

Divergence Visualization in Lasp program execution

Reunion 4 (14/10/2020)

Table des matières

1. Questions.....	2
1.a Delivery.....	2
1.b Determinism Conditions.....	3
2. ORSWOT	4
2.a Vector clock	4
2.b ORSWOT representation:	4
2.c Add an element	5
2.d Remove an element	6
2.e Merge entre deux replicas	6
2.f Conclusion.....	8
3. Clustering with remote nodes	9
3.a Remote node	9
3.b Clustering remote nodes with local nodes: ERROR.....	10
4. TFE Roadmap.....	11

1. 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.

Notation for Replication and Method Executions Assume a replicated object with n replicas and one state per replica. We use the notation s_i^k for the state of replica i after k method executions. The vector $(s_0^{k_0}, \dots, s_{n-1}^{k_{n-1}})$ of the states of all replicas is called the object's *configuration*. A state is computed from the previous state by a method execution, which can be either an update or a merge. We have $s_i^k = s_i^{k-1} \circ f_i^k(a)$ where $f_i^k(a)$ is the k -th method execution at replica i . An update is an external operation on the data structure. A merge is an operation between two replicas that transfers state from one to another. A method execution that is an update is denoted $u_i^k(a)$ (it updates replica i with argument a). A method execution that is a merge is denoted $m_i^k(s_{i'}^{k'})$ (where $i \neq i'$; it merges state $s_{i'}^{k'}$ into replica i).

Definition 4.1. Causal order of method executions Method executions $f_i^k(a)$ have a causal order \leq_H (H for *happens before*) defined by the following three rules:

1. $f_i^k(a) \leq_H f_i^{k'}(a')$ for all $k \leq k'$ (causal order at each replica)
2. $f_{i'}^{k'}(a) \leq_H m_i^k(s_{i'}^{k'})$ (causal order of replica-to-replica merge)
3. $f_i^k(a) \leq_H f_{i'}^{k'}(a')$ if there exists $f_{i''}^{k''}(a'')$ such that $f_i^k(a) \leq_H f_{i''}^{k''}(a'')$ and $f_{i''}^{k''}(a'') \leq_H f_{i'}^{k'}(a')$ (transitivity)

Definition 4.2. Delivery Using causal order we define the concept of delivery: an update $u_i^k(a)$ is *delivered* to a replica i at state $s_{i'}^{k'}$ if $u_i^k(a) \leq_H f_{i'}^{k'}(a)$.

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.

Therefore, to correctly use an OR-Set in Lasp, it is important to impose conditions that ensure determinism. The following two conditions are sufficient to guarantee determinism for all merge schedules:

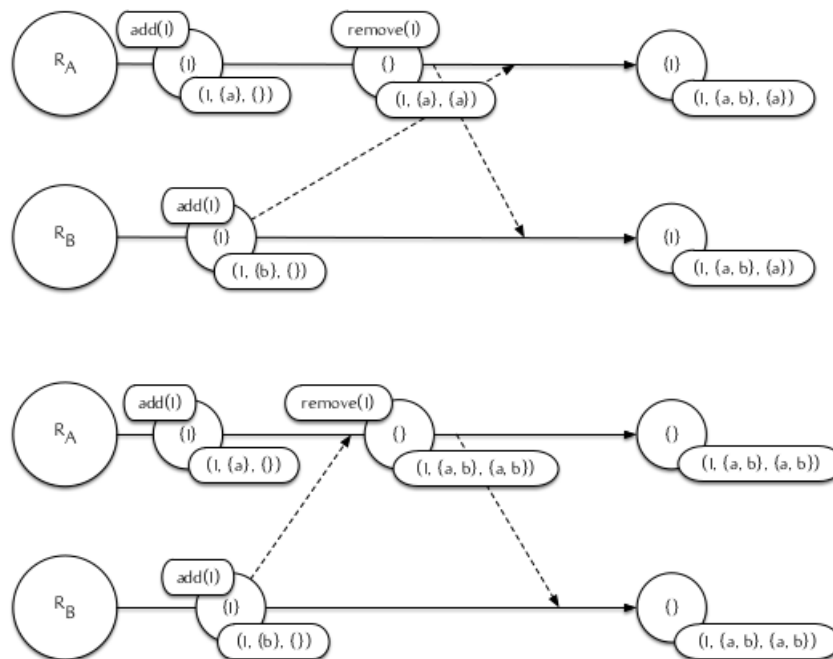


Figure 5: Example of nondeterminism introduced by different replica-to-replica merge schedules. In the top example, merging after the remove results in the item remaining in the set, where merging before the remove results in the item being removed.

- A $\text{remove}(v)$ is only allowed if an $\text{add}(v)$ with the same value has been done previously at the same replica.
- An $\text{add}(v)$ with the same value of v may not be done at two different replicas.

2. 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 :

Each process has a vector v_p of size n (n is equal to the number of nodes in the system) where each number in the vector corresponds to the number of events on that corresponding process that are causally before this one. Initially $v_p[i]=0$ for all i . For each transition on node P update $v_p[p]=v_p[p]+1$. When receiving a message from node q , $v_p[i] = \max(v_p[i], v_q[i])$ for every i . In other words, the vector starts with all 0 then when you make a compute, you update your own value in the vector and when receiving a message, you update your own vector with the received vector, only updating values where the received ones are higher.

$V_p \leq V_q$ iff $V_p[i] \leq V_q[i]$ for all i .

$V_p < V_q$ iff $V_p \leq V_q$ and for some i , $V_p[i] < V_q[i]$.

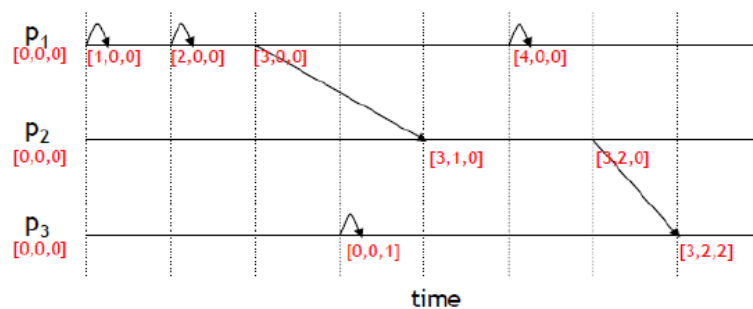
It is much more powerful than Lamport Clock. Indeed:

If $V(a) < V(b)$ then $a <_h b$

If $a <_h b$ then $V(a) < V(b)$

V_p and V_q are concurrent iff not $V(a) < V(b)$ and not $V(b) < V(a)$ and thus a and b are concurrent.

Example of Vector Clocks



2.b ORSWOT representation:

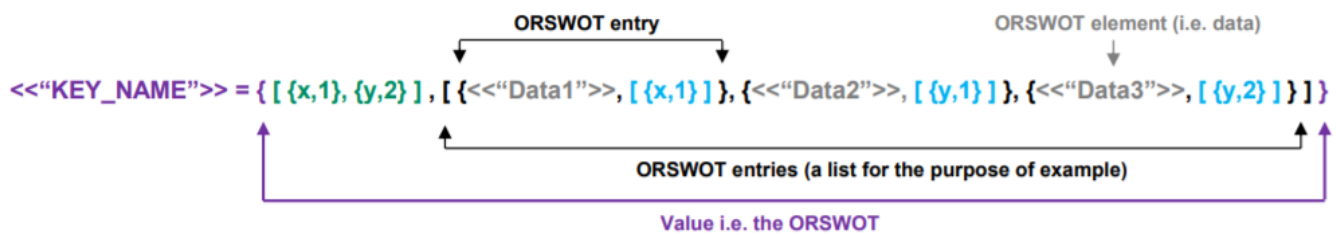
La représentation du ORSWOT avec ses métadonnées est la suivante :

Un version vector suivi des ORSWOT entries.

Le version vector consiste en un set de tuples (tuples de taille 2). Chacune de 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 constituée 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'ORSWOT 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'élémentX, 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 :

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'>>]`

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.

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

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

Results in the new ORSWOT:

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

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 a 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 a 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 :

ORSWOT A: $\{ [\{x,1\}, \{y,2\}], [\{ \ll "Data1" \gg, [\{x,1\}] \}, \{ \ll "Data2" \gg, [\{y,1\}] \}, \{ \ll "Data3" \gg, [\{y,2\}] \}] \}$

Seen:

1. Adding element $\ll "Data1" \gg$ via actor x
2. Adding element $\ll "Data2" \gg$ via actor y
3. Adding element $\ll "Data3" \gg$ via actor y

ORSWOT B: $\{ [\{x,1\}, \{y,1\}, \{z,2\}], [\{ \ll "Data2" \gg, [\{y,1\}] \}, \{ \ll "Data3" \gg, [\{z,1\}] \}, \{ \ll "Data4" \gg, [\{z,2\}] \}] \}$

Seen:

1. Adding element $\ll "Data1" \gg$ via actor x
2. Adding element $\ll "Data2" \gg$ via actor y
3. Adding element $\ll "Data3" \gg$ via actor z
4. Adding element $\ll "Data4" \gg$ via actor z
5. Removing element $\ll "Data1" \gg$ via actor z

Merged ORSWOT: $\{ [\{x,1\}, \{y,2\}, \{z,2\}], [\{ \ll "Data2" \gg, [\{y,1\}] \}, \{ \ll "Data3" \gg, [\{y,2\}, \{z,1\}] \}, \{ \ll "Data4" \gg, [\{z,2\}] \}] \}$

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.

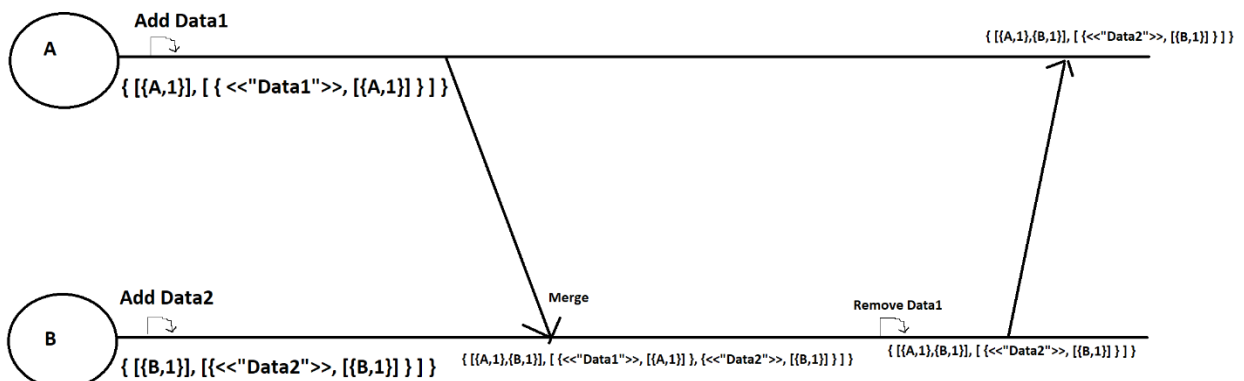
Premièrement, le version vector est mis à jour en prenant pour chaque UniqueActorName, la valeur la plus haute.

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 conserve 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 :

$\{ [\{x,1\}, \{y,2\}, \{z,2\}] , [\{<< \text{Data2} >>, [\{y,1\}] \}, \{<< \text{Data3} >>, [\{y,2\}, \{z,1\}] \}, \{<< \text{Data4} >>, [\{z,2\}] \}] \}$

Here is an other little visual example I wrote:

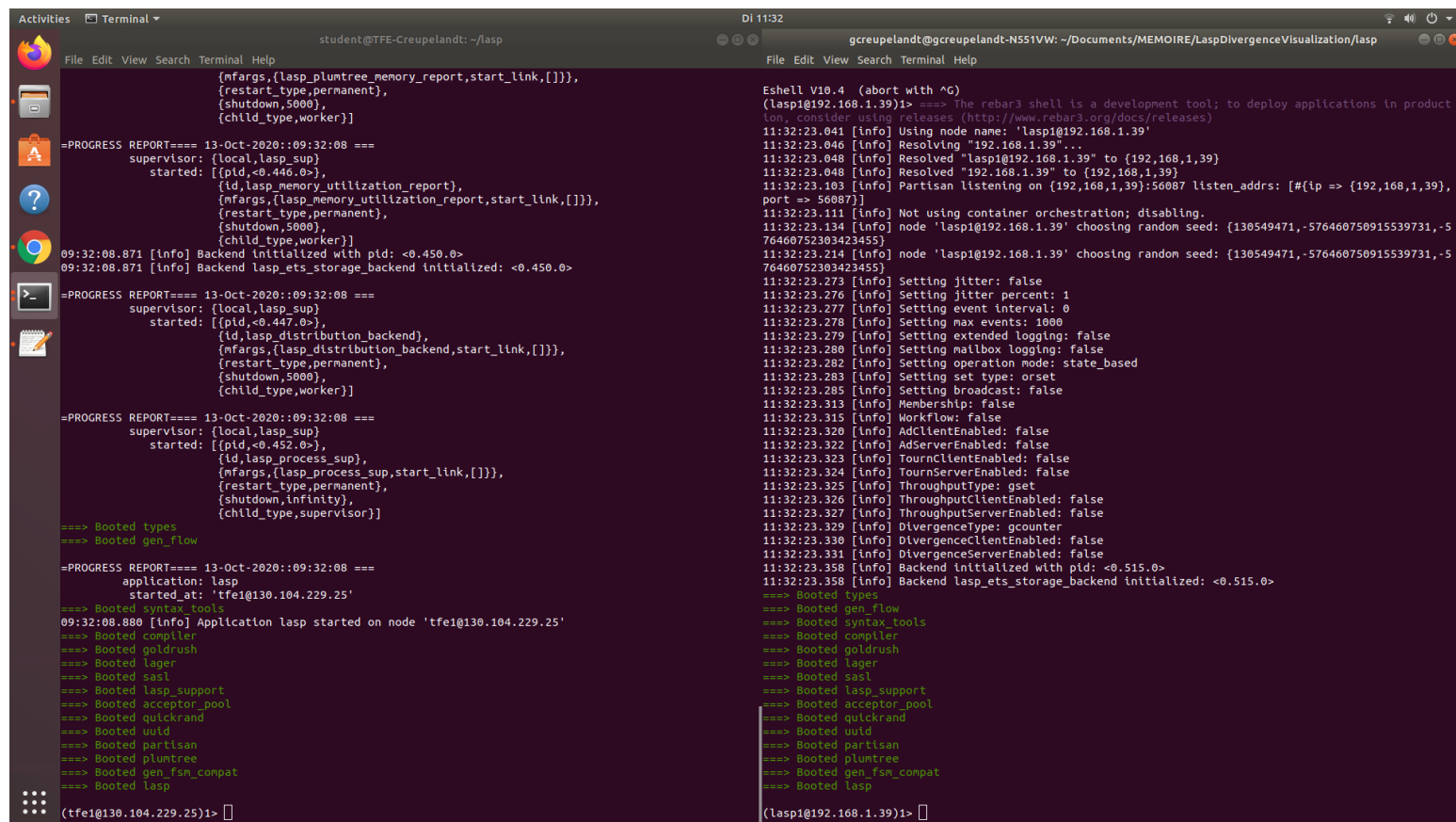
2.f Conclusion

On a un 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 tombstones (contrairement au OR-set on ne doit pas garder trace des éléments qui ont été remove).

3. 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.



```
student@TFE-Creupelandt: ~/lasp
{nfargs,{lasp_plumtree_memory_report,start_link,[]}},
{restart_type,permanent},
{shutdown,5000},
{chld_type,worker}}

=PROGRESS REPORT=== 13-Oct-2020:09:32:08 ===
supervisor: {local,lasp_sup}
started: [{pid,<0.446.0>},
          {id,lasp_memory_utilization_report},
          {nfargs,{lasp_memory_utilization_report,start_link,[]}},
          {restart_type,permanent},
          {shutdown,5000},
          {chld_type,worker}}]

09:32:08.871 [info] Backend initialized with pid: <0.450.0>
09:32:08.871 [info] Backend lasp_ets_storage_backend initialized: <0.450.0>

=PROGRESS REPORT=== 13-Oct-2020:09:32:08 ===
supervisor: {local,lasp_sup}
started: [{pid,<0.447.0>},
          {id,lasp_distribution_backend},
          {nfargs,{lasp_distribution_backend,start_link,[]}},
          {restart_type,permanent},
          {shutdown,5000},
          {chld_type,worker}}]

=PROGRESS REPORT=== 13-Oct-2020:09:32:08 ===
supervisor: {local,lasp_sup}
started: [{pid,<0.452.0>},
          {id,lasp_process_sup},
          {nfargs,{lasp_process_sup,start_link,[]}},
          {restart_type,permanent},
          {shutdown,infinity},
          {chld_type,supervisor}}]

==== Booted types
==== Booted gen_flow

=PROGRESS REPORT=== 13-Oct-2020:09:32:08 ===
application: lasp
started_at: 'tfe1@130.104.229.25'

09:32:08.880 [info] Application lasp started on node 'tfe1@130.104.229.25'

==== Booted syntax_tools
==== Booted compiler
==== Booted goldrush
==== Booted lager
==== Booted sasl
==== Booted lasp_support
==== Booted acceptor_pool
==== Booted quickrand
==== Booted uuid
==== Booted partisan
==== Booted plumtree
==== Booted gen_fsm_compat
==== Booted lasp

(tfe1@130.104.229.25)1>

gcreupelandt@gcreupelandt-NS51VW: ~/Documents/MEMOIRE/LaspDivergenceVisualization/lasp
Eshell V10.4 (abort with ^G)
(lasp1@192.168.1.39)1> ==== The rebar3 shell is a development tool; to deploy applications in product
ton, consider using releases (http://www.rebar3.org/docs/releases)
11:32:23.041 [info] Using node name: 'lasp1@192.168.1.39'
11:32:23.046 [info] Resolving "192.168.1.39" to {192,168,1,39}
11:32:23.048 [info] Resolved "lasp1@192.168.1.39" to {192,168,1,39}
11:32:23.048 [info] Resolved "192.168.1.39" to {192,168,1,39}
11:32:23.103 [info] Partisan listening on {192,168,1,39}:50007 listen_addrs: [{(ip => {192,168,1,39},
port => 50007)}]
11:32:23.111 [info] Not using container orchestration; disabling.
11:32:23.134 [info] node 'lasp1@192.168.1.39' choosing random seed: {130549471,-576460750915539731,-5
76460752303423455}
11:32:23.214 [info] node 'lasp1@192.168.1.39' choosing random seed: {130549471,-576460750915539731,-5
76460752303423455}
11:32:23.279 [info] Setting jitter: false
11:32:23.276 [info] Setting jitter percent: 1
11:32:23.277 [info] Setting event interval: 0
11:32:23.278 [info] Setting max events: 1000
11:32:23.279 [info] Setting extended logging: false
11:32:23.280 [info] Setting mailbox logging: false
11:32:23.282 [info] Setting operation mode: state_based
11:32:23.283 [info] Setting set type: orset
11:32:23.285 [info] Setting broadcast: false
11:32:23.313 [info] Membership: false
11:32:23.315 [info] Workflow: false
11:32:23.320 [info] AdClientEnabled: false
11:32:23.322 [info] AdServerEnabled: false
11:32:23.323 [info] TournClientEnabled: false
11:32:23.324 [info] TournServerEnabled: false
11:32:23.325 [info] ThroughputType: gset
11:32:23.326 [info] ThroughputClientEnabled: false
11:32:23.327 [info] ThroughputServerEnabled: false
11:32:23.329 [info] DivergenceType: gcounter
11:32:23.330 [info] DivergenceClientEnabled: false
11:32:23.331 [info] DivergenceServerEnabled: false
11:32:23.350 [info] Backend initialized with pid: <0.515.0>
11:32:23.350 [info] Backend lasp_ets_storage_backend initialized: <0.515.0>

==== Booted types
==== Booted gen_flow
==== Booted syntax_tools
==== Booted compiler
==== Booted goldrush
==== Booted lager
==== Booted sasl
==== Booted lasp_support
==== Booted acceptor_pool
==== Booted quickrand
==== Booted uuid
==== Booted partisan
==== Booted plumtree
==== Booted gen_fsm_compat
==== Booted lasp

(lasp1@192.168.1.39)1>
```

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 :

```
Terminal
gcreupelandt@gcreupelandt-N551VW: ~/Documents/MEMOIRE/LaspDivergenceVisualization/lasp

==> Booted syntax_tools
==> Booted compiler
==> Booted goldrush
==> Booted lager
==> Booted sasl
==> Booted lasp_support
==> Booted acceptor_pool
==> Booted quickrand
==> Booted uuid
==> Booted partisan
==> Booted plumtree
==> Booted gen_fsm_compat
==> Booted lasp

(lasp1@192.168.1.39)1> lasp_peer_service:join('tfe1@130.104.229.25').
** exception exit: {{function_clause,{{lists,foldl,
    [#Fun<partisan_util.1.120488044>,
     {dict,0,16,16,8,80,48,
      {{[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]},
       {{[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]},
       {badrpc,nodedown}}},
     [{file,"lists.erl"},{line,1262}]}],
    {lists,foldl,3,[{file,"lists.erl"},{line,1263}]},
    {partisan_pluggable_peer_service_manager,internal_join,3,
     [{file,"/home/gcreupelandt/Documents/MEMOIRE/lasp/_build/default/lib/partisan/src/partisan_pluggable_peer_service_manager.erl",
      {line,1452}}],
     {partisan_pluggable_peer_service_manager,handle_call,3,
      [{file,"/home/gcreupelandt/Documents/MEMOIRE/lasp/_build/default/lib/partisan/src/partisan_pluggable_peer_service_manager.erl",
       {line,526}}],
     {gen_server,try_handle_call,4,
      [{file,"gen_server.erl"},{line,661}]},
     {gen_server,handle_msg,6,
      [{file,"gen_server.erl"},{line,690}]},
     {proc_lib,init_p_do_apply,3,
      [{file,"proc_lib.erl"},{line,249}]}]}},
  {gen_server,call,
   [partisan_pluggable_peer_service_manager,
    {join,#{listen_addr => {badrpc,nodedown},
      name => 'tfe1@130.104.229.25'}},
    infinity]}}
    in function gen_server:call/3 (gen_server.erl, line 223)
(lasp1@192.168.1.39)2> 13:47:20.654 [error] gen_server partisan_pluggable_peer_service_manager terminated with reason: no function clause matching lists:foldl(#Fun<partisan_util.1.120488044>, {dict,0,16,16,8,80,48,{{[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]},{{[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]},{badrpc,nodedown}}}) line 1262
13:47:20.654 [info] node 'lasp1@192.168.1.39' choosing random seed: {130549471,-576460750996246794,-576460752303423455}
13:47:20.654 [error] CRASH REPORT Process partisan_pluggable_peer_service_manager with 0 neighbours crashed with reason: no function clause matching lists:foldl(#Fun<partisan_util.1.120488044>, {dict,0,16,16,8,80,48,{{[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]},{{[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]},{badrpc,nodedown}}}) line 1262
13:47:20.655 [error] Supervisor partisan_sup had child partisan_pluggable_peer_service_manager started with partisan_pluggable_peer_service_manager:start_link() at <0.438.0> exit with reason no function clause matching lists:foldl(#Fun<partisan_util.1.120488044>, {dict,0,16,16,8,80,48,{{[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]},{{[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]},{badrpc,nodedown}}}) line 1262 in context c
hild_terminated
(lasp1@192.168.1.39)2>
```

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 spécifiant 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.

4. 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 donné que je dispose de deux fois moins de temps, j'ai supposé que produire une première roadmap 1 mois après avoir commencé (plutôt que 2 mois) 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. Important: 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 permettre 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 (1 fois par minute par exemple) afin de rafraîchir 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.