Table of contents

1. Introduction

This chapter presents the general content and context of this manuscript, describing what is Lasp, what are CRDTs and what are their innovative aspects. 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.

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.

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.

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 fist sight (since it does not involve recording the distributed variable state in a specific place such as a database), 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].

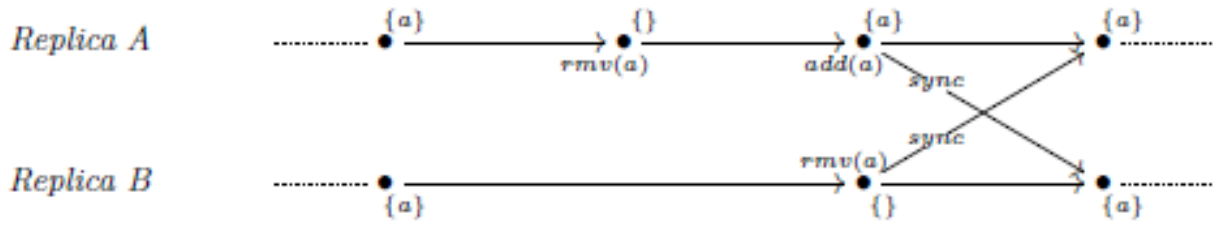
1.2.1 Principles

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.

No doubt the good features and properties described above together with the excellent CAP theorem compromise it allows are the reasons why CRDT usage is growing quickly[8] and has been adopted by some big compagnies as previously mentioned.

1.3 Lasp

Lasp is an experimental implementation of CRDTs developed by the Ecole Polytechnique de Louvain (EPL) research team and initiated in 2013 with the impetus of two European projects; 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 when this is not especially the case for the values we got from querying it 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 update 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.

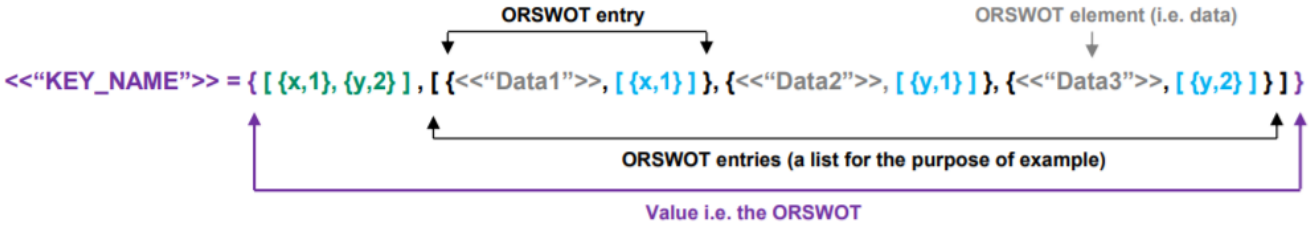
Thus, the ORSWOT is the general selected CRDT for this master thesis. As a little remark, the generic name ORSWOT comes from orset without tombstones due to the fact, as just explained, it addressed the tombstone issue from the generic orset. As a note, the rest of this document might use the name “awset” 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}] }] }

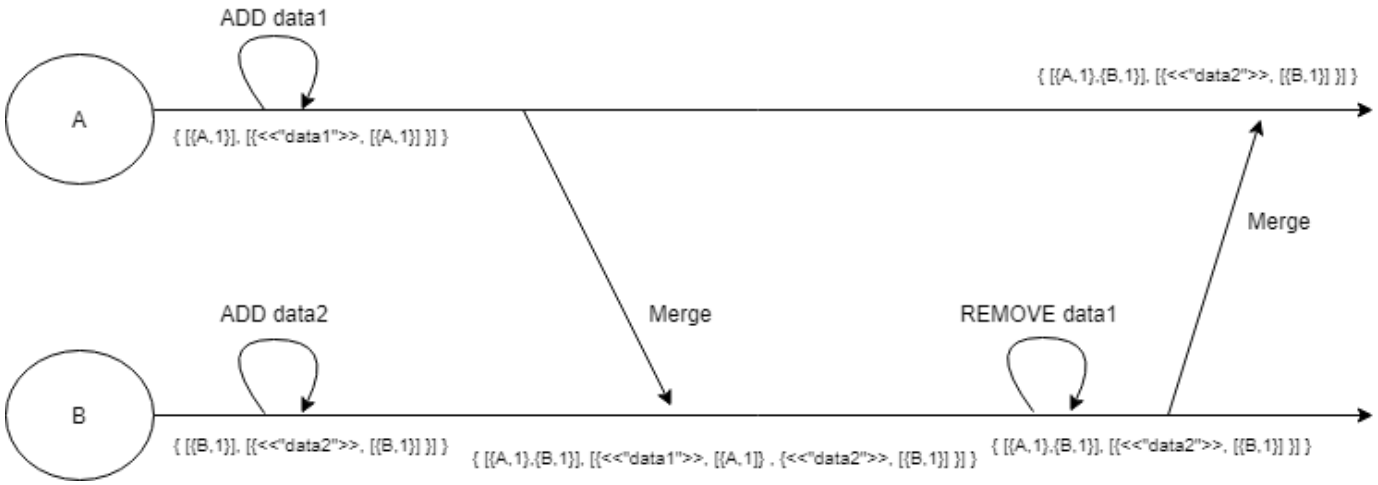
Seen:

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

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

1.4.1 Improved API

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

In this optic, a first goal was to improve the API in a particular direction related to the convergence principle described in section 1.2.1. Indeed, convergence is a fantastic feature allowing every node to eventually end up with a consistent state without requiring any heavy synchronization algorithm but how much time does it take for a specific cluster of nodes to converge? There was, up to now, no

directly available tool to easily know such information from an end developer perspective. This is an important issue for practical usage since an application would probably not work as intended on a cluster than converges in 10sec instead of 1sec for example and the developer had no tool to easily detect that. Therefore, a first objective was to develop a measurement tool that could be easily incorporated in Lasp to measure convergence time together with network utilisation.

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

1.4.2 Measurements

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

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

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

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

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

- CRDT operation
- CRDT value update speed

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

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

1.5 Summary and structure

This work will be based on Lasp implementation of CRDT, using the orswot (named awset in Lasp) as use case. The contributions to Lasp will be presented in the next chapter (chapter2) including the new developed tools and a few improvements that were proposed to enrich Lasp or make it a bit more user-friendly. The scripts used for this work will also be briefly presented. Following, in chapter 3, the measurement experimental work will be presented explaining the different scenarios and how they were measured. Chapter 4 presents the results and tries to analyse them while final chapter 5 concludes with some summary on observations, a personal opinion on Lasp and the general methodology followed during this work.

2. Contributions

Introduire le fait que cette section développe ce que j'ai codé et apporté à Lasp.

2.1 Measurement tools

2.1.1 Principle

2.1.2 API

2.1.3 Examples

2.2 Adaptation tools

2.2.1 Principle

2.2.2 API

2.2.3 Examples

2.3 Scripts

Expliquer que cette section va résumer les différents scripts.

2.3.1 Static measurement scripts

Expliquer ce que ça fait. Le fait que ce sont les premiers scripts que j'ai utilisé pour mes premières mesures, qu'ils sont assez basiques mais qu'ils permettent déjà de donner une rapide idée des performances.

2.3.2 Dynamic measurement scripts

Expliquer qu'il s'agit d'une version déjà un peu plus évoluée qui fait des mesures en continue.

2.3.3 Quality-of-life scripts

Expliquer qu'il y a tout une série de scripts qui permettent simplement de faciliter et accélérer certaines manipulations comme : recompiler Lasp en entier, créer un cluster de 5 nodes, mesurer l'utilisation mémoire des process, etc…

2.4 Lasp minor additions

2.4.1 Readme improvement

Expliquer que j'ai rencontré parfois quelques petits problèmes, asez simples à résoudre mais qui, de mon point de vue, pourraient être explicités sur le github officiel. Example : Préciser qu'il peut être

nécessaire de partager au préalable un fichier erlang.cookie. TODO : Ouvrir une pullrequest proposant d'améliorer le readme.

2.4.2 Memory leak

Expliquer le souci de mémoire avec la table ets et le fait que ça stockait une redondance inutile qui finissait par faire exploser le process. Expliquer l'impact que cela avait et la façon dont ça a été résolu.

3. Measures

Expliquer ce qui est mesuré (convergence time, memory usage, nombre de messages/sec).
Rappeler que j'ai décidé d'utiliser le awset (orswot) pour toutes mes mesures. Faire un petit rappel en une ou deux phrases sur ce qu'est Lasp, ce qu'est le Awset et ce qu'il permet.
Expliquer le mode opératoire (quels scripts ont été utilisés), combien d'itérations ont été réalisées, etc.
Expliquer que cette partie a pour but d'expliquer ce qui a été mesuré et comment. Il s'agit simplement de paragraphes expliquant comment j'ai mesuré telle ou telle chose. Les résultats seront dévoilés au chapitre suivant (graphes).

3.1 Parameters

Bref rappel des paramètres sur lesquels j'ai joué (nombre de nodes, nombre d'éléments dans le CRDT, distance entre les nodes, add ou remove, update speed, partition).

3.2 Initial measures

Expliquer qu'il s'agit là des mesures réalisées sur Lasp tel qu'il était fourni sur le github officiel avant d'avoir développé l'API pour modifier le convergence time.

3.2.1 Number of Nodes

3.2.2 Nodes distance

3.2.3 CRDT size

3.2.4 CRDT operation

3.2.5 Partition

3.2.6 Update speed

3.3 Adapted Lasp

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

4. Results and analysis

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

4.1 Results

4.1.1 Number of Nodes

Dévoiler le graphe pour initial_lasp 5,10,15,20 nodes.
Dévoiler le graphe pour final_lasp 5,10,15,20 nodes (c'est-à-dire la version avec 100ms de interval).
Discuter à propos des graphes. Essayer de comprendre ce qui cause cela.

4.1.2 Nodes distance

Meme principe que précédemment

4.1.3 CRDT size

Meme principe que précédemment

4.1.4 CRDT operation

Meme principe que précédemment

4.1.5 Partition

Même principe que précédemment.

4.1.6 Update speed

Même principe que précédemment.

4.2 Analysis

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

5. Conclusion

5.1 Summary

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

5.2 Analysis conclusion

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

5.3 Personal opinion and 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

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