Rapport de DEA : Etude de la Cohérence Dans les jeux en réseaux distribués

Juillet 2001

Auteur : Christian BRUNETTE

Encadreurs : Frédéric BOUSSINOT et Jean Ferdinand SUSINI

Rapport de DEA : Etude de la Cohérence Dans les jeux en réseaux distribués

Juillet 2001

Dans le cadre du projet PING

Christian BRUNETTE
D.E.A. Réseau et Systèmes Distribués

Université de Nice – Sophia Antipolis
Ecole Doctorale STIC
930, route des Colles – Bât ESSI
B.P. 145
06903 SOPHIA ANTIPOLIS CEDEX

&

Frédéric BOUSSINOT
et
Jean Ferdinand SUSINI
Equipe MIMOSA

I.N.R.I.A.
Unité de Recherche SOPHIA ANTIPOLIS
2004, route des Lucioles
B.P. 93
06902 SOPHIA ANTIPOLIS CEDEX

REMERCIEMENTS

Je tiens tout d'abord à remercier Frédéric BOUSSINOT et Jean Ferdinand SUSINI pour m'avoir accueilli au sein de l'équipe MIMOSA et pour m'avoir apporté leur connaissance et un soutien permanent dans mon stage. Leur aide m'a été précieuse et m'a permis d'effectuer, dans une bonne ambiance, cette étude.

Je tiens aussi à remercier Julien DEMARIA, stagiaire de DEA Informatique, pour son aide, sa bonne humeur quotidienne et pour avoir supporté ma musique pendant toute la durée du stage.

Enfin, je tiens aussi à remercier Raoul ACOSTA pour sa présence dès que j'avais besoin de lui et son humour.

SOMMAIRE

REMERCIEMENTS 3
SOMMAIRE 4
TABLE DES FIGURES 5
INTRODUCTION 6
PREMIERE PARTIE : Contexte du stage 7
I. Présentation du projet PING 8
1. Qu'est-ce que le projet PING ? 8
2. Les diverses tâches 8
3. Les acteurs 9
4. Organisation du projet 10
II. Présentation des mondes distribués 11
III. Présentation de Junior 12
1. Qu'est-ce que Junior ? 12
2. Les instructions 13
3. Les Icobjs 15
IV. Les jeux en réseau : état de l'art 16
1. Différents types d'architectures 16
a. Architecture centralisée 16
b. Architecture distribuée 17
2. Les recherches effectuées 18
3. Discussion 18
SECONDE PARTIE : Analyse du problème et solutions proposées 19
V. Notion de cohérence 20
1. Cohérence spatiale 20
2. Cohérence temporelle 21
3. Discussion 22
VI. Présentation du jeu 23
1. Principe du jeu 23
2. L'architecture 24
3. Les communications 25
4. La complexité 25
5. Quelques points importants 26
VII. Problèmes de cohérence 27
1. Types d'incohérence 27
2. Causes de ces incohérences 29
VIII. Cohérence : solutions proposées 30
1. Solution 1 : ajout de l'écart moyen 31
2. Solution 2 : prise en compte de la différence de rapidité 32
3. Solution 3 : estimation de la prochaine position 33
4. Solution 4 : rattrapage d'écart 33

IX. Résultats 35
1. Implémentation des solutions 35
2. Résultats de ces tests 35
a. Contexte des tests 35
b. Durée des instants 36
c. Tests sans modifications 37
d. Tests avec modifications 38
CONCLUSIONS ET PERSPECTIVES 40
Conclusions 40
Perspectives 40
REFERENCES 41

TABLE DES FIGURES

Figure 1 : Schéma de l'organisation du projet PING 10
Figure 2 : Répartition d'un monde virtuel entre différentes simulations 11
Figure 3 : Notion d'instant et de concurrence 12
Figure 4 : Exemple de code en Junior 15
Figure 5 : Programmation par Icobjs 15
Figure 6 : Mesure d'incohérence 20
Figure 7 : Evolution d'une vue partielle d'une simulation 21
Figure 8 : Exemple d'agrandissement de cohérence 22
Figure 8 : Capture d'écran du jeu 23
Figure 9 : Architecture du jeu 24
Figure 10 : Schéma de communication 25
Figure 11 : Incohérence au moment de la collision 27
Figure 12 : Incohérence temporelle 29
Figure 13 : Modification de la première solution 31
Figure 14 : Modifications apportées par la deuxième solution 32
Figure 15 : Modification de la première solution 33
Figure 16 : Durée des instants en millisecondes sans correction 36
Figure 17 : Durée des instants en millisecondes avec correction 36
Figure 18 : Trajectoire sur l'axe des X d'un répliqua maître (sans correction) .. 37
Figure 19 : Trajectoire sur l'axe des X d'un répliqua esclave (sans correction). 37
Figure 20 : Trajectoire sur l'axe des X d'un répliqua maître (avec correction).. 38
Figure 21 : Trajectoire sur l'axe des X d'un répliqua esclave (avec correction) 38
Figure 22 : Graphe de différence de rapidité entre les simulations 39

INTRODUCTION

Le domaine des jeux en réseau est de plus en plus étudié avec la montée grandissante de l'Internet. Actuellement, on peut jouer en réseau, mais en nombre peu élevé. Le projet européen PING (Platform of Interactive Network Game) a pour but de développer une plate-forme pour jouer en réseau de façon massive, c'est-à-dire une plate-forme qui fonctionne à l'échelle de l'Internet.

La problématique principale est que les jeux en réseau, en général, fonctionnent dans une architecture centralisée où un serveur se charge de calculer, comme pour Quake, la majeure partie des modifications du plateau de jeu qu'il doit renvoyer au client, ce qui permet de maintenir un état de cohérence forte entre chaque client. Ce système fonctionne, mais toute la complexité et le travail se situant au niveau du serveur, il est clair que cette méthode ne passe pas à l'échelle ou du moins ne tient pas dans le cas où il y a un très grand nombre de participants. Si l'on veut pouvoir jouer en réseau de façon massive, il faut donc réussir à distribuer les différentes tâches entre toutes les machines qui participent au jeu, comme par exemple dans le jeu Diablo. La majeure partie des études (cf.[4] et [6]) qui se font dans le domaine des jeux en réseau se place dans une architecture distribuée pour justement permettre d'augmenter le nombre de joueurs sur le jeu. Le problème vient du fait que, si on distribue les différentes tâches entre les différents clients, il risque d'apparaître des états incohérents entre les différentes simulations. C'est dans ce cadre que se situe mon étude. La tâche était de définir les différents types d'incohérences qu'il pouvait y avoir à partir d'un petit jeu, de donner des mesures physiques de cette incohérence et de trouver des moyens pragmatiques pour améliorer l'état de cohérence global.

L'équipe MIMOSA travaille dans ce projet pour essayer de définir des comportements pour les objets simulés dans les jeux. C'est dans ce but qu'ils ont créé un petit jeu pour démontrer les différents comportements qu'ils peuvent fournir grâce à leur langage réactif. C'est à partir de ce jeu qu'on a défini les différents types d'incohérences.

Les systèmes réactifs réagissent de manière continue aux activations provenant de leur environnement. L'approche réactive pour programmer ces systèmes est actuellement développée en Java, sous la forme d'un langage appelé Junior.

Ce rapport va se présenter de la façon suivante. Dans une première partie, nous verrons le contexte dans lequel s'est passé mon stage : nous présenterons d'abord le projet dans lequel s'inscrit mon étude, c'est-à-dire le projet PING ; puis nous verrons quelques notions portant sur les mondes virtuels distribués ; puis nous verrons le langage réactif utilisé pour programmer des comportements dans les jeux, Junior ; et pour conclure cette première partie, nous ferons un petit état de l'art sur les jeux en réseau. Ensuite, dans une seconde partie, nous présenterons de plus près la notion de cohérence, le petit jeu réalisé par l'équipe MIMOSA. Ensuite à partir de ce jeu, nous détaillerons les différents types d'incohérence que l'on peut y trouver et enfin, avant de conclure, nous décrirons les différentes solutions pour résoudre ou du moins réduire ces incohérences.

PREMIERE PARTIE : Contexte du stage

I. Présentation du projet PING

1. Qu'est-ce que le projet PING ?

PING signifie Platform of Interactive Networked Games. Ce projet a pour but de spécifier, développer une architecture ouverte pour l'Internet qui supporte des applications distribuées de mondes virtuels et qui peut supporter plusieurs milliers de connexions simultanées dispersées géographiquement. L'infrastructure du projet consiste en des composants de communication, de stockage et de simulations temps réel. PING propose aussi des facilités de programmations à partir d'objets actifs de haut niveau. Plusieurs techniques à différents niveaux utilisant la structure spatiale et la sémantique des grands mondes virtuels seront utilisées pour permettre à l'utilisateur d'avoir une vue cohérente de son environnement tout en maintenant une interactivité et une navigabilité fluide.

Donc, le but est de créer une plate-forme qui permettra de contenir aussi bien des simulations de courte durée avec peu d'utilisateurs, sans contrainte temps réel et non persistantes que des simulations temps réel, persistantes, de longue durée avec un grand nombre d'utilisateurs et de gros volumes de données à partager, et qui passera sur l'Internet.

La plate-forme PING doit répondre à certains critères. Entre autre, elle doit pouvoir :

- contenir un espace partagé pour des centaines voire des milliers d'utilisateurs répartis à travers le monde ;
- elle doit être capable de supporter des mondes virtuels partagés de grandes tailles (niveau espace mémoire) qui nécessite de grandes ressources de calculs ;
- elle doit permettre de garder les traces et de continuer à faire vivre le monde même quand les utilisateurs ne sont pas présents ;
- permettre de partager l'espace du monde entre les objets contrôlés par les utilisateurs et des objets autonomes et de les faire interagir en temps réel ;
- être supportée par des plates-formes et des environnements hétérogènes.

2. Les diverses tâches

Pour l'architecture, le projet définira une plate-forme ouverte, flexible et extensible pour des mondes virtuels partagés sur l'Internet. Les applications visées par le projet ont des besoins différents en terme de modèles de communication, de cohérence, de persistance… Ces besoins évoluent très rapidement. C'est pour cela que le projet PING n'a pas pour but de créer une plate-forme ancrée dans certaines technologies, mais a pour but de définir d'une architecture ouverte, évolutive et adaptable qui pourra donc être étendue et personnalisée selon l'application.

Pour l'infrastructure, le projet doit développer différents composants pour le stockage des informations, pour gérer les communications et l'exécution pour un grand éventail d'applications multi-utilisateurs. Ces différents composants doivent comprendre aussi des outils de programmation. Quatre aspects ont été retenus pour ce projet :

- ***Une gestion des objets.*** Les objets diffèrent par leur fréquence de changement et la durée de leur cycle de vie. Par exemple, certains peuvent être modifiés très régulièrement et avoir une durée de vie très courte comme par exemple les projectiles dans une bataille ou, au contraire, être atteint par très peu de modifications ou des modifications de façon assez irrégulière mais avoir une durée de vie très longue. Donc nous avons des objets avec des caractéristiques très particulière, et donc PING doit procurer des services de gestion de ces objets et permettre à un très grands nombres d'utilisateurs d'y accéder assez rapidement.

- ***Une gestion de la cohérence***. Chaque utilisateur doit avoir une vue cohérente du monde virtuel qu'il a sous les yeux. On ne peut pas avoir une cohérence parfaite ou l'application deviendrait inutilisable au point de vue de l'interactivité, c'est ce que l'on va voir de plus près dans la suite de ce rapport. Donc le projet PING a pour but de définir des contraintes pour garantir la plus grande cohérence possible tout en proposant une interactivité raisonnable.
- ***Une infrastructure réseau qui passe à l'échelle de l'Internet.*** Dans ce domaine, PING a pour but de combiner tous les atouts d'un serveur centralisé et ceux d'une approche multicast distribuée permettant les communications clients serveurs et les communications point à point. Le projet conduira aussi à rechercher comment modifier les infrastructures réseaux existantes pour qu'elles supportent les applications à grands nombres d'utilisateurs.
- ***Le comportement des objets.*** Le projet PING ne veut pas vraiment se focaliser sur l'aspect graphique des objets, mais veut aussi permettre de coder le comportement de ces objets. Donc, un framework de haut niveau sera développé pour permettre de programmer des comportements d'objets actifs qui réagiront aux événements de l'environnement, c'est-à-dire aux interactions avec d'autres objets par exemple.

Pour ce qu'il en est du réseau, le projet prend comme base de départ le prototype des jeux en réseau actuels. Le choix des jeux en réseau vient du fait que c'est un domaine qui a de fortes contraintes en matières d'interactivité et de passage à l'échelle. C'est pourquoi les résultats de la plate-forme PING seront testés par un jeu ce qui permettra de voir les caractéristiques de la plate-forme et de tester sa « scalabilité » (le passage à l'échelle) et son interactivité. La plate-forme doit pouvoir s'adapter aussi bien à des machines connectées directement à un LAN qu'à des machines connectées par modem. Evidemment, le projet tiendra compte des avancées technologiques dans le domaine des réseaux (ADSL…) et s'assurera que la plate-forme sera le reflet des technologies réseau utilisées par la majorité des clients de la plate-forme. Il est bien évident que si l'on veut maintenir une bonne interactivité aux participants de l'application, il faut réussir à leur cacher le manque de bande passante ou les effets de la latence.

3. Les acteurs

Le projet PING consiste en plusieurs groupes de travail qui se concentrent chacun sur différents problèmes allant du niveau applicatif au niveau système. PING est un projet européen dont les acteurs sont :

- **France Telecom R&D** (France) a pour but de coordonner le projet. Il amène ses compétences dans le domaine plate-forme distribuée, orientée-objet. France Telecom a récemment ouvert un site de jeu en réseau (cf. [11]).
- **ENST** (Ecole Nationale Supérieure des Télécommunications de Paris, France) amène ses connaissances dans le domaine des systèmes distribués temps réel et dans le domaine des protocoles multicast.
- **ARMINES-CMA** (Centre de Mathématiques Appliquées de l'école des Mines de Paris, France) amène ses compétences dans le domaine des systèmes réactifs et de la programmation réactive.
- **SICS** (Swedish Institute of Computer Science, Suède) amène ses compétences dans le domaine des mondes virtuels partagés et des applications collaboratives. SICS a d'ailleurs développé l'une des plate-formes de recherches sur les environnements virtuels multi-utilisateurs les plus utilisées, DIVE (cf. [12]).

- **IMAG** (Institut d'Informatique et de Mathématiques Appliquées de Grenoble, France) amène ses compétences dans le domaine des bases de données orientées-objet multimédia.
- **University of Reading** et **Lancaster University** (Royaume-Uni) apportent leur expertise dans la conception et la modélisation de systèmes multimédia distribués et coopératifs.
- **Kalisto** (France) amène ses connaissances dans le domaine de la conception de jeux dont les jeux en réseau bien évidemment. Kalisto est une cible potentielle pour l'utilisation de la plate-forme PING.

4. Organisation du projet

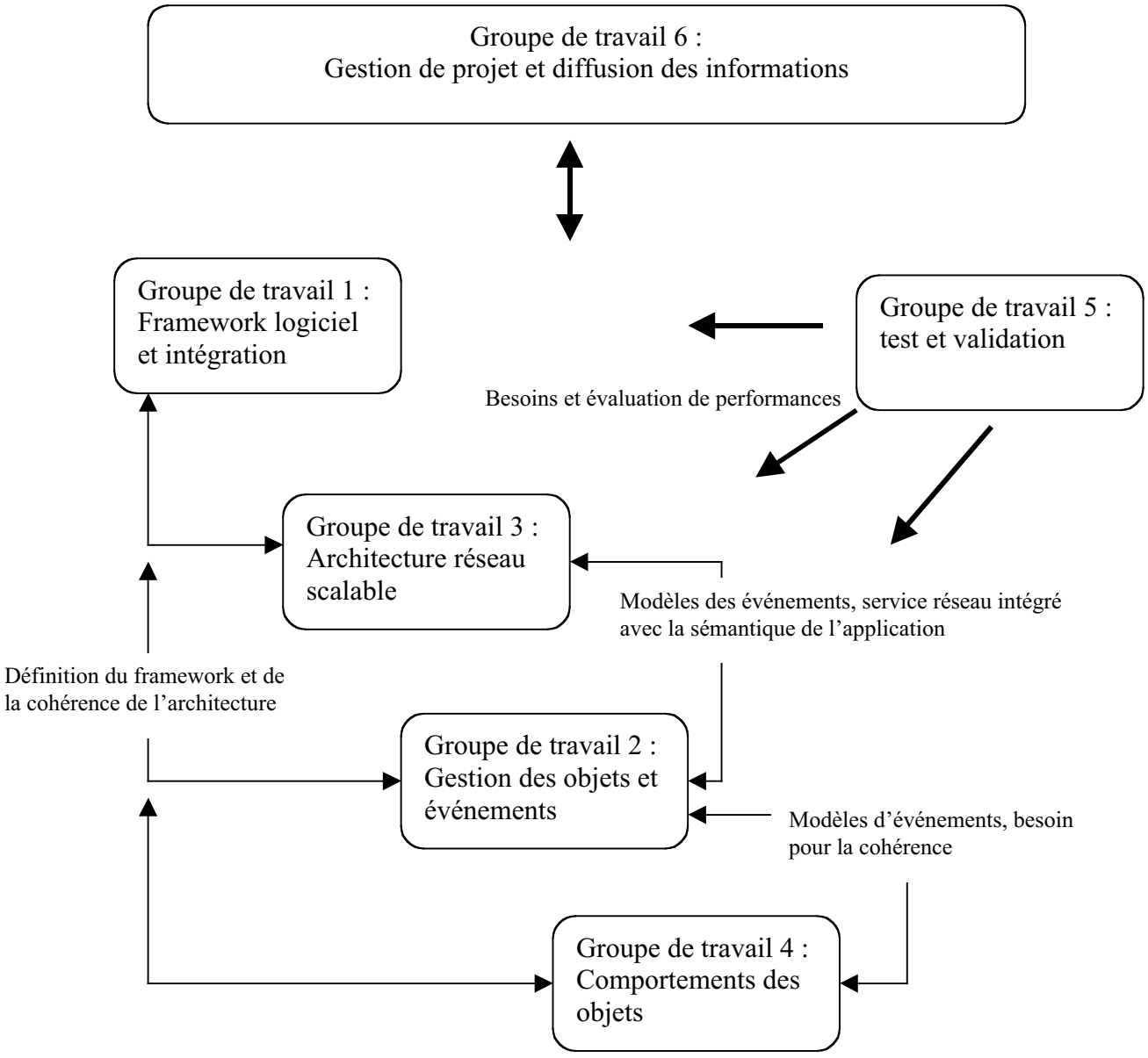


Figure 1 : Schéma de l'organisation du projet PING

Pour plus d'informations sur le projet PING, cf. [10].

II. Présentation des mondes distribués

Dans cette partie, nous allons préciser quelques notions qui sont en rapport avec les mondes virtuels distribués. Pour commencer, un schéma étant plus parlant que tous les textes, nous pouvons déjà nous référer à la Figure 2 pour décrire chaque notion.

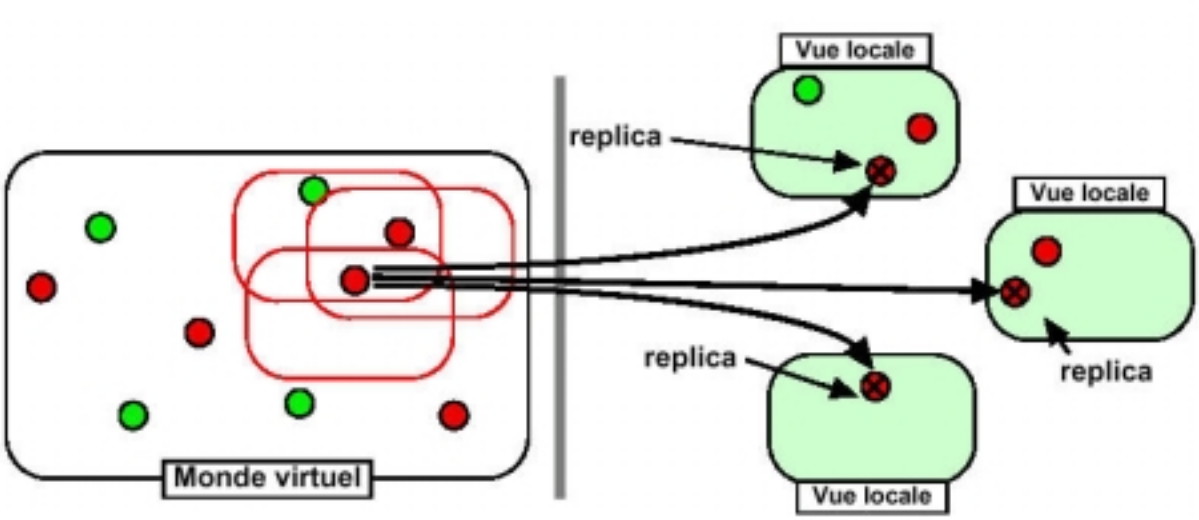


Figure 2 : Répartition d'un monde virtuel entre différentes simulations

Un monde virtuel est un environnement, qui peut être modifié, composé de plusieurs types d'objets comme nous pouvons le voir sur le schéma. Ces objets sont représentés par des ronds sur le schéma. Les ronds verts représentent les entités passives que l'on peut trouver dans un monde virtuel comme des murs, des décors ; ce sont des objets qui n'évoluent pas au cours du temps et donc qui n'ont aucun comportement. Les ronds rouges représentent les entités actives, c'est-à-dire des entités qui évoluent au cours du temps par l'intermédiaire de leur comportement ou du comportement des autres objets.

Un monde distribué est un monde virtuel dont les modifications sont gérées par plusieurs simulations en parallèle. Donc un des avantages que l'on peut y trouver, c'est que chaque simulation n'a pas besoin de représenter la totalité du monde virtuel, c'est ce que l'on peut remarquer sur la Figure 2. Une simulation locale représente donc le point de vue de l'utilisateur (ou joueur) sur le monde.

Puisqu'il y a plusieurs points de vue sur le même monde, il y a donc plusieurs occurrences de la même entité. La représentation d'une entité dans une simulation est appelée un « répliqua ». Ces répliquas sont séparés en deux catégories : d'un côté les maîtres et de l'autre les esclaves.

Pour chaque répliqua, il y a un maître présent dans une des simulations. Ce maître est ce que l'on peut appeler le guide de l'entité, c'est-à-dire c'est de lui qu viennent les initiatives concernant l'entité. Le maître a l'ensemble des comportements de l'entité. Les autres répliquas de la même entité qui ne sont pas le maître sont tous des répliquas esclaves. Ces derniers possèdent une partie des comportements que possède son maître ou plus précisément des comportements dégradés. On appelle ces répliquas des esclaves, car ils reproduisent ce que fait leur maître.

Une autre définition qu'il faut préciser est celle de Dead-reckoning. C'est un mécanisme d'anticipation de mouvements, utilisé dans les simulations distribuées pour donner un mouvement au répliqua esclave entre deux resynchronisations par rapport à celle du maître et non faire des sauts d'une position à une autre à chaque resynchronisation.

III. Présentation de Junior

Comme nous venons de le voir, le projet PING contient une partie de création des comportements des objets pour la plate-forme. C'est la tâche qui incombe à l'équipe MIMOSA. C'est dans cette optique qu'il utilise Junior et les Icobjs que nous allons décrire maintenant.

1. Qu'est-ce que Junior ?

Pour définir Junior, il faut d'abord définir ce qu'est l'approche réactive. Les programmes réactifs réagissent, en continu, aux activations provenant de l'extérieur par l'intermédiaire d'événements. Pour plus d'informations sur l'approche réactive, la lecture de [1] donnera le complément. Les notions de base qu'il faut définir pour l'approche réactive sont celles d'instant, de concurrence et de diffusion d'événements.

Notion d'instant

Dans le mot réactif, on voit la notion de réaction. Dans le langage réactif, une réaction est appelée un *instant* (cf. Figure 3). Aucune hypothèse de durée n'est faite sur l'instant. Au niveau du langage, c'est l'exécution des instructions qui permet de savoir si un instant est fini ou non, c'est-à-dire on détermine la fin d'un instant si plus aucune instruction ne peut être exécutée dans l'instant courant et si on est sûr que plus aucun événement sera généré pendant cet instant.

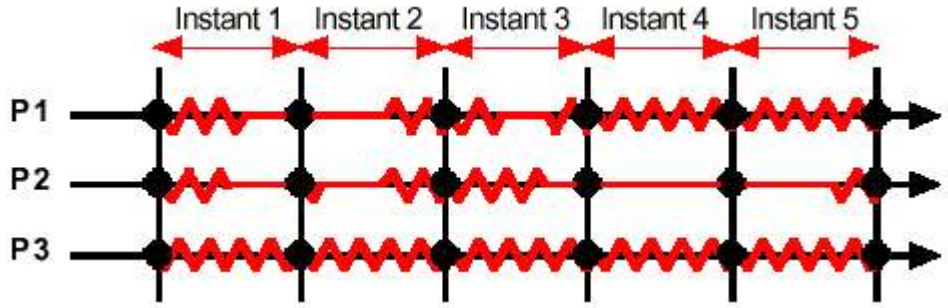


Figure 3 : Notion d'instant et de concurrence

Concurrence

Dans l'approche réactive, la notion de concurrence est liée à la notion d'instant, c'est-à-dire que, puisque le temps est cadencé par les instants, on peut exécuter en parallèle plusieurs programmes réactifs, ce qu'on peut d'ailleurs remarquer sur la Figure 3. D'ailleurs, comme nous le verrons par la suite dans le langage, il existe une instruction qui permet de paralléliser des instructions.

Diffusion d'événements

Un événement est une entité qui existe dans l'environnement et qui est réinitialisée à chaque nouvel instant. Dès qu'un événement est généré, il est présent dans l'environnement pendant tout l'instant courant, c'est ce qu'on appelle la diffusion instantanée d'événements. A partir de ce moment, plus aucun programme réactif qui se trouve dans la machine réactive ne

peuvent nier la présence de l'événement au cours de cet instant. Cette diffusion d'événements permet de synchroniser plusieurs instructions. Un autre point important à préciser est qu'en Junior, on ne peut déterminer qu'un événement est absent durant un instant qu'à la fin de celui et donc on ne peut réagir à l'absence d'un événement qu'à l'instant suivant celui du test.

Programmer en Junior

La programmation réactive permet d'utiliser les aspects de concurrence, de diffusion d'événements et, par l'intermédiaire de plusieurs primitives, d'avoir un contrôle fin sur l'exécution des programmes réactifs. Les deux notions principales sont les instructions réactives dont la sémantique est liée à la notion d'instants et les machines réactives qui exécutent les instructions réactives dans un environnement où les événements sont diffusés de façon instantanée.

Junior est un langage réactif basé sur Java pour programmer des comportements réactifs. Pour effectuer ces comportements, il suffit de :

- décrire le comportement par des instructions réactive que nous verrons par la suite,
- de créer une machine réactive,
- de mettre les instructions réactives dans cette machine,
- et de faire réagir cette machine.

Un autre aspect de Junior est que l'on peut mettre des instructions réactives dans la machine de façon dynamique au cours de l'exécution de cette dernière. Ces nouvelles instructions n'auront pas besoin d'attendre la fin de celles qui étaient déjà dans la machine pour pouvoir être exécutées, elles sont directement exécutées en parallèle de celles qu'il y avait déjà. Cet aspect est à comparer avec le langage Esterel où il n'est pas possible d'ajouter des instructions dynamiquement car cela engendre des problèmes de cohérence à l'exécution, mais nous ne rentrerons pas plus à fond dans ces considérations.

Junior est aussi une alternative à l'utilisation des threads en Java et règle même quelques problèmes inhérents à ces threads. Pour plus d'informations sur Junior, on peut lire un article sur les SugarCubes (cf. [3]) qui est l'ancêtre de Junior. Pour plus d'informations sur Junior, son noyau et sur ses différentes implémentations, on peut lire [7], [8] et [9].

2. Les instructions

Dans Junior, il y a un minimum d'instructions qui nous permettent de créer des comportements qui s'exprimeraient de façon plus complexe sans le langage réactif. Sans rentrer trop dans les détails de la sémantique que l'on peut consulter en [3], [8] et [9], voici les différentes instructions dont est composé Junior.

Instructions de base :

- **Nothing()** : c'est l'instruction qui ne fait rien et se finit immédiatement.
- **Seq(instruction1 , instruction2)** : met en séquence deux instructions. La seconde débutera qu'à partir du moment où la première est finie. Chaque instruction peut durer un ou plusieurs instants.
- **Par(instruction1 , instruction2)** : permet de paralléliser deux instructions dans la machine réactive. Les deux instructions seront exécutées en même temps.
- **Stop()** : interrompt l'exécution de la branche d'exécution dans laquelle le programme réactif courant se trouve jusqu'à l'instant suivant. Il ne met pas fin à tout les programmes qui sont dans la machine réactive. On a précisé plus haut que plusieurs

instructions pouvaient être parallélisées et l'instruction Stop() interrompt uniquement l'instruction (ou suite d'instructions) parallélisée dans laquelle il se trouve.

- **Loop(instruction)** : permet de boucler à l'infini sur l'instruction.
- **Repeat(expression , instruction)** : permet de répéter l'instruction un nombre de fois selon l'expression.

Instructions événementielles :

- **Generate(événement)** : permet de générer un événement pur à l'instant courant.
- **Generate(événement , valeur)** : permet de générer un événement valué à l'instant courant. Cette valeur est un objet Java quelconque.
- **Await(événement)** : se met en attente de l'événement. Cette instruction est bloquante tant que l'événement n'est pas apparu.

Les opérations suivantes permettent de faire des combinaisons d'événements.

- **And(événement1 , événement2)** : est vraie si les deux événements sont présents lors du même instant.
- **Or(événement1 , événement2) :** est vraie si un des deux événements est présent lors de l'instant.
- **Not(événement)** : est vraie si l'événement est absent lors de l'instant.

Instructions de contrôle :

- **If(condition , instruction1 , instruction2)** : teste la condition lors du premier instant d'exécution et si la condition est vraie, on exécute l'instruction1, sinon on exécute l'instruction2.
- **When(événement , instruction1 , instruction2)** : teste la présence et si l'événement est présent, alors on exécute l'instruction1, sinon on exécute l'instruction2 à l'instant suivant.
- **Until(événement , instruction1 , instruction2)** : permet d'exécuter l'instruction1 tant que l'événement n'est pas apparu. Dès que l'événement apparaît, on finit l'instant courant sur l'instruction1. Si elle a fini son exécution pendant l'instant, on passe à l'instruction2 dans le même instant, sinon on attend l'instant suivant pour commencer l'instruction2. Dans le cas où l'instruction1 est finie avant la réception de l'événement, on sort de l'instruction Until.
- **Control(événement , instruction)** : permet d'exécuter l'instruction uniquement pendant les instants où l'événement est présent.

Instructions d'interfaçage avec Java :

- **Atom(Action)** : permet d'exécuter un programme JAVA classique. Il suffit que la classe JAVA implémente l'interface Action. Cette instruction sera forcément exécutée et finie dans l'instant où elle a commencé.
- **Link(Objet , instruction)** : permet de lier un programme réactif à un objet Java.
- **linkedObject()** : permet d'accéder à l'objet qui est lié au comportement (à l'instruction réactive) courant.

Instructions de migration :

- **Freezable(événement , instruction)** : permet de geler l'instruction dans la machine réactive à partir de la réception de l'événement.
- **GetFrozen(événement)** : permet de récupérer les instructions gelées qui sont dans la machine réactive à la réception de l'événement.

Le code suivant boucle à l'infini sur un comportement. Ce comportement génère l'événement « step » à chaque instant (l'instruction Stop() permet d'attendre l'instant suivant pour réémettre l'événement « step ») tant que l'événement « suspend » n'a pas été émis. Dès que l'événement « suspend » est émis, le comportement précédent est préempté sur une instruction Stop qui stoppera l'exécution du nouveau comportement jusqu'au prochain instant où le programme se bloquera sur l'attente de l'événement « await ». Dès que ce dernier sera émis, alors l'instruction Until sera terminée.

```
Program controller =
        Jr.Loop(
                Jr.Until(« suspend »,
                        Jr.Loop(Jr.Seq(Jr.Generate(« step »), Jr.Stop())),
                        Jr.Seq(Jr.Stop(), Jr.Await(« resume »))))) ;
```

Figure 4 : Exemple de code en Junior

3. Les Icobjs

Les Icobjs sont des objets (« objs ») qui possèdent une représentation graphiques (« icon »). C'est un langage de programmation graphique basé sur le langage réactif Junior. Un icobj est un objet ayant une représentation graphique qui peut être animée et un comportement décrit en réactif. Ce langage comprend aussi un mécanisme capable de créer de nouveaux icobjs en combinant plusieurs comportements d'autres icobjs. Cette programmation a les mêmes avantages que Junior, elle permet de paralléliser des tâches, et permet la communication par diffusion d'événements. Ce langage permet à ses utilisateurs, une plus grande intuition dans leur programmation. Une des utilisations possibles des icobjs serait de l'utiliser directement à l'intérieur des jeux et permettre ainsi à chacun de programmer des petits comportements aux personnages des jeux. De part la sémantique sur laquelle est basée Junior, et donc de la même manière les icobjs, il est possible de créer des automates et d'accepter et de refuser ce programme en fonction de la complexité de cet automate.

Comme nous pouvons le voir sur la Figure 5, la programmation par Icobjs est vraiment intuitive. L'icobj « right » se déplace d'une certaine distance vers la droite, l'icobj « down » se déplace d'une certaine distance vers le bas. Ensuite, par la programmation par Icobj, on met en séquence les deux, ce qui donne un icobj qui se déplace d'une certaine distance vers la droite puis d'une certaine distance vers le bas.

Dans le cadre du petit jeu réalisé par l'équipe MIMOSA, la programmation par Icobjs ne servent qu'à avoir l'aspect graphique des icobjs, ce qui a simplifié la tâche, car, par exemple, l'environnement nécessaire à la diffusion d'événements était déjà en place. Pour plus d'informations sur les icobjs, vous pouvez consulter [2] ou sur le site [13], vous pouvez trouver un tutorial et quelques exemples su l'utilisation des icobjs.

Figure 5 : Programmation par Icobjs

IV. Les jeux en réseau : état de l'art

Actuellement, le monde des jeux en réseau est en pleine expansion grâce à la croissance des réseaux et évidemment de l'Internet. Les jeux en réseau permettent d'effectuer des tests très poussés sur les architectures réseaux que ce soit au niveau de la demande en bande passante, au niveau de la réaction face au temps de réponse, face aux variations de comportements du réseau. Cela est dû aux changements d'états très fréquents des objets que l'on peut observer dans les jeux, et à la grande demande en interactivité des jeux.

Donc dans cette partie, nous allons d'abord faire un bref état de l'art sur les différentes architectures qui existent dans les jeux en réseau et sur des recherches effectuées. Ensuite, nous pourrons voir une description du petit jeu réalisé par l'équipe MIMOSA pour montrer les comportements qui peuvent être créés par l'intermédiaire du langage réactif Junior et la facilité avec laquelle on peut coder ces comportements.

Les informations concernant les jeux en réseau ne sont en général très pas diffusées par les éditeurs. Mais nous pouvons quand même faire des constatations à partir de ce que l'on voit. Il y a tout de même quelques recherches qui ont été publiées dont surtout [4] que nous verrons de plus près.

1. Différents types d'architectures

Actuellement, nous pouvons différencier deux types d'architecture dans les jeux en réseau : d'une part, nous avons les architectures centralisées avec des jeux « à la » Quake et d'autre part, des architectures distribuées comme on peut les trouver en jouant à Diablo sur un LAN. Les avantages d'une des architectures sont les inconvénients de l'autre.

a. Architecture centralisée

Donc d'un côté, nous avons un serveur qui supporte toutes les connexions des clients et qui se charge de faire la majeure partie des calculs. La seule tâche du client est de transmettre au serveur les réactions du joueur et d'afficher le résultat.

Avantages

+ Dans une architecture centralisée, on n'a pas à s'occuper de l'état de cohérence générale, car, tout étant calculé sur une seule machine, il n'y a qu'une seule décision prise pour chaque cas de figure, donc il n'y a aucun risque de trouver deux versions différentes sur deux simulations qui se trouvent dans le même monde sur le même serveur.
+ Un des avantages que l'on peut noter, c'est qu'une architecture centralisée est bien plus simple à mettre en œuvre, bien plus sécurisée et bien plus simple au niveau de la cohérence globale. Il n'est pas nécessaire, par exemple, de mettre en place des synchronisations puisqu'elles existent de manière naturelle.
+ De plus, la puissance de la machine n'a pas forcément besoin d'être très élevée, car la majeure partie des calculs est effectuée par le serveur. Dans un jeu « à la » Quake, c'est le serveur qui a toute la charge de calculs alors que le client ne fait que transmettre des informations et d'afficher les résultats calculés par le serveur.
+ Enfin, on peut aussi remarquer qu'une architecture centralisée est bien plus sûre, puisque toutes les informations proviennent d'une seule machine, le serveur. Donc, il est beaucoup moins évident d'introduire des erreurs.

Inconvénients

- Si le serveur plante, la partie est finie, puisque c'est lui qui se charge de calculer toutes les simulations, donc on peut noter une faible tolérance aux pannes.
- Un autre inconvénient de l'architecture centralisée est que tout dépend du serveur (et des connexions qui permettent d'atteindre le serveur) qui devient le « goulot d'étranglement » des performances, puisque, plus il y a de machines, plus il doit gérer de connexions et de calculs.
- On peut aussi dire que l'on perd beaucoup du temps dans les communications réseaux puisque les informations transitent d'abord par le serveur avant d'aller vers le client.
- En plus, le serveur doit simuler l'ensemble du monde et pas uniquement la partie où se trouve les joueurs.
- Le nombre de joueurs qui peuvent participer à un jeu en réseau dans cette architecture sont limités par ce que le serveur peut fournir.

b. Architecture distribuée

D'un autre côté, nous avons une architecture qui distribue les tâches entre les différentes simulations ou plus généralement entre les différentes ressources de calculs. Pour tout ce qu'il en est des calculs, ils sont effectués sur chacune des machines à partir des informations échangées entre chacun des participants.

Il faut quand même un serveur qui, en général, permet d'accepter les connexions à la partie et qui met en relation tous les joueurs. Il faut tout de même préciser que l'utilisation du serveur peut être remplacé par la mise en place d'adresses multicast.

Avantages

+ Le premier point à noter est une grande tolérance aux pannes, puisque toutes les tâches sont distribuées, si une machine disparaît, les autres peuvent continuer à tourner avec un minimum à faire pour redistribuer les tâches.
+ Dans cette architecture, il n'y a pas de goulot d'étranglement comme l'est le serveur dans une architecture centralisée. Puisque les tâches sont distribuées, les calculs le sont évidemment aussi.
+ Enfin, on peut aussi dire que l'on perd beaucoup moins de temps sur les architectures distribuées dans les communications réseaux puisque les informations sont directement transmises aux clients alors que dans une architecture centralisée où toutes les informations transitent d'abord par le serveur avant d'aller vers le client.
+ Par contre, on peut noter qu'une simulation distribuée n'est pas forcée de calculer les modifications qui ont lieu dans tout le monde simulé, il lui est seulement nécessaire de calculer ce que le joueur voit.

Inconvénients

- Un des premiers problèmes à noter est que, puisque chaque simulation calcule ses propres modifications du monde, il est très difficile d'avoir un état de cohérence générale
- Pour palier au premier problème, il faut mettre en place des mécanismes qui permettent de garder au mieux un état cohérent, donc cela entraîne des difficultés supplémentaires dans la programmation d'un système distribué. Il faut aussi mettre en place, par exemple, des systèmes de synchronisations puisque les différentes simulations ne tournent pas forcément au même rythme.
- De plus, on peut noter un problème de sécurité, puisque chacune des simulations possède tout le code du monde, donc peut le modifier à son aise.

– De plus, pour un jeu, la puissance de la machine doit être plus élevée, car elle doit effectuer bien plus de calculs que si c'était un serveur qui se chargeait de tout.

2. Les recherches effectuées

La majeure partie des recherches que l'on peut trouver s'oriente vers ce type d'architecture. Par exemple, dans [6], R.M. Fujimoto nous fait un panorama des simulations distribuées, autant au niveau des simulations de calculs que des simulations distribuées d'environnement virtuel. Il parle des différents avantages des simulations distribuées par rapport à celles centralisées (réduction du temps d'exécution, tolérance aux pannes…). Il montre les différents mécanismes à mettre en place dans des mondes virtuels distribués (dead-reckoning, diffusion des données…). Il décrit aussi certains algorithmes pour la synchronisation entre les différentes simulations. Enfin, il évoque très brièvement un système qui utilise des simulations distribuées pour des exercices militaires appelé SIMNET. Ce système a réussi à tourner sur plus de 250 simulations reliées par réseau sur 11 sites géographiques en 1990.

Dans [4], on peut voir un travail, dans le même sens que celui du projet PING, réalisé par l'équipe RODEO de l'INRIA. Ils ont réalisé un jeu (un PACMAN en 3D) totalement distribué sur Internet. Ils ont défini principalement les bases d'une application distribuée sans serveur avec des mécanismes pour garantir la cohérence des simulations quelque soit le délai du réseau. Ils ont beaucoup travaillé au niveau protocolaire et ils ont utilisé RTP sur UDP sur IP additionnés évidemment à leur propre mécanisme. Leur application suit les règles définies par la norme DIS, c'est-à-dire :

- Que le délai d'interaction doit être inférieur à 100ms
- Qu'il puisse servir un très grand nombre de participants (de l'ordre de 10000)
- Que les messages sont courts mais fréquents
- Qu'il soit dynamique…

Dans [4], ils expliquent aussi leur mécanisme de synchronisation en utilisant une horloge globale fournie grâce à NTP, leur mécanisme de dead-reckoning et de transmission d'informations. On peut voir que ce qu'offre leur petit jeu est vraiment assez proche de ce que la plate-forme PING veut pouvoir offrir aux différentes applications qu'elle doit supporter.

3. Discussion

Pour toutes les solutions qui existent dans les jeux en réseau, on passe souvent par l'utilisation d'une horloge commune ou d'une horloge logique qu'il faut synchroniser entre les différentes simulations qui participent au jeu. Il existe un mécanisme similaire dans le langage réactif que l'on peut trouver dans un article sur les machines réactives distribuées (cf. [5]). Le but est d'utiliser un processus (un synchroniseur) qui permet de, comme le dit son nom, synchroniser les différentes machines réactives, c'est-à-dire de faire qu'il y ait un instant commun à toutes les machines réactives.

L'inconvénient de cette technique, du moins pour notre problème, c'est qu'elle réduit considérablement l'interactivité que pourrait avoir un joueur. En effet, le synchroniseur permet, à la fois de déclarer la fin des instants et de transmettre à toutes les machines réactives des événements globaux à la simulation. Le problème, c'est que toutes les machines réactives iront forcément plus lentement que la vitesse de la machine la plus lente, car il faut encore ajouté les communications réseaux. De plus, cette technique peut fonctionner sur un petit réseau local, mais pas à l'échelle de l'Internet.

SECONDE PARTIE : Analyse du problème et solutions proposées

V. Notion de cohérence

Dans cette partie, nous allons essayer de définir la notion de cohérence. Les moyens proposés pour essayer de la définir ne sont pas exhaustives (loin de là), mais ce sont des premières direction de recherches. Il y a plusieurs moyens de définir la notion de cohérence : d'un côté, on peut la voir comme une incohérence spatiale et d'un autre côté, comme une incohérence temporelle.

1. Cohérence spatiale

Quand on parle de cohérence spatiale, on part sur une notion de mesure physique de distance entre le « réalité » (le comportement du maître) et sa reproduction (le comportement de l'esclave) que l'on peut voir sur la Figure 6.

- La première solution est assez intuitive, c'est de faire la somme des distances entre la position de chaque répliqua maître et la position du répliqua esclave correspondant, ce qui nous permettra de donner une mesure physique à l'incohérence. Cette mesure doit être évidemment effectuée sur chaque simulation entre la position des maîtres des autres simulations et celle de leur esclave sur la simulation où est faite la mesure. Puisque nous nous situons dans un cadre réactif, il y a une notion d'instant commun à chaque maître et à chaque esclave se trouvant sur la même simulation. Evidemment, il est impossible de maintenir un instant commun sur l'ensemble du jeu entre les différents sites qui peuvent être bien éloignés et ce serait vraiment trop coûteux au niveau réseau et au niveau de l'interactivité que pourrait avoir le joueur. Donc, une façon de caractériser le niveau d'incohérence serait, à chaque instant, de calculer cette somme d'écart.
- L'inconvénient de cette première technique est que s'il y a beaucoup de répliquas dans le monde, on ne peut pas déterminer s'il y a de grandes incohérences. Pour montrer cela, on peut partir sur un exemple. Admettons que nous sommes dans un jeu à 8 utilisateurs, si la somme fait 8, on ne peut pas déterminer s'il y a une distance de 8 pour l'un et de 0 pour tous les autres ou s'il y a une somme de 1 pour tout le monde... Donc il serait peut être plus indicatif d'utiliser la distance maximum entre un maître et son esclave.

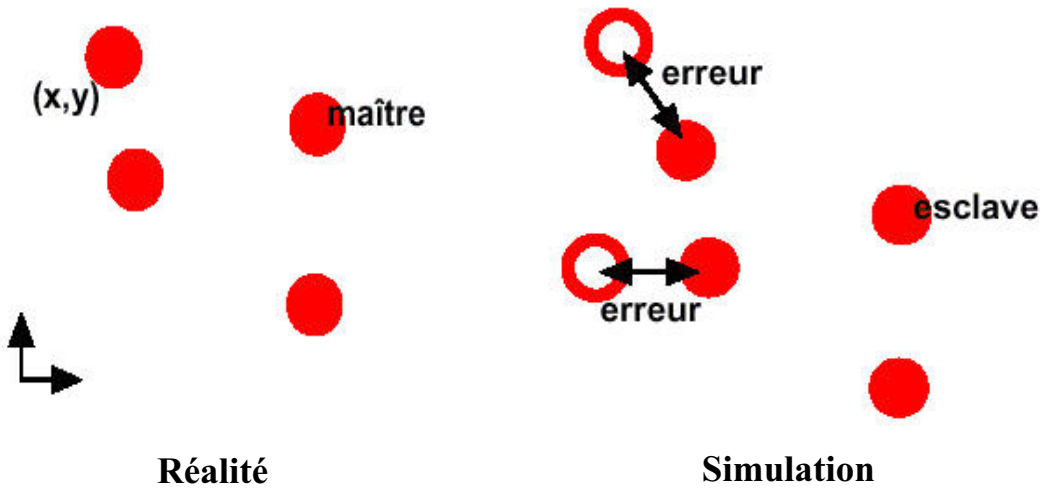


Figure 6 : Mesure d'incohérence

- L'inconvénient de cette seconde méthode est qu'on ne peut pas déterminer l'ampleur des dégâts sur toute la simulation, ce qui peut être déterminé à partir de la première méthode, il serait donc peut être judicieux de valuer l'incohérence par une paire <x, y>, où x représente la somme des distances et y représente le maximum de la distance entre un maître et son répliqua esclave.
- Enfin, dernière solution que je propose dans ce cas d'incohérence, c'est un moyen de définir cette cohérence au cas par cas pour chaque entité présente dans la simulation. En fait, chaque entité qui n'est pas un répliqua maître dans une simulation « représente » une communication point à point entre deux répliquas : le maître dans une autre simulation et le répliqua dans la simulation locale, donc il représente des conditions particulières qu'on ne peut pas nécessairement généralisé, comme les routes empruntés par les paquets sur le réseau, comme la différence entre les deux machines, etc. On traite donc au cas par cas selon la simulation qu'on a en face de nous. Sur un exemple à 3 simulations, si l'une des deux autres simulations est bien plus rapide et le troisième bien plus lente et que l'on réagit de la même manière par rapport aux deux simulations, on risque de créer de bien plus grosses incohérences.

2. Cohérence temporelle

Un autre point de vue important à définir dans la notion de cohérence, c'est la cohérence temporelle. Cette cohérence est assez simple à définir. De la même manière que la cohérence spatiale qui nous permet d'évaluer un écart de distance, la cohérence temporelle permet d'évaluer un écart de durée (dans notre cas en nombre d'instant) qui sépare un maître et son esclave pour une position donnée. Il est possible que les positions d'un avatar soient exactes, mais le problème n'est pas que la trajectoire elle-même, il faut aussi que la transition par chaque point ait lieu au même moment (ou du moins, à des niveaux équivalent sur une certaine échelle de temps).

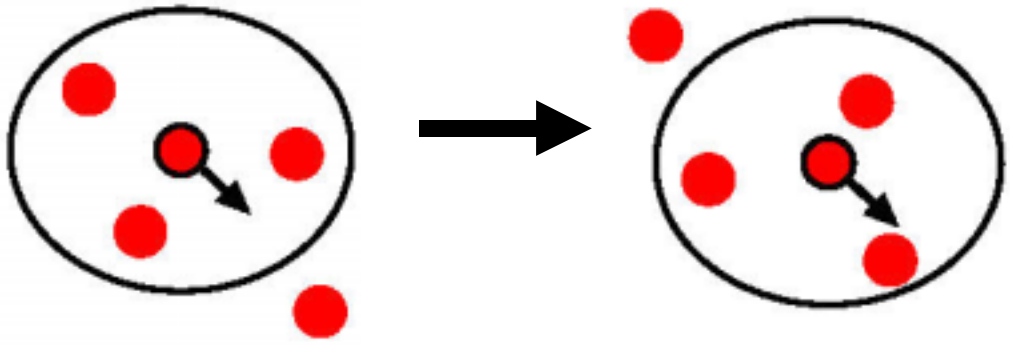


Figure 7 : Evolution d'une vue partielle d'une simulation

De même, si un événement (je ne parle pas d'événements réactifs ici) doit arriver et que deux simulations ont dans leur champ de vision la zone où cet événement doit se passer, il faut réussir à trouver des moyens pour qu'il arrive à se passer vraiment au même moment dans les deux simulations. Comme on peut le voir sur la Figure 7, on a une vue partielle du monde virtuelle, donc il faut que des éléments qui sont hors du champ de vision apparaissent au bon moment.

3. Discussion

Un autre problème est à soulever : "est-ce que, pour rattraper une incohérence, on peut se permettre d'en commettre une encore plus grande ?". Par exemple, pour rattraper une position est-ce que l'on va autoriser notre répliqua a traverser des obstacles infranchissables ? Si ce n'est pas le cas, alors il faut considérer un nouveau moyen de calculer la distance. La distance ne sera plus alors une ligne droite mais la trajectoire la plus courte pour revenir à la position réelle.

La question reste assez ouverte. On peut partir du principe que la nouvelle incohérence ne choquera pas le joueur puisqu'elle est consécutive à une autre. Pour ma part, il est possible qu'à certain moment l'incohérence s'accroît au lieu de se réduire, car les moyens pragmatiques pour réussir à améliorer l'état de cohérence ne sont pas efficaces dans tous les cas de figure. De plus, je pense qu'il est assez important de réduire le plus rapidement possible l'état d'incohérence dans lequel on peut se trouver, car plus on le fait durer, plus il peut s'intensifier.

Prenons un petit exemple pour illustrer ce fait (cf. Figure 8). Pour cela nous allons schématiser une situation ou l'incohérence peut s'agrandir en voulant le réduire. Sur ce schéma, le scénario est le suivant : notre esclave a un certain retard sur son maître et il s'en rend compte au moment de la première resynchronisation. Pour rattraper ce retard, un moyen, par exemple, serait d'augmenter la vitesse de l'esclave jusqu'à la prochaine resynchronisation. Si la seconde resynchronisation est retardée, l'esclave aura une vitesse plus grande que son maître sur une trop longue période et le retard qu'il avait se transforme en avance qui peut être bien plus grande que le retard qu'il avait auparavant.

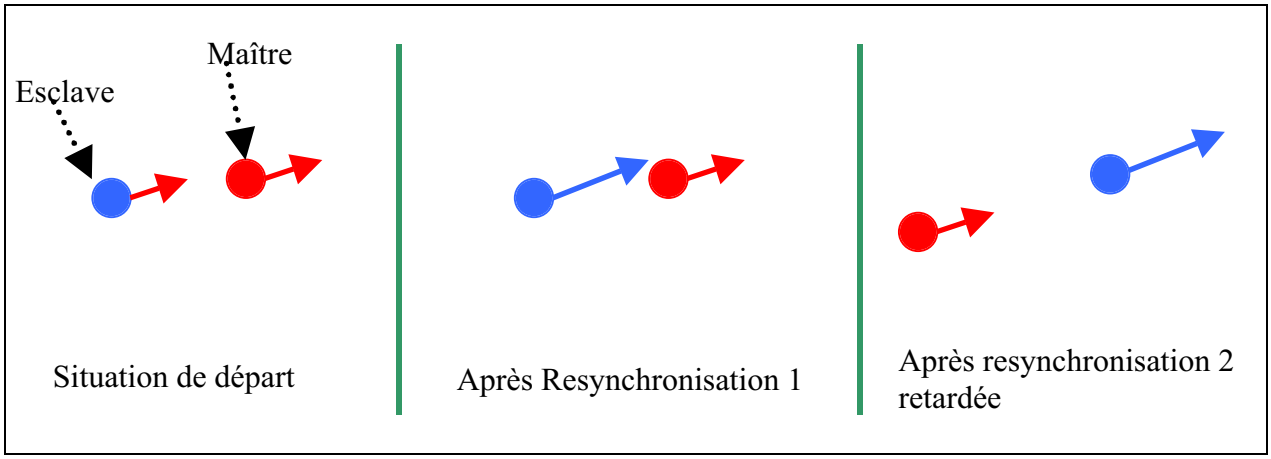


Figure 8 : Exemple d'agrandissement de cohérence

De plus, je pense qu'il est assez important de réduire le plus rapidement possible l'état d'incohérence dans lequel on peut se trouver, même si c'est au prix peut être d'une incohérence plus grande pendant une courte période (ce que l'on peut avoir quand un esclave essaie de rattraper une trajectoire sans passer par les positions où sont passées son maître), car plus on le fait durer, plus il peut s'intensifier.

VI. Présentation du jeu

Maintenant, nous allons parler du petit jeu sur lequel j'ai dû me baser pour déduire les différents types d'incohérences et pour améliorer ces états.

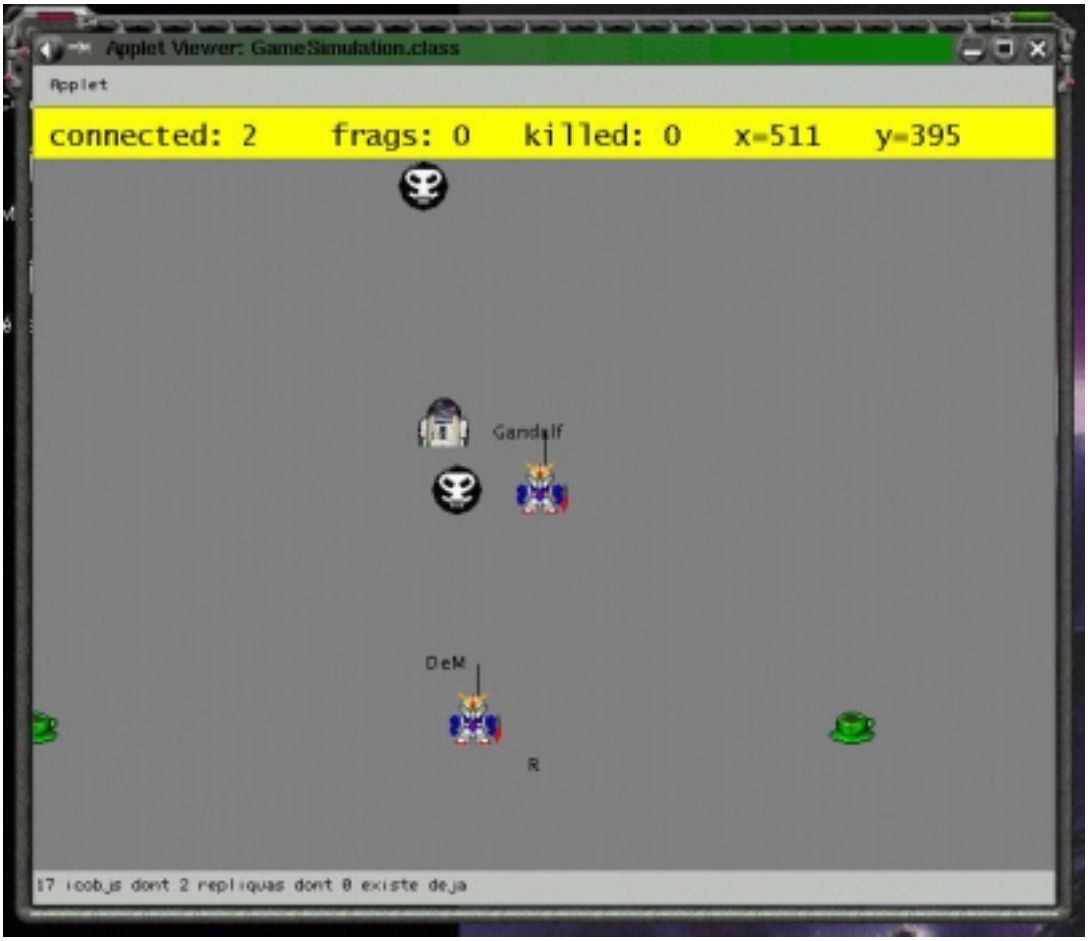


Figure 8 : Capture d'écran du jeu

1. Principe du jeu

Le jeu consiste en une sorte de bomberman (cf. Figure 8). Chaque participant a son propre avatar dans le monde. Son avatar est une icône (au choix du participant) et un trait représentant le canon du joueur. L'avatar a 2 moyens pour se déplacer :

- Soit il envoie une bombe dans la direction vers laquelle pointe son canon et il reçoit alors une impulsion identique a la bombe mais en sens inverse.
- Soit il reçoit simplement l'impulsion dans le sens inverse de la direction du canon (le canon est dirigé par l'intermédiaire des touches fléchées).

Le mouvement de l'avatar est ensuite inertiel tout en rebondissant sur les bords du monde. Il y a tout de même un frottement au sol qui ralentit l'avatar et on ne peut envoyer une bombe qu'une fois tous les N instants (pour nos tests, nous avons paramétré N à 10 instants) L'avatar gère aussi les collisions avec les autres avatars et avec les bombes qui sont dans le monde. Le but du jeu est évidemment de détruire les avatars des autres participants par l'intermédiaire des bombes, sachant que la bombe explose au bout d'un certain nombre d'instant (pour notre petit jeu, c'est 50 instants) et quand la bombe explose, elle détruit en même temps tous les avatars qui se trouvent dans sa zone d'influence.

Tous les comportements qui ont été cités, c'est-à-dire le comportement inertiel, celui de rebond sur les bords du monde, celui de collision, celui de lancer des bombes, sont des comportements écrit en langage réactif.

2. L'architecture

Le choix pour l'architecture du jeu a bien évidemment été une architecture distribuée, puisque le projet PING a pour but de supporter un nombre massif de joueurs. Bien évidemment, le jeu réalisé n'a pas été créé pour montrer ses performances sur la plate-forme PING, mais surtout pour montrer les comportements qu'il est possible de réaliser.

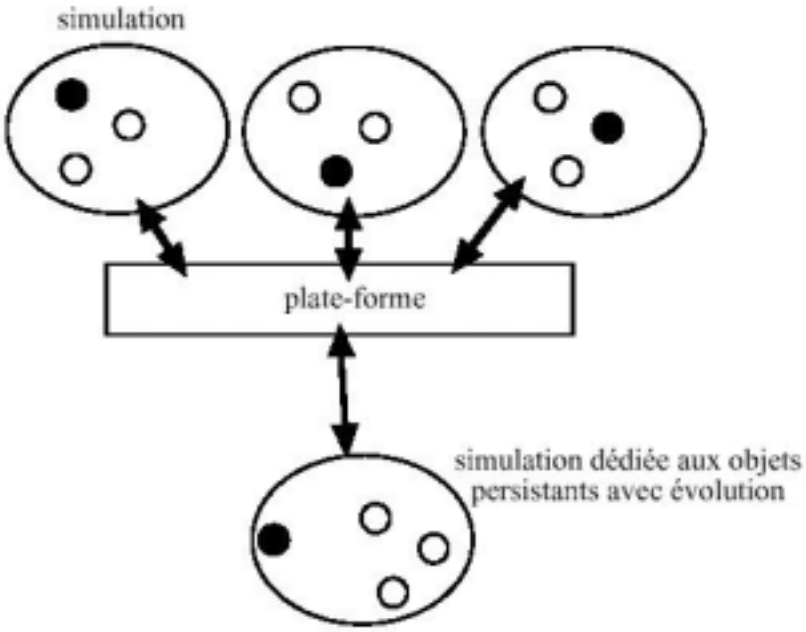


Figure 9 : Architecture du jeu

Pour l'instant, le jeu tourne sur Java RMI puisqu'il n'y a pas encore de version de la plate-forme. Donc, à l'heure actuelle, le protocole qui est sous le jeu est un protocole d'architecture centralisée. Le but principal du jeu, pour l'instant, c'est de monter que tous les calculs sont effectués par les différents clients.

Sur la Figure 9 où l'on peut observer l'architecture utilisée pour ce jeu, les ronds noirs correspondent aux répliquas maîtres et les ronds blancs aux répliquas esclaves.

Pour ce qu'il en est des bombes, elles sont gérées par, ce qu'on appelle, le serveur de persistance. Ce choix a été effectué pour que, dans le cas où la simulation de celui qui a envoyé la bombe se plante, la bombe continue à avoir un comportement cohérent et qu'elle ne disparaisse pas en même temps que le répliqua maître qui l'a créé. Tout ce qui se trouve dans le monde et qui est en dehors de ce que voit le joueur n'est pas traité par la simulation, ce qui permet d'alléger les calculs. Le problème c'est que la bombe a un comportement bien à elle et qu'elle peut très bien sortir du champ de vision de la simulation. Donc un serveur de persistance existe pour faire évoluer tous les répliquas maîtres des objets persistants, donc, dans le cas du jeu, des bombes. Dans le cas du serveur de persistance, il n'a pas à se charger de toute la partie graphique, mais par contre il doit gérer l'ensemble du monde, puisque c'est le champ d'action des bombes.

3. Les communications

Hormis les communications pour la connexion, les seules communications qui existent vont du répliqua maître vers tous les répliquas esclaves comme nous pouvons le voir sur la Figure 10. Evidemment, actuellement, toutes ses informations passent au niveau du serveur RMI puisqu'il n'y a pas encore de version de la plate-forme PING. Donc, il n'y a aucune communication réseau entre plusieurs répliquas différents. Le seul endroit où il peut y avoir des communications entre les différentes entités du jeu sur la simulation locale, c'est dans la machine réactive locale.

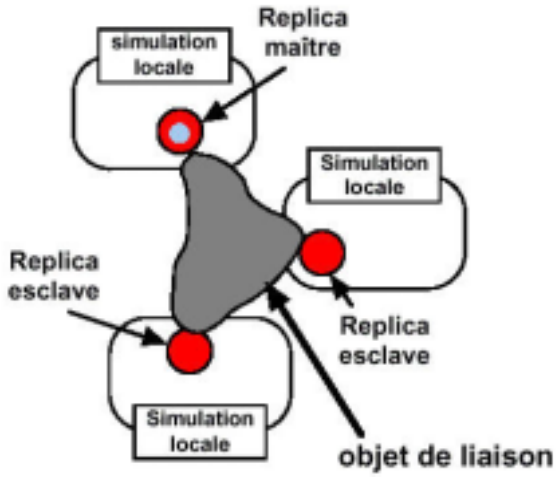


Figure 10 : Schéma de communication

Le mécanisme actuel pour mettre à jour les données des répliquas esclaves par rapport à celles du maître est :

- Tous les X instants, un message est envoyé, pour les maîtres, des simulations vers le serveur RMI. Cette mise à jour est aussi effectuée au bout d'un nombre Y d'instants ou dès qu'une nouvelle impulsion est donnée au répliqua, c'est-à-dire dès que le joueur appuie sur une touche de déplacement .
- Le serveur RMI contient une table de tous les entités qui existent dans le monde virtuel et donc il met à jour ses informations.
- Tous les Z instants, les répliquas esclaves récupèrent, au niveau du serveur RMI, les informations correspondantes à leur maître.

Le fait d'avoir différents paramètres pour les différentes mises à jour permet d'avoir une grande liberté pour paramétrer au mieux la simulation.

L'objet de liaison décrit sur la Figure 10 est donné en fait par la plate-forme PING et est « en gros » une adresse multicast. Un des intérêts à avoir un canal de communication par objet est que ça permet d'accélérer le traitement des informations concernant chaque entité et de ne pas avoir un seul canal pour toutes les informations qui pourraient s'encombrer par des informations qui nous seront pas forcément utiles. On peut aussi remarquer qu'avec cette technique, on ne recevra que les informations nécessaires à notre simulation puisqu'on ne sera connecté qu'aux adresses multicast concernant notre zone.

4. La complexité

Pour ce jeu, la principale complexité vient du comportement et surtout de calculs engendrés par le comportement de collision. Le but de la plate-forme étant d'offrir le moyen

de jouer en très grand nombre, le nombre d'entités dans le monde augmente de la même manière, donc devient vraiment le comportement le plus coûteux.

Dans le cas d'une architecture centralisée, toutes les collisions sont calculées au niveau du serveur, donc, pour N entités dans le monde, le nombre de collisions a géré est de N*(N-1)/2, car pour chaque entité, il y a N-1 collisions possibles et puisqu'elles sont toutes gérées au même endroit, il n'est pas nécessaire de les calculer deux fois. Donc on a une complexité en Θ(N_).

Dans le cas d'une architecture distribuée, la gestion des collisions sur chacune des simulations distribuées est de N – 1, car il n'y a que le maître qui gère les collisions (les répliquas esclaves ont uniquement un comportement dégradé par rapport à celui de leur maître), donc la complexité est en Θ(N), ce qui est bien meilleur que la complexité dans le cas d'une architecture centralisée. Si on calcule la complexité globale, on trouve qu'elle est égale à N*(N-1), donc de l'ordre Θ(N_), ce qui est plus grand que dans le cas d'une architecture centralisée, car la même collision est gérée deux fois (une fois par chaque simulation qui se charge du maître des entités concernées). Mais ceci n'est pas grave puisqu'elle est répartie sur plusieurs sites, ce qui confirme donc qu'en partant d'une architecture distribuée, on peut augmenter le nombre de participants.

5. Quelques points importants

Voici quelques points importants qui précisent l'état du jeu à mon arrivée. Tout ce qui est décrit dans cette partie sera des fonctions qui seront modifiées par la suite. Dans un premier temps, on peut voir ce qui se passe quand les informations du répliqua esclave sont mises à jour. Donc, au moment de la resynchronisation, le répliqua esclave modifie sa position actuelle et se positionne à la position où son maître se trouvait lors de la dernière mise à jour des informations et il prend aussi le même vecteur vitesse, comme cela est décrit ci-dessous :

X = info.X
Y = info.Y
speedX = info.speedX
speedY = info.speedY

Avec **X** et **Y** les coordonnées actuelles du répliqua esclave
speedX et **speedY**, les vitesses pour le répliqua esclave
info.X et **info.Y**, les coordonnées du répliqua maître au moment de sa dernière mise à jour
info.speedX et **info.speedY**, les paramètres de la vitesse du maître

Un des principaux comportements que possède le répliqua esclave, c'est le comportement inertiel. Ce comportement est le mécanisme de Dead-reckoning du jeu, c'est-à-dire que c'est lui qui donne le mouvement du répliqua esclave entre deux resynchronisations. Ce mécanisme, décrit ci-dessous, est exécuté à chaque instant consiste en une réduction de la vitesse due aux forces de frottement du sol suivie d'une modification de la position selon le nouveau vecteur vitesse.

speedX = (1 – absorption) * speedX
speedY = (1 – absorption) * speedY
X = X + speedX
Y = Y + speedY

Avec **absorption** le ratio de frottement du répliqua sur la surface du jeu

VII. Problèmes de cohérence

Pour ce faire, on va étudier de plus près le jeu réalisé par l'équipe MIMOSA et que l'on peut trouver à l'adresse [14] et on va lister les différents cas d'incohérence que l'on peut relever dans un monde aussi simple (Il est évident que plus le monde sera complexe, plus le nombre d'incohérences possibles sera grand). Pour être assez généraliste, les problèmes d'incohérences proviennent du fait que l'un des deux réagit à une situation et que l'autre n'y est pas confronté dû à des décalages de position.

1. Types d'incohérence

La première incohérence que l'on peut noter est évidemment la simple différence de position (dont on peut voir un exemple sur la Figure 6) qu'il peut y avoir entre un maître et son répliqua esclave. Tant que cela n'engendre pas d'autres incohérences sur les autres simulations, cela est tout de même assez minime et peut être réglé assez facilement.

La seconde incohérence qu'il faut prendre en compte est la différence de position qui engendre certains comportements sur un site, comportements qui ne seraient pas répercutés sur les autres sites. Par exemple, si on rentre en collision avec un autre répliqua sur notre simulation et que le maître correspondant à ce répliqua ne rentre pas en collision avec notre répliqua esclave sur sa simulation, la collision ne sera donc détectée que par un des deux partis, cela entraîne des trajectoires complètement incohérentes que ce soit pour l'un ou pour l'autre (c'est-à-dire l'un changera sa trajectoire et verra que l'autre a continué dans sa direction originale et sur l'autre simulation l'un verra un brusque revirement de trajectoire sans comprendre comment il a pu changer de trajectoire aussi rapidement).

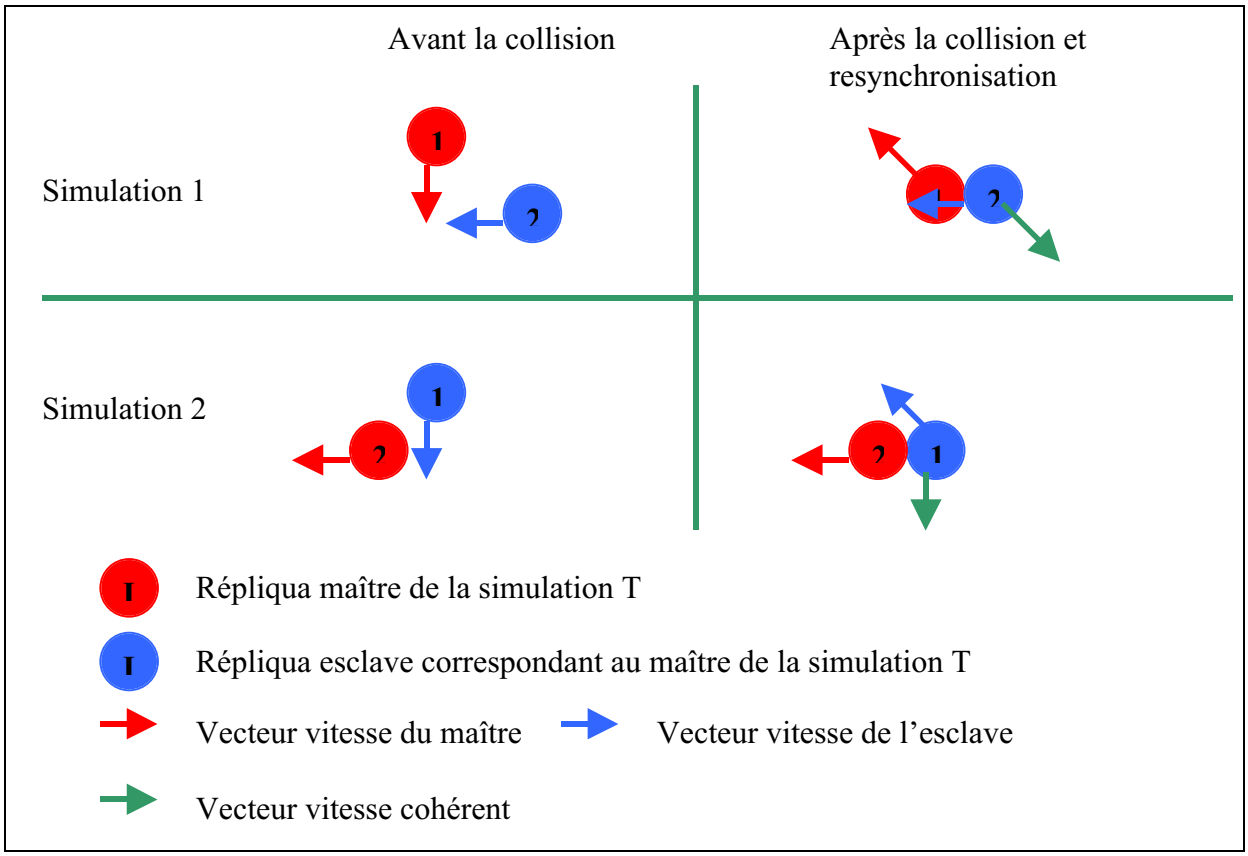
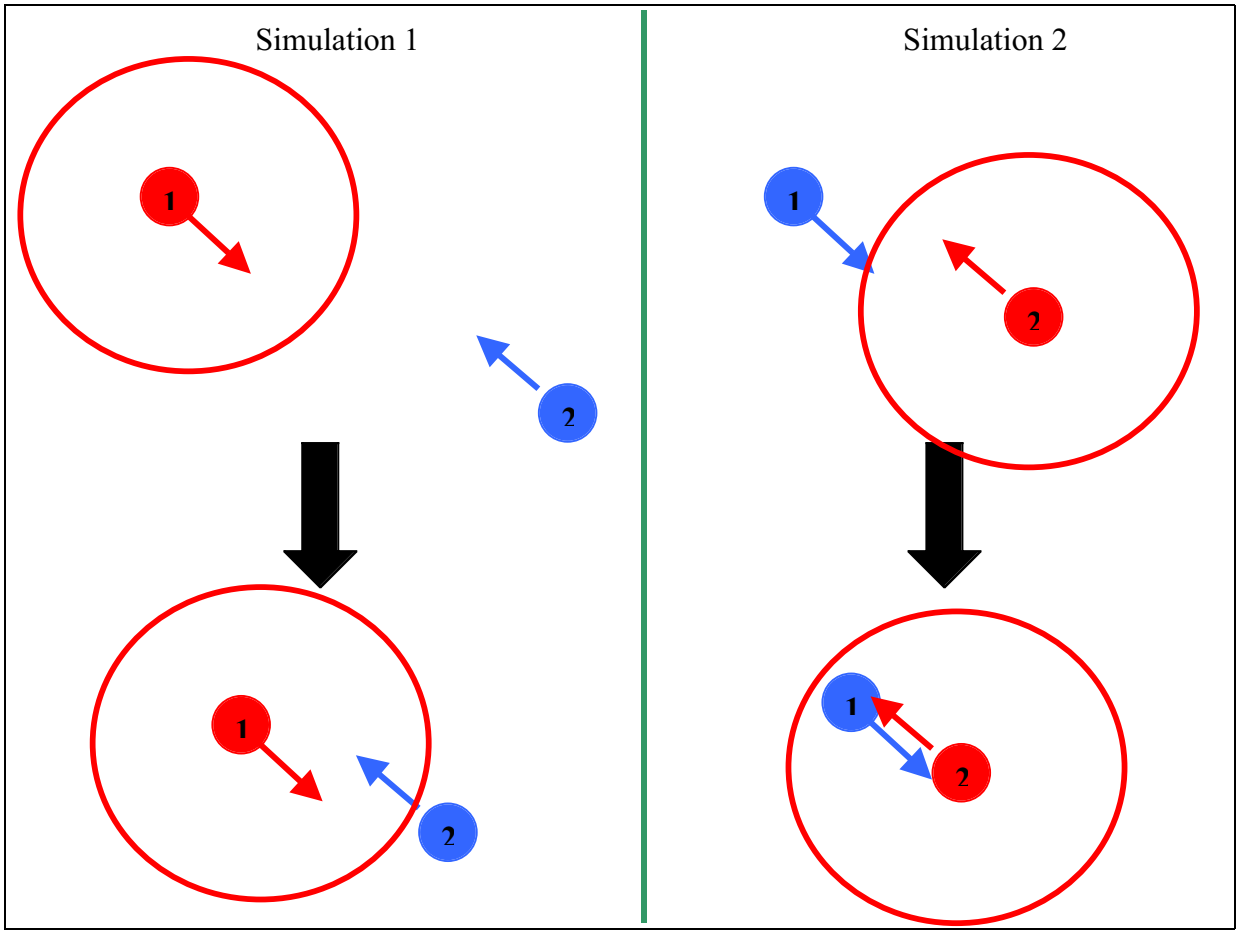


Figure 11 : Incohérence au moment de la collision

Dans la Figure 11, on voit le problème typique de cohérence au moment d'une collision. On voit dans la simulation 1, avant la collision, un problème de positionnement du répliqua esclave 2 par rapport à la position de son maître dans la simulation 2, ce qui entraîne que dans la simulation 2, on ne se rend pas compte de la collision alors que dans la simulation 1, c'est le cas. Il s'avère donc que le maître de la simulation 1 a un comportement correct dans sa simulation ce qui n'est pas le cas de l'esclave de la simulation 1, et inversement dans la simulation 2. La flèche verte représente le vecteur vitesse que devrait prendre le répliqua esclave par rapport à la situation de la simulation correspondante pour que la situation soit cohérente.

Une autre incohérence que l'on peut noter est au niveau des bombes. Si un maître envoie une bombe sur un autre répliqua dans sa simulation et voit qu'il l'a touché et qu'à l'inverse le maître correspondant au répliqua a évité cette bombe dans sa simulation, il y a incohérence totale pour savoir à qui il faut donner raison, surtout que les bombes étant gérées par la simulation virtuelle, les positions de ses répliqua esclaves sur les deux simulations concernées peuvent être aussi inexactes sur l'une ou sur l'autre des simulations.. Dans la version actuelle, c'est le répliqua maître qui prend la décision de savoir s'il a ou non été touché.

On peut aussi noter l'apparition d'incohérence temporelle. En effet, comme on l'a signalé auparavant et comme on peut le voir sur la Figure 7, les simulations n'ont qu'une vue partielle du monde où l'avatar évolue. Donc, il peut arriver qu'une des simulations voit l'arrivée d'un répliqua qui était jusqu'à ce moment hors de son champ de vision, alors que la simulation qui gère le maître de ce répliqua ne voit pas l'entrée du répliqua du premier ou, dans le meilleur des cas, ne la voit pas au même moment (Il faut savoir que le répliqua maître géré par la simulation est toujours au centre de cette même simulation, et donc les champs de vision de tous les maîtres sont identiques).



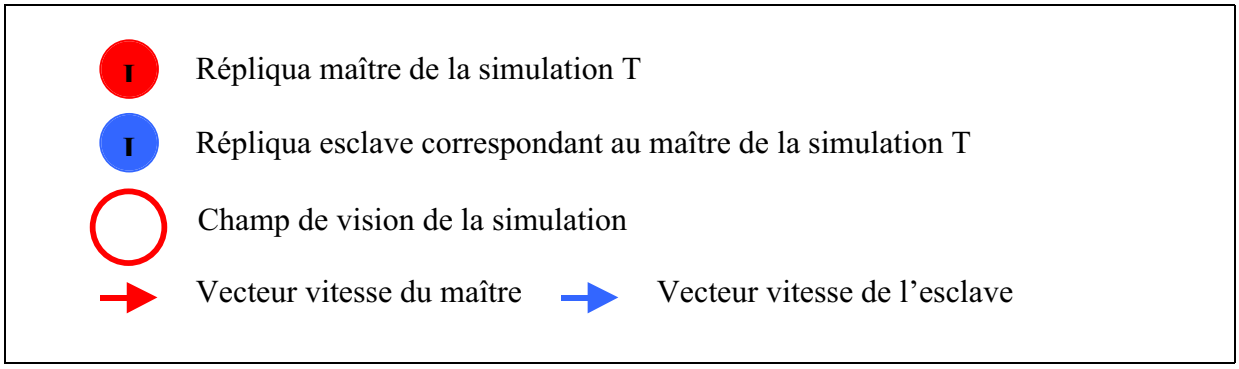


Figure 12 : Incohérence temporelle

Dans la Figure 12, on voit le problème typique de cohérence temporelle sur lequel on peut tomber, c'est-à-dire que l'un des deux voit apparaître l'autre dans sa simulation alors que ce n'est pas le cas pour l'autre. Ce cas de figure peut apparaître quand une des simulations est bien plus rapide que l'autre.

Le problème des incohérence temporelle est très important dans le cas des objets qui ont une courte durée de vie, comme par exemple, les bombes. Puisque ces dernières disparaissent très vite du monde, si elles n'apparaissent pas dans une simulation alors qu'elle doit y effectuer des modifications, ces modifications ne seront pas effectuées et il pourra y avoir des problèmes de cohérence encore plus important.

2. Causes de ces incohérences

Les causes d'incohérences sont assez aisées à comprendre. Dans un premier temps, on peut tout de suite parler des délais engendrés par les communications réseaux. Pour ce point, nous ne pouvons intervenir puisque nous sommes limités à travailler uniquement à un niveau comportemental dans l'application, car c'est la plate-forme PING qui se charge des problèmes ayant un rapport avec le réseau.

Une autre cause à mettre en avant et sur laquelle nous pouvons par contre influer c'est la mise en place d'une horloge commune ou du moins, puisque nous avons bien vu que la technique dans [5] est bien trop gourmande en ressource, mettre en place des mécanismes d'équivalence entre les horloges des différentes machines. Mais nous reviendrons sur ce point dans la partie suivante.

Enfin, il faut noter que la puissance et la charge de la machine qui fait tourner la simulation jouent un rôle assez important, car plus la machine est rapide, plus la machine réactive qui tourne dessus avance rapidement. Il est clair aussi que le jeu favorisera plutôt les machines rapides, du moins jusqu'à un certain seuil, puisqu'il y a un paramètre au niveau de la machine réactive qui permet de limiter le nombre d'instants par seconde. Par contre, il n'y a pas de bornes inférieures pour les machines lentes.

VIII. Cohérence : solutions proposées

Ces mécanismes sont des mécanismes à mettre en place au niveau du comportement des répliquas et non à des niveaux réseau en faisant des protocoles ou en utilisant certains protocoles qui nous donneraient des mécanismes pour amoindrir le niveau d'incohérence comme on pouvait le voir en [4] avec l'utilisation de NTP pour avoir une horloge globale. De plus, les mécanismes que je dois mettre en place doivent être assez simple et donc une utilisation d'un mécanisme de « bucket » n'est pas de mise dans notre projet. Ce mécanisme sera d'ailleurs sûrement mis en place au niveau de la plate-forme et ne dépend donc pas de notre travail. Plusieurs approches sont possibles, mais il faut trouver celle dont l'efficacité est réelle dans la majeure partie des cas.

Il faut tout de même rappeler qu'il aurait été possible de rajouter tous les comportements du maître au répliqua esclave, mais le principe étant que l'esclave doit avoir un comportement dégradé par rapport à celui de son maître, il fallait que les mécanismes ne soient pas trop gourmands en terme de calcul.

On pourra faire directement, au moment de la resynchronisation (ce qui est d'ailleurs le cas actuellement) des recalages brutaux de position du répliqua esclave par rapport à celle de son maître, mais cela engendre des problèmes de « cohérence » quant à la vision qu'en ont les autres simulations sur le déplacement de l'avatar. C'est à la fois un problème de fluidité de déplacement, mais aussi un problème de cohérence puisqu'il a tendance à faire des aller-retour autour de la position du maître.

Donc, une des solutions évidentes pour améliorer l'état d'incohérence, c'est d'améliorer justement ces mécanismes de dead-reckoning. Mais cette technique ne suffit pas à éviter toutes les incohérences, car même en affinant les trajectoires obtenues par ces mécanismes, il y a forcément des différences de rapidité entre les différentes simulations et il faut en tenir compte. Plusieurs paramètres rentrent en jeu dans la mise en place d'un mécanisme de dead-reckoning :

- ***Les délais de resynchronisation*** : selon le temps entre chaque resynchronisation, l'état d'incohérence peut être plus ou moins importants selon les règles d'anticipation. Par contre, si on diminue trop le temps de resynchronisation, on risque de submerger le réseau de message et finalement obtenir des délais plus importants.
- ***La taille de la zone d'influence de chaque avatar*** : autour de chaque Icobj est défini une zone d'influence sur laquelle l'icobj peut intervenir ou du moins influer. Il serait donc possible de faire varier la taille de cette zone, ce qui permettrait de diminuer la précision avec laquelle, par exemple, on pourrait détecter une collision, ou mettre en place plusieurs zones concentriques qui serait un bon principe pour le problème des bombes. La zone dans laquelle on pourrait toucher le répliqua serait plus grande, mais elle ne détruirait pas forcément le répliqua, il y aurait un effet croissant en se rapprochant du centre de la zone d'influence.

La première chose qui a été faite, a été de diminuer, au niveau de la machine réactive, le nombre d'instants maximal qui pouvait s'exécuter par seconde. Nous avons préféré diminuer ce paramètre, plutôt que de diminuer les instants séparant deux resynchronisations, car il ralentissait en même temps le jeu et qu'il était difficile pour un Pentium 200 de suivre un G4 et donc il était possible que cette différence joue sur les différents tests effectués.

Un autre point important, voire le plus important du mécanisme, que j'ai mis en place, c'est d'ajouter aux données de mise à jour le numéro d'instant auquel a été faite la mise à jour. Cela permet, au fur et à mesure des resynchronisations, de déterminer à chaque fois la rapidité de l'autre machine et de pouvoir déterminer la dérive que notre simulation subie. Evidemment, ces valeurs étaient différentes selon le répliqua pris en compte. Donc on a décidé de mesurer l'écart d'instant entre chaque maître et son répliqua dans chaque simulation et la mesure d'écart entre deux resynchronisations. Ces données ont permis de mettre en place une sorte de mécanisme d'apprentissage, c'est-à-dire qu'au début de la simulation, on ne connaît strictement rien et donc les corrections de trajectoire se font moins brutalement que le recalage direct mais avec quelques petites saccades et au fur et à mesure, il obtient des informations de plus en plus précise sur la vitesse des autres simulations.

1. Solution 1 : ajout de l'écart moyen

Le mécanisme précédent a d'abord donné lieu à de premières modifications. En fait, en calculant la dérive d'une simulation par rapport à une autre, une première idée a été de calculer la différence moyenne d'écart de distance entre l'esclave et son maître et puisqu'on a la durée moyenne entre deux resynchronisations, il était possible de modifier la vitesse de façon suivante :

speedX = info.speedX + (EcartMoyenX / EcartMoyenSynchro)
speedY = info.speedY + (EcartMoyenY / EcartMoyenSynchro)

Avec **speedX** et **speedY**, les vitesses pour le répliqua esclave
Info.speedX et **info.speedY** les vitesses pour le répliqua maître distant
EcartMoyenX et **EcartMoyenY** les écarts moyens de distance sur les axes X et Y
Et **EcartMoyenSynchro**, l'écart moyen en nombre d'instants entre 2 resynchronisations

Cette modification permettait de reporter l'écart moyen de distance et de le reporter sur la vitesse que doit prendre son esclave.

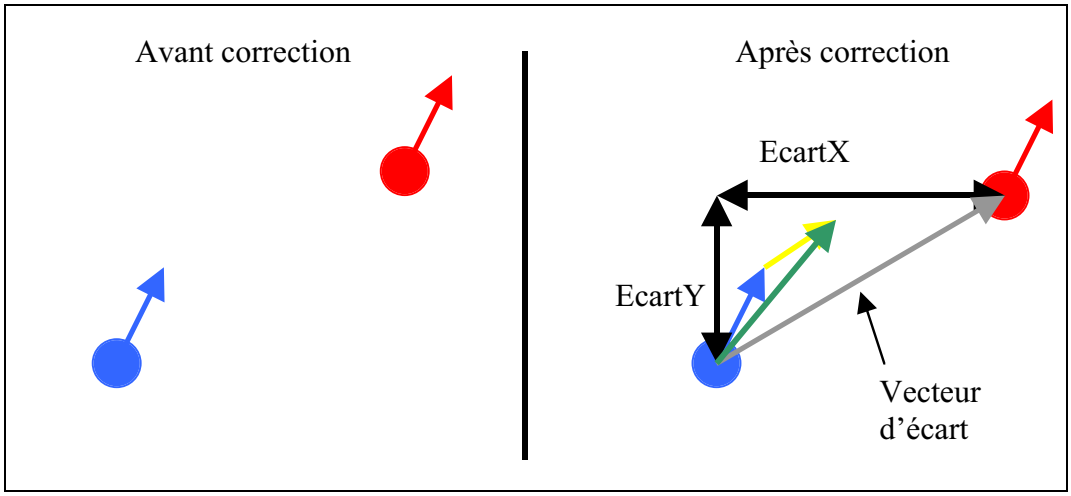


Figure 13 : Modification de la première solution

Sur la Figure 13, on peut observer les modifications apportées par cette première solution. Avant la correction, le vecteur vitesse du répliqua esclave (en bleu) est simplement mis à jour sur celui de son maître (en rouge). Après la correction, on part de la mise à jour du vecteur vitesse, puis on prend l'écart moyen en X et en Y et on trouve un vecteur d'écart. A partir de ce vecteur d'écart (en gris), on obtient un nouveau vecteur de vitesse (en jaune), par

l'intermédiaire de la formule ci-dessus, à rajouter au vecteur vitesse normal pour rattraper l'écart entre le maître et l'esclave, on obtient donc un vecteur vitesse (en vert) qui est la somme du vecteur de vitesse du maître et du vecteur vitesse de rattrapage.

2. Solution 2 : prise en compte de la différence de rapidité

Le problème de cette première méthode était qu'il y avait vraiment de grandes amplitudes d'écarts d'avance ou de retard donc l'écart moyen de distance n'était pas un très bon indicateur ou du moins il n'était pas un très bon modifieur. Donc la seconde idée a été de modifier la vitesse du répliqua esclaves correspondant, c'est-à-dire si on calcule qu'une simulation va 1,5 fois plus vite que celle sur laquelle on se trouve, on va multiplier le vecteur vitesse par 1,5. Cela permettait d'avoir la même vitesse du répliqua à l'écran et donc un mouvement relativement cohérent, on avait donc les fonctions suivantes :

SpeedX = info.speedX * DeriveMoyenne
SpeedY = info.speedY * DeriveMoyenne

Avec **DeriveMoyenne**, le rapport moyen entre les vitesses des 2 simulations concernées

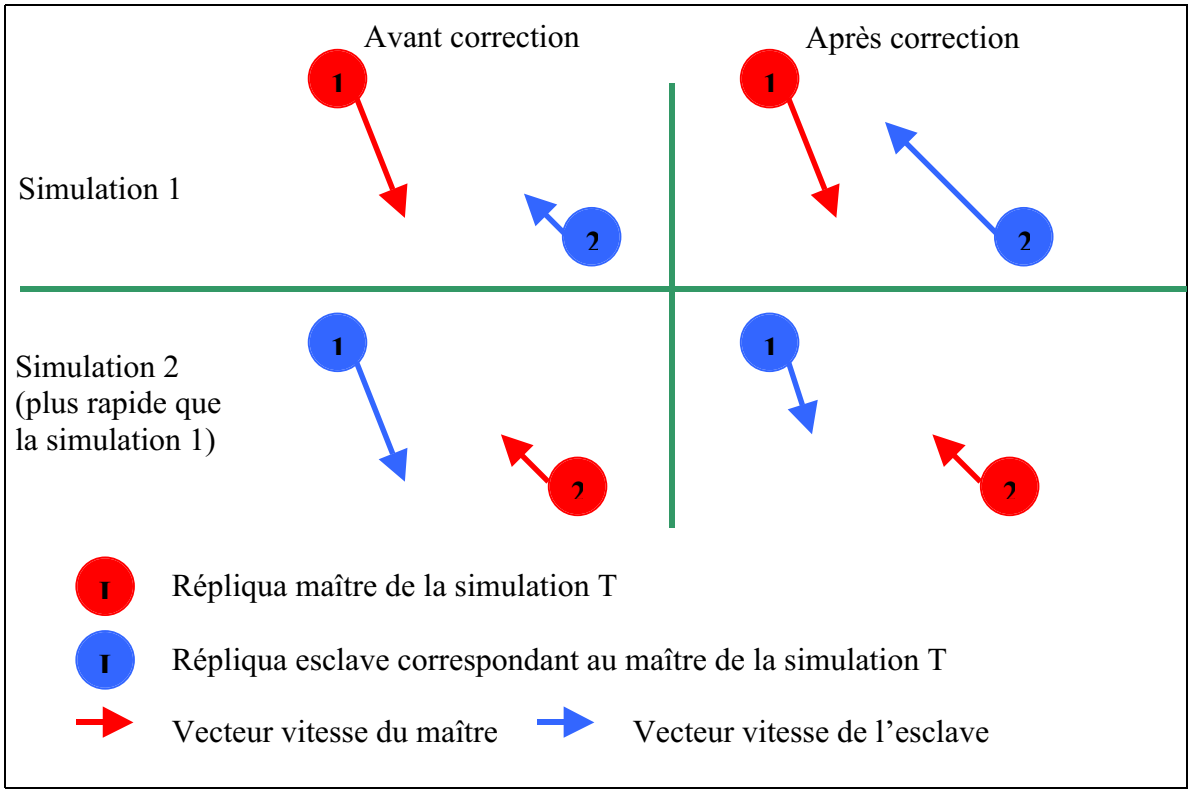


Figure 14 : Modifications apportées par la deuxième solution

Sur la Figure 14, on peut voir les corrections apportées par la deuxième solution. Avant la correction, la différence de rapidité entre deux simulations ne changeait rien pour le comportement de l'esclave. Après la correction apportée par ma solution, on voit une modification des vecteurs vitesse des répliquas esclaves en fonction de la simulation sur laquelle il tourne.

3. Solution 3 : estimation de la prochaine position

Le problème de cette solution, c'est que les durées entre les resynchronisations pouvant être assez variables, le répliqua esclave pouvait, si la durée avant la prochaine resynchronisation était plus grande que la moyenne, aller trop loin et donc le mouvement de l'esclave partait bien dans des directions parallèles à celle du maître, ce qui était assez flagrant et la position réelle du maître ne pouvait jamais être rejointe, donc, j'ai testé une nouvelle solution. Le principe en était de calculer ou plus précisément d'estimer la prochaine position du maître et une fois qu'on avait cette position, on pouvait calculer le vecteur de vitesse pour rejoindre cette position.

EstimationX = info.X + (info.speedX * EcartMoyenSynchro)
EstimationY = info.Y + (info.speedY * EcartMoyenSynchro)
speedX = (EstimationX – X) / EcartMoyenSynchro
speedY = (EstimationY – Y) / EcartMoyenSynchro

avec **info.X** et **info.Y** les coordonnées du maître lors de sa dernière mise à jour
X et **Y** les coordonnées actuelles du répliqua esclave
EstimationX et **EstimationY** les coordonnées estimées à la prochaine resynchronisation du maître

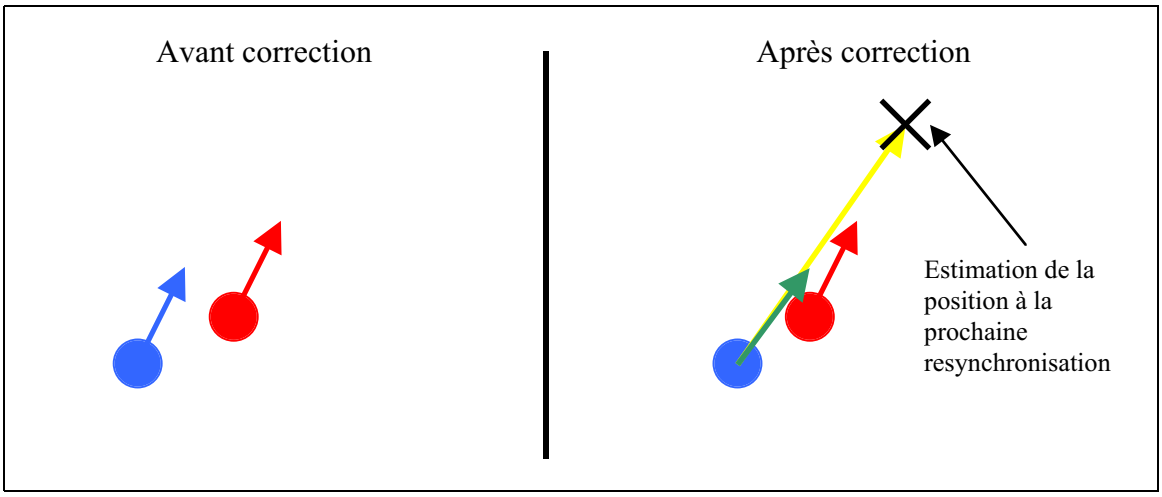


Figure 15 : Modification de la première solution

Sur la Figure 15, on peut voir le fonctionnement de ma troisième proposition de correction. On estime la position du maître à la prochaine resynchronisation (croix noire), puis on estime la distance à parcourir jusqu'à ce point (en jaune), puis on détermine le vecteur vitesse à adopter (en vert).

4. Solution 4 : rattrapage d'écart

Le problème de cette dernière méthode réside aussi dans les variations de durées entre deux resynchronisations et donc ce qu'il se produisait, c'est qu'au fur et à mesure l'écart avec la vraie position du maître est de plus en plus grand, donc la vitesse augmente d'autant plus pour rattraper cet écart, mais ça crée un écart encore plus grand et ça devient un « cercle vicieux ». Donc j'ai repris en partie la solution où on modifie le vecteur vitesse par rapport à l'écart de rapidité entre les simulations ce qui donnait déjà de relativement bons résultats et pour régler le problème de rattraper la position du maître, il a fallu ajouter une modification de la fonction inertielle des répliquas esclaves

Pour rattraper la position réelle, on calcule l'écart de distance au moment de la resynchronisation et on espace le rattrapage de cet écart sur plusieurs instants en modifiant le comportement inertiel de la manière suivante :

```
speedX = (1 - absorption) * speedX
speedY = (1 - absorption) * speedY
Si  EcartX - VitesseX_ratt > 0
  Alors
          X = X + VitesseX_ratt + speedx
          EcartX = EcartX - VitesseX_ratt
  Sinon
        X = X + speedX + EcartX
        EcartX = 0
Finsi
Si Ecarty - VitesseY_ratt > 0
  Alors
          Y = Y + VitesseY_ratt +  speedY
          EcartY = EcartY - VitesseY_ratt
  Sinon
          Y = Y + speedY + EcartY
          EcartY = 0
FinSi
```

Pour cette dernière solution, les paramètres **VitesseX_ratt** et **VitesseY_ratt** sont calculés au moment de la dernière resynchronisation, ils représentent respectivement la distance à rattraper sur l'axe des X et celui des Y divisée par la durée moyenne entre deux resynchronisations. Donc le répliqua va rattraper son retard au fur et à mesure entre deux resynchronisations et il garde un mouvement fluide, ce qui permet d'éviter les problèmes où la collision ne se faisait pas parce que le répliqua avait fait un saut de recalage de position ou ceux où on croyait que notre bombe avait détruit le répliqua. Cette technique n'est évidemment pas efficace dans 100% des cas, mais elle permet déjà d'améliorer de façon assez sensible la vision que le joueur a du jeu. Dans la prochaine partie de ce rapport, nous pourrons voir les résultats obtenus pour cette dernière solution.

Une autre modification que l'on pourrait apporter est de mettre en place des compromis (genre de mécanisme que l'on peut trouver dans Quake), c'est-à-dire de mettre en place un mécanisme qui crée un « flou artistique » autour de l'état d'incohérence pour le cacher ou au moins l'atténuer. Par exemple, les techniques usuelles pour essayer d'amoindrir l'impression d'incohérence que pourrait avoir un joueur pour savoir si une bombe a détruit un répliqua ou non, sont de donner, à chaque avatar, des points de vie et de définir différentes aires de dégâts autour de la bombe qui enlève plus ou moins de dégâts. Ce mécanisme contente en général les deux parties : d'un côté, celui qui croyait l'avoir détruit a au moins la satisfaction d'avoir enlevé des points de vie et d'un autre côté, celui qui croyait avoir évité la bombe restera tout de même en vie. Ce mécanisme ne diminue pas vraiment l'état d'incohérence, mais il le rend acceptable par les joueurs.

IX. Résultats

1. Implémentation des solutions

Une fois que la notion d'incohérence avait été définie et que l'on avait trouvé des solutions aux différents types d'incohérence trouvés dans le jeu, il a fallu mettre en place les différentes solutions. La première chose à faire a été d'instrumenter le code du jeu pour pouvoir obtenir les différents paramètres dont on avait besoin pour pouvoir influer correctement sur les comportements des répliquas esclaves. Pour cela, il nous a fallu loguer plusieurs types d'informations sur chaque simulation :

- à chaque instant, la durée de l'instant pour pouvoir avoir une première vision des différences de vitesse entre chaque machine.
- à chaque instant, la position du répliqua maître
- à chaque instant, les positions des répliquas esclaves
- à chaque resynchronisation, les écarts de position, les écarts de vitesse entre les différentes simulations.
- les instants où sont apparus et disparus les répliquas esclaves qui se trouvent dans le champ de vision du maître de la simulation locale.
- l'instant et la position où a eu lieu une collision entre le maître de la simulation locale et un quelconque répliqua esclave.

Ensuite, il a évidemment fallu modifier les comportements en cause pour voir si les modifications que je propose sont adéquats et améliorent bien les situations d'incohérence. Donc, après quelques modifications en Java, j'ai pu enfin me lancer dans les tests.

Pour les besoins des tests, il a fallu rajouter quelques petites fonctionnalités au jeu pour bien mettre en valeur certains aspects des incohérence, c'est-à-dire par exemple de pouvoir se déplacer sans envoyer de bombes, car la bombe ajoutait de la complexité à la simulation ; ou aussi une petite fonctionnalité qui permet de passer du mode avec les corrections de départ qu'il y avait dans le jeu au mode corrigé que j'ai mis en place.

Il faut aussi préciser que bien évidemment les solutions proposées dans la partie précédente ont toutes été testées au fur et à mesure de leur élaboration.

2. Résultats de ces tests

Dans cette partie, nous allons montrer les résultats des tests que nous avons obtenus avec le jeu au départ sans mes modifications puis ceux que nous avons obtenus avec mes modifications. Les différences que l'on peut voir sur les graphes peuvent paraître assez minime, car les principaux problèmes que l'on pouvait remarquer était des recalages brutaux de position, ce qui a été remplacé par un rattrapage progressif de l'écart ou on pouvait aussi remarquer au départ des aller-retour autour de la position réelle du maître.

a. Contexte des tests

Tous les tests ont été effectués sur le réseau interne de l'INRIA, donc sur un réseau 100 Mbits/s. Certaines machines ont, pour le test, tourné sur un réseau 10 Mbits/s dont le serveur RMI. Il est clair que ces tests ne seront pas très probants quant à l'influence du réseau sur les comportements. Par la suite, des tests seront sûrement effectués pour mettre en avant les variations que le réseau pourraient introduire dans la simulation. Les principaux points qu'on peut mettre en avant sont les différences de puissance entre les machines et l'impact de la charge de la machine.

b. Durée des instants

Sur les deux figures suivantes, on peut voir la durée des instants sur un machine qui fait tourner le jeu sans les corrections (cf. Figure 16) et la durée des instants sur la même machine en faisant cette fois tourner le jeu avec les corrections que j'y ai apportées. On remarque que la durée n'est pas plus importante avec mes modifications et donc que les comportements que j'y ai apporté n'augmente pas la complexité générale ou de façon très légère.

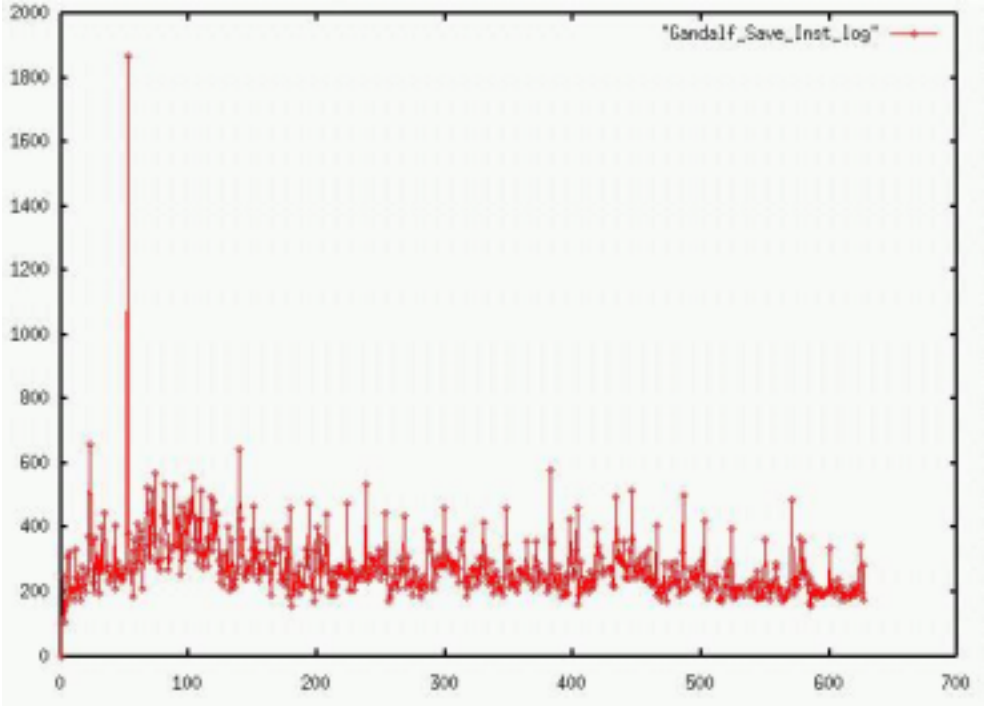


Figure 16 : Durée des instants en millisecondes sans correction

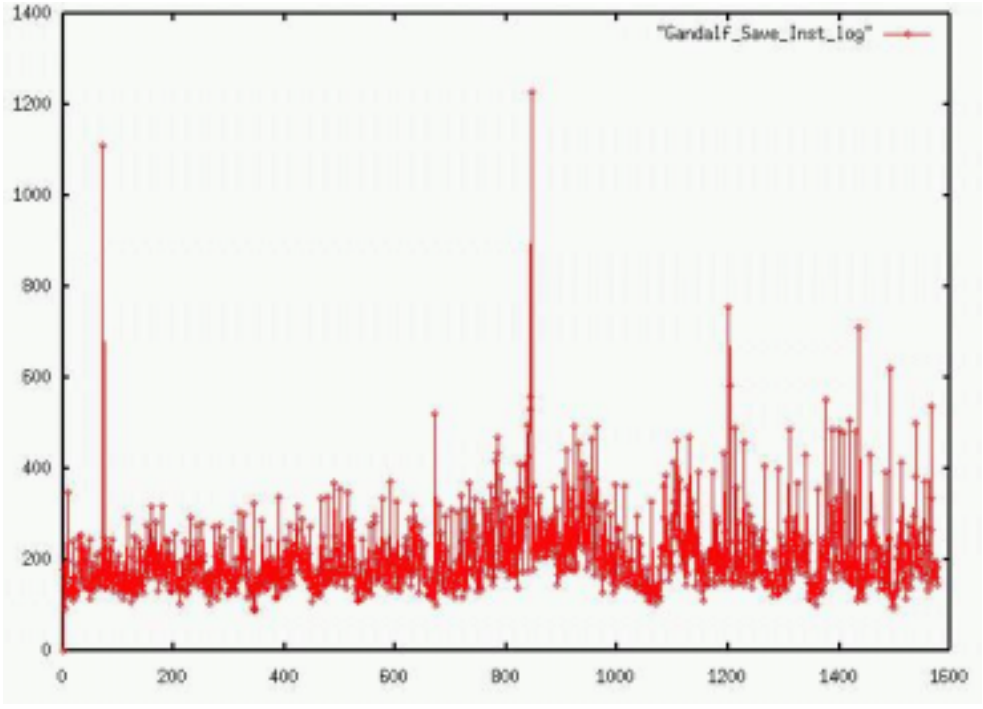


Figure 17 : Durée des instants en millisecondes avec correction

c. Tests sans modifications

Sur les deux figures qui suivent (cf. Figure 18 et 19), nous pouvons voir la trajectoire d'un répliqua maître sur l'axe des X et celle de son répliqua esclave correspondant sur une simulation distante. Sur la figure 19, on ne voit qu'une partie de la trajectoire qu'a fait le maître puisque les mesures ont uniquement lieu quand le répliqua esclave est dans le champ de vision du maître de la simulation distante. Il ne faut pas non plus faire attention à l'échelle de temps, puisqu'elle est différente selon la simulation sur laquelle la mesure est faite, il faut surtout faire attention à l'allure de la trajectoire.

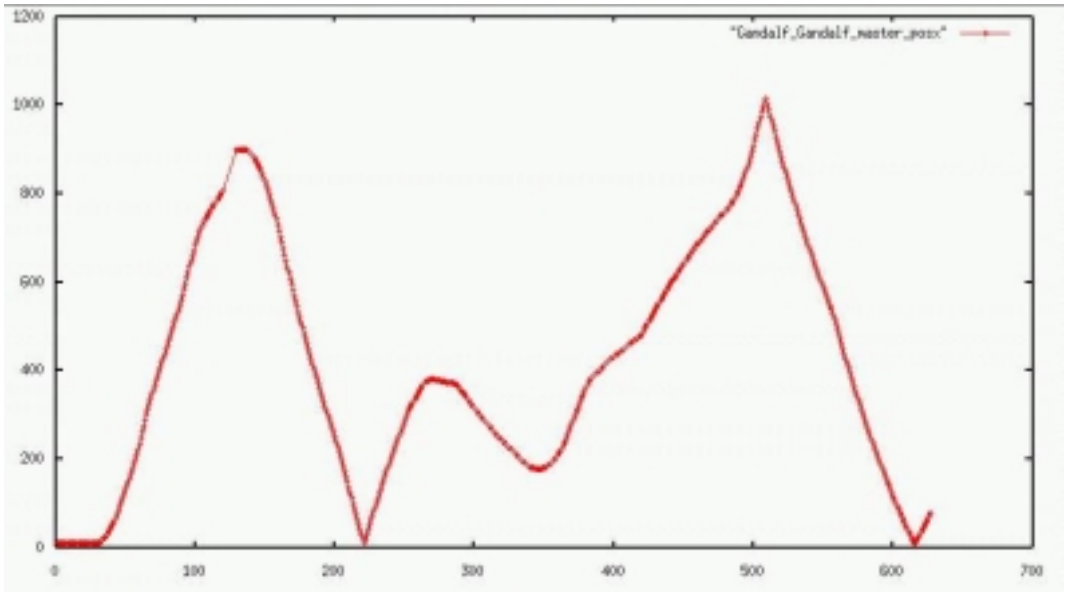


Figure 18 : Trajectoire sur l'axe des X d'un répliqua maître (sans correction)

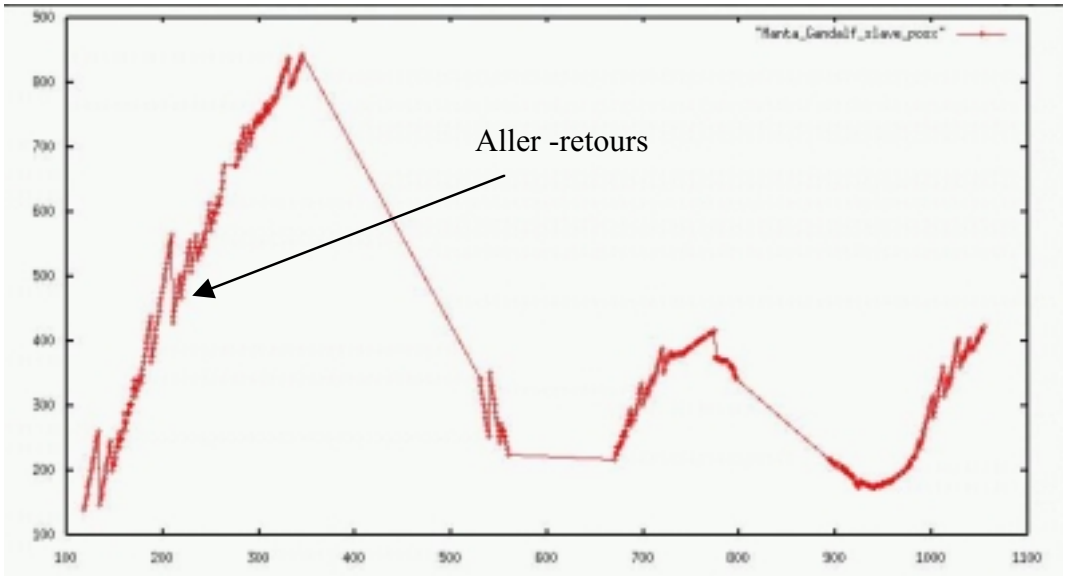


Figure 19 : Trajectoire sur l'axe des X d'un répliqua esclave (sans correction)

Sur la Figure 19, on remarque bien les aller-retours effectués par le répliqua esclave alors que la trajectoire du maître (cf. Figure 18) est bien lissée. Et c'est ces allers-retours qui entraîne des comportements incohérents.

d. Tests avec modifications

Sur les deux figures qui suivent, nous pouvons observer comme pour les Figure 18 et 19, la trajectoire d'un maître et de son esclave sur une simulation distante, mais cette fois-ci en ayant apporter mes modifications au jeu pour lisser les trajectoires et donc diminuer l'incohérence qu'il pouvait y avoir dans le jeu.

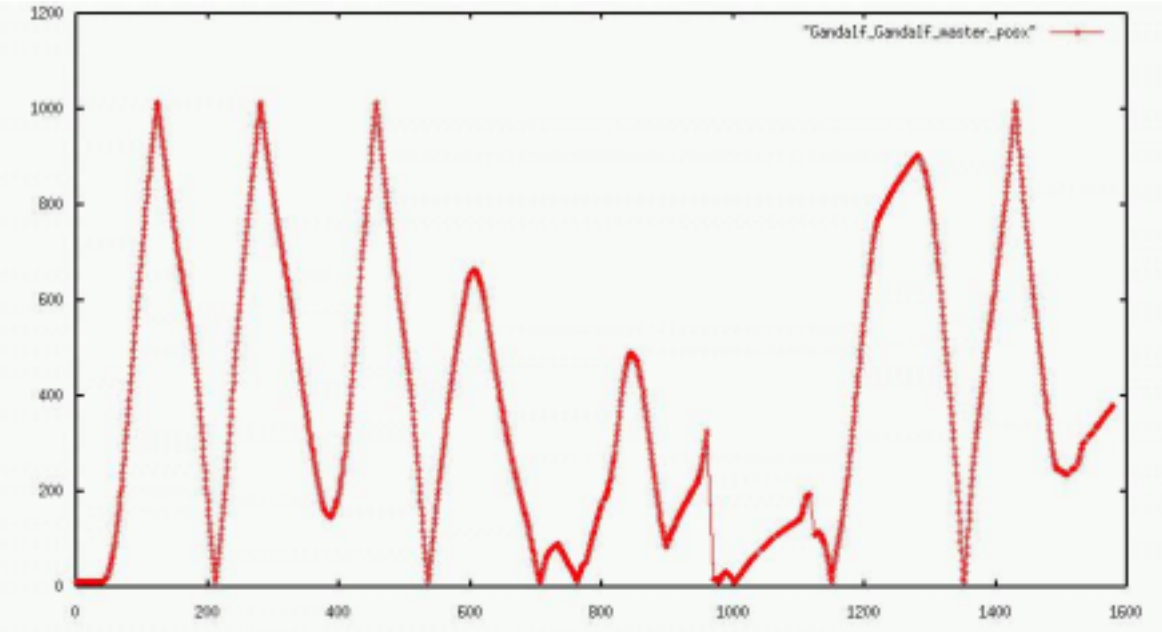


Figure 20 : Trajectoire sur l'axe des X d'un répliqua maître (avec correction)

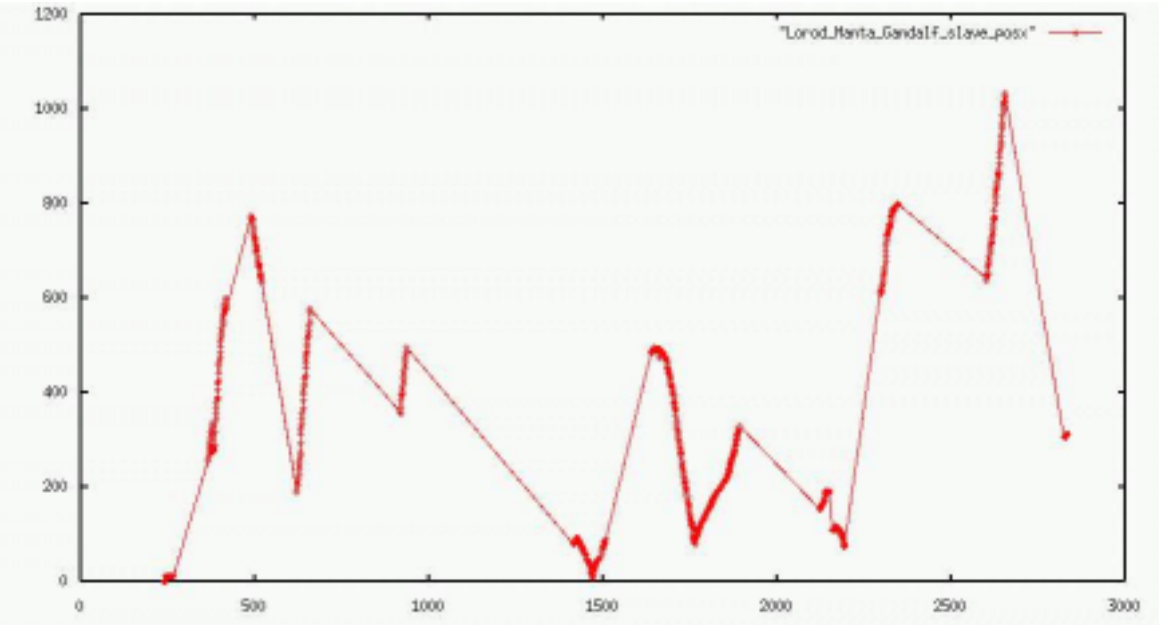


Figure 21 : Trajectoire sur l'axe des X d'un répliqua esclave (avec correction)

Cette fois-ci, on peut remarquer que le comportement que l'on pouvait voir auparavant, c'est-à-dire des allers-retours autour de la position réelle du maître. On remarque que le trajectoire de l'esclave est bien plus régulière, ce qui rend le déplacement beaucoup plus cohérent et en même temps permet de garder une cohérence bien plus grande à la simulation.

Enfin, sur la Figure 22, nous pouvons observer les différences de rapidité entre les quatre simulations. Sur ce graphe, nous pouvons voir que la simulation que nous avons pris pour exemple pour les graphes précédents (la courbe bleue) est bien plus lente que toutes les autres simulations, ce qui montre l'efficacité de mes modifications même si on se trouve face à des machines de puissance différente.

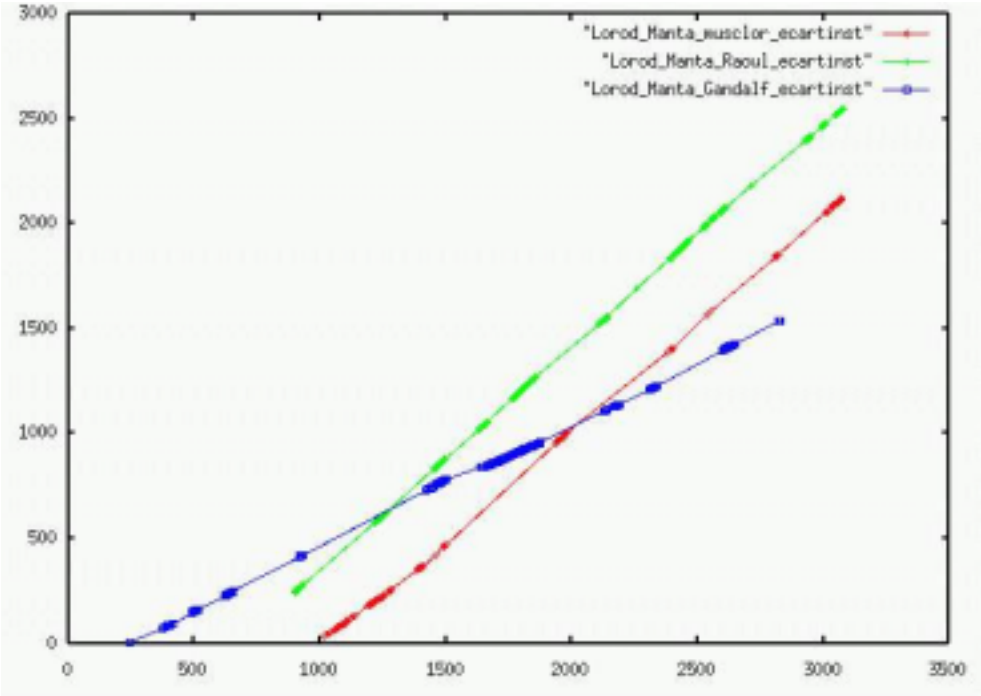


Figure 22 : Graphe de différence de rapidité entre les simulations

CONCLUSIONS ET PERSPECTIVES

Conclusions

Comme nous l'avons signalé plutôt, le but du projet PING est de définir une plate-forme qui permette de jouer en réseau en grand nombre (à l'échelle de l'Internet). Pour l'équipe MIMOSA, la tâche consiste à créer des comportements réactifs pour les jeux en réseau à partir d'un langage réactif basé sur Java, Junior. La première tâche à accomplir était de définir la notion d'incohérence. Comme nous l'avons vu, il y a plusieurs moyens de définir cette notion tant au niveau spatial et qu'au niveau temporel, c'est-à-dire nous devions trouver une mesure physique à l'incohérence. Il est clair qu'on peut trouver d'autres moyens de définir cette notion.

Ensuite, nous devions définir, à partir d'un petit jeu codé en Java et en Junior, tous les types d'incohérences que l'on pouvait trouver à partir d'un exemple très simple. Il est évident que plus on rend le jeu complexe, plus le nombre d'incohérences va être élevé. La recherche de ces incohérences se faisait surtout sur la base de constatations. Une fois que ces types avaient été bien définis, la tâche était de pouvoir améliorer l'état d'incohérence par de petits mécanismes pragmatiques, donc définis surtout au cas par cas. La majeure partie de la tâche était surtout de mettre en place des mécanismes d'anticipation de déplacements (dead-reckoning) qui permettent d'éviter les sauts de resynchronisations des répliquas esclaves. Une fois que ces solutions avaient été élaborées, nous avons pu remarquer au fur et mesure leur performance par rapport à la solution de départ.

Perspectives

Pour ce qu'il en est des perspectives futures, on peut déjà remarquer que nous avons testé notre petit jeu sur Java RMI qui est très gourmand en ressources par rapport à ce qu'il faudrait au projet PING. Actuellement, des tests sont en cours sur une première ébauche de ce que sera la plate-forme PING et qui a été réalisée par France Telecom R&D. Comme nous l'avons vu précédemment dans une architecture distribuée, la simulation n'a qu'une vue partielle du monde. Et, cette plate-forme se charge de la population de chaque simulation, et d'après les premiers tests effectués, il y aurait quelques incohérences qui apparaîtraient pendant cette population de simulations. Dans tous les cas, il est déjà prévu de tester le jeu sur lequel je me suis basé sur cette plate-forme.

Il est possible aussi d'ajouter des mécanismes de « flous artistiques » (comme par exemple rajouter des points de vie aux avatars et donner des airs d'effet à l'explosion des bombes) que je n'ai pas encore pu mettre en place, c'est-à-dire des règles qui permettent d'effectuer des compromis dans les situations où il est assez difficile de prendre une décision claire envers l'un des partis. Ces mesures permettraient de réduire en apparence quelques types d'incohérences possibles, c'est-à-dire de les cacher à l'utilisateur.

On peut aussi continuer de définir de nouvelles notions d'incohérences qui donneront peut être de meilleures approximations et qui permettra de définir de nouvelles fonctions d'extrapolation sans trop augmenter la complexité de la tâche des simulations de chaque client ce qui est tout de même le but originel.

REFERENCES

[1] « Reactive Object » de Frédéric Boussinot, Guillaume Doumenc et Jean-Bernard Stephani, Octobre 1995, ISSN 0249-6399, ISRN INRIA/RR-2664--FR
[2] « Icobj Programming » de Frédéric Boussinot, Octobre 1996, ISSN 0249-6399, ISRN INRIA/RR-3028—FR
[3] « The SugarCubes Tool Box » de Frédéric Bousssinot et Jean-Ferdy Susini, Septembre 1997, ISSN 0249-6399, ISRN INRIA/RR-3247—FR
[4] « Mimaze, a multiuser Game on the Internet »de Laurent Gautier et Christophe Diot, Septembre 1997, ISSN 0249-6399, ISRN INRIA/RR-3248—FR
[5] « Distributed Reactive Machines » de Frédéric Boussinot, Jean-Ferdy Susini et Laurent Hazard, Mars 1998, ISSN 0249-6399, ISRN INRIA/RR-3376—FR
[6] « Parallel and Distributed Simulation » de Richard M. Fujimoto, Proceedings of the 1999 Winter Simulation Conference
[7] « The Junior Reactive Kernel » de Laurent Hazard, Jean-Ferdy Susini et Frédéric Boussinot, Juillet 1999, ISSN 0249-6399, ISRN INRIA/RR-3732--FR
[8] « Objets réactifs en Java » de Frédéric Boussinot, Presses Polytechniques Universitaires Romandes, Collection Technique et Scientifique des Télécommunications, 2000
[9] « Programming with Junior » de Frédéric Boussinot, Laurent Hazard et Jean-Ferdy Susini, Octobre 2000, ISSN 0249-6399, ISRN INRIA/RR-4027--FR
[10] http:www.arttic.com/Projects/PING/
[11] http:www.goa.com
[12] http:www.sics.se/dive/
[13] http:www.inria.fr/mimosa/rp/Icobjs/
[14] http:chabee.inria.fr/