ORSWOT

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

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

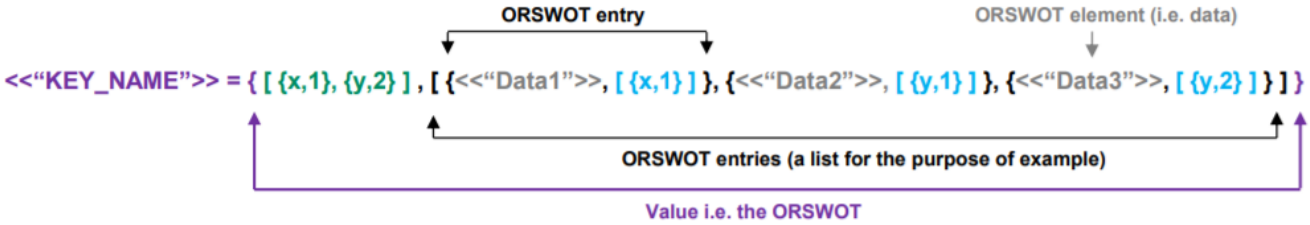
Thus, the ORSWOT is the general selected CRDT for this master thesis. As a little remark, the generic name ORSWOT comes from orset without tombstones due to the fact, as just explained, it addressed the tombstone issue from the generic orset. As a note, the rest of this document might use the name “awset”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

Finally, since it is the CRDT used for this work as a core use case, let's go a little bit more in depth about its implementation then let's illustrate with an example. A very good presentation from Bet365[12] shows the ORSWOT principle, which is why my explanation will re-use some of their own examples.

The data-structure is as follow:

- It starts with a version vector. It is a set of tuples of size 2 (pairs).
    - Each pair consists of a unique actor name (unique identifier for a replica) and a counter. It is represented in green on the example image.

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



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

- When a node adds an element:
  The version counter is updated, incrementing by 1 (or setting to 1 if not currently present) the pair for that unique actor.
  A pair is added in the entries with the added element and the updated pair {UniqueActorName, Counter} as its Dots. If a pair already existed for that element, it is replaced by the new one.

  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">>]

- 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}] }] }

- 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.

Merging example:

ORSWOT A: { [{x,1}, {y,2}] , [{<<"Data1">>, [{x,1}] }, {<<"Data2">>, [{y,1}] }, {<<"Data3">>, [{y,2}] }] }

Seen:

1. **Adding element** <<"Data1">> **via actor** x
2. **Adding element** <<"Data2">> **via actor** y
3. **Adding element** <<"Data3">> **via actor** y

ORSWOT B: { [{x,1}, {y,1}, {z,2}] , [{<<"Data2">>, [{y,1}] }, {<<"Data3">>, [{z,1}] }, {<<"Data4">>, [{z,2}] }] }

Seen:

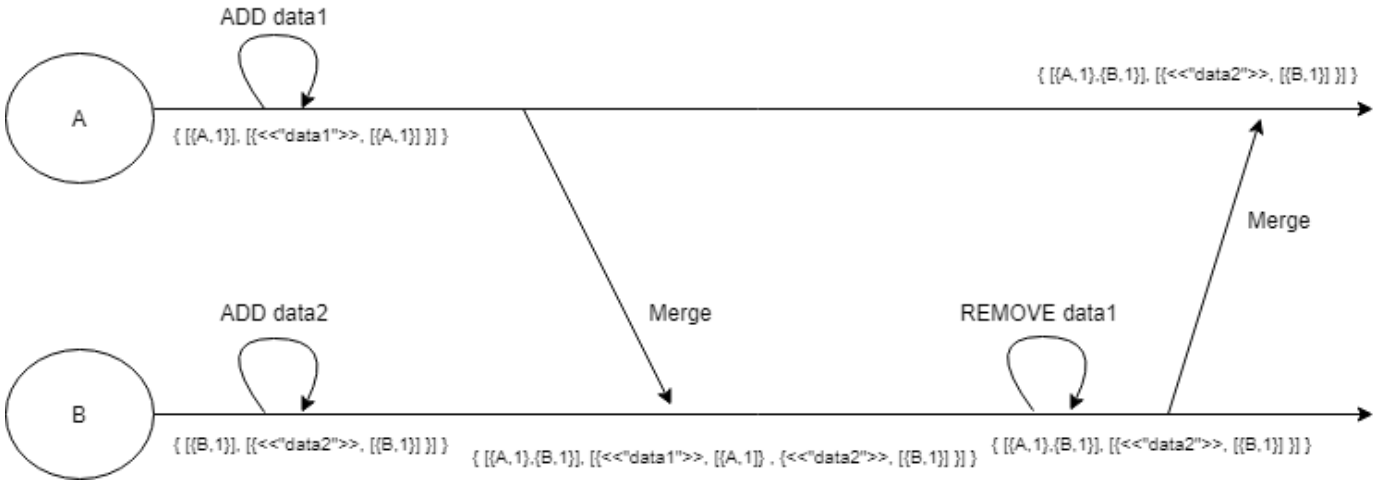
1. **Adding element** <<"Data1">> **via actor** x
2. **Adding element** <<"Data2">> **via actor** y
3. **Adding element** <<"Data3">> **via actor** z
4. **Adding element** <<"Data4">> **via actor** z
5. **Removing element** <<"Data1">> **via actor** z

Merged ORSWOT: { [{x,1}, {y,2}, {z,2}] , [{<<"Data2">>, [{y,1}] }, {<<"Data3">>, [{y,2}, {z,1}] }, {<<"Data4">>, [{z,2}] }] }

CRDTs implemented in Lasp are state-based, meaning the peer-to-peer messages simply contain the CRDT state (and no operation as opposed to operation-based). This means the three operations;

adding, removing and merging are everything that is needed to implement and understand the ORSWOT (awset in Lasp).

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



1.4 Goals and contributions

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

1.4.1 Improved API

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

In this optic, a first goal was to improve the API in a particular direction related to the convergence principle described in section 1.2.1. Indeed, convergence is a fantastic feature allowing every node to eventually end up with a consistent state without requiring any heavy synchronization algorithm but how much time does it take for a specific cluster of nodes to converge? There was, up to now, no directly available tool to easily know such information from an end developer perspective. This is an important issue for practical usage since an application would probably not work as intended on a

cluster than converges in 10sec instead of 1sec for example and the developer had no tool to easily detect that. Therefore, a first objective was to develop a measurement tool that could be easily incorporated in Lasp to measure convergence time together with network utilisation.

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

1.4.2 Measurements

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.

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

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

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

- Cluster size
- Geographical distance (at a small scale)
- Nodes under partition
- CRDT size
- CRDT operation

- CRDT value update speed

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

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

1.5 Summary and structure

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

2. Contributions

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

2.1 Measurement tools

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

2.1.1 Principle

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

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

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

A first idea was to allow every node to be the source of the measurement signal to allow convergence time measurement from any source on the cluster. This is probably a good idea if we are only

interested in measurements and precision. But since the tool is designed to be run easily in background on a real under-use cluster, it was preferable to only have one node initiating and orchestrating the measurement where other nodes simply answer. This allows smaller impact on the performance (nodes workload and network usage) while still giving some general information about the cluster convergence time.

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

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

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

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

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

2.1.2 API

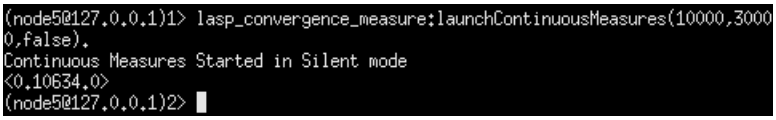
The API to use the measurement tool is straight forward. There is a function that must be called on every node to initiate the measurement process with some specific arguments (it can be launched at node booting for example) then a few getters to get information about the cluster whenever needed. As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

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

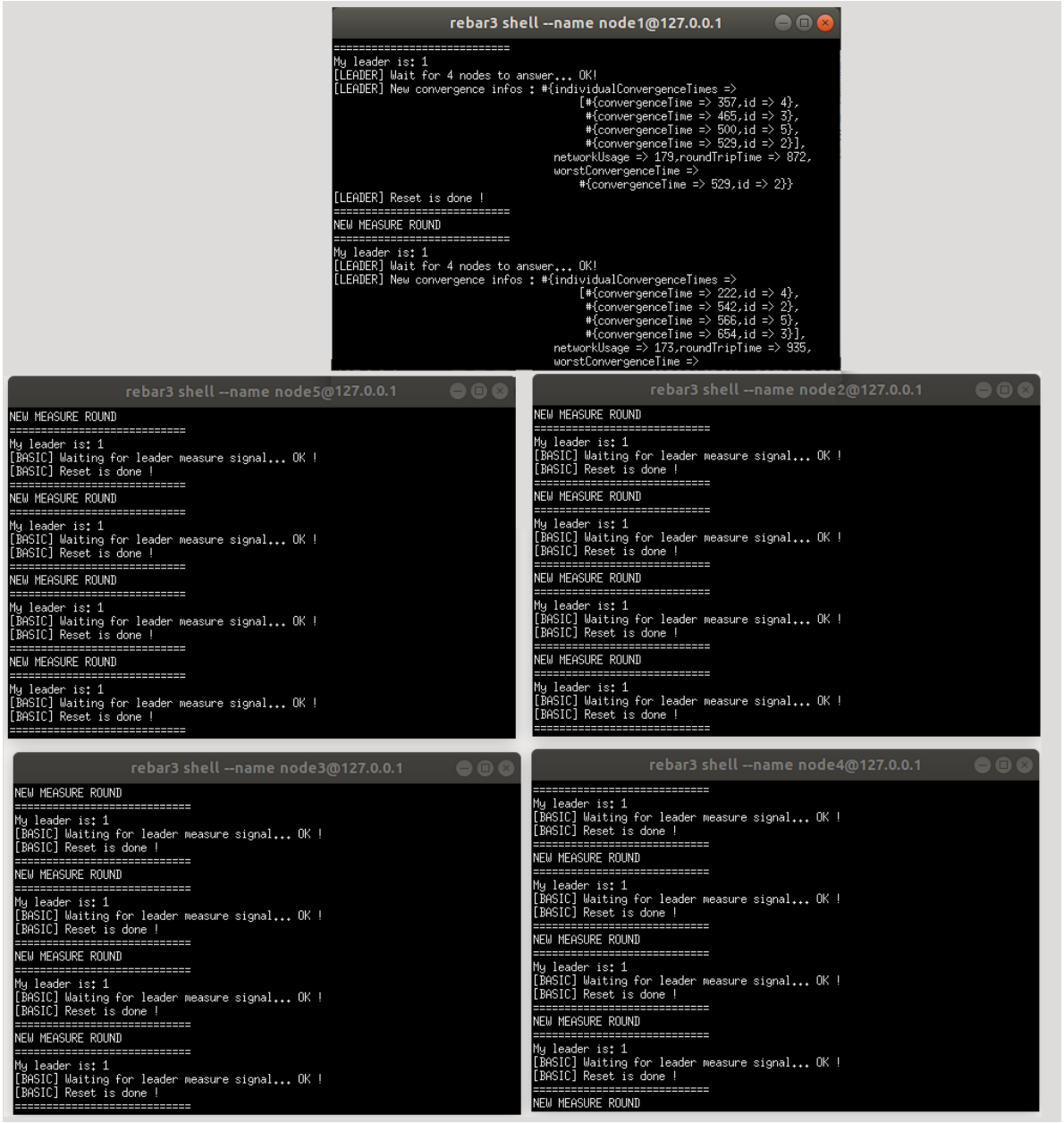
Here is the complete API with some little examples on a local cluster of 5 nodes:

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

Example usage, Debug=false :



Example usage, Debug=true :



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

Example usage :

```
(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>
```

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>
```

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>
```

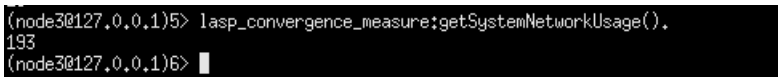
Function Name : **getSystemRoundTrip**	Queries the information from the cluster most recent measurements to get the round-trip time.
Output	Returns the round-trip time (ms) from the most recent measurement information. As a reminder, the round-trip timed is considered as the time between leader signal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

Example usage :

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

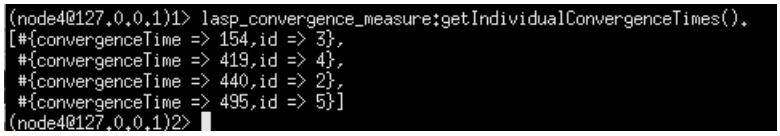
Function Name : **getSystemNetworkUsage**	Queries the information from the cluster most recent measurements to get the network usage information.
Output	Returns the number of messages per sec delivered on the cluster. It is computed as the sum of all the nodes received messages per sec.

Example usage :



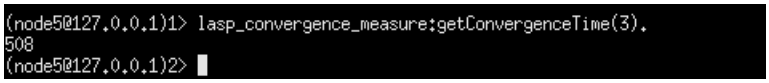
Function Name : **getIndividualConvergenceTimes**	Queries the information from the cluster most recent measurements to get the convergence times for every nodes.
Output	Returns a map containing the convergence times (ms) for every node. Convergence time is considered here as the time for that node to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

Example usage :



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

Example usage :



2.2 Adaptation tools

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

2.2.1 Principle

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

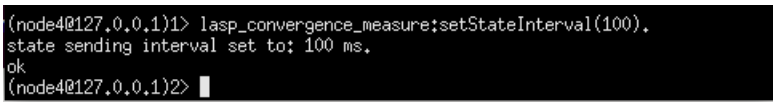
2.2.2 API

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

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

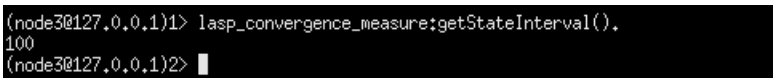
Function Name: **setStateInterval**	Modify the state_interval which defines the time interval between each state sending from the nodes.
Argument1: newStateInterval	The new desired state_interval value in term of ms.
Output	No output. Simply modify the value which may take up to one convergence time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

Example usage :



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

Example usage :



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

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

2.3 Scripts

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

2.3.1 Static measurement scripts

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

These scripts include:

- LaunchSet1.sh: It launches a cluster of 5 nodes. The cluster shares an initially empty awset where nodes will each add elements and wait for convergence. Each node adds 10 unique elements (all at once) on the CRDT and wait to detect the final 50 elements (since there are 5 nodes and each adds 10 elements). The script runs this experiment 50 times (iterations) with the same parameters then switch to another version with other parameters. The different versions are:
  - Nodes puts 10 elements (all at once) and wait to detect 50 elements.
  - Nodes puts 100 elements (all at once) and wait to detect 500 elements.
  - Nodes puts 1000 elements (all at once) and wait to detect 5000 elements.
  - Nodes puts 5000 elements (all at once) and wait to detect 25000 elements.
  -Nodes put elements then join cluster (as if every node added their elements on local state while under partition then resolved partition).

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

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

- LaunchSet2.sh: This uses the exact same principle and structure as the previous one, again with a cluster of 5 nodes but runs the following experimentations:
  - Awset starts with 50 elements, every node removes 10 elements and wait until it is empty.
  - Awset starts with 500 elements, every node removes 100 elements and wait until it is empty.
  - Awset starts with 5000 elements, every node removes 1000 elements and wait until it is empty.
  -Two versions of each case is run, joining before or after removing elements.

- LaunchSet3.sh: This is exactly similar to LaunchSet1.sh with nodes adding 10,100,1000 elements but on a cluster of 10 nodes, thus nodes wait to detect respectively 100, 1000 and 10000 elements.

- LaunchSet4.sh: This is exactly similar to LaunchSet2.sh with nodes removing 10,100,1000 elements but on a cluster of 10 nodes, thus starting with CRDT of respectively 100, 1000 and 10000 elements and waiting to detect it is empty.

- LaunchSet5.sh: This is exactly similar to LaunchSet1.sh with nodes adding 10 or 100 elements but on a cluster of 20 nodes, thus nodes wait to detect respectively 200 and 2000 elements.

- LaunchSet6.sh: This is exactly similar to LaunchSet2.sh with nodes removing 10 or 100 elements but on a cluster of 20 nodes, thus starting with a CRDT of respectively 200 and 2000 elements and waiting to detect it is empty.

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

2.3.2 Dynamic measurement scripts

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

- LaunchSet7.sh: This script launches a cluster of 5 nodes that will share an awset. Every node will remove an unique element and add another one (the CRDT number of elements eventually stays the same) on a regular time basis while outputting to files the added element, removed element and current awset local state. The iteration is not, as previously done, handled by killing the cluster and restarting it again for next iteration but by letting it run longer to allow more elements addition/removal and thus more measurements. Here are the update speed tested:
  - One element is added/removed every 0.5 sec (2/sec).
  - One element is added/removed every 0.25 sec (4/sec).
  - One element is added/removed every 0.05 sec (20/sec).
  - One element is added/removed every 0.01 sec (100/sec).

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

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

2.3.3 Quality-of-life scripts

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

These little handy scripts include:

- LaunchBasicNode.sh that simply launches a local node that does not nothing special but can be used for little manual tests.

- LaunchBasicCluter5.sh that simply launches a cluster of 5 nodes that does nothing automatically but allow manual tests on the cluster. Also present in version LaunchBasicCluster10.sh for a 10 nodes cluster.

- Clean_measures.sh that simply removes every trace from previous measures. This is normally automatically run when starting a new measurement script. While handy, this means if you want to run the measurements scripts, you must save the outputs or result files in a remote folder to avoid deleting them.

- Recompile.sh that simply recompiles the entire lasp code including CRDT types, partisan (communication layer), lasp core modules, etc. This should not be used unless you want to modify lasp code and test your modifications.

- LaunchLeaderElection5.sh is a simple script that launches a cluster of 5 nodes that tests the leader election protocol that is used for the continuous measurement tool. Basically, it is just a debug tool to check that the leader election works correctly. It is also available in 10 nodes version with LaunchLeaderElection10.sh. If this does not work smoothly, verify that you entered your IpAddress (or 127.0.0.1) in the related file (see readme files).

- LaunchContinuousMeasurementSilent5.sh launches a cluster of 5 nodes that run continuous measurement in background. In other words, it is a real example of the developed tool that runs on a cluster that you can use for tests, modify convergence time and check the impact on measurement in a dynamic way. See section 2.1 and 2.2 for more details on the API to use. This is also available in 10 nodes version with LaunchContinuousMeasurementSilent10.sh.

- LaunchContinuousMeasurementTalkative5.sh launches a cluster of 5 nodes that run continuous measurement under debug mode. While not practical due to all the prints, it is very interesting to analyse to understand in real time the measurement tool principle.

- 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 additions

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

2.4.1 Memory leak

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

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

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

This simple modification allowed the process memory size to stay smaller than 0Mb where previously starting at 0 Mb it escaladed quickly to hundreds Mb. This was also translated into convergence

measurements being more constant where previously it was progressively slowing down (obviously due to the process eating all the memory).

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

TODO: modify this section since it was not exactly what I thought. Explain that the memory leak was due to the CRDT update which was not correctly used due to the wrong readme file from official github.

2.4.2 Readme improvement

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

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

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

3. Measures and results

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

3.1 Parameters

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

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

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

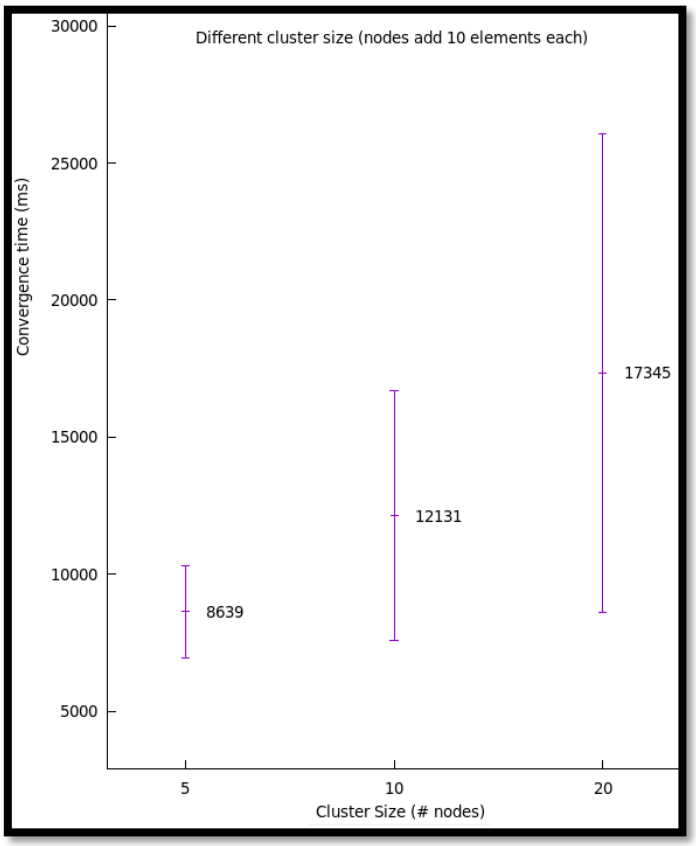
3.2 Measures

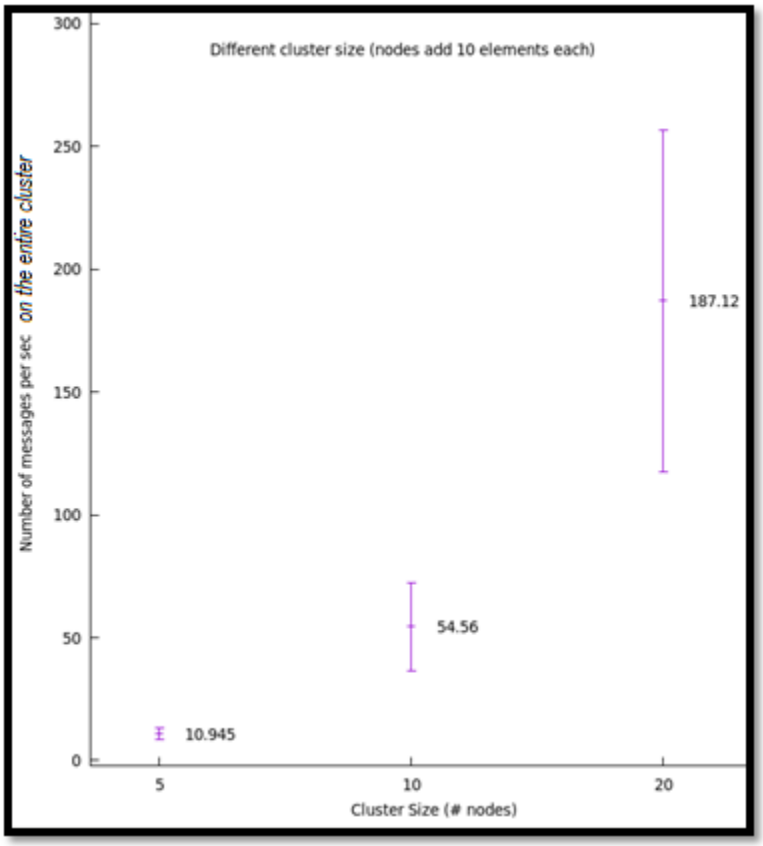
The number of iterations for every experiment is always 50. The majority of the measurements were executed with the scripts described in section 2.3. While all the measurements are not present on this work github page, some examples are (in folder mylasp/lasp/Memoire/Saved_measures). More details are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

3.2.1 Number of Nodes

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

Lasp (as given from Lasp official github) and was used for the mast majority of the measurements below.



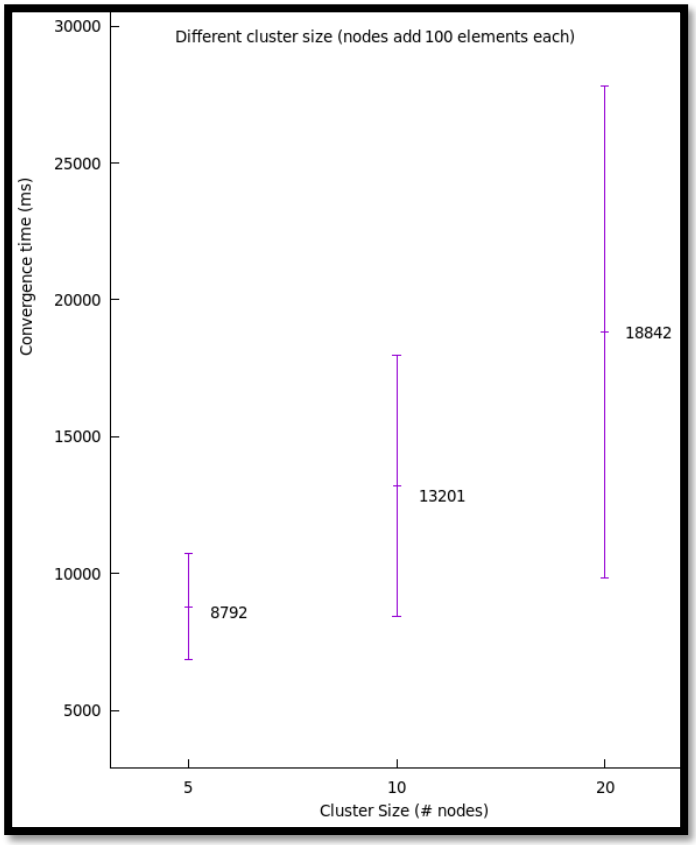


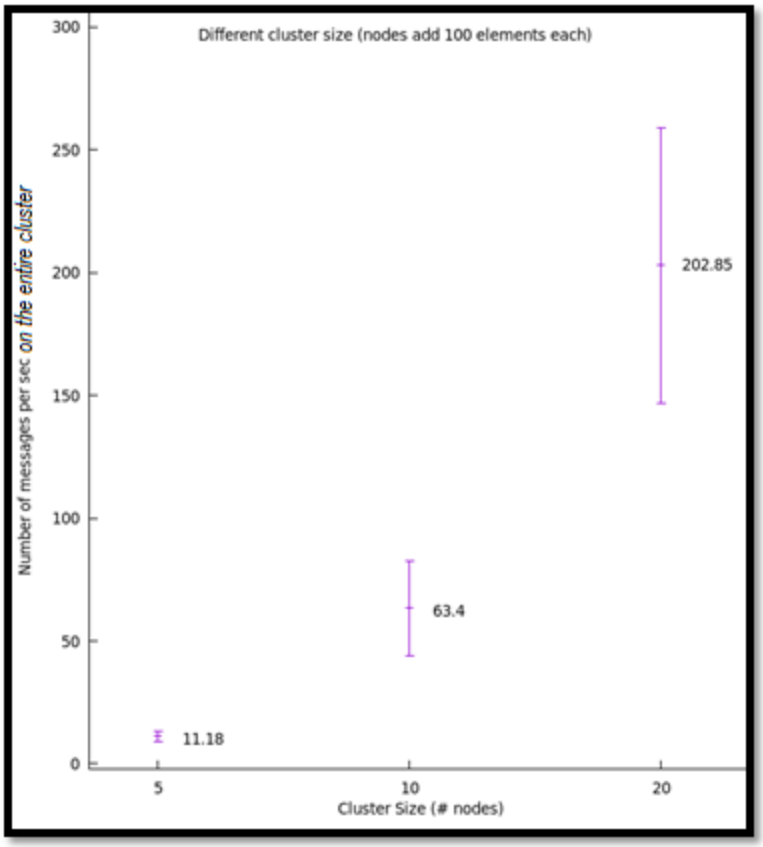
At first sight, these convergence time values might look very big (order of magnitude of 10 seconds) but this is due to the state_interval Lasp default value that is 10 seconds. The fact, the convergence time is of same magnitude as the state_interval is already a quite logical sign about the measurement 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 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 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 assume 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.

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. To be sure the variations in the results are only 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.

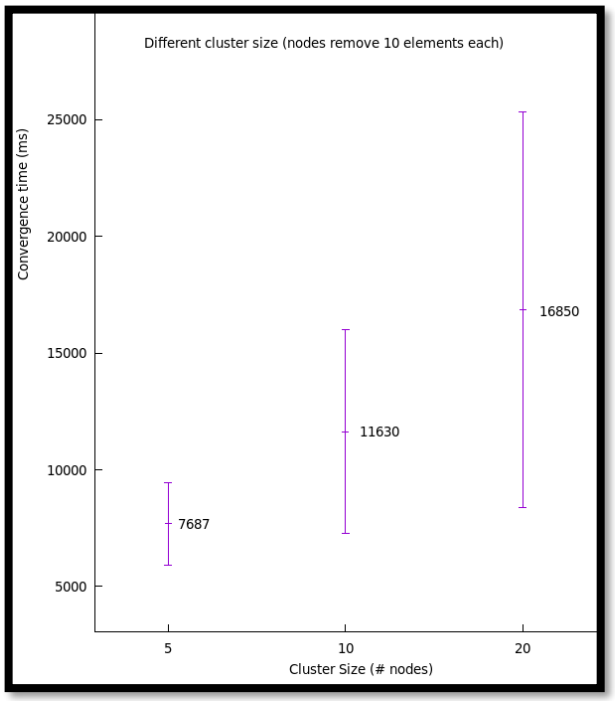


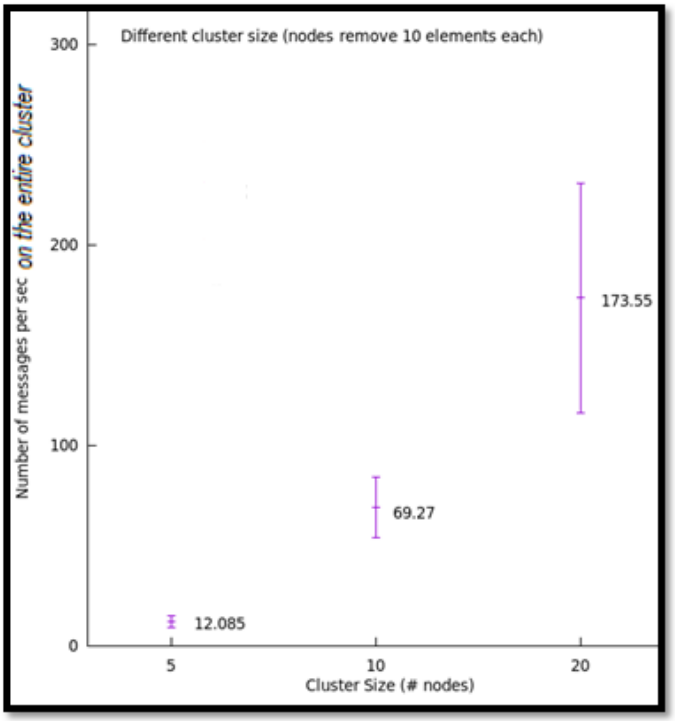


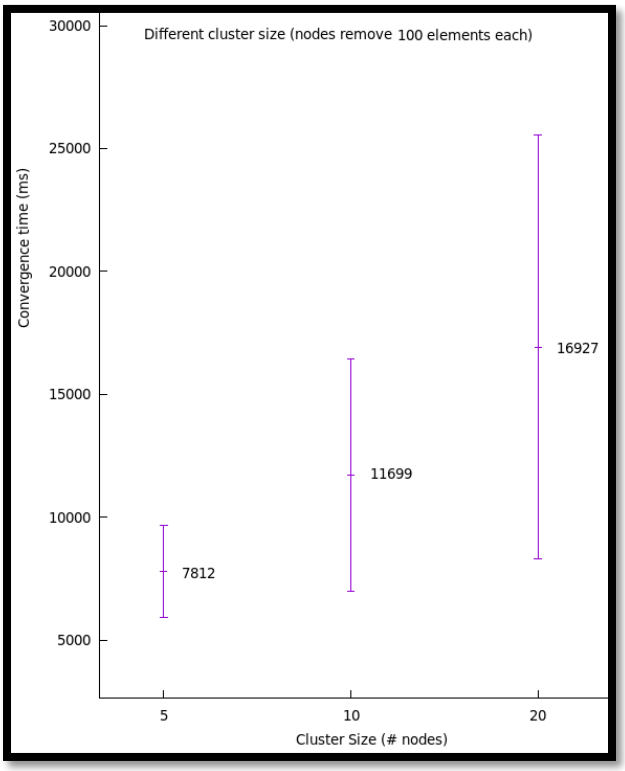
Clearly, with nodes adding 100 elements instead of 10, the exact same behaviour shows up, confirming the results were reflecting the modification in number of nodes and not the slight variation in the CRDT

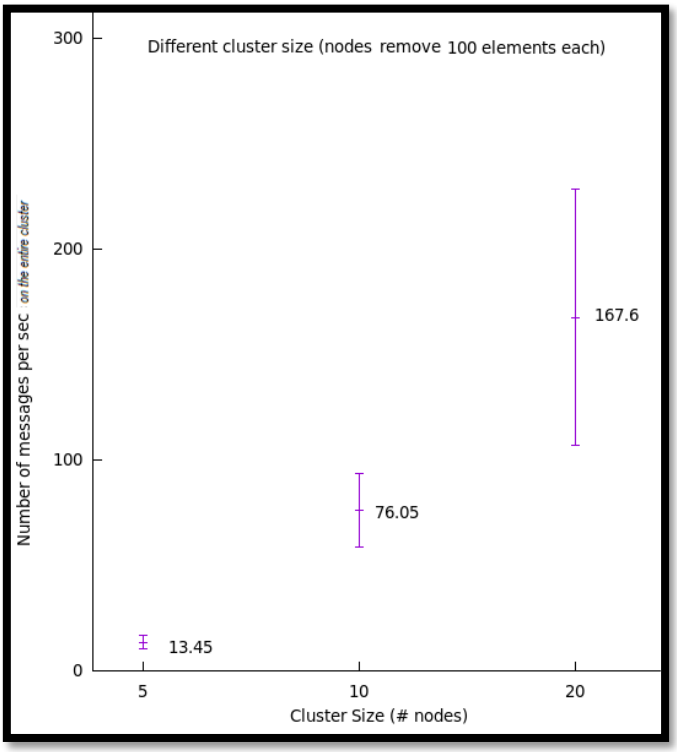
size. Still, the finest readers might remark the convergence time is very slightly higher on this new graph but the impact of the CRDT size on the convergence will be discussed in section 3.2.3.

Finally, one last question 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 are above but for elements removal instead of addition.









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 size 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, meaning if the 5 nodes cluster was run with the following setup: 2 computers running 2 nodes each and one little (slower) computer running one single node, the 20 nodes cluster was much more complicated with one computer running 10 nodes, one other computer running 9 and the smaller (slower) computer running one. This obviously is not ideal at all since while modifying the number of nodes (intended modified parameter) it also modify 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 precise 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 a good sign for Lasp future.

3.2.2 Nodes distance

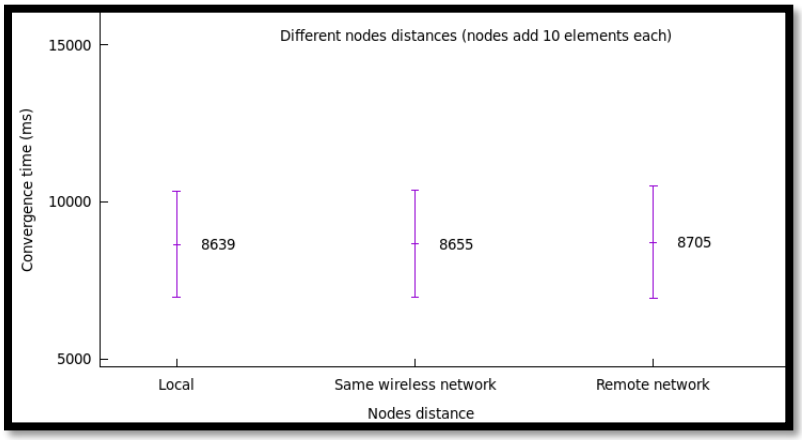
After checking the impact of the number of nodes, another important parameter in regard to the nodes is the distance between them. To measure this, three different nodes setup where tested:

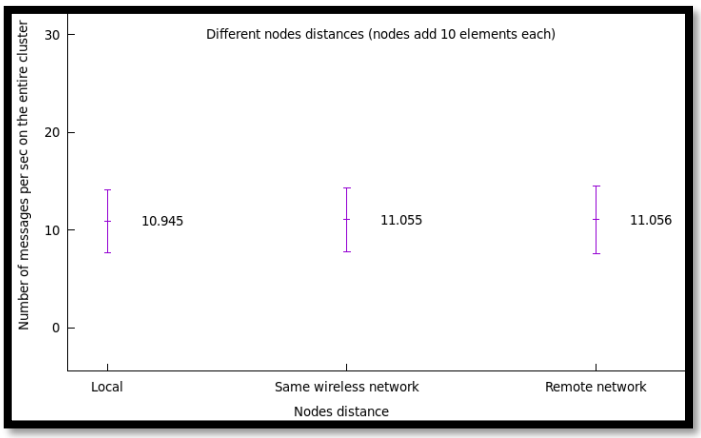
- Only local nodes (on a single computer)
- Nodes under a same wireless network (small distance)
- Remote nodes (nodes locally and nodes on a remote network which is around 20 km distance)

One difficult task for this set of experimentations was the orchestration part. Since it was not as simple as 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 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 which are of the magnitude order of seconds not ms anyway.

Now that the orchestration question is resolved, 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).





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 full local 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).

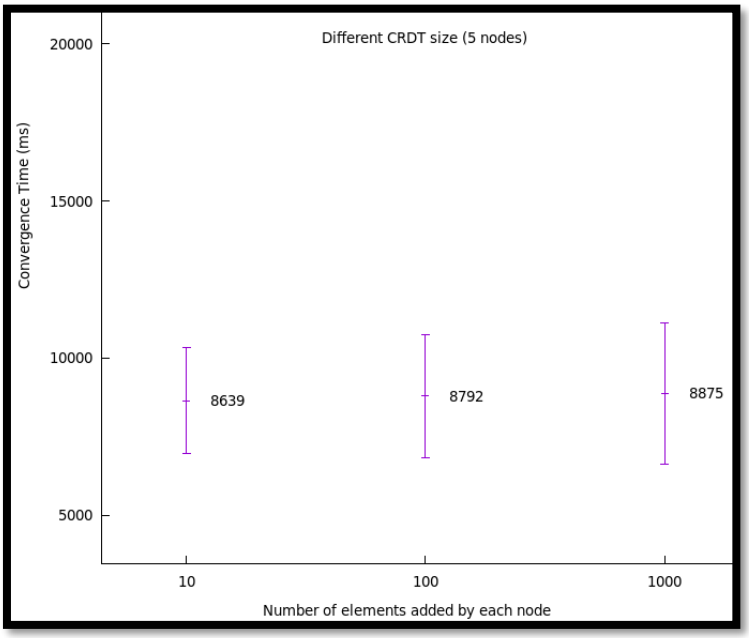
Again, for a more rigorous experimental approach, this experiment was run with nodes adding 10 and 100 elements then with nodes removing elements instead of adding them. While showed in the case of the first parameter (number of nodes) to present the entire experimental approach to the reader, I decided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

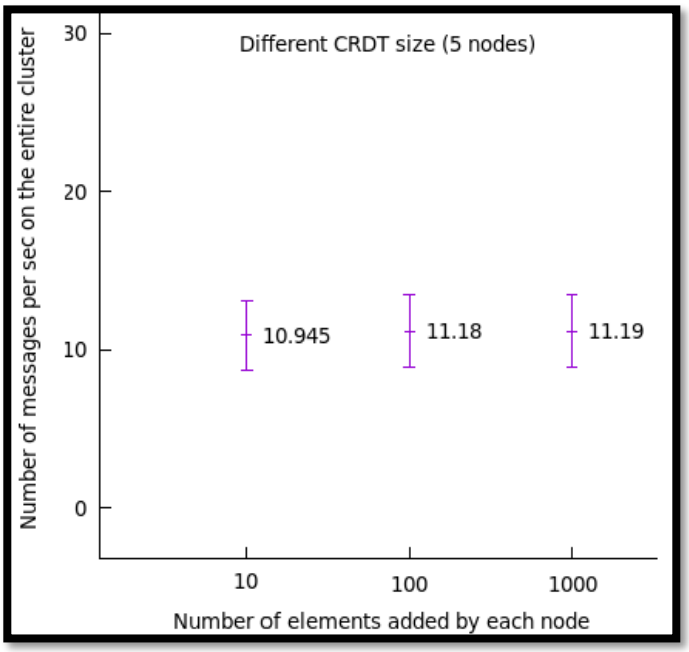
3.2.3 CRDT size

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, as an opposite way, let's try to not modify other parameters and focus on the CRDT size itself and thus the number of elements in the CRDT.

As a remind about the experimental procedure, 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 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).

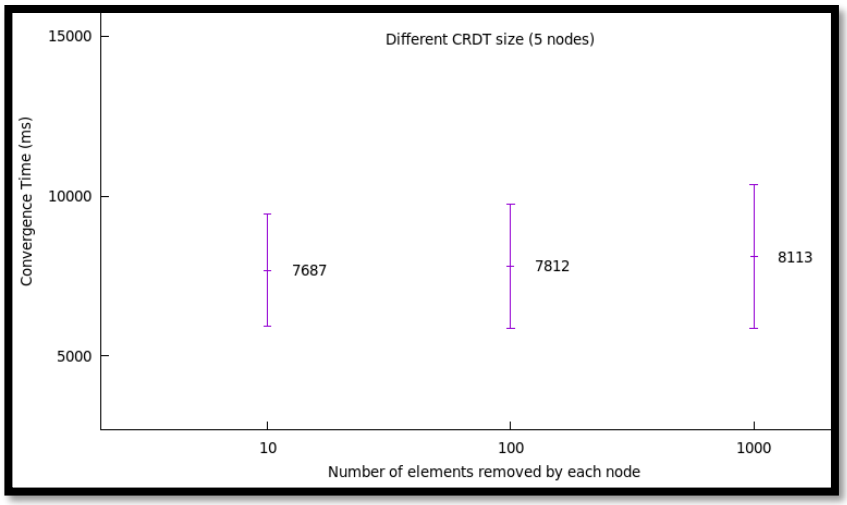


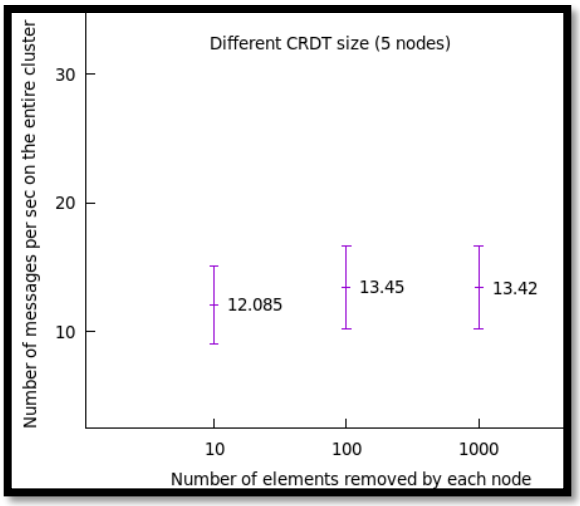


From these results, it looks like the convergence time for an awset (CRDT) containing a total of 50, 500 or 5000 elements is hardly impacted by its size. Indeed, the convergence time seems to be very slightly longer when the number of elements is bigger, but the variation is very little. Since a CRDT 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).



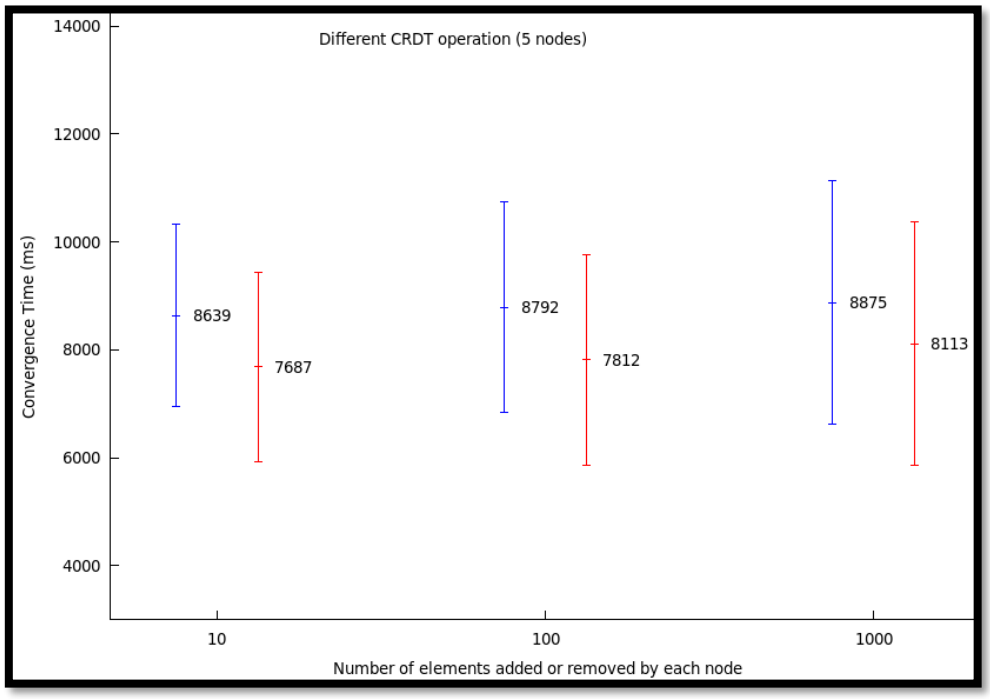


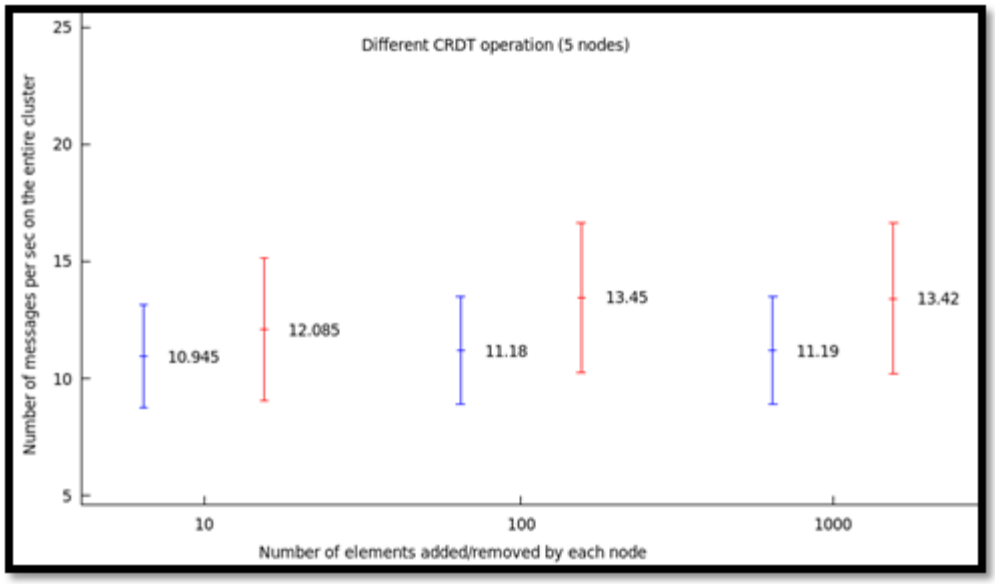
When regarding these new graphs, we see globally the same profile with CRDT initially containing 50 elements, 500 or 5000 not having a big impact on the convergence time. Still, we see, as for additions, a small increase in convergence time with the number of elements. About the network usage, we see globally that the number of messages per second is not really impact 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.

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). For more details, see annexes. TODO add annexe.

3.2.4 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.



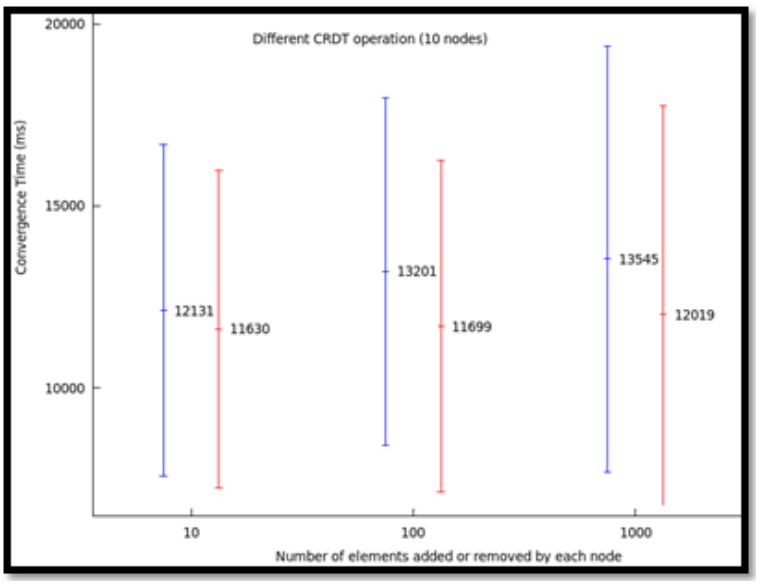


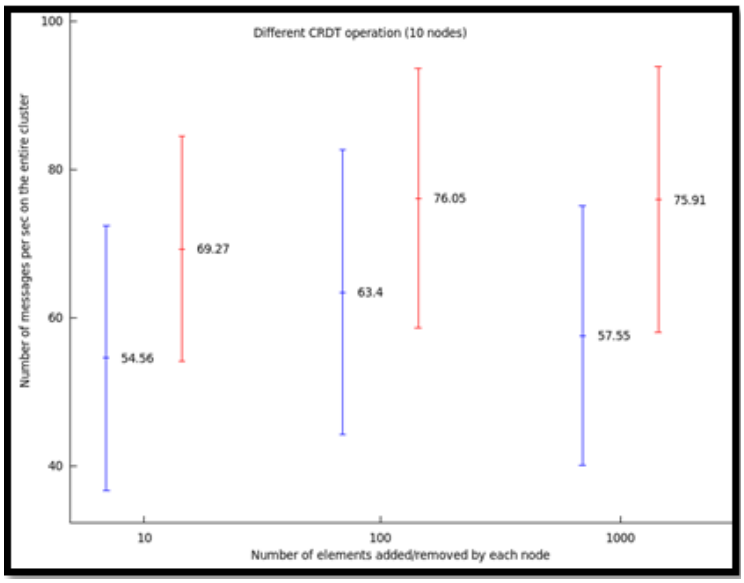
On these graphs, in blue is the elements addition and in red is the elements removals. Measures for adding/removing 10 elements, 100 elements and 1000 elements (by each node) are represented. We see that, as it was shown in previous section (3.2.3), convergence time tends to be a bit longer when the CRDT size increases. 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 CRDT size, 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 nearly 1 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.

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





Again, is 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*(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*(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)

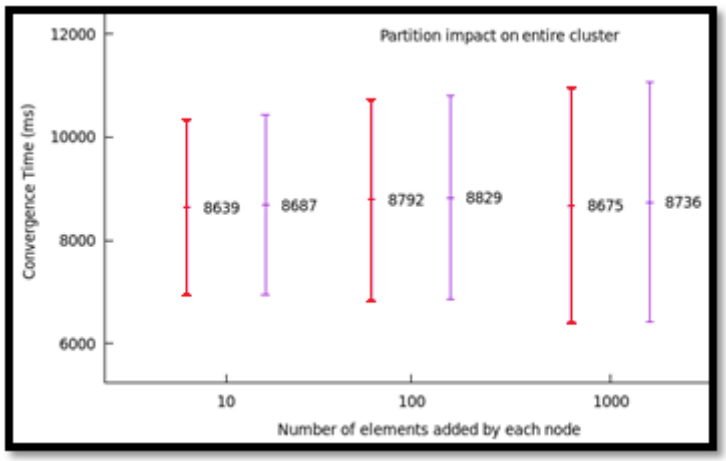
As a note about the network measurement, nodes count the number of messages received while waiting for convergence then divide this value by the convergence time to have the number of messages per second. On the graphs, the represented value is the sum of all the nodes messages per second to represent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

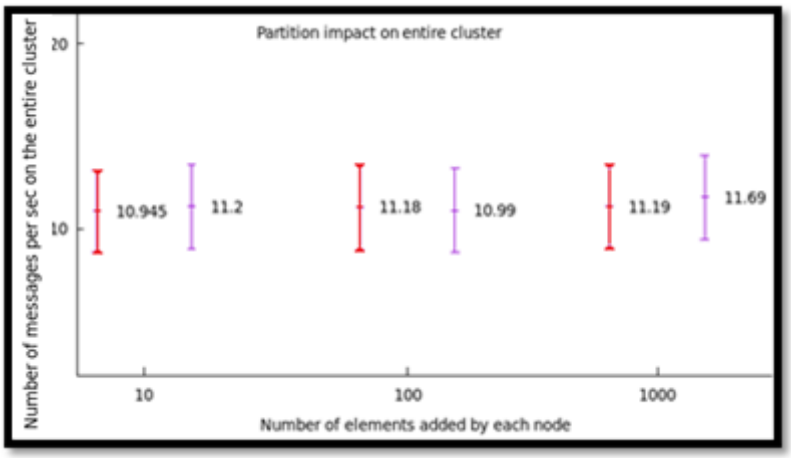
As a conclusion for this specific element under test (CRDT operation), while the measurements show some interesting results, no precise explanation have been found yet to explain such behaviour, nor in theoretic elements nor in the experimental approach.

3.2.5 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 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.

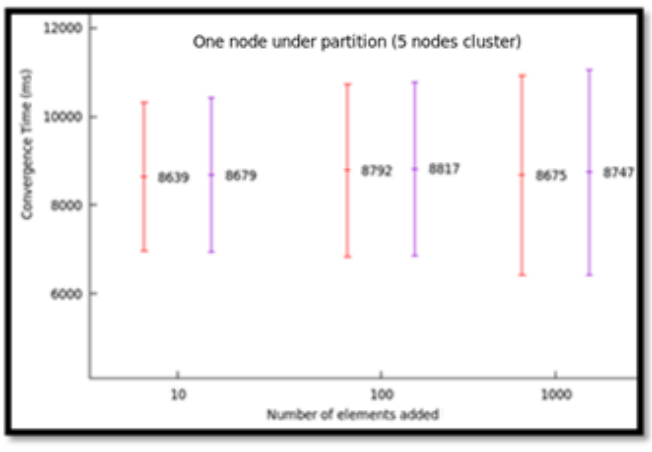
This is theoretically well handled, let's see if it's the case in practice. Firstly, I wanted to check if there was an impact on the cluster if a node was under partition while updating its local state then resolved partition. In other words, does the updates done while under partition are directly taken 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.

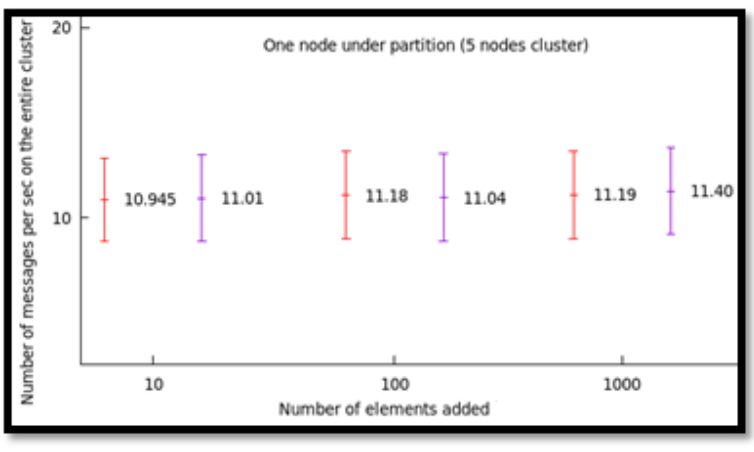




In red is the normal scenario with no partition (previous measurements) and in purple is the new scenario measured where nodes are under partition when updating the awset then resolve partition. Again, as for previous measurements, it was tested with updates adding 10 elements, 100 elements or 1000 elements 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 exact 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 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 were nodes add elements in an awset (again 10,100 or 1000 elements), before having the time to converge (for this following measurements, actually 1 second after the elements addition), one node gets partitioned 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.

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 real 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 (graph: one node under partition) goes logically with the first two graphs (partition impact on the entire cluster) 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. Fore more details, see annexes. TODO add annexe.

3.2.6 Value update speed

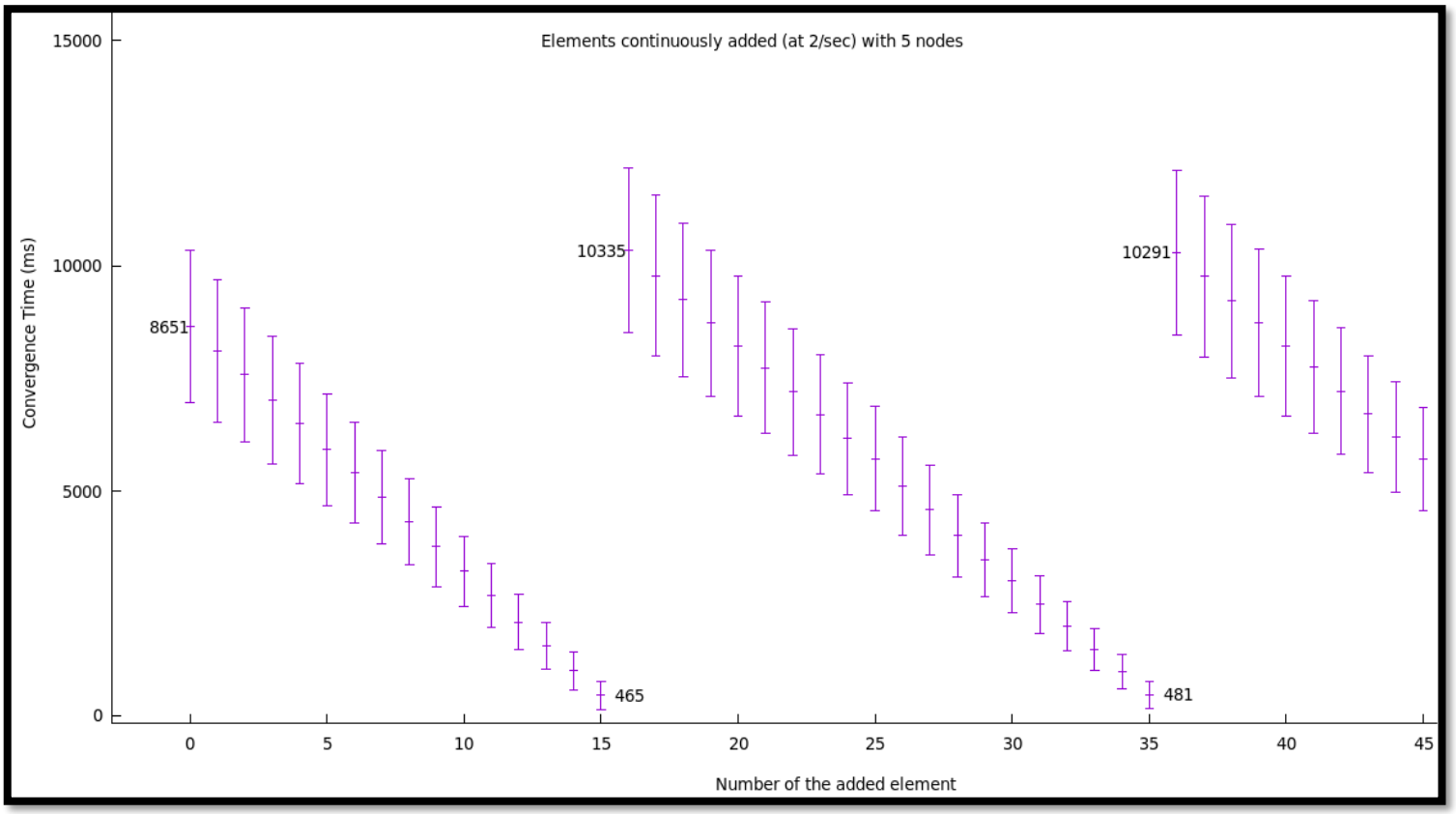
All the previous measurements were from a static approach. Indeed, nodes did some update then started measuring while doing nothing other than waiting for convergence. This approach while easy to implement in practice for measurements does not cover many real usage cases where a CRDT value is updated 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.

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, 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 update speed. By printing on regular time basis to files the nodes local states together with timestamps and information on the removed and added element, it is possible to analyse the files after the experimentation to measure the time it required for the nodes to receive other nodes updates (elements).

Since there are a lot of measurements which can become incommodious to analyse all at once, let's just show on a graph the measured convergence time for other nodes to detect elements added by node 1. 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 presented on the github page of this work (the readme file there can easily guide you to the results you may be looking for).



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 arounds 0.5 second less time (around 8.15 seconds) to be detected by other nodes, element 3 takes again 0.5 second less time to be detected and so on… Element 15 is detected very quickly then the next element (16) gets extremely long (slightly more than 10 seconds) to be detected, starting again the decreasing convergence times.

This can be easily explained by looking at Lasp principle. Since nodes (here we focus on node 1) send their states every 10 seconds (default state_interal is 10000ms), every update done during that time interval simply affects local state, then on timer trigger, the state is sent. This means only one specific state (the most recent one) is sent 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 at 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:

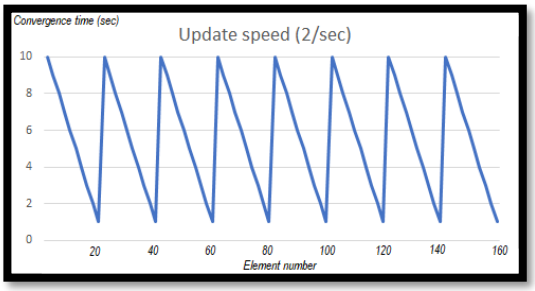
- Updating the awset elements at a faster rate than the state_interval will result in a cluster that only receive some of the source states. Indeed, if a node acts like a source continuously modifying elements in the awset, only its most recent state will be sent at every state_interval period hiding 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 and thus there will be no moment where the state on every node (including souce node) would be consistent. To achieve a real consistent state on every single node including the continuously updating source node, it would require the state_interval to be at least shorter than the value update speed. 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.

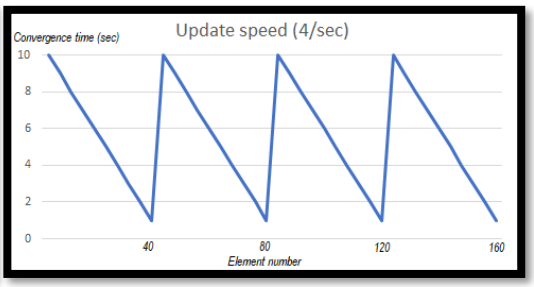
- The previous measurements (3.2.1 to 3.2.5) had to start the experiment and measuring convergence time at a moment which was not especially at the start of a new state sending interval. This means, as clearly visible on the graphs, some big standard deviations are measured since the moment when an update is done (element added for example) has an impact on the 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

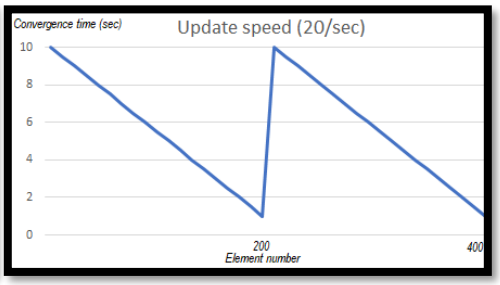
delay between node booting and start of experiment to avoid the exact phenomenon described above which, otherwise, would have resulted in totally inconsistent measurements.

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 slower.

Basically, to have a quick overview, the different cases can be resumed as these summaries (which as simply overview figures):







The only difference is the number of updates 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), the first 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 all the 40 first 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.

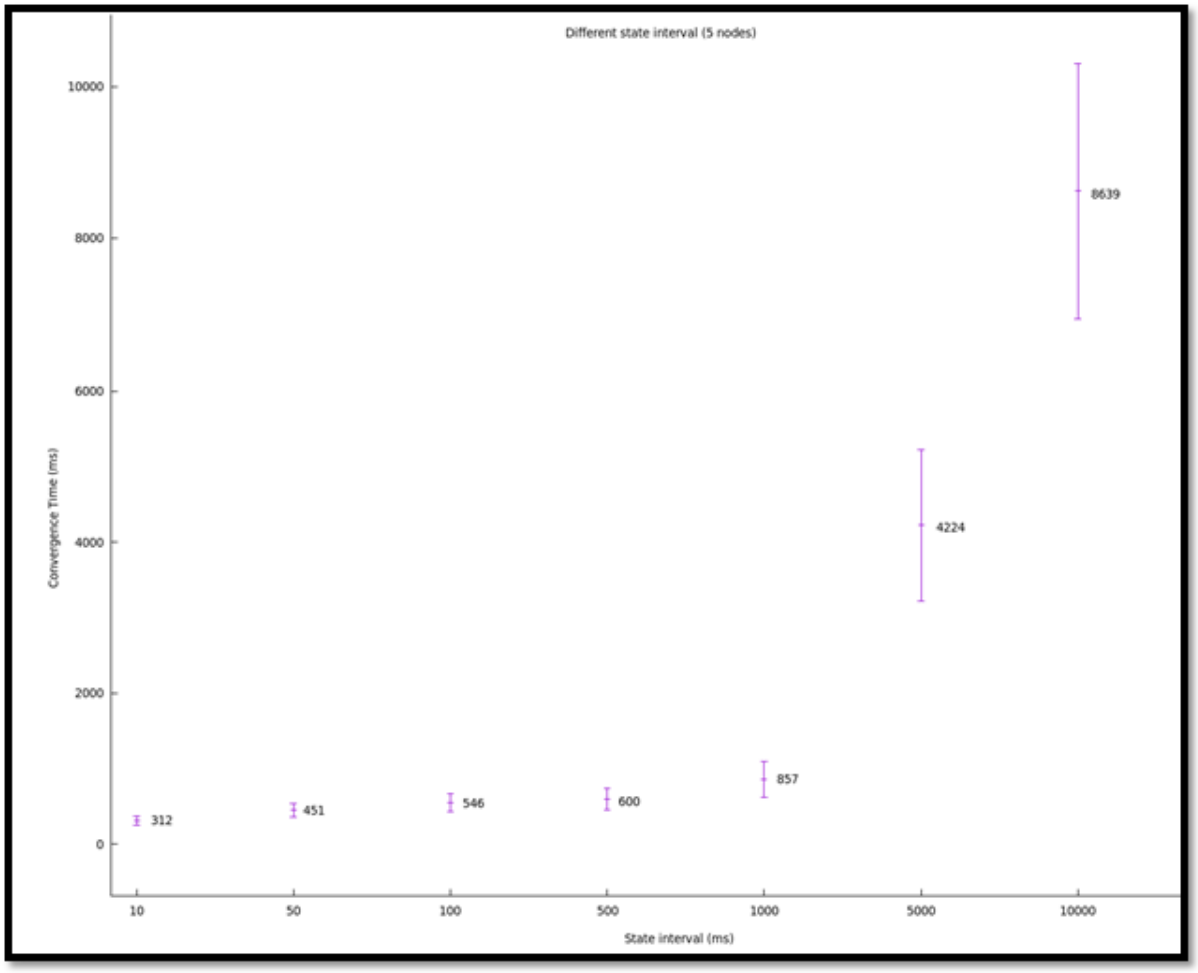
As a conclusion for that parameter, the value update speed is not really a parameter that affects the convergence, it is more of an element that is limited by convergence speed. Indeed, if a programmer wants a cluster to be convergent with perfect consistent and thus every single state from any source to converge on all the nodes without hiding any temporary state, he must 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 3.2.1).

3.2.7 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 (thus allowing higher value 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 :



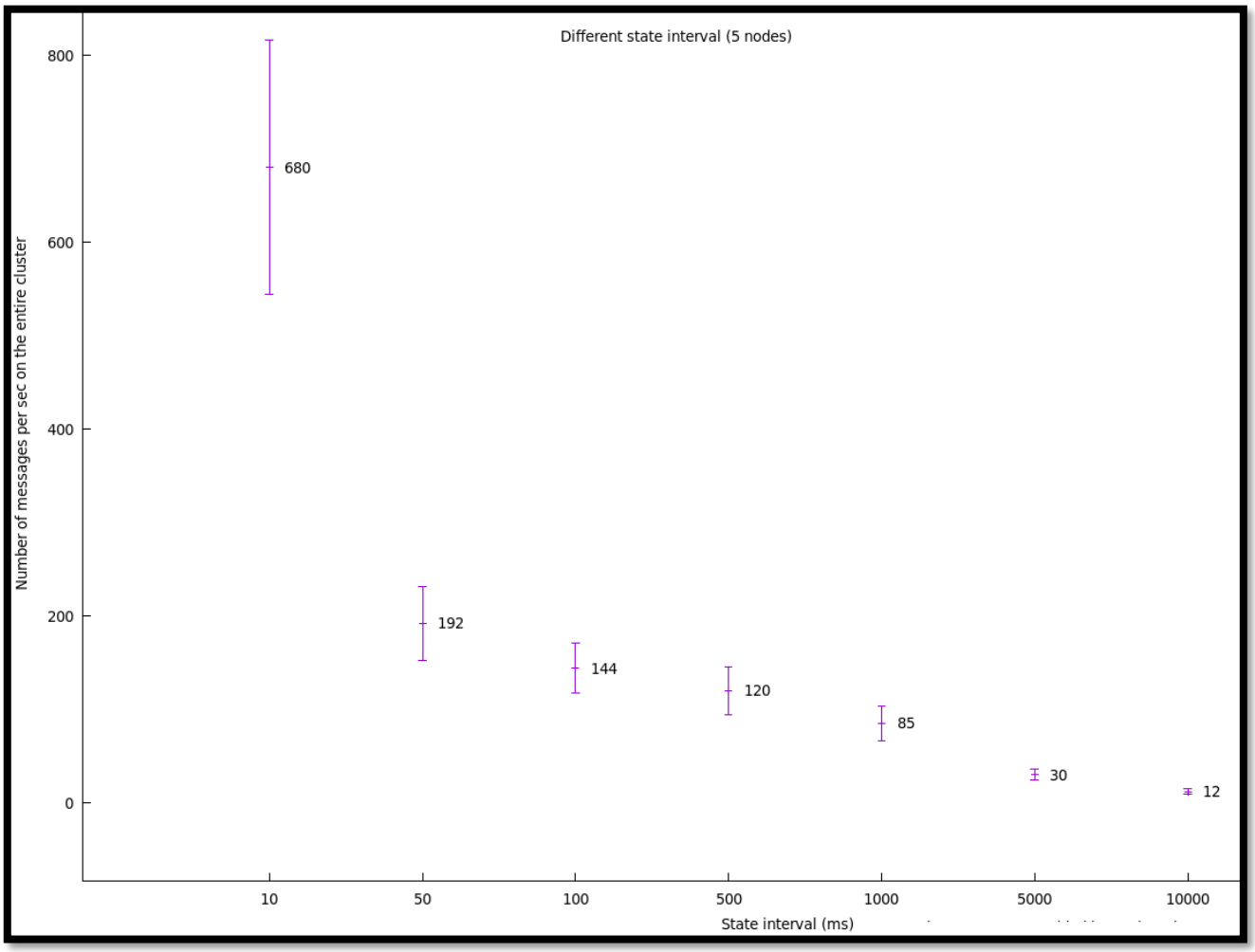
This was measured directly with the same script that was used for previous measurements (see section 2.3.1 for details). This result graph shows measurements for a cluster of 5 nodes each adding 10 elements (all at once) then waiting for convergence. The only modified parameter is the state_interval value that goes from 10 ms to 10000ms (which is the default initial value). The same measurements with nodes adding more elements (all at once) such as 100 elements or 1000 elements instead of 10 is now shown here since these measurements represent the exact same behaviour with extremely little variation due to the number of elements (parameter already measured in section 3.2.3).

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

Talking about this last point, let's see the impact on the number of messages per second on the cluster:



As expected, we clearly notice that the smaller the state_interval, the bigger the number of messages per second goes. 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 logical 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 3.2.1 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 (3.2.6) 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 to update the CRDT value every 0.5 second.

As for the other experimentations, more measurements are available within the results no the github page of this work.

4. Conclusion

4.1 Summary

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

4.2 Developed tools conclusion

Expliquer ce que je pense des outils que j'ai développé. Peut-être placer ici l'exemple où je measure, je modifie et je remesure pour voir que c'est mieux. Expliquer en quoi c'est bien mais également les défauts. Revenir sur le fait que globalement, l'API pour modifier le convergence time fonctionne et montrer exemple clair d'utilisation.

4.3 Results analysis conclusion

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

Dire que globalement Lasp a montré à travers les mesures de très bonnes propriétes et se montre assez peu impacté par les différents parametres testés. A la limite le seul parametre qui a l'air d'avoir un assez gros impat est le nombre de node mais c'est également la mesure malheureusement la moins fiable dans ce travail à cause du nombre de devices limités et du coup plus de nodes par device ce qui peut fausser les mesures.

4.4 Future work

Ameliorer l'outil de mesure pour qu'il soit plus paramétrable (type de CRDT, taille, etc…)
Permettre à l'outil de mesure d'être couplé avec l'outil d'adaptation pour automatiquement modfier le state_interval pr atteindre une convergence souhaitée.
Faire des mesures plus précises avec plus de matériel pour le nombre de nodes.
Faire des mesures qui comparent les performances de Lasp avec un système plus conventiel (non CRDT) en utilisant les mêmes machines.
Améliorer la démarche expérimentale pour faire des mesures plus précises et avec moins de variations (écart type). En profiter pour valider et expliquer la différence entre addition et removals.
…

4.5 Personal opinion

4.6 Methodology

Expliquer ce que je pense personnellement de Lasp. Est-ce que je l'utiliserais personnellement.
Expliquer que je trouve cela encore assez expérimental (pour le cas de Lasp) bien que prometteur car certaines grandes entreprises utilisent déjà du CRDT, etc…

Expliquer comment j’ai travaillé, chaque semaine, avec mes réunions hebdomadaires avec Peter Van Roy etc.

6. Bibliography

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

1. CAP theorem (trouver une bonne source)

2. CRDT

3. document : crdts_overview

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

5. Example CRDT addwins, document crdts-overviews

6. Projet SyncFree (leur site web)

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

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

9. SyncFree

10. LighKone

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

12. présentation de BET365 sur ORSWOT

TODO : Ajouter des references utiles et leurs sources