**Divergence Visualization in Lasp program execution  
Reunion 4 (14/10/2020)**

Table des matières

[1. Questions 2](#_Toc53510584)

[1.a Delivery 2](#_Toc53510585)

[1.b Determinism Conditions 3](#_Toc53510586)

[2. ORSWOT 4](#_Toc53510587)

[2.a Vector clock 4](#_Toc53510588)

[2.b ORSWOT representation: 4](#_Toc53510589)

[2.c Add an element 5](#_Toc53510590)

[2.d Remove an element 6](#_Toc53510591)

[2.e Merge entre deux replicas 6](#_Toc53510592)

[2.f Conlusion 8](#_Toc53510593)

[3. Clustering with remote nodes 9](#_Toc53510594)

[3.a Remote node 9](#_Toc53510595)

[3.b Clustering remote nodes with local nodes: ERROR 10](#_Toc53510596)

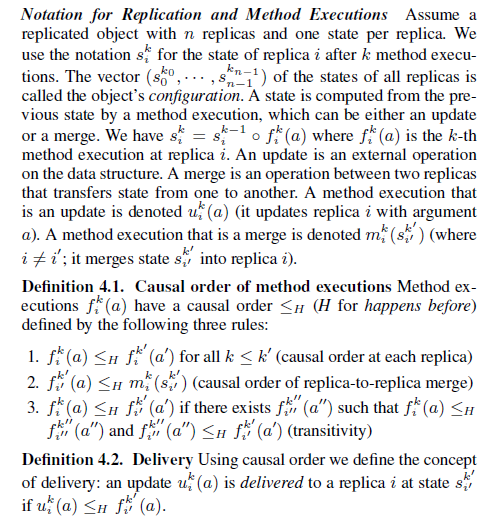
[4. TFE Roadmap 11](#_Toc53510597)

# Questions

Parmi les différentes questions abordées la semaine passée, deux restent encore obscures à mes yeux :

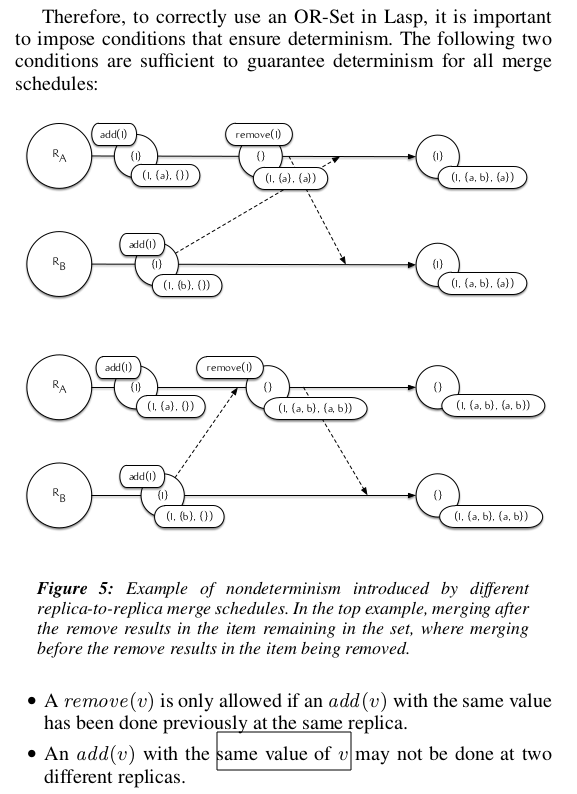
## 1.a Delivery

La première concerne la définition de « delivery » dans le document « Lasp, a language for Distributed, Coordination-Free programming » (ppdp 2015).  
En effet, j’ai visiblement du mal à bien interpréter la définition proposée.



## 1.b Determinism Conditions

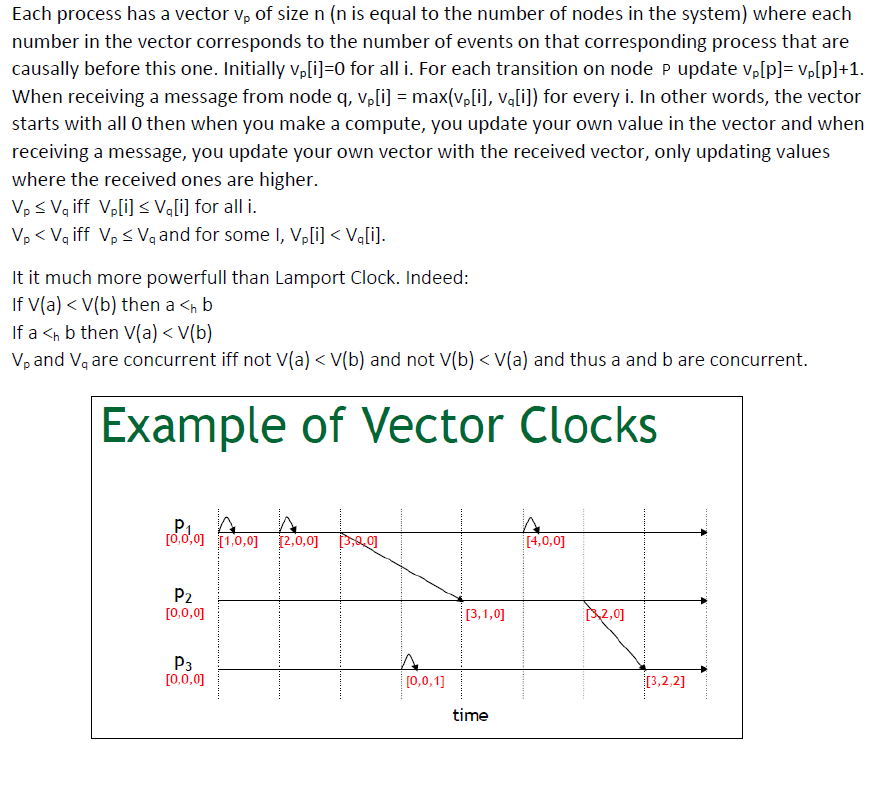
Ma deuxième question concerne les deux conditions à imposer pour éviter tout non-déterminisme quel que soit le scheduler des merges. C’est-à-dire que je comprends les deux conditions et je vois pourquoi cela règle le problème mais ces deux conditions me semblent très contraignantes en pratique pour le end-user (programmeur). Je serais curieux d’en débattre avec vous afin de mieux comprendre comment cela affecte le programmeur.  
J’ai remarqué que ces conditions s’appliquent aussi aux ORSWOT.



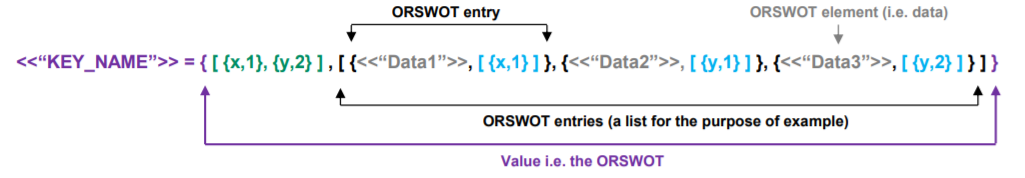
# ORSWOT

Je tiens à préciser qu’une grosse partie des informations qui vont suivre à propos des ORSWOT sont issues du document « Using Erlang, Riak and the ORSWOT CRDT at bet365 for Scalability and Performance » écrit par Michael Owen.

## 2.a Vector clock

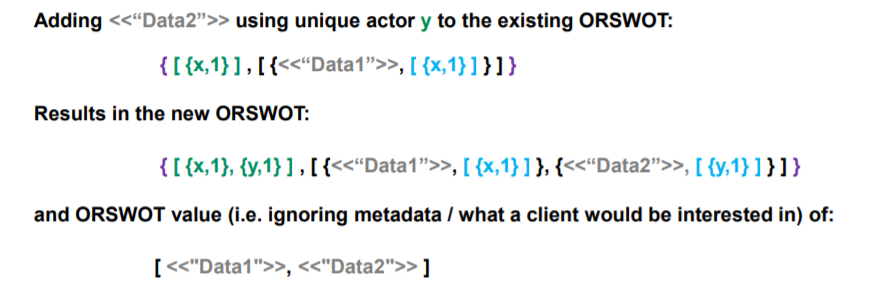
Commençons par un rappel des Vector Clock. Ceci est une brève définition avec un exemple tiré de mes notes de cours de LSINF2345 :

## 2.b ORSWOT representation:

La représentation du ORSWOT avec ses métadonnées est la suivante :  
Un version vector suivi des ORSWOT entries.  
  
**Le version vector** consiste en un set de tuples (tuples de taille2). Chacune ces paires consiste en un UniqueActorName (c’est-à-dire un identifiant unique identifiant un replica) et un counter. Ensuite, après le version vector viennent les entries.  
**ORSWOT entries** est un set contenant des paires. Chaque paire est constitué d’un élément (data telle que visible par le client) et d’un set de Dots. Ces Dots sont eux-même composé d’une paire UniqueActorName et counter.  
Cela sera plus simple avec une représentation visuelle :  
  
Ici, on a un ORSWOT avec en vert son version vector.  
Vient ensuite le set d’OSWOT entries. Chacun composé d’un élément ici appelé data et d’un Dot en bleu (ici chaque Dot est composé d’une seule paire mais on peut parfois en avoir plusieurs).

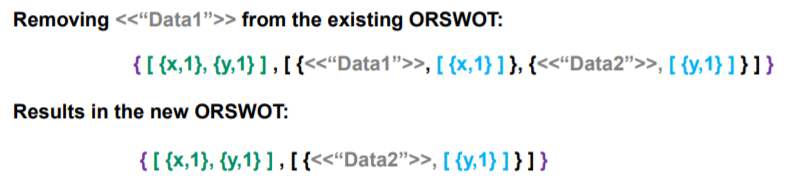
## 2.c Add an element

A noter qu’il s’agit de state-based CRDT. Un replica n’enverra donc jamais un message du type add elementX vers d’autres replicas. Au lieu de cela, si elle décide d’ajouter l’élementX, elle mettra à jour son state en modifiant le ORSWOT et par la suite ce state sera merge avec les autres replicas.  
Une approche simple pour comprendre le fonctionnement est donc de regarder localement quel est l’impact d’un add ou d’un remove puis de regarder comment fonctionnent les merges.

  
Lorsqu’une replica y ajoute un élément, son counter correspondant dans le version vector est augmenté de 1 ou sa paire est ajoutée et mise à 1 s’il n’était pas encore dans le version vector.  
Si l’élément n’était pas encore présent dans le ORSWOT, une nouvelle entry est créée, comportant l’élément et en Dots la paire (UniqueActorName, updated counter).   
On peut voir ci-dessous un exemple simple :

Ici, le replica y ajoute l’élément Data2. Vu qu’il n’y avait pas encore de paire associée à y dans le version vector, une paire {y,1} y est ajoutée. L’entry comportant l’élément Data2 et la paire {y,1} est elle aussi ajoutée.

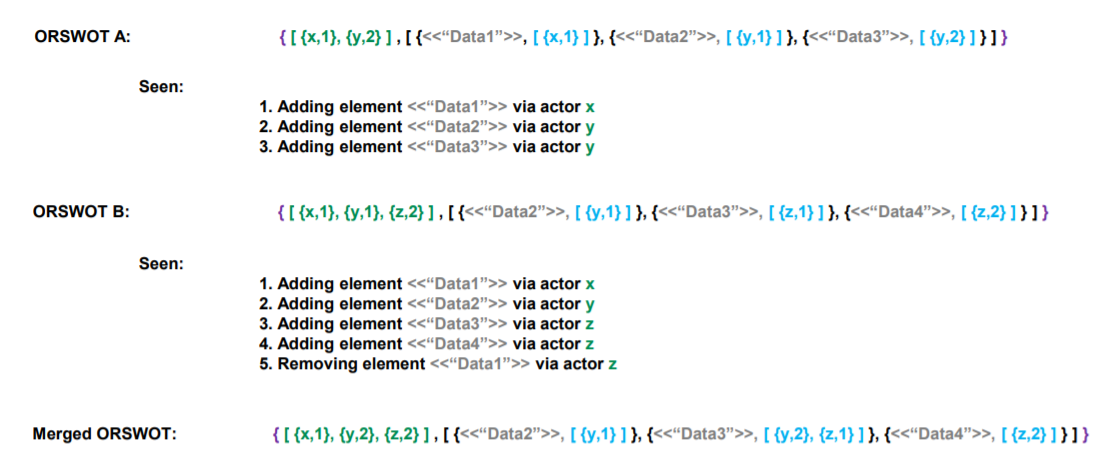
## 2.d Remove an element

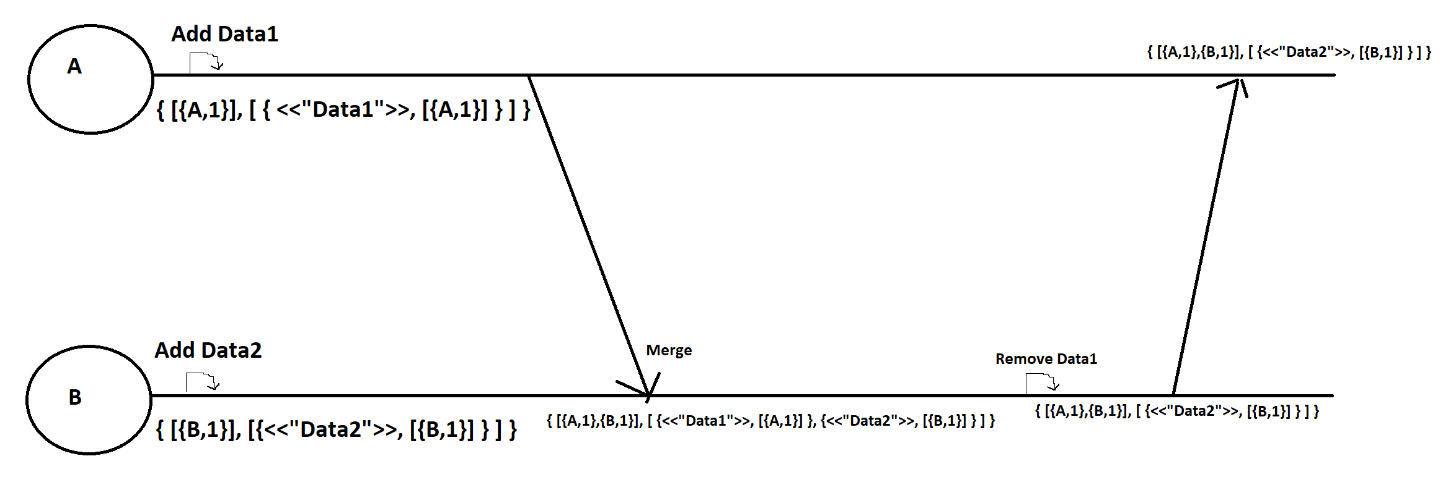
Lorsqu’une replica remove un élément, son counter (dans le version vector) n’est pas modifié, contrairement à l’opération add. L’entrée avec l’élément et son Dot est simplement supprimée et retirée du ORSWOT sans laisser de trace, contrairement au OR-set.

Dans ce petit exemple, un replica (peu importe lequel) décide de remove l’élément Data1. L’entry en question est simplement retirée sans laisser de tombstone.

## 2.e Merge entre deux replicas

Voyons, enfin, la façon dont deux replicas mergent leur state.  
  
**Version Clock :**  
Les deux version vector sont mergent ensemble à la façon des vector clocks. C’est-à-dire que pour chaque UniqueActor, seule la plus grande valeur est gardée.  
  
**Common elements (élément déjà présents dans les deux replicas avant le merge):**  
Les dots communs pour cet élément (même UniqueActorName et même valeur de counter) sont conservés.  
Pour chaque paire présente seulement dans la replica A, on garde la paire si le counter est supérieur au counter pour cet UniqueActorName dans le version vector de la replica B.  
Et inversement, pour chaque paire présente seulement dans la replica B, on garde la paire si le counter est supérieur au counter pour cet UniqueActorName dans la version vector de la replica A.  
Autrement dit, en réalité, on garde la paire UniqueActorName/Counter uniquement si c’est une information plus récente que ce qu’on a déjà.  
  
**Non-common elements (Eléments qui se trouvaient dans un seul des deux replicas) :**Pour les éléments seulement présents dans la replica A, on conserve les dots dont le counter est supérieur au counter pour cet UniqueActorName dans le version vector de la replica B.  
S’il n’y aucun dot qui correspond à ce critère pour cet élément, il est tout simplement supprimé du ORSWOT.  
Et pour les éléments seulement présents dans la replica B, on conserve les dots dont le counter est supérieur au counter pour cet UniqueActorName dans le version vector de la replica A.  
Et à nouveau, s’il n’y aucun dot qui correspond à ce critère pour cet élément, il est tout simplement supprimé du ORSWOT.  
En résumé, à nouveau, cela revient à garder uniquement les informations les plus récentes.

Voyons un exemple concret :

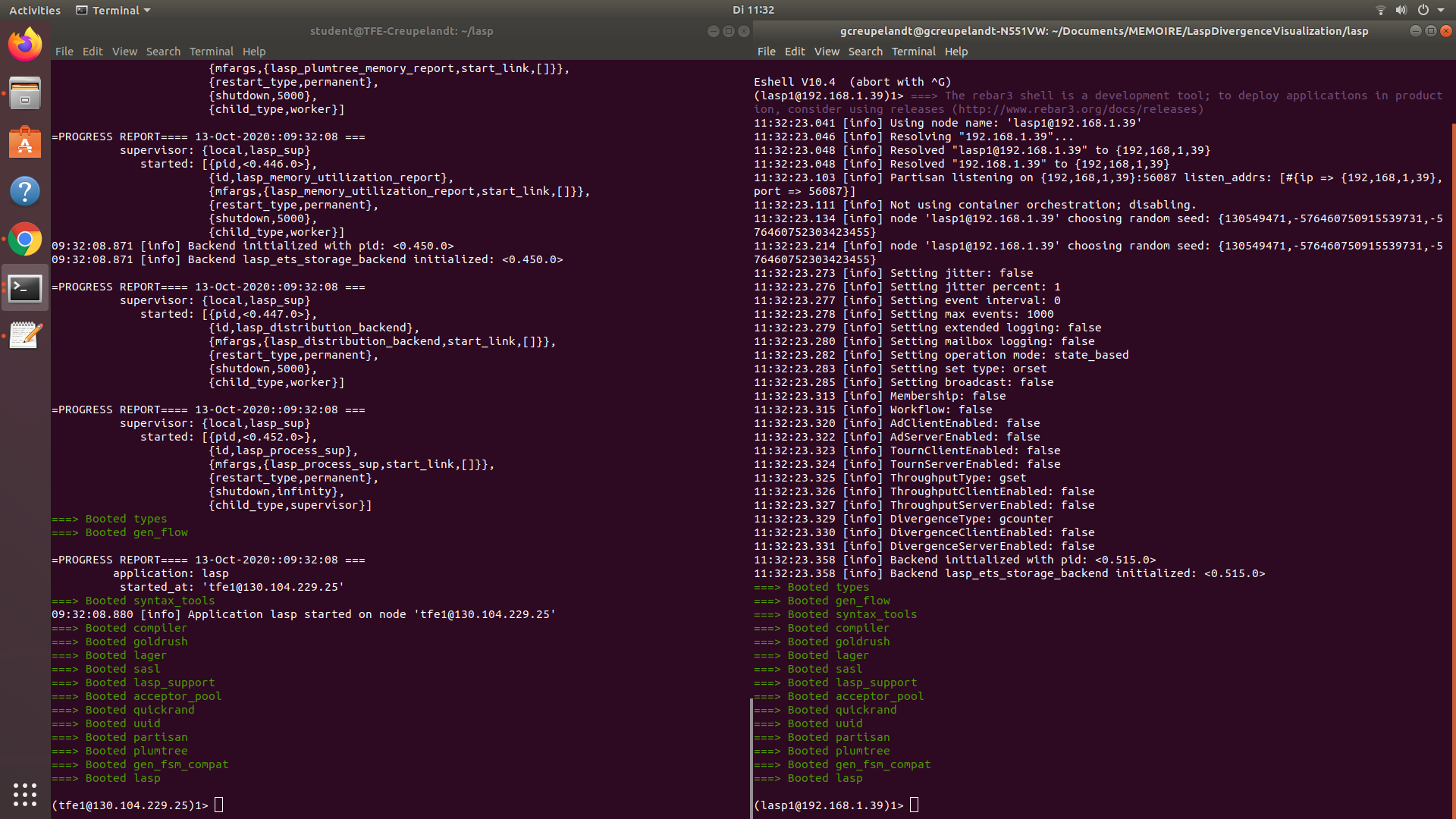
Ici, on a deux replicas A et B qui vont merger leur ORSWOT. A noter qu’ils disposent, avant le merge, d’ORSWOT différent car ils ont tous les deux vus des événements différents, on peut d’ailleurs facilement reconstruire leurs state avant le merge sur base des indications données sur l’image.  
Procédons ensuite au merge.  
**Premièrement, le version vector** est mis à jour en prenant pour chaque UniqueActorName, la valeur la plus haute.   
**Ensuite, regardons les éléments en communs**. On a Data2 qui est dans ORSWOT A et ORSWOT B. Pour cet élément, il n’y a que la paire {y,1} et cette dernière est commune. On conserve donc l’élément Data2 avec son dot.  
On a également Data3 commun entre les deux. Commençons dans le replica A où on a la paire {y,2}. C’est supérieur à la valeur du counter pour y dans le version vector de la replica B. On conserver donc l’élément Data3 avec la paire {y,2}. Ensuite dans la replica B, on a la paire {z,1}. A nouveau, c’est supérieur au counter pour z dans la replica A (car à 0).  
On a donc l’élément Data3 avec les deux paires {y,2} et {z,1}.  
**Enfin, regardons les éléments qui ne sont pas communs** aux deux replicas. On a Data1 qui est uniquement présent dans Replica A avec le Dot {x,1}. Dans la replica B, on a déjà la valeur 1 pour le counter associé à x dans le version vector. On ne garde donc pas cette pair {x,1} car la condition était d’être strictement supérieur. Vu qu’on a donc un Dot vide pour Data1, on le supprime simplement.  
Regardons finalement Data4, cet élément est uniquement présent dans la replica B avec le Dot {z,2}.  
Etant donné que le replica A n’avait pas de counter pour z, l’élément est gardé.  
On obtient donc au final :  
   
  
Here is an other little visual example I wrote:

## 2.f Conlusion

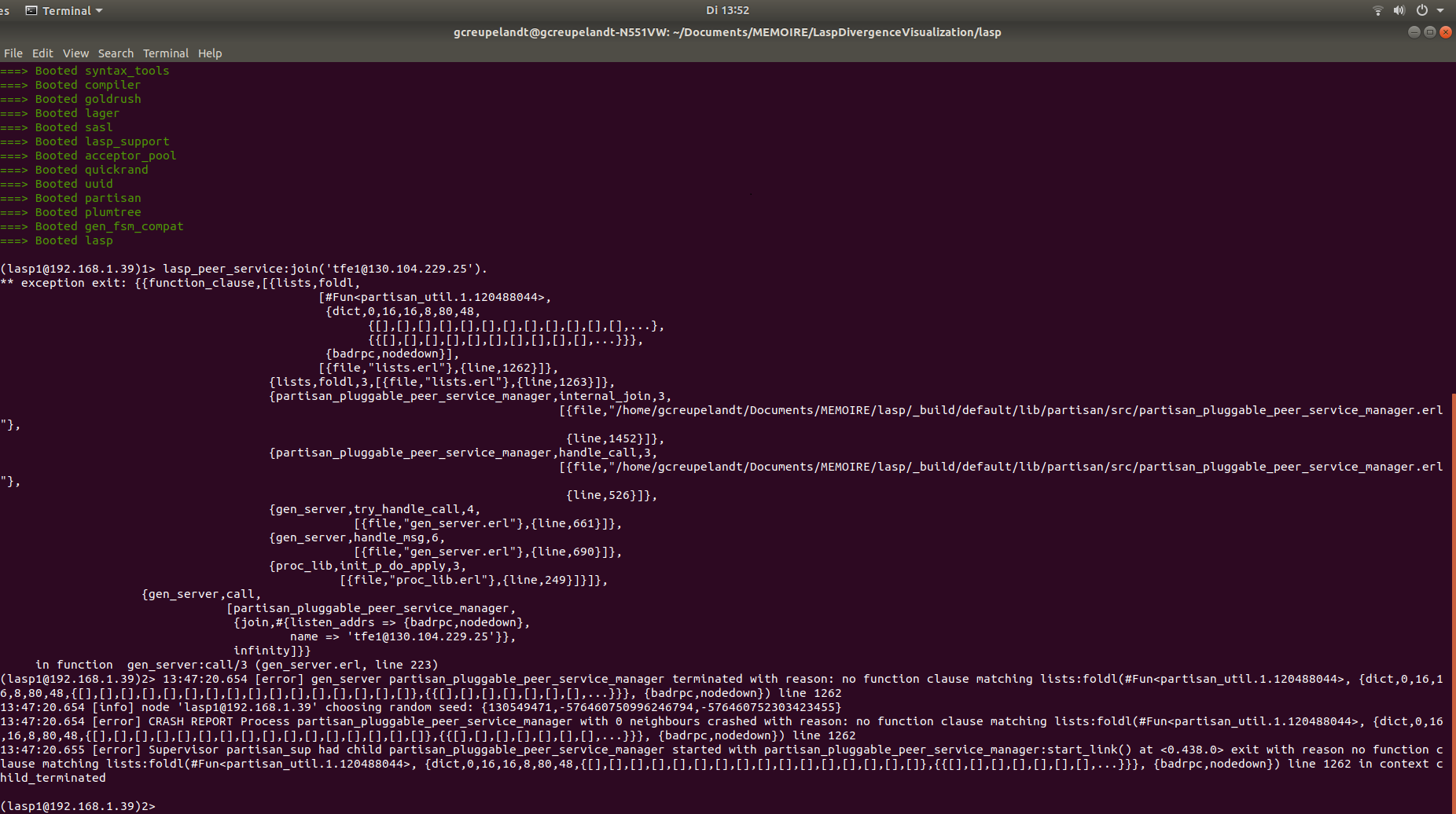
On a comportement et un principe très similaire au OR-set. Cela dit, l’utilisation des version vectors (adaptation de vector clock) permet de garder un nombre limité de counters (au maximum N counters pour un cluster de N nodes) et d’utiliser ceux-ci pour résoudre les merges de façon déterministe sans nécessiter de tumbstones (contrairement au OR-set on ne doit pas garder trace des éléments qui ont été remove).

# Clustering with remote nodes

## 3.a Remote node

J’ai accès à une VM sur des serveurs UCL sur laquelle j’ai les droits Sudo, j’ai donc pu y installer Lasp et ses quelques dépendances. J’arrive à lancer des nodes Lasp depuis ma machine ainsi que sur la VM (via SSH). Jusque-là, pas de souci. Vous pouvez voir, par exemple sur le screen ci-dessous, deux terminaux, un sur ma machine et l’autre représentant la VM. Chacun run un node avec leur adresse IP correspondante. J’arrive d’ailleurs à cluster ensemble des nodes tournant sur la VM.

## 3.b Clustering remote nodes with local nodes: ERROR

C’est lorsque j’essaye de rassembler ensemble des nodes locaux (sur ma machine) avec des nodes remotes (sur la VM), qu’un problème survient.   
En effet, il semblerait que ce soit Partisan qui ne parvient pas à trouver le node que je lui demande de join. Ci-dessous, vous pouvez voir, par exemple, l’erreur obtenue lorsque, depuis un node local, j’essaye de join un node sur la VM :

J’ai vérifié les adresses IP utilisées, je ne pense pas que le problème vienne de là car de toute façon si j’utilise une autre adresse IP (non localhost bien entendu), le node ne se lance tout simplement pas car crash au démarrage. J’ai également désactivé mon pare-feu mais cela n’a pas résolu le problème non-plus. J’ai essayé de trouver de l’information sur la page github ou sur le site de documentation de Lasp, testant notamment d’appeler lasp\_peer\_service :join() en specifiant de façon complète l’adresse ip, le port, etc…   
Mais je suis un peu à court d’idée et ne comprend malheureusement pas vraiment le message d’erreur (no function clause matching…), qui, je suppose, signifie simplement que Partisan n’arrive pas à communiquer avec le node spécifié.  
  
Si vous savez ce qui pourrait clocher, je serais heureux d’entendre vos idées car j’ai malheureusement du mal à trouver des informations pratiques sur le clustering des nodes.

# TFE Roadmap

J’ai appris lors d’une séance d’information sur les TFE qu’il était demandé aux étudiants de fournir à leur promoteur une roadmap détaillant le planning qu’ils comptent suivre et ce, 2 mois après le début de leur travail. Dans mon cas, étant donne que je dispose de deux fois moins de temps, j’ai supposé que produire une première roadmap 1 mois après avoir commencé (plutôt que 2mois) serait plus approprié. Voici donc ce que j’ai en tête en considérant 4 semaines par grosse étape :

**Mercredi 23 septembre – Mercredi 21 octobre :**1. Découverte du sujet.  
2. Prise en main des outils (Erlang, Rebar, Lasp).  
3. Lecture d’articles.  
4. Premiers petits prototypes de mesures de convergence.  
5. Clustering de nodes locaux avec nodes à distance.   
  
**Mercredi 21 octobre – Mercredi 18 novembre :**1. Développement d’un outil de mesure de la vitesse de convergence précis.  
2. Mesures réalisées pour un CRDT donné face à différents scénarios (crash et recovery de nodes, partition, update depuis différents nodes…).  
3. Mesures réalisées (avec les différents scénarios) pour plusieurs CRDT différents (encore à définir lesquels).  
4. Décision quant à la façon d’incorporer cet outil de mesure de façon efficace et utile dans Lasp\*.  
  
**Mercredi 18 novembre – Mercredi 16 décembre :**  
1. Suite et fin : incorporation de l’outil de mesure au sein du code de Lasp  
2. Preuve d’efficacité. Scénarios d’utilisation et démonstrations.  
3. Mise au propre des documents rédigés au cours des mois précédents.  
4. Imporant: Encodage du titre du mémoire et du jury pour le 29 novembre !

**Mercredi 16 décembre – Mercredi 6 janvier :**  
1. Suite et fin rédaction au propre du mémoire.  
2. Important : Remise du mémoire pour le 10 janvier !

\* Concernant la façon d’incorporer mon travail au code de Lasp, j’ai pour l’instant plusieurs idées mais il faudra voir, selon ce que j’ai produit, la façon la plus intéressante d’incorporer cela. Cela pourrait être, par exemple, une méthode qui, une fois appelée, lancerait l’outil de mesure et retournerait un set de valeurs (avec X valeurs représentant les vitesses de convergence des X nodes) et permettant donc de prendre des décisions sur base de ces valeurs. Cela aurait comme désavantage que l’appel à cette méthode demanderait immédiatement à procéder à des mesures pour ensuite retourner les valeurs et permette la prise de décision en fonction. Une probablement meilleure implémentation serait d’avoir l’outil de mesure lancé automatiquement en tâche de fond tous les X (1fois par minute par exemple) afin de rafraichir le set de valeur représentant les vitesses de convergence. Ainsi la méthode utilisée par le programmeur ne ferait que retourner le set de valeur actuel et permettre donc une prise de décision sans délais.