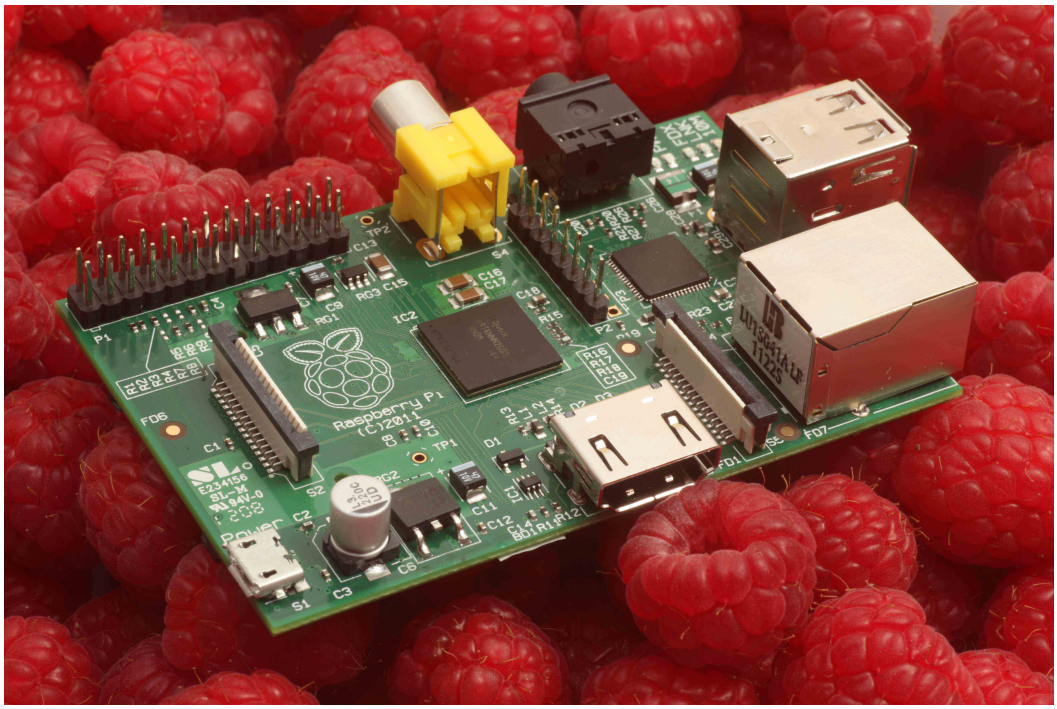
UNIVERSITÉ DE NANTES

**Rapport de projet**
UFR Sciences et Techniques de Nantes
M1 ALMA

# Raspbuddies

Maël BARBIN - Julien BIZEUL - Stanislas KOBAK - Florian FAGNIEZ




Nantes, le 13 mai 2013

# Table des matières

# Introduction

Le cloud computing est une nouvelle technologie destinée à être une évolution majeure du web social où des clients (utilisateurs lambda ou organisations) peuvent mettre en ligne d'énormes quantités de données qui étaient auparavant stockées localement. Le déploiement de données et d'applications sur un Cloud est centralisé de façon à pouvoir répliquer les données si l'un des serveurs sur lequel sont stockées les données venait à tomber en panne.

Au cours des dernières années, un nouveau défi est apparu, consistant à fournir des structures de données évolutives qui pourraient être utilisées pour créer des applications sociales.

Des structures de données telles que les CRDT (Commutative Replicated Data Type) ont été développées dont le type est capable d'assurer la cohérence éventuelle sur les infrastructures sociales décentralisées.

Notre travail consista à expérimenter le comportement de ce type sur un dispositif expérimental construit à l'aide d'un cluster et de 48 Raspberry Pi. Plus précisément, notre travail consista à évaluer les performances de livraison causale probabiliste et des opérations de recovery d'un CRDT en cas d'erreur de livraison de causalité.

# 1 QVV

Les besoins décrits en introduction nécessitent de nous assurer que la causalité, c'est à dire le fait que certains messages doivent impérativement être reçus avant d'autres, soit respectée.

Il existe plusieurs types d'évènements : les évènements causaux et concurrents. Un évènement est causal par rapport à un autre si leur exécution doit respecter un certain ordre (Pour deux évènements a et b, si a doit être exécuté avant b, alors l'exécution de b avant a est invalide). En revanche, deux évènements sont concurrents si l'ordre dans lequel ils sont exécutés ne compte pas (Pour deux évènements a et b, a peut être exécuté avant b et vice versa).

On utilise pour respecter cela des horloges logiques. Ces horloges permettent de savoir si un message arrive avant un autre.

Parmi ces horloges, nous avons vu l'horloge de Lamport qui fonctionne comme suit :
Chaque processus envoyant des messages possède une entrée dans un vecteur(représentant les horloges de chaque processus). Lors d'un évènement, le processus incrémente son entrée dans le vecteur, ce qui permet d'assurer la causalité en toutes circonstances.

Le problème majeur de cette solution est qu'elle n'est pas adaptée à un nombre élevé de processus. En effet, la taille du vecteur augmente en même temps que le nombre de processus. Hors, le processus doit envoyer, en plus de son message, l'état de son vecteur. On comprend donc que cette solution ne passe pas à l'echelle car cela augmente grandement la taille des message et les temps de traitement du vecteur pour chaque processus.

Nous avons donc vu plusieurs solutions pour parer à ce problème. La première est celle des *plausible clock*. Elle consiste juste en une réduction du vecteur d'horloges, plusieurs processus partageant la même entrée. Cette solution est probabiliste, et donc sujette aux erreurs.

Afin de réduire les probabilités d'erreurs, nous avons vu une deuxième solution, qui consiste à attribuer plusieurs entrées pour un même processus. Cette solution est celle que nous devions tester lors de cette initiation à la recherche, et s'appelle *QVV*(pour *Quasi Version Vector*). Plusieurs paramètres peuvent varier pour améliorer l'efficacité des QVV :
- La taille du vecteur d'horloges
- Le nombre d'entrées à incrémenter pour un processus dans le vecteur

Cette solution a été testée sur simulateur (donc avec un contrôle sur les différents paramètres tels que la latence, ...). Nous devions donc la tester dans des conditions réelles, sur un cluster de *raspberry pi*.

# 2 Un nouveau langage : BUD

Pour implémenter le protocole causal, nous avons du apprendre un nouveau langage : BUD (Bloom Under Development) qui possédait les outils étant en corrélation avec le contexte de recherche dans lequel notre projet s'inscrivait. Bud est un DSL (Domain Specific language) utilisant **Ruby**. Ce langage déclaratif à été mis en place par Neil CONWAY et d'autres doctorants de l'université de Berkeley. On retrouve chez BUD les caractéristiques suivantes :

**Un langage de programmation déclarative :** Un programme Bud est constitué d'un ensemble de règles ou bloc de règles, de la forme *left op right*. Quand la partie droite des règles est vraie, Bud applique la partie gauche de la règle ce qui permet à l'application de passer dans un nouvel état.

**Temps logique :** L'évaluation des règles ou blocs de règles s'effectuent à chaque unité de temps appelée tick. Le changement de tick s'opère lorsque les règles ne peuvent plus être évaluées. Par conséquent, une règle ou un bloc de règles peuvent être évalués plusieurs fois lors d'un même tick.

**Structures de données :** Les collections Bud s'inspirent de modèles comme le MapReduce et les stores (système de Clé/Valeur). Les structures standards de Bud sont donc des collections désordonnées. Ces structures de données permettent de refléter le non-déterminisme inhérent aux systèmes distribués.

**Clarté/Qualité du code :** Bud étant un langage déclaratif de haut niveau, le code produit est plus court et plus intuitif qu'un même code créé dans un langage impératif.

**Les opérateurs de BUD :** Il existe trois modes d'opérations dans Bud :
- **Synchrone** : l'opération est effectuée immédiatement, pendant le tick en cours.
- **Asynchrone** : l'opération sera effectuée dans les prochains ticks, sans garantie sur le temps.
- **Différé** : l'opération s'effectuera au prochain tick.

Il existe cinq opérateurs en Bud, réalisant trois types d'opérations :
- **La fusion de données**
  - <= : merge synchrone
  - < : merge asynchrone
  - <+ : merge différée

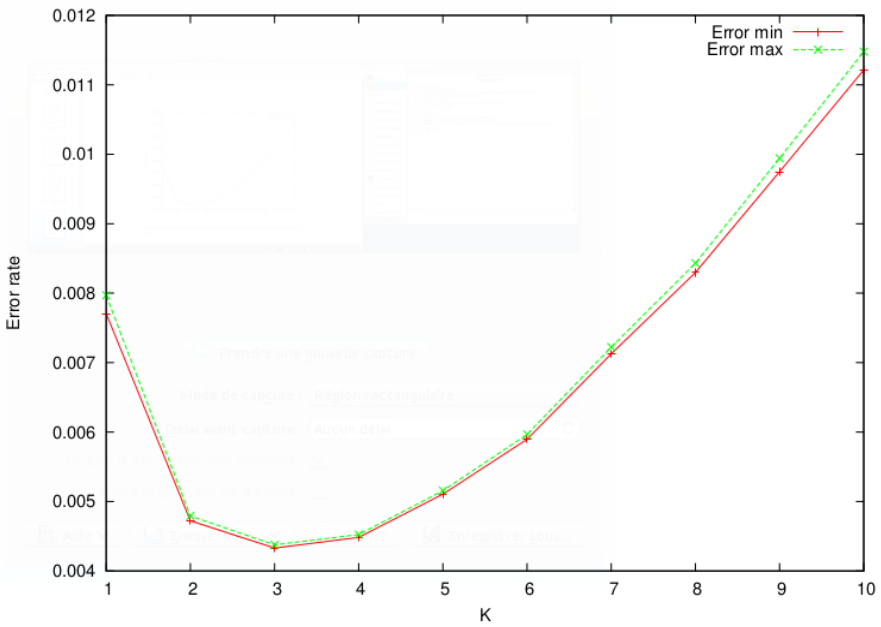
- **la suppression de données** <- : suppression différée
- **la mise à jour de données** <+- : mise à jour différée.
  Pour chaque élément présent dans la collection de droite, cet opérateur retire de la collection à gauche tous les tuples correspondant à l'élément (comparaison sur les clés), puis insert l'élément dans la collection.

# 3 Une expérimentation à l'échelle

Nous avons présenté ci-dessus l'ensemble des concepts nécéssaires pour comprendre le but de ce projet. Nous nous devons maintenant de présenter l'expérimentation que nous avons mis en place ainsi que les moyens technologiques utilisés.

## 3.1 But de l'expérimentation

Le but premier de notre expérimentation est de valider ou d'invalider les résultats de *Achour Mostefaoui* (*équipe GDD*) obtenu sur un simulateur. L'expérimentation faite sur un simulateur consistait à determiner le nombre d'entrées minimum à incrémenter afin de minimiser le taux d'erreur. La courbe suivante à alors pu être établie.




Afin de comprendre l'utilité de notre projet, il est indispensable de savoir que sur un simulateur l'ensemble des paramètres est contrôlé. Par exemple, la latence est fixée par l'utilisateur du simulateur. D'autre part, un simulateur permet de faire tourner un programme dans des conditions idéales. Ainsi, le but de notre projet est de tester un programme similaire dans des conditions réelles et de vérifier que les résultats ressemblent à ceux obtenus dans l'expérimentation de *Achour Mostefaoui*.

De plus, nous ajoutons une plus-value à l'expérimentation précédemment décrite. En effet, lors du traitement, de nos résulats, nous calculons certaines métriques non calculées lors de l'expérience sur simulateur. Effectivement, lors du post-processing, nous mesurons le nombre d'opérations à remonter lorsqu'une erreur est détectée afin de la replacer au bon endroit. Ceci permet de mesurer le travail que devra faire la procédure de *recovery* afin de replacer une instruction au bon endroit.

## 3.2 Mise en place de l'expérimentation

Afin de mettre en place notre expérimentation, nous avons dû prendre en compte plusieurs facteurs tels que les aspects matériel et logiciel à utiliser. Nous allons donc voir dans cette partie, l'environnement matériel dans lequel se déroule notre expérience. Nous décrirons également le dévelopement du programme à lancer afin de construire une trace d'exécution qui sera traitée par la suite. Ce traitement sera décrit dans la partie suivante.

### 3.2.1 Configuration matérielle

Dans un premier temps, il a fallu mettre en place un réseau afin de faire tourner un programme dessus. Ce réseau a été mis en place par *Jean Yves LEBLIN*, il est constitué de 43 *Raspberry Pi* (ordinateur monocarte à processeur ARM). Les cartes ont un linux modifié en tant que système d'exploitation, nous y avions accès via *SSH* depuis le *CIE*. Le programme que nous souhaitons faire tourner sur ces raspberry étant rédigé en *BUD* ce dernier se doit donc d'être installer sur l'ensemble de ces machines.

### 3.2.2 Développement logiciel

Le but de ce projet étant d'observer la communication entre plusieurs processus, notre programme envoie $n$ messages à l'ensemble des processus lancés. Chaque processus dispose d'un $QVV$ qui est incrémenté lors de l'envoi d'un message. Lorsqu'un message est reçu par un processus, ce dernier enregistre dans un fichier les informations suivantes :

- l'identifiant du processus
- le QVV
- le message transmis
- un booléen ( 1 ou 0) permettant de préciser si l'envoi de ce message a généré une erreur.

Le choix des entrées à incrémenter pour chacun des QVV est déterminé via un module BUD conçu par les *Master 2 ALMA* de l'année dernière. Nous avons donc simplement intégré leur module au sein de notre programme.

Chacun des processus nécéssite de se voir attribuer un identifiant unique. Ce dernier est fourni au lancement du processus par un script bash permettant de parametrer l'ensemble des processus pour l'expérimentation.

### 3.2.3 Problème à résoudre

La dernière version de notre programme n'est à ce jour pas fonctionnelle. En effet, notre compréhension de *BUD* fut longue et difficile. L'apprentissage d'un tel langage n'est pas chose aisé lorsque nous sommes habitué à la programmation séquentielle. D'autre part, proche de la date de rendu du projet, nous nous sommes aperçu que certains problèmes ne pourraient être résolus dans les temps.

Dans un premier temps, lors d'un entretien avec *Brice NEDELEC*, nous nous sommes rendu compte que le code métier nous qui nous a été fourni pour les $QVV$ n'était pas complétement fonctionnel. Ainsi, l'incrémentation des ces derniers lors de l'envoi d'un message posait problème.

Une fois ce soucis résolu par *Brice NEDELEC*, il nous restait le problème de la communication entre plusieurs machines physiques en *BUD*. En effet, il nous a été indiqué une manipulation à faire afin de permettre la communication entre différentes machines. Cette manipulation consiste en fait à modifier le code de *BUD* afin de modifier le comportement d'une communication inter-machines. Ensuite, il nous faut recompiler cette version et s'assurer que c'est bien celle-ci qui sera utilisée par notre programme. Cependant, même avec cette manipulation la communication ne semble pas fonctionner. Sur le *Github* de *BUD*, est disponible un exemple de chat respectant le patron de conception client/serveur. Ce dernier fonctionne parfaitement lorsqu'il est utilisé sur une seule machine. Cependant, lorsque nous tentons d'envoyer un message depuis une machines vers une autre ceci ne fonctionne pas. D'autre part, nous avons réussi à déterminer à l'aide de notre programme que le processus lancé sur une machine différente de celle du serveur ne parvient pas à se connecter à ce dernier.

## 3.3 Résultats attendus

. Avant et pendant le développement de notre programme, nous avons pensé aux différentes configurations possibles pour l'exécution de ce dernier ainsi qu'à l'impact de ces dernières sur les résultats pouvant être observés Nous avons de ce fait établit plusieurs configurations à tester et fait des prévisions sur les résultats que nous devrions obtenir.

Ainsi, nous pouvons faire varier la latence sur le réseau via un outils : *netem*. Nous pensons quand introduisant de la latence sur le réseau, le nombre d'erreur augmentera.

# 4 Présentation des résultats

## 4.1 Traitement des données après expérimentation

Le traitement des données consiste à exploiter les logs générés à partir du programme afin de déterminer le nombre d'opérations en état de recovery. Il s'agit de compter le nombre d'opérations concurrentes et causales comprises entre l'opération en erreur et l'opération précédant causalement celle-ci.

Après avoir parsé le log, il faut pouvoir déterminer qu'une opération est causale à une autre. Pour cela nous construisons un graphe de causalité à partir des $QVV$ et des entrées incrémentées contenus dans le log. Cette opération effectuée, nous pouvons maintenant savoir si une opération est causale ou concurrente à une autre en testant la présence d'un chemin dans le graphe entre celles-ci.

Enfin nous parcourons le log afin de détecter les opérations en erreur. Chaque opération en erreur est alors remontée dans le log juste avant l'opération qui la suit causalement. Dans l'intervalle d'opérations, la distinction est faite entre les opérations causales et concurrentes. Ces dernières sont replacées juste avant l'opération en erreur rectifiée. Nous nous retrouvons donc après avoir traiter une erreur un log organisé de la façon suivante avec dans l'ordre :

- les opérations concurrentes replacées,
- l'opération en erreur replacée,
- les opérations causales à cette dernière non déplacées,
- le reste des opérations qui se situent après l'emplacement initial de l'opération en erreur.

En fin de traitement le log est réorganisé et nous obtenons pour chaque erreur le nombre d'opérations concurrentes et le nombre d'opérations causales.

## 4.2 Présentation des résultats

# Conclusion

Nous sommes déçus de ne pas avoir pu terminer ce sujet de recherche à cause de nombreux problèmes qui n'ont été décelés qu'à la toute fin de notre projet. L'intégralité de notre code semble fonctionnel à première vue :

- Le code Bud permettant l'attribution des $QVV$, leur incrémentation et la communication entre Rasp semble fonctionner correctement
- le code concernant le post-processing à également été mis en place et semble fonctionner

C'est à la fin de l'implémentation de notre code Bud que nous nous sommes aperçus d'un problème dans le code métier des QVV qui a été par la suite résolu.

Cependant quelques problèmes persistent :

- nous n'avons pas été capable de faire communiquer les Rasp entre elles via la passerelle SSH même en modifiant le code BUD (remplacement de **@ip, @options[ :port]** par **ip, port** et **:ext_ip** et **:ext_port** pour les adresses externes.
- Nous avons été dans l'incapacité de terminer le Setup et ce à cause du fait que nous ne pouvions accèder au serveur des Rasp par moment (impossible de « pingguer » celui-ci et donc de s'y connecter)

Malgré ces soucis, ce sujet de TER s'est révélé intéressant même si nous avons consacré la majeur partie de notre temps à comprendre Bud et ses mécanismes. Etant contraints par la taille de ce rapport nous avons préféré exposer davantage en détail notre travail que de revenir sur des notions comme Bud et les QVV qui seront davantage expliqués lors de la soutenance.