DISTRIBUTION D'ANTOID : UN MOTEUR DE JEU EXPÉRIMENTAL MASSIVEMENT MULTIJOUEURS AU MONDE PERMANENT

par

Jules Martin

Mémoire présenté au Département d'informatique
en vue de l'obtention du grade de maître ès sciences (M.Sc.)

FACULTÉ DES SCIENCES

UNIVERSITÉ DE SHERBROOKE

Sherbrooke, Québec, Canada, 30 décembre 2019

Le 30 décembre 2019

Le jury a accepté le mémoire de Jules Martin dans sa version finale

Membres du jury

Gabriel Girard (Directeur de recherche)
Université de Sherbrooke

Fabien Michel (Co-directeur de recherche)
Laboratoire d'informatique, de Robotique et de Microélectronique de Montpellier

Sylvain Giroux (Membre interne)
Université de Sherbrooke

Froduald Kabanza (Président-rapporteur)
Université de Sherbrooke

Sommaire

Le projet de maîtrise présenté dans ce mémoire, constitue le prolongement d'un projet initié en 2014 à L'UQÀC, dans le cadre de notre baccalauréat en conception de jeu vidéo. Notre objectif était alors de développer les fondations d'une version multijoueurs du jeu « Moria ». Notre jeu comprend une partie client et une partie serveur, cette dernière comprenant le moteur du jeu (Antoid) et l'interface avec la partie client. Notre objectif, en maîtrise, était de distribuer Antoid sur une grappe d'ordinateurs. L'enjeu étant d'obtenir une version massivement multijoueurs.

Quand un événement se produit, il n'affecte pas toutes les entités du jeu, mais seulement celles qui se situent dans sa zone d'effet. Partant de cette observation, nous avons élaboré une stratégie consistant (en passant au paradigme Acteur de C. Hewitt) à doter chaque entité du jeu de sa propre boucle d'événements, à transformer le monde virtuel du jeu en un maillage de cases extensible et enfin à développer divers algorithmes de traitement et de diffusion de l'information événementielle, capables d'opérer dans ce nouvel environnement sans nécessiter de supervision. Notre hypothèse était qu'avec cette nouvelle version d'Antoid, il serait possible d'accroître massivement le nombre d'entités en interaction (donc d'événements) en ajoutant simplement de nouveaux nœuds à la grappe d'ordinateurs, sans effondrement des performances. Pour tester notre hypothèse, nous avons mesuré l'évolution des délais de traitement liés au déplacement des personnages (non-joueurs) dans le monde virtuel du jeu, quand on augmente progressivement le nombre de personnages et de cases d'une part, puis le nombre de nœuds de la grappe d'ordinateurs d'autre part. L'expérience, comprenant 60 itérations et impliquant jusqu'à 279936 cases, 7776 personnages et 4 nœuds, a montré une grande stabilité, en moyenne, de ces délais, ainsi qu'un accroissement linéaire du nombre d'entités additionnelles par nœud additionnel.

Sommaire

Mots-clés: jeux massivement multijoueurs ; Paradigme acteur ; architectures pilotées par les événements ; systèmes répartis ; Erlang ; Beam ; Node.js ; Javascript

Remerciements

Quand j'ai décidé en 2015 de me lancer dans une maîtrise en cheminement recherche en informatique, je ne me doutais absolument pas de ce qui m'attendais. En effet, l'exercice s'est avéré très difficile et je n'aurais pas pu mener mon projet à son terme sans la patience et les réorientations pertinentes aux moments opportuns de mon directeur de recherche Gabriel Girard. Malgré toutes ses responsabilités en tant que directeur du département informatique, et malgré le fait que mon sujet dans le domaine du jeu vidéo n'entrait pas nécessairement dans ses centres d'intérêt, il a fait preuve d'une grande ouverture d'esprit et a réussi à trouver le temps nécessaire pour me guider dans cet exercice délicat.

Je veux également remercier mon codirecteur Fabien Michel. Alors que dans une première version de mon mémoire, l'emphase était mise sur les personnages, il m'a permis de comprendre que l'élément essentiel était constitué par l'environnement d'Antoid : son microcosme. Je tiens également à le remercier de m'avoir permis de réaliser un stage au Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier où j'ai pu me recentrer sur le paradigme acteur qui constitue l'ossature de ma solution.

Je tiens aussi à remercier Bruno Bouchard pour avoir accepté de me laisser créer Antoid durant mon baccalauréat à l'Université du Québec à Chicoutimi et sans qui ce projet de maîtrise auraît été très différent.

Mes derniers remerciements vont aux personnes de mon entourage pour leurs encouragements et leur soutien tout au long de mon parcours.

Table des matières

Liste des figures

Liste des tableaux

Liste des programmes

Introduction

Antoid est un projet de moteur expérimental de jeu vidéo initié à l'Université du Québec à Chicoutimi en 2014. Au départ, ce projet avait pour objectif de créer un moteur pour un jeu inspiré de « Moria » (1986) qui propose une expérience mono-joueur où l'on incarne un avatar cherchant à s'échapper d'un labyrinthe peuplé de monstres à l'aide d'artefacts magiques (potions, parchemins ...) [1].

Antoid reprend ce principe en ajoutant deux fonctionnalités absentes du moteur de jeu original, soit la possibilité d'une part de jouer en mode multijoueurs et d'autre part, d'évoluer dans un monde permanent (i.e. qui ne s'arrête jamais). Une première version d'Antoid a vu le jour et permettait bel et bien à quelques dizaines de joueurs d'explorer un labyrinthe qui ne disparaissait pas même quand aucun joueur n'était présent. Suite à cette première version, le projet est devenu plus ambitieux et nous souhaitons désormais rendre le moteur Antoid massivement multijoueurs [2].

Ce projet de maîtrise consiste à expérimenter des algorithmes distribués rarement appliqués au domaine du jeu vidéo massivement multijoueurs, afin d'étudier la possibilité de développer ce type de jeu (généralement onéreux) à faible coût. Nous insistons sur le côté expérimental de ce projet. En effet, notre but, aujourd'hui, n'est pas de développer un jeu complet, mais uniquement des éléments de jeu permettant de tester le moteur.

1. Moria est un jeu vidéo appartenant au genre des « rogue-like ». Notons que le genre qui est né dans les années 1970 a été redéfini par sa communauté en 2008 [5], bien que cette définition soit parfois considérée trop restrictive [23]. Précisons aussi que nous parlons ici d'une version particulière de Moria [46].

2. Un jeu massivement multijoueurs est un jeu capable d'accueillir un très grand nombre de joueurs (généralement plusieurs milliers) dans un monde permanent.

Conceptuellement, Antoid est composé de ce que nous appelons des « objets d'Antoid » qui sont répartis entre deux systèmes en interaction. Le premier système est l'environnement du jeu que nous appelons "Microcosme" et qui adopte la topologie d'un réseau maillé de cases. Le deuxième système comprend tous les objets d'Antoid évoluant au sein de ce microcosme. Il peut s'agir de personnages ou de leurs artefacts.

Un jeu vidéo étant immanquablement basé sur le concept d'événement (comme les commandes envoyées par les joueurs), Antoid devait se doter d'une architecture pilotée par les événements (Event Driven Architecture). Dans la première version d'Antoid, notre choix s'est porté sur Node.js [30] pour la partie serveur du jeu et HTML5 [10] (l'interface de programmation) via Google Chrome pour la partie client [3]. Ce choix avait plusieurs avantages. Premièrement, il nous évitait d'avoir à installer cette partie cliente sur les machines des joueurs qui possédaient déjà en majorité un navigateur Chrome. En second lieu, nous profitions du moteur javascript V8 [47] de Google, réputé rapide et présent sur la partie cliente comme sur la partie serveur. Finalement, cela nous permettait de n'utiliser qu'un seul langage de programmation à la fois orienté objets et événements : Javascript [4].

Dans cette première version d'Antoid, le serveur Node.js est installé sur un simple micro-ordinateur personnel et n'utilise qu'une seule boucle d'événements. Le nombre d'objets d'Antoid en interaction qu'il est possible de supporter est donc limité. Or, pour rendre Antoid massivement multijoueurs, ce nombre doit augmenter radicalement. En effet, si le nombre de joueurs devient massif, le monde qui les accueille doit devenir tout aussi massif et comme il n'est pas très intéressant de parcourir un monde vide, celui-ci doit bien évidemment être entièrement peuplé de personnages non joueurs et d'artefacts. Ainsi, pour supporter le nombre explosif d'interactions que ce type de jeu exige, nous avons choisi de procéder par ajout progressif de processeurs, en nous imposant de ne pas interrompre le jeu, de ne pas toucher au code et de ne

3. Notons cependant, qu'il est tout à fait possible d'utiliser d'autres navigateurs capables d'interpréter du Javascript tel que Firefox.

4. Il est à noter que Javascript est un langage de programmation réputé particulièrement difficile à maîtriser. Dans le cadre du développement de cette première version d'Antoid, nous avons eu recours à la série de livres « You Don't Know JS » [43], [38], [39], [42] , [40], [41].

pas perturber les joueurs.

En pratique, augmenter progressivement, mais massivement le nombre de personnages, de cases susceptibles de les accueillir et le nombre d'interactions entre tous ces objets d'Antoid revient en définitive à augmenter le nombre d'événements. Cependant, dans un système n'utilisant qu'une boucle d'événements unique, une augmentation de cette ampleur risque de provoquer, tôt ou tard, un goulot d'étranglement. Dans ces conditions, exploiter de multiples boucles d'événements semble une option raisonnable ; chaque boucle d'événements prenant en charge une partie des événements émis. Toutefois, multiplier le nombre de boucles d'événements risque de rendre plus difficile le maintien de la cohérence globale du système.

Précisons que lorsque nous parlons de cohérence, nous nous basons sur la définition proposée par Maarten van Steen et Andrew S. Tanenbaum dans leur ouvrage « Distributed systems » [44] : « A distributed system is coherent if it behaves according to the expectations of its users ». En d'autres termes, un système est cohérent tant qu'il apparaît comme tel aux yeux des utilisateurs (dans notre cas les joueurs) et qu'il se comporte de la même manière pour tous les utilisateurs.

Notre solution a consisté, via le passage du paradigme objet au paradigme acteur de C. Hewitt [25] et l'adoption du langage de programmation Erlang, à doter chaque objet d'Antoid d'une part, d'un exemplaire de boucle d'événements et d'autre part, d'un nouveau mécanisme leur permettant de contrôler l'enchaînement de leurs actions (i.e. déterminer la prochaine action à réaliser en fonction de la situation).

Notons qu'une des originalités de cette deuxième version d'Antoid est que tous nos objets d'Antoid sont des acteurs, que ce soit les cases qui forment l'environnement, les personnages et leurs artefacts. On peut aussi noter que ce choix fait écho à l'un des axiomes proposés par Carl Hewitt pour son paradigme acteur : «everything is an actor» (tout est acteur).

Lorsqu'un événement se produit, celui-ci a ce que nous appelons une zone d'effet. C'est à dire qu'il n'impactera que les entités situées dans cette zone. Par exemple, lors-

qu'un personnage chuchote, seuls ceux qui sont très près de celui-ci peuvent l'entendre alors que lorsqu'un personnage hurle, tous les personnages alentours l'entendent.

Dans la première version d'Antoid, la tâche consistant à acheminer les événements auprès des entités concernés revient à une boucle d'événements unique, en l'occurrence celle de Node.js. Or, comme dit précedemment, lorsque le nombre d'événements augmente, cette boucle risque de devenir un goulot d'étranglement.

Pour notre deuxième version d'Antoid, nous avons distribué ce rôle aux cases composant l'environnement de notre jeu (le microcosme) organisées de manière à former un réseau maillé. L'idée est que lorsqu'un événement se produit sur une case, celui-ci sera diffusé via un algorithme local de case en case de manière à recouvrir sa zone d'effet. Par conséquent seuls les objets d'Antoid situés dans la zone d'effet seront informés de cet événement laissant aux autres objets le loisir de vaquer à d'autres tâches et limitant de ce fait le risque d'apparition de goulot d'étranglement.

La distribution de cette boucle d'événements dans les cases apporte donc un résultat équivalent à une boucle unique tout en réduisant le risque d'apparition de goulot d'étranglement. Cependant, multiplier le nombre de boucles d'événements risque de rendre plus difficile la tâche de maintenir la cohérence du système.

Dans un système utilisant une boucle d'événements unique, celle-ci maintien cette cohérence via son fonctionnement en mode bloquant (équivalent à ce qu'on appelle un fonctionnement au tour par tour). C'est à dire qu'elle ne traite un nouvel événement que lorsque l'exécution de l'action associée à l'événement précédent est terminée. De cette manière il est impossible que deux actions modifient une partie de l'état d'une entité concurremment et risquent donc de provoquer une incohérence. Cependant, même si le système reste cohérent d'un point de vu technique, il peut tout de même devenir incohérent du point de vu des utilisateurs soit dans notre cas des joueurs.

Pour illustrer ce phénomène nous allons présenter un exemple où deux personnages tentent de se déplacer sur une même case inoccupée (la case ne pouvant être occupée

que par un personnage à la fois). Cet exemple est basé sur l'une des règles de notre jeu consistant à permettre l'interruption du déplacement d'un personnage. Cette règle pouvant, dans le cadre de notre première version d'Antoid, créer une situation incohérente du point de vu des joueurs. Précisons que notre exemple se déroule dans une architecture où les appels de fonction associées aux événements sont traités par la boucle d'événements.

Notre exemple est le suivant :

1. la boucle d'événements commence par le traitement d'un événement (E1 P1→C1) correspondant à une tentative de déplacement d'un premier personnages P1 sur une case de destination C1. La boucle d'événements sollicite alors la case C1 ;
2. la case C1 traite cette tentative de déplacement. Étant inoccupée elle réserve son emplacement et émet l'événement (E1 C1→P1) correspondant ;
3. la boucle d'événements traite alors un événement (E1 P2→C1) correspondant à une tentative de déplacement d'un deuxième personnage P2 sur la case C1. Elle sollicite à nouveau la case C1 ;
4. la case C1, ayant réservé sont emplacement, refuse le déplacement. Elle émet alors l'événement (E1 C1→P2) correspondant ;
5. la boucle d'événements traite ensuite l'événement (E1 C1→P1). Elle sollicite alors le personnage P1 ;
6. le personnage P1 met à jour sa position et émet l'événement (E2 P1→C1) correspondant ;
7. la boucle d'événements traite l'événement (E1 C1→P2). Elle sollicite alors le personnage P2 ;
8. le personnage P2, son déplacement ayant échoué, conserve son état ;
9. la boucle d'événements traite l'événement (E2 P1→C1). Elle sollicite alors la case C1 ;
10. la case C1 change son état de manière à accueillir le personnage P1.

Dans cet exemple, aucune incohérence n'apparaît. Mais supposons maintenant qu'un événement ayant pour effet d'immobiliser le personnage P1 soit traité avant l'événement (E1 C1->P1), de sorte que ce dernier soit dans l'incapacité de finaliser son déplacement, en ce cas aucun des deux personnages ne parvient à se déplacer [5]. Bien que le système reste techniquement cohérent, pour le joueur contrôlant le personnage P2, il n'est pas normal que son déplacement vers une case vide ait échoué rendant, de son point de vu, le système incohérent.

Pour cette nouvelle version d'Antoid, les objets d'Antoid deviennent des acteurs intégrant l'équivalent d'une boucle d'événements et interagissant entre eux via l'émission et la réception de messages. Chaque objet d'Antoid devient donc capable de différer le traitement d'un événement offrant un niveau de maintien de la cohérence supérieur à celui de la solution précédente. En effet, si nous reprenons notre exemple, la case C1 plutôt que de refuser le déplacement du personnage P2 pourra le différer. Ainsi, une fois le déplacement du personnage P1 interrompu, l'événement précédement différé pourra être traité, cette fois-ci avec succès. Le joueur contrôlant le personnage P2 n'observera alors pas d'incohérence.

Dans ce mémoire, nous allons étudier via plusieurs expériences, l'efficacité de notre solution pour cette deuxième version d'Antoid en observant l'évolution du nombre d'objet d'Antoid qu'il est possible de supporter sur une grappe d'ordinateurs lorsque le nombre d'ordinateurs augmente.

Ce mémoire est découpé en quatre chapitres.

Le premier chapitre présente les différents concepts nécessaires à la compréhension de ce mémoire.

Le second chapitre expose les raisons de notre choix du langage de programmation Erlang (et sa machine virtuelle Beam).

5. Précisons que pour éviter que la case attende indéfiniment une réponse du personnage P1 il serait nécessaire de mettre en place un mécanisme d'annulation comme un compte à rebours.

Introduction

Le troisième chapitre, présente notre solution dont l'élaboration a connu deux phases. Premièrement une phase consistant à doter chaque objet d'Antoid de sa propre boucle d'événements et deuxièmement, une phase consistant à doter ces mêmes objets d'Antoid d'un nouveau mécanisme leur permettant de contrôler l'enchaînement de leurs actions.

Le quatrième chapitre, présente les résultats de notre expérience qui a consisté à distribuer Antoid sur notre grappe de RaspberryPI : Olifan. Ces résultats se présentent sous la forme de différentes statistiques relatives à d'une part, l'évolution du nombre maximal d'objets d'Antoid en interaction qu'Antoid et Olifan peuvent supporter lorsqu'on augmente le nombre de RaspberryPI et d'autre part, l'évolution du temps de traitement nécessaire pour réaliser les déplacements des personnages (non-joueurs) dans le microcosme quand on augmente progressivement le nombre d'objets d'Antoid en interaction et le nombre de RaspberryPI.

Chapitre 1

Etat de l'art

L'objectif de ce premier chapitre est de présenter les différents concepts nécessaires à la compréhension de ce mémoire.

Antoid est un moteur de jeu vidéo conçu pour le genre massivement multijoueurs, la première section fait donc un tour d'horizon du genre. Un jeu vidéo étant immanquablement piloté par les événements, la seconde section est consacrée aux architectures pilotées par les événements. Notre solution fait appel au paradigme acteur, nous exposons donc les points clefs du paradigme acteur de Carl Hewitt dans la troisième section. Enfin, le projet étant un projet de distribution et plus particulièrement de distribution de la boucle d'événements nous parlons de la problématique des architectures pilotées par les événements distribuées dans la quatrième et dernière section de ce chapitre.

1.1 Les jeux massivement multijoueurs

Contrairement à d'autres types de jeux dont le qualificatif est tiré de leurs mécaniques de jeu et la manière dont on y joue (e.g. les jeux de tir à la première personne ou les jeux de courses automobiles), les jeux massivement multijoueurs se caractérisent par deux particularités de leur monde virtuel, soit la capacité à accueillir un

nombre massif de joueurs en même temps (aujourd'hui plusieurs milliers) et le fait que ce monde soit permanent. En d'autres termes, ces jeux offrent aux joueurs la possibilité d'entrer dans un monde qui est capable de fonctionner et d'évoluer sans eux tout en leur permettant de vivre des expériences nécessitant l'association d'un très grand nombre de joueurs.

Avant de commencer, précisons que notre objectif dans cette section n'est pas de proposer un état de l'art exhaustif du genre massivement multijoueurs, mais bien de fournir une introduction aux lecteurs peu familiers avec ce type de jeu.

1.1.1 Évolution du genre massivement multijoueurs

Le concept de monde permanent est indissociable des jeux massivement multijoueurs. En effet, techniquement, les serveurs hébergeant le monde virtuel de ces jeux ne sont pas supposés s'arrêter (sauf en cas de panne ou de maintenance).

Les premiers jeux où cette caractéristique apparaît sont les donjons multijoueurs ou *Multi User Dungeon* (MUD) apparus à la fin des années 70. Ces jeux à l'interface textuelle réunissaient plusieurs joueurs dans un donjon persistant [1].

Ces MUD s'inspirant des jeux de rôles comme *Donjon et Dragon* ont posé les bases toujours utilisées dans les jeux massivement multijoueurs d'aujourd'hui. Une de ces idées, est la différenciation entre les personnages que contrôlent les joueurs que l'on appelle les « personnages joueurs » ou « PJ », et les personnages contrôlés par une intelligence artificielle (très limitée à l'époque) et que l'on appelle les « personnages non-joueurs » ou « PNJ ». Ces derniers peuvent être des monstres qui attendent de rencontrer des personnages joueurs à affronter, des marchands qui permettent d'acheter ou vendre des objets ou tout simplement des personnages dont la seule utilité est de rendre le monde du jeu plus vivant et immersif pour les joueurs.

1. Il est d'ailleurs intéressant de noter que plusieurs adaptations françaises de ce type de jeu, renommé « Multi Access Dungeon », ont vu le jour dans les années 80.

Cependant, du fait de la puissance limitée des machines de l'époque, il n'était pas possible d'accueillir un nombre réellement massif de joueurs. En effet, ces jeux n'en accueillaient au maximum que quelques dizaines.

Le qualificatif de jeu massivement multijoueurs est arrivé plus tard, lorsque les machines et les réseaux ont été capables de supporter un nombre réellement massif de joueurs. Ainsi, c'est *Neverwinter nights* qui pouvait alors accueillir jusqu'à 500 joueurs à la fois qui est souvent considéré comme le premier « vrai » jeu massivement multijoueurs.

Aujourd'hui, le nombre de joueurs que ces jeux peuvent accueillir dépasse facilement les quelques milliers. À titre d'exemple, *World of warcraft* est capable d'accueillir jusqu'à 5000 joueurs à la fois dans son monde virtuel.

Avant de poursuivre, notons que ce n'est pas seulement le nombre de joueurs qui a augmenté ,mais aussi le nombre de personnages non-joueurs nécessaires à l'immersion de ces joueurs. De plus, ces personnages non-joueurs ne font pas ici office de figurants, mais sont des entités actives dans le jeu ayant des capacités similaires à celles des personnages joueurs.

Le point que nous cherchons à développer ici est que comme ces jeux accueillent un nombre encore plus massif de personnages non-joueurs et que les moteurs supportant ces jeux accueillent donc un nombre d'interactions nettement plus élevé qu'un jeu simplement massivement multijoueurs, il pourrait être plus pertinent de parler de jeu massivement multi-personnages.

Le moteur d'Antoid prolonge en quelque sorte cette évolution puisque les artefacts (e.g. des murs) et les cases (qui composent le monde virtuel du jeu) entrent en interaction d'une manière équivalente aux personnages joueurs et non-joueurs. Ainsi, il serait peut être encore plus pertinent de parler de jeu massivement multi-acteurs faisant écho au paradigme de Carl Hewitt, où tout est acteur.

Ajoutons enfin que cette similarité permet à nos expériences de produire des résultats pertinents quoiqu'elles n'incluent que des cases, des personnages non-joueurs et des artefacts, le moteur Antoid ne faisant aucune différence entre les acteurs.

1.1.2 Difficultés posés par un nombre massif de joueurs

Un nombre d'acteurs aussi massif peut être un défi autre que technique. En effet, lorsqu'un monde virtuel accueille un nombre aussi grand d'interactions, l'expérience de jeu telle qu'elle a été prévue par les développeurs peut rapidement être mise à mal.

À titre d'exemple, nous pourrions citer le jeu *Ultima Online* où les joueurs, tels une nuée de sauterelles, ont détruit l'écosystème du monde virtuel du jeu à son lancement.

En effet, il existait dans ce monde, un certain nombre de créatures (représentées par des personnages non-joueurs) mettant en scène un petit écosystème contenant des végétaux, des herbivores et des carnivores. Le principe de cet écosystème étant que la population de chaque type de créature devait s'équilibrer via ses interactions avec celles des autres types. Par exemple, lorsque le nombre de carnivores devenait trop élevé pour le nombre de proies (les herbivores), le nombre de carnivores baissait, car ils ne pouvaient pas tous se nourrir et se reproduire.

Ce qui n'avait pas été pris en considération par les développeurs dans la conception de cet écosystème est le comportement des joueurs. En effet, dès leur arrivée, les joueurs ont tué toutes les créatures qu'ils rencontraient, que ce soit les carnivores (qui attaquaient les joueurs) ou les herbivores (pacifiques). De cette manière, après un certain temps, il ne restait plus aucune créature capable de se reproduire ne laissant plus que les joueurs.

Pour permettre à tous les joueurs de profiter des expériences de jeux prévues par les développeurs, plusieurs solutions ont été conçues.

Conceptuellement, la majorité des jeux mettent en place une règle moins immersive consistant à rendre les ressources illimitées. Ainsi, à chaque fois qu'une ressource est

consommée par un joueur (e.g. une créature est tuée par un joueur, une plante est cueillie) elle réapparaîtra après un certain temps de telle manière qu'un autre joueur puisse à son tour la consommer.

Techniquement, une autre solution a ironiquement consisté à réduire le nombre de joueurs pouvant se trouver en même temps dans le monde virtuel, l'idée étant d'aménager le nombre de personnages joueurs et des autres interacteurs afin de trouver un équilibre permettant de conserver une expérience de jeu satisfaisante. Pour ce faire, plusieurs instances du monde du jeu (initialement appelées « Shards ») limitant le nombre de joueurs maximal ont été mises en place. Il serait par exemple possible de répartir les joueurs entre différentes instances ne pouvant accueillir que 5000 joueurs chacune au maximum.

Il est à noter que ce concept d'instance a aussi été utilisé au sein d'un même monde virtuel afin de créer des copies d'une même zone de jeu de manière à ce que chaque instance ne puisse accueillir qu'un nombre très réduit de personnages joueurs ; l'intérêt étant de pouvoir contrôler le type d'expérience qui sera offert aux joueurs. Il devient par exemple possible de mettre en place des affrontements prévus pour un nombre limité de joueurs qui devront redoubler d'inventivité pour les remporter. Un autre exemple d'utilisation des instances se retrouve dans *Guild Wars*, où ce ne sont pas seulement les affrontements qui sont mis en avant, mais la narration. Une instance devient ici un moyen de s'assurer qu'une histoire puisse être racontée dans les conditions voulues.

Avant de poursuivre, il nous faut toutefois nuancer les difficultés posées par un nombre massif de joueurs. En effet, de nouvelles expériences totalement imprévues peuvent émerger de toutes ces interactions sans pour autant être négatives. Une expérience de jeu imprévue a par exemple émergé du jeu *World of warcraft*, où les joueurs ont affronté une véritable pandémie virtuelle.

À l'origine de cette pandémie, on retrouve l'introduction dans le monde du jeu d'une instance, constituant un défi pour les joueurs les plus puissants, contenant une

créature capable d'ensorceler ses assaillants. Ce sort avait pour effet de se transmettre entre les personnages joueurs comme une maladie contagieuse et de leur faire perdre progressivement leurs points de santé (une fois le nombre de points arrivé à zéro, le personnage joueur mourrait).

Conceptuellement, ce sort devait être confiné dans l'instance et les joueurs infectés ne pouvaient en sortir qu'une fois guéris du sort (en cas de victoire sur la créature ou la mort de leur personnage). Cependant, un effet imprévu fit que le sort pouvait aussi contaminer les animaux de compagnie des joueurs sans pour autant les affecter (les animaux devenant en quelque sorte des porteurs sains d'une maladie). Or, ces animaux pouvaient ne pas êtres guéris du sort avant de sortir de l'instance. Ainsi, lorsque les joueurs accompagnés de leurs animaux de compagnie retournèrent dans les villes du jeu, ceux-ci contaminèrent les autres personnages (joueurs et non-joueurs) créant la première épidémie à l'intérieur d'un jeu vidéo.

Les effets furent tellement importants, qu'une toute nouvelle expérience de jeu vit le jour : une simulation de pandémie où les joueurs sains fuyaient les grandes villes, tandis que les joueurs contaminés instauraient des quarantaines pour contenir le sort ou au contraire cherchaient à contaminer le plus de personnages possible.

Il est d'ailleurs à noter qu'outre l'expérience de jeu unique que cet événement a provoqué, il a aussi fait l'objet de divers articles dans le domaine de l'épidémiologie et offre un aperçu des capacités des jeux massivement multijoueurs comme simulateurs [3].

1.1.3 Architecture client-serveur

Les jeux massivement multijoueurs utilisent en règle générale une architecture client-serveur (comme c'est le cas pour Antoid). La partie client est distribuée entre les joueurs. Elle se charge de transmettre les commandes des joueurs à la partie serveur (e.g. déplacer le personnage du joueur) et de traiter les messages envoyés par la partie serveur concernant le joueur (e.g. des informations visuelles à afficher à l'écran).

La partie serveur (qui peut être distribuée), héberge le monde virtuel du jeu et les différents acteurs qui le peuplent. Elle est aussi responsable entre autres de l'application des lois physiques du jeu (e.g. la collision entre les acteurs), du comportement (i.e. l'intelligence artificielle) des personnages non-joueurs, du traitement des commandes des joueurs et de la transmission des messages à destination des différents clients connectés.

Un avantage notable de cette architecture est qu'elle permet de contrôler plus facilement l'évolution du monde du jeu. Il est en effet plus facile d'arrêter complètement le monde virtuel afin d'installer une mise à jour ou de procéder à une maintenance par exemple.

Il existe une troisième partie à cette architecture qui est la base de données centralisée qu'utilise le serveur pour stocker toutes les informations nécessaires au bon fonctionnement du jeu, notamment pour conserver un état stable du jeu auquel retourner en cas de panne.

Il est à noter que cette base de données ne contient toutefois pas toutes les données du jeu. En effet, les données relatives à l'affichage telles que les modèles 3D, les icônes ou les animations sont toutes copiées sur les machines des joueurs. En revanche, toutes les données concernant les caractéristiques des personnages joueurs et non-joueurs sont stockées dans cette base de données.

Notons aussi que certaines de ces données ne sont pas stockées sur les machines des joueurs. En effet, cela offre un meilleur contrôle pour les développeurs, qui n'ont ainsi pas besoin de mettre à jour la partie client des joueurs à chaque modification. Ajoutons que cela permet aussi de réduire les risques de tricherie de la part des joueurs. Ces données peuvent aussi avoir un usage autre que faire fonctionner le jeu. Elles peuvent aussi permettre la production de statistiques. Ces statistiques pouvant être utilisées dans l'optique d'équilibrer le jeu, de trouver des bogues ou même de remarquer plus facilement des comportements parfois indésirables tels que l'utilisation de robots.

Dans le cadre du moteur Antoid, nous avons fait le choix de ne pas utiliser de base de données centralisée, mais de répartir les données entre les différentes machines composant notre grappe d'ordinateur. Techniquement, les données sont distribuées dans la mémoire vive gérée par les différentes instances de la machine virtuelle d'Erlang (Beam), elles-mêmes distribuées sur les différents nœuds de notre grappe d'ordinateurs.

En ce qui concerne les données nécessaires pour paramétrer le monde virtuel lors de son initialisation ou conserver le dernier état stable du jeu, nous avons choisi d'utiliser des fichiers répartis entre les différentes machines de la grappe d'ordinateur ; ces fichiers étant aussi utilisés dans le cadre de nos expériences pour sauvegarder les informations nécessaires à nos calculs statistiques.

Distribution du monde virtuel

Les jeux les plus populaires du genre massivement multijoueurs emploient tous une architecture distribuée pour leur serveur (i.e. une grappe d'ordinateurs) afin de permettre une mise à l'échelle du nombre de joueurs tout en maintenant une certaine stabilité dans le temps de réponse du serveur (cette caractéristique étant capitale dans un jeu vidéo multijoueurs). Notons toutefois que dans ce contexte, l'utilisation d'une base de données centralisée impose une limite à cette mise à l'échelle, car celle-ci risque de devenir un goulot d'étranglement lorsque le nombre de transactions deviendra trop élevé. À titre d'exemple, le jeu *EVE online* a recours à un serveur SQL centralisé prenant en charge autour de 2500 transactions à la seconde pour une quantité de données s'élevant en moyenne à 1.4 TB [13] et a donc nécessité une optimisation extrême de son architecture matérielle et logicielle [15] .

Compte tenu du contexte commercial dans lequel ces jeux sont développés, les techniques d'implémentation de ces serveurs ne sont en règle générale pas divulguées au grand public. Il existe toutefois certains articles présentant une approche générale des stratégies de distribution entre les différentes machines composant le serveur. Nous avons déjà parlé du concept d'instance (ou « shards ») qui consiste à créer de

multiples copies d'un même monde virtuel de manière à réduire le nombre de joueurs maximal qu'un même monde peut héberger.

Il reste cependant évident qu'abuser de cette stratégie fait perdre de son intérêt au concept même de jeu massivement multijoueurs. C'est pourquoi l'on peut aussi retrouver des stratégies prenant place dans un même monde virtuel.

Dans un article traitant du jeu *EverQuest* [28], on retrouve par exemple une idée consistant à diviser le monde virtuel en régions qui sont alors distribuées dans la grappe d'ordinateurs. En revanche, cette stratégie pose un problème lorsqu'un certain nombre de joueurs se massent au niveau d'une frontière entre deux serveurs (le « crowding »). Pour résoudre cette problématique, une solution est de placer des obstacles naturels sur ces frontières comme des chaînes de montagnes ou de grandes étendues d'eau dans lesquelles les joueurs n'ont que peu d'intérêt à rester.

Il faut toutefois noter que contrairement au concept d'instance, cette stratégie n'empêche pas les joueurs qui le désirent de se masser dans une seule région du monde virtuel. Nous pourrions citer à titre d'exemple le jeu *EVE online*, où presque 3000 joueurs se sont livrés bataille dans une de ces régions [14].

La littérature propose pour ce cas de figure, différentes solutions de répartition de la charge entre les différentes machines composant le serveur. L'idée principale étant de produire une répartition pilotée par les joueurs où un système est chargé de diviser dynamiquement le monde virtuel en fonction des mouvements des joueurs ; la charge étant alors répartie entre les machines composant le serveur en fonction des besoins [16], [26], [48].

Dans le cadre d'Antoid, nous avons fait le choix de conceptuellement diviser notre monde virtuel (que nous appelons microcosme) dans un réseau maillé de cases (une case étant assez grande pour contenir un personnage à la fois) avant de le distribuer dans une grappe d'ordinateurs de manière à ce que chaque ordinateur héberge une partie des cases formant le microcosme.

Précisons que bien que dans les expériences que nous avons réalisées, le nombre de cases qu'héberge chaque machine est le même mais il serait tout à fait possible de répartir les cases et donc des régions du microcosme autrement.

Ajoutons enfin que ces cases ne sont pas de simples réceptacles adressables via des coordonnées ; elles jouent le rôle de processeur virtuel capable de communiquer localement avec d'autres cases. Ainsi, il serait tout à fait possible de concevoir des algorithmes collaboratifs non supervisés permettant aux cases de s'organiser et de se réorganiser en fonction des joueurs en temps réel, et cela sans nécessiter la production de statistiques globales.

1.2 Architectures pilotées par les événements

Dans une application basée sur une architecture pilotée par les événements (Event Driven Architecture), sans événements, pas d'actions ; chaque action spécifique dépendant de l'occurrence d'un événement spécifique. Ce type d'architecture est particulièrement adapté aux jeux vidéo qui sont des applications interactives et doivent donc être capables de réagir aux actions qu'effectuent les joueurs et les différentes entités du jeu. Dans un contexte d'application basée sur les événements, les événements sont le lien entre les actions. Ils en sont à la fois leurs effets et leurs causes.

Imaginons le cas suivant : un personnage agit en se déplaçant dans le monde du jeu. Cette action de jeu a pour effet (en simplifiant) de le faire disparaître d'un endroit pour le faire apparaître à un nouvel endroit. Nous pourrions aussi imaginer qu'en apparaissant dans ce nouvel endroit, il entre dans le champ de vision d'un autre personnage. Le fait pour cet autre personnage de voir quelqu'un apparaître pourrait alors être cause d'une nouvelle action de jeu comme saluer le nouvel arrivant. Cette action peut alors avoir à son tour des effets qui pourront être causes de nouvelles actions de jeu, le personnage voit qu'on le salue et agit en saluant à son tour.

Notons aussi qu'il existe, en plus des événements conçus par le développeur, des événements standards. Ces événements peuvent par exemple correspondre à un clic

de souris ou à la pression sur une touche de clavier.

1.2.1 Définition d'un événement

Commençons par tenter de définir ce qu'est un événement. Etzion et Niblett [18] postulent que le terme d'événement regroupe deux acceptions :

1. un événement est quelque chose qui se produit dans le monde réel ou dans d'autres systèmes tel qu'un système informatique.
2. un événement est une structure (programming entity) représentant la chose qui s'est produite.

Etzion et Niblett distinguent ici l'occurrence d'un événement des informations relatives à celui-ci. Pour marquer cette distinction, ces auteurs et d'autres dont Chandy et Schulte [11] qualifient cette structure incorporant ces informations de « event object ». Toutefois, dans le cadre de ce mémoire, nous lui préférerons le terme de message, plus en accord avec l'idée que ces informations sont transmises à ceux qui sont situés dans la zone d'effet d'un événement.

Cette distinction est importante, car elle montre qu'un observateur ne peut avoir connaissance d'un événement qu'indirectement au travers d'un message incorporant des informations relatives à l'événement en question. Précisons, que cette distinction n'est pas toujours faite dans la littérature, où le terme d'événement est parfois utilisé indifféremment pour ces deux acceptions.

À ces deux acceptions, nous souhaiterions en ajouter une troisième, celle du patron d'un événement (Event Pattern), qui est un schéma contenant l'ensemble des attributs qu'un message correspondant à un événement particulier doit utiliser (comme un horodatage, l'identifiant des intervenants, etc.). Ce patron est utilisé pour reconnaître les messages reçus.

Avant de poursuivre, ajoutons que Chandy, Charpentier et Capponi [9] proposent une autre définition pour le concept d'événement : un événement est un changement

d'état significatif dans l'univers, où l'univers est l'association entre un système piloté par les événements et son environnement.

Cette définition apporte une distinction entre les changements significatifs et non-significatifs, où un changement significatif provoque une réaction, alors qu'un changement non-significatif sera ignoré par le système. Nous pouvons rapprocher cette définition des messages reconnus (significatifs) et des messages non reconnus (non-significatifs).

1.2.2 Cycle de vie des événements

Les applications basées sur une architecture pilotée par les événements, bien qu'elles puissent différer les unes des autres, comprennent au minimum trois types de composants (comme illustré sur la figure 1.1).

1. des producteurs d'événements ;
2. des consommateurs d'événements ;
3. un système de diffusion qui transmet les messages incorporant les informations relatives aux événements des producteurs aux consommateurs.

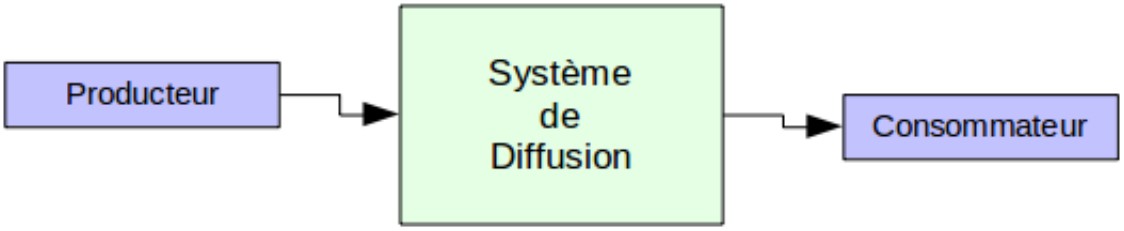

Figure 1.1 – Composants d'une architecture pilotée par les événements

Notons que les producteurs ou les consommateurs peuvent changer de rôle, un producteur pouvant devenir un consommateur et vice-versa. Un exemple de ce cas de figure est illustré par le paradigme acteur de Carl Hewitt, où un même acteur peut envoyer et recevoir des messages.

Le cycle de vie d'un événement peut être divisé en trois étapes principales, son émission, sa réception et son traitement.

Emission

L'étape d'émission consiste à envoyer un message incorporant les informations relatives à l'événement produit à tous ceux concernés par celui-ci. Dans notre exemple, cela consiste à envoyer un message concernant la disparition et l'apparition d'un personnage à d'autres personnages qui peuvent le voir.

Il existe plusieurs méthodes de diffusion [9], les deux méthodes principalement utilisées sont :

- la méthode consistant à expédier les messages à l'adresse des récepteurs (la méthode en flux poussé ou « push ») illustrée à la figure 1.2 ;
- la méthode consistant à retirer les messages depuis l'adresse de l'émetteur (la méthode en flux tiré ou « pull ») illustrée à la figure 1.3.

Dans la première méthode, les messages sont transmis à tous ceux qui sont intéressés par le type d'événement qui a été produit. Techniquement, chaque destinataire possède un espace mémoire dans lequel les messages sont déposés.

Dans la deuxième approche, les messages sont stockés dans l'espace mémoire de l'émetteur. Ceux qui sont intéressés par les événements provenant de cette source récupèrent les messages depuis cet espace mémoire.

Réception

L'étape de réception correspond à la fin de la diffusion du message, soit à l'obtention du message par son destinataire. Dans le cadre des deux méthodologies que nous avons citées, cela consiste d'une part pour la méthode en flux poussé à l'enregistrement du message par l'émetteur, dans l'espace mémoire dédié au récepteur et d'autre part pour la méthode en flux tiré à l'enregistrement du message par le récepteur dans son espace mémoire une fois celui-ci retiré de l'espace mémoire de l'émetteur.

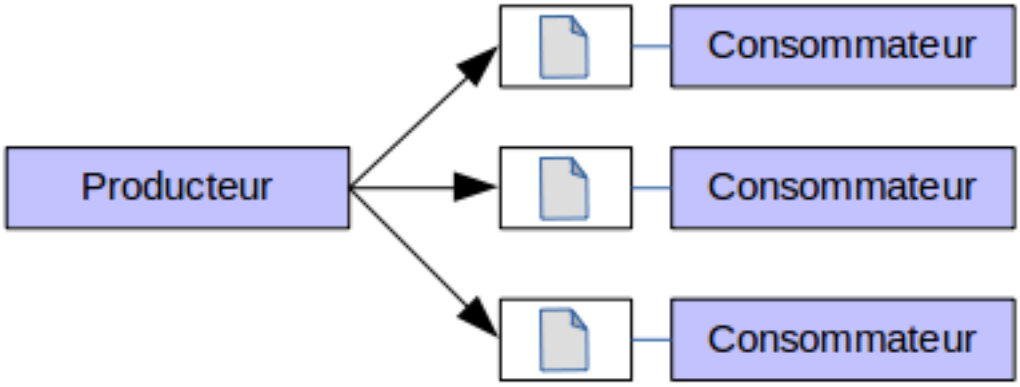

Figure 1.2 – Méthode orientée Push

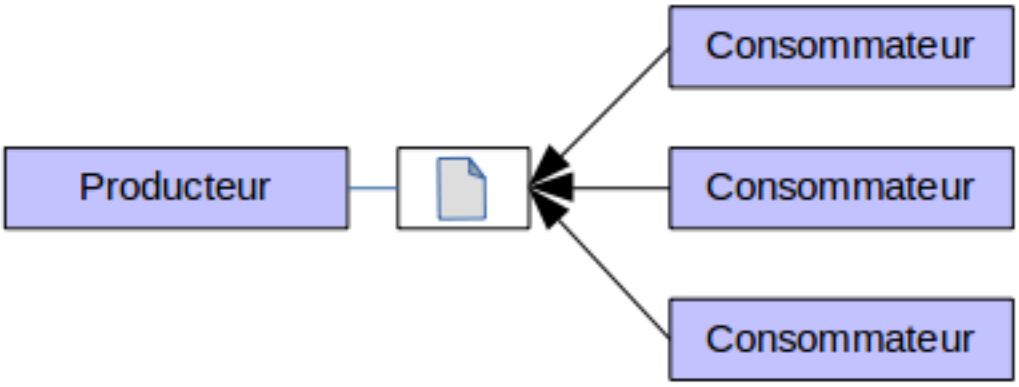

Figure 1.3 – Méthode orientée Pull

Il est à noter que dans le cadre du paradigme acteur, ce sont les deux méthodes qui sont utilisées. En effet, chaque acteur dispose d'une boîte aux lettres dont le rôle est de stocker les messages qui lui sont destinés. Ainsi, lorsqu'un acteur envoie un message à un autre acteur, celui-ci va effectuer un flux poussé vers la boîte aux lettres de son destinataire. Alors que lorsqu'un acteur reçoit un message, il effectue un flux tiré depuis sa propre boîte aux lettres. Pour plus de détails sur le fonctionnement des boîtes aux lettres, vous pouvez consulter la section 1.3.1.

Traitement

Une des particularités des architectures pilotées par les événements, est que les messages incorporent exclusivement des informations relatives aux événements et n'indiquent donc pas quelle action effectuer [11]. Le message seul est donc insuffisant pour traiter l'événement.

Pour déterminer de quelle manière traiter un message, les consommateurs utilisent une liste de couples associant un message particulier à une action particulière à entreprendre lorsque ce message est reçu. Pour reconnaître les messages, les consommateurs ont recours au concept de reconnaissance de formes ou « pattern matching ». La reconnaissance de formes passe par la comparaison entre la forme des messages reçus et des formes conservées en mémoire correspondant à ce que nous avons appelé les patrons d'événement.

Précisons que ces listes de couples ne sont pas statiques. En effet, les deux environnements que nous avons utilisés dans le cadre d'Antoid (Node.js et Erlang) permettent la modification de ces listes et donc d'adapter les réactions de nos objets d'Antoid par rapport à un même événement.

L'étape de traitement d'un événement consiste donc, dans un premier temps, à tenter de reconnaître le message en le comparant aux formes de messages conservées en mémoire. Si le message n'est pas reconnu, il est ignoré et le traitement s'arrête ici. Si au contraire le message est reconnu, l'action associée est lancée.

Un autre apport des boîtes aux lettres des acteurs est que les messages non reconnus lors de cette étape peuvent ne pas être supprimés. Ils sont alors redéposés dans la boîte aux lettres pour un traitement ultérieur (dans l'attente de l'ajout d'une association dans la liste de couples permettant son traitement).

1.2.3 Dispositif de traitement des événements

Il existe différents mécanismes de traitement d'événements. L'un de ces mécanismes est la boucle d'événements (voir figure 1.4). Cette boucle est rattachée d'une part à une file d'attente où sont entreposés les différents messages émis et d'autre part à une liste associant messages et actions. Son rôle consiste à prendre en charge la réception et le traitement des événements.

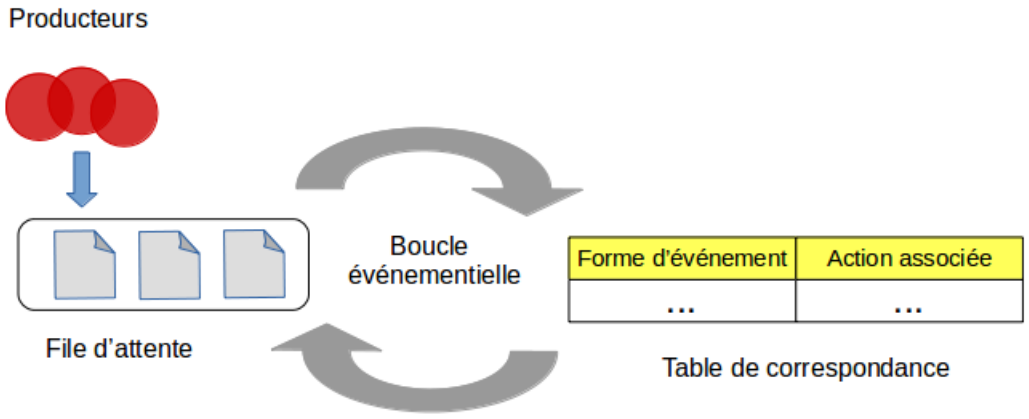

Figure 1.4 – Boucle événementielle

Il est à noter que bien que Node.js utilise un dispositif qui soit aussi qualifié de boucle d'événements, son fonctionnement diffère de celui d'une boucle d'événements standard. En effet, la gestion des événements dans Node.js ne passe pas par sa boucle d'événements, mais par des objets appelés « EventEmitter ». Ces objets sont donc responsables de la réception des messages et de leur traitement. Il est à noter que ce rôle est en contradiction avec le terme « EventEmitter ». En effet, ces objets n'émettent aucun événement.

Chaque objet « EventEmitter » conserve dans ses attributs une liste de couples associant messages et actions (les actions étant dans ce contexte des méthodes à appeler lors de la réception d'un message particulier). Cette liste est principalement modifiée via les méthodes « .on() » et « .off() », qui permettent respectivement d'ajouter un couple à la liste ou d'en retirer un.

Techniquement, pour émettre un événement dans Node.js, il faut appeler la méthode « .emit » d'un des objets « EventEmitter » disponible, avec comme argument le message incorporant l'événement désiré. Le rôle de cette méthode est de reconnaître, parmi les messages contenus dans la liste de l'objet, ceux qui correspondent à celui passé en argument et de déposer les appels de méthode associés dans la file d'attente centrale associée à la boucle d'événements de Node.js.

En définitive, le rôle de la boucle d'événements de Node.js revient à lancer les appels de méthodes entreposés dans sa file d'attente en respectant leur ordre d'arrivée (en mode premier entré, premier sorti). Le terme de boucle d'événements est donc peu pertinent dans ce contexte.

Précisons qu'il existe, en plus des objets « EventEmitter » conçus et créés par le développeur, des objets « EventEmitter » automatiquement créés au lancement de l'environnement d'exécution et qui sont conçus pour des événements standards. Ils s'occupent par exemple de traiter les messages liés à une connexion entrante ou à un clic de souris. Naturellement, dans le cas de ces objets, l'appel de la méthode « .emit » est géré automatiquement. Il est aussi à noter qu'il est possible de créer autant de ces objets que l'on souhaite, rendant donc possible la distribution des étapes de réception et de traitement des messages dans Node.js.

Dans la deuxième version d'Antoid, la boucle d'événements standard se retrouve dans chaque objet d'Antoid sous la forme d'une boucle de messages ; nous ne faisons toutefois pas la synthèse de la manière dont ils gèrent les événements ici, car nous la détaillons dans la section 3.2.

1.3 Le paradigme acteur

Dans sa première version, Antoid exploite le paradigme objet via le langage de programmation Javascript. Aujourd'hui, pour sa deuxième version, Antoid exploite le paradigme acteur défini par Carl Hewitt [25] via le langage de programmation Erlang et sa machine virtuelle Beam.

L'acteur est l'unité de traitement fondamentale de ce paradigme. Un acteur ne peut pas fonctionner seul, il doit faire parti d'un système où il travaille concurremment avec d'autres acteurs. Il communique avec ces derniers via l'envoi et la réception de messages.

1.3.1 Boîte aux lettres

Dans le paradigme acteur, les messages ne sont pas directement transmis à leurs destinataires, mais à la boîte aux lettres (Mail Box) dont dispose chaque acteur. Le rôle d'une boîte aux lettres est donc d'entreposer les messages destinés à l'acteur auquel elle est associée en attendant que ce dernier vienne les retirer.

Techniquement, un acteur délègue l'enregistrement des messages qui lui sont destinés à sa boîte aux lettres, ce qui permet de découpler l'émission et la réception d'un message. En effet, l'émetteur n'a pas besoin d'attendre que le récepteur soit prêt à recevoir son message, il le transmet à la boîte aux lettres qui le transmettra à son tour lorsque l'acteur auquel elle est associée souhaitera le retirer. Précisons que c'est bien l'opération de retrait qui correspond à l'étape de réception du message et non pas l'opération de dépôt du message dans la boîte aux lettres.

Lorsqu'on étudie le modèle acteur, la première propriété énoncée par son créateur, Carl Hewitt, est que tout est acteur. On pourrait donc légitimement se poser la question : une boîte aux lettres ne pourrait-elle pas être un acteur ? Carl Hewitt répond de la manière suivante : « the first property is that everything is an actor. [...] an actor has a mailbox, while a mailbox is an actor so now the mailbox needs a mailbox, this is a disaster ». Ainsi, si chaque boîte aux lettres était un acteur, chacune d'entre elles devrait avoir sa propre boîte aux lettres et ainsi de suite ... nous amenant à une infinité de boîtes aux lettres à partir d'un unique acteur. Ainsi, il est évident que les boîtes aux lettres ne peuvent pas être des acteurs, mais ce faisant, le paradigme produit une contradiction avec sa première propriété, en effet, en définitive, tout n'est pas acteur.

Bien que les boîtes aux lettres ne soient pas des acteurs, elles demeurent des entités définies dans le cadre du paradigme au même titre que les acteurs eux même. Ce point a son importance lorsque l'on étudie comment la mémoire de ces entités est gérée. En effet, le mode de gestion de la mémoire dépend du niveau dans lequel on se place. Il existe plusieurs niveaux, au premier (le niveau conceptuel) se trouve le paradigme acteur où chaque entité (boîte aux lettre ou acteur) gère elle-même sa propre mémoire. En revanche au niveau sous–jacent, qui dans notre contexte correspond au niveau de la machine virtuelle d'Erlang, c'est bien la machine virtuelle qui alloue et gère la mémoire des différentes entités. Nous pourrions aussi mentionner un niveau plus profond qui est celui du système d'exploitation. À ce niveau, c'est le système d'exploitation qui alloue et gère la mémoire de la machine virtuelle.

Avant de poursuivre, il est à noter que dans le travail de Carl Hewitt sur le paradigme acteur, le concept de boîte aux lettres a remplacé celui de « messager » qui était un acteur créé par un autre acteur pour porter un message à son destinataire. Or ce concept, contrairement à celui de la boîte aux lettres, ne contredisait pas la propriété : tout est acteur.

1.3.2 Le paradigme acteur dans un contexte événementiel

Dans une architecture pilotée par les événements, sans événements, pas d'actions. Les acteurs suivent une logique équivalente, s'ils n'ont pas de message à traiter, ils ne réalisent aucune action. Pour agir, un acteur traite les messages entreposés dans sa boîte aux lettres via une boucle de réception de messages que l'on peut assimiler à une boucle d'événements. À chaque itération, cette boucle retire un message contenu dans la boîte aux lettres de l'acteur afin de le traiter.

En introduction de ce mémoire, nous avons énoncé notre objectif de rendre notre moteur Antoid capable de supporter des jeux massivement multijoueurs et donc par extension capable de traiter le nombre massif d'événements produit par ces jeux. Toutefois, la première version d'Antoid ne dispose que d'une boucle d'événements unique (celle de Node.js) risquant d'engendrer, tôt ou tard, un goulot d'étranglement.

Rappelons cependant que techniquement, la boucle d'événements de Node.js ne traite pas des événements (ce rôle étant celui des objets « EventEmitter »), mais bien les appels de méthode associés à ces événements (pour plus de détails à ce sujet, vous pouvez consulter la section : 1.2.3). Il n'empêche qu'elle demeure un goulot d'étranglement potentiel lorsque le nombre d'événements et donc le nombre de fonctions à appeler augmente.

En transformant chacun de nos objets d'Antoid (les cases, les personnages et leurs artefacts) en acteur, nous distribuons notre boucle d'événements dans tous nos objets d'Antoid, réduisant de ce fait les risques d'apparition de goulots d'étranglement.

1.3.3 Actions des acteurs

Lorsqu'un acteur traite un message, il est en capacité de réaliser trois choses :

1. créer de nouveaux acteurs ;
2. envoyer des messages à d'autres acteurs dont il connaît l'adresse ;
3. déterminer comment réagir au prochain message qu'il reçoit.

Adresses des acteurs

L'unique moyen à disposition des acteurs pour communiquer avec d'autres acteurs est l'envoi de messages. Cependant, un acteur ne peut pas communiquer avec n'importe quel autre acteur du système. En effet, chaque acteur est associé à un identifiant que l'on pourrait assimiler à une adresse postale. Si un acteur ne connaît pas l'adresse d'un autre acteur, celui-ci est alors dans l'incapacité de lui envoyer un message.

Pour être informé de l'adresse d'un autre acteur, un acteur peut soit créer un nouvel acteur (l'addresse d'un nouvel acteur étant connue par l'acteur parent) soit recevoir cette adresse via un message. Un acteur peut par exemple communiquer sa propre addresse à un autre acteur dont il connaîtrait l'adresse.

Notons enfin qu'un acteur ne peut pas fabriquer l'adresse d'un autre acteur, dans le sens où il ne peut pas la deviner.

Modulation de l'enchaînement des action

Dans une architecture pilotées par les événements, les événements ne peuvent pas êtres traités dans n'importe quel ordre. Il serait par exemple étrange de voir un personnage brandir une épée avant de l'avoir dégainée de son fourreau. Pour éviter ce type d'incohérence, nous avons recours au concept de chaîne d'événements imposant un ordre dans lequel les événements liés doivent être traités (nous revenons sur ce concept dans la section 1.4.1).

C'est ici que la capacité des acteurs consistant à déterminer quoi faire du prochain message reçu intervient. Il est en effet possible pour un acteur de différer le traitement d'un message s'il ne correspond pas à un événement qui soit cohérent avec la situation. De cette manière, il lui est possible de traiter les événements dans un ordre particulier.

1.4 distribution de processus communicants

Le moteur de jeu Antoid est composé de deux sous-systèmes en interaction, d'une part l'environnement composé de cases et d'autre part les personnages et leurs artefacts qui évoluent en son sein ; chaque sous-système étant réparti entre les objets d'Antoid qui le compose.

Dans cette section, nous traitons de la répartition des données et des opérations dans le contexte d'une architecture pilotée par les événements en suivant les étapes essentielles du cycle consistant à transformer chaque événement émis en commande exécutable. Notre étude consiste pour chacune de ces étapes à déterminer comment les données sont réparties entre les différents espaces de stockage et comment les opérations sont réparties entre les différents opérateurs.

Nous choisissons un découpage en deux parties. Une première regroupant trois étapes applicables aux événements et une deuxième regroupant trois étapes applicables aux commandes.

1.4.1 Répartition relative aux événements

Dans cette sous-section, nous étudions comment les données et les opérations relatives aux événements sont réparties. Ces opérations correspondant aux étapes d'émission, de réception et de sélection applicables aux événements.

Étape d'émission

Chaque émetteur est un producteur d'événements ; les données événementielles émises et les opérations d'émission sont donc réparties entre les différents producteurs.

Dans Antoid, les événements proviennent des joueurs qui transmettent leurs commandes et des différents objets d'Antoid qui réagissent aux événements dont ils sont informés.

Il existe deux grands modes de diffusion des données événementielles émises. Le premier consiste pour l'émetteur à expédier les messages incorporant ces données à l'adresse des consommateurs (le flux poussé ou « push »). Le deuxième consiste pour le consommateur à copier les messages depuis l'adresse de l'émetteur (le flux tiré ou « pull »).

Dans le mode en flux tiré, les données événementielles sont réparties dans la mémoire de chaque émetteur, ce qui peut créer un goulot d'étranglement lorsque le nombre de consommateurs copiant les données émises par un même émetteur devient trop grand. Dans le mode en flux poussé, ces données sont réparties dans la mémoire de chaque consommateur, réduisant ainsi le risque d'apparition de goulots d'étranglement, mais augmentant sensiblement le travail de chaque émetteur.

Dans le domaine du jeu vidéo, les deux modèles peuvent cohabiter ; le système du jeu utilisant l'un ou l'autre en fonction de la situation. Millington et Funge [29] notent dans leur livre que le flux tiré est plus rapide que le flux poussé, mais qu'il perd en efficacité quand l'application est mise à plus grande échelle comme dans un jeu massivement multijoueurs où la quantité de consommateurs explose.

Rappelons que dans le paradigme des acteurs de Hewitt [25], les données événementielles ne leurs sont pas directement transmises. En effet, ces derniers utilisent un intermédiaire : leur boîte aux lettres (que nous détaillons dans la section : 1.3.1). En pratique, les acteurs combinent donc les deux modes de diffusion. Lorsque le message est émis, il est diffusé via un flux poussé de l'émetteur dans une boîte aux lettres. Lorsqu'il est reçu, il est diffusé via un flux tiré de cette boîte aux lettres vers le consommateur. Cette particularité est intéressante, car elle évite aux acteurs de se bloquer durant l'émission de messages.

Étape de réception

Chaque récepteur est un consommateur d'événement ; les données événementielles reçues et les opérations de réception sont donc réparties entre les différents consommateurs.

Dans le cadre du paradigme acteur, il existe autant de consommateurs qu'il existe d'acteurs alors que dans le cas d'une boucle d'événements standard celle-ci joue le rôle de consommateur unique. Il est aussi à noter que comme les acteurs avec leur boîtes aux lettres, la boucle d'événements standard dispose elle aussi une file d'attente dans laquelle les événements à traiter sont entreposés.

Dans Node.js, les consommateurs d'événements sont les objets « EventEmitter ». Rappelons que le qualificatif de « EventEmitter » est trompeur. En effet, ces objets ne sont pas des émetteurs d'événements, mais bien des consommateurs d'événements (nous présentons ces objets en détail dans la section : 1.2.3). Dans la première version d'Antoid, nous utilisions une dizaine de ces objets, chacun gérant un type d'événement particulier comme les connexions entrantes, les commandes des joueurs ou certains événements de jeu.

L'étape de réception termine le mécanisme de communication entre producteurs et consommateurs. Il est à noter que les communications peuvent s'effectuer soit de manière synchrone, soit de manière asynchrone.

On retrouve par exemple le concept de communications synchrones dans le modèle des processus séquentiels communicants, (Communicating Sequential processes, CSP) [27] proposé par C. A. R. Hoare via ce que l'on appelle des rendez-vous. Ces communications impliquent que l'émetteur et le récepteur se connaissent, soient reliés logiquement (e.g via une couche logicielle) ou matériellement (e.g par un câble) et que l'émission et la réception aient lieu concomitamment. En effet, si le récepteur n'est pas en train d'écouter pendant que le message est émis ou que l'émetteur n'émet pas alors que le récepteur est en train d'écouter, la communication ne peut avoir lieu.

On retrouve le concept de communications asynchrones par exemple dans le travail de Baker et Hewitt [6] ou celui de Brookes [8] avec le modèle des processus parallèles communicants (Communicating Parallel Processes, CPP). Celles-ci se différencient des communications synchrones en ce qu'elles ne nécessitent pas que l'émission et la réception se produisent concomitamment. L'attrait de ce type de communication pour l'émetteur étant qu'il n'a pas besoin d'attendre que le récepteur soit prêt à recevoir son message.

La solution permettant ce découplage entre l'émission et la réception de messages consiste à utiliser un tiers, capable non seulement de jouer le rôle de relais entre émetteur et récepteur, mais également de conserver durablement les messages de sorte qu'ils ne se perdent pas. La communication est alors divisée en deux temps : dans un premier temps, le tiers se synchronise avec l'émetteur et conserve le message émis (l'émetteur pouvant se libérer et faire autre chose) puis dans un second temps, il se synchronise avec le récepteur, celui-ci recevant son message.

Précisons que toute communication utilise un tiers. Par exemple, un câble est un tiers (que la communication soit synchrone ou asynchrone). Il existe cependant une différence significative entre un câble et la boîte aux lettres des acteurs. un message peut persister durablement dans une boîte aux lettres alors qu'un signal ne le peut pas dans un câble. Deux tiers donc, mais deux rôles différents. Le rôle d'un câble se limite à acheminer des signaux alors qu'une boîte aux lettres peut conserver des messages jusqu'à ce que le destinataire le retire.

Précisons enfin que la réception d'un message dans une communication asynchrone correspond au moment où le récepteur en prend connaissance et non pas quand il est mémorisé par le tiers.

Le micro-noyau L4 [17] est un exemple de couche logicielle capable de gérer ces deux types de communication. Dans le cas d'une communication synchrone, l'émetteur est bloqué par le système tant que le récepteur n'est pas prêt. En ce cas, c'est le tiers (logiciel) qui synchronise l'émetteur et le récepteur. Dans le cas d'une communication asynchrone le système sauvegarde le message, libère l'émetteur et transmet le message une fois le récepteur disponible. Précisons que l'émetteur et le récepteur peuvent se synchroniser sans l'intervention d'un tiers. En effet, l'émetteur peut émettre un même message autant de fois que nécessaire, tant qu'il n'est pas certain qu'il a été reçu. En ce cas, aucun intermédiaire ne mémorise le message. Il est aussi possible pour l'émetteur de contrôler que son message a bien été reçu en demandant un accusé de réception. Dans ce cas, la communication sera naturellement ralentie. Par exemple, le protocole TCP met en œuvre une telle option. Il faut toutefois noter que dans le cadre d'un jeu vidéo en ligne, cette augmentation de trafic est généralement indésirable. C'est pourquoi certains jeux font le choix de ne pas demander d'accusé de réception au risque de perdre certains messages. Par exemple, le protocole UDP met en œuvre cette option.

Étape de sélection

Ce sont les consommateurs qui opèrent la sélection des événements. Les opérations de sélection de l'événement qui doit être transformé en commande exécutable et les données relatives aux événements sélectionnés sont donc réparties entre les consommateurs. Ainsi, par exemple, avec la boucle d'événements standards elles sont centralisées alors qu'avec Erlang ou Node.js elles sont décentralisées (via les objets « EventEmitter » pour Node.js).

Que ce soit dans un jeu vidéo ou toute autre application pilotée par les événements, les événements ne peuvent pas êtres traités dans n'importe quel ordre. Il serait par exemple curieux de voir apparaître une bosse sur la tête d'un personnage avant que

celui-ci ne se cogne. Pour éviter ce type d'incohérence, certains événements doivent donc être traités dans un ordre précis.

Pour imposer un ordre strict dans lequel traiter les événements, Irene G. Greif [22], Carl Hewitt et Henry Baker [6] proposent, dans le cadre des processus parallèles communicants, le concept d'événements ordonnés consistant à organiser certains événements de manière à former des chaînes. Un événement ne peut alors être traité que si l'événement qui le précède dans sa chaîne a déjà été traité.

L'étape de sélection consiste à choisir le prochain événement à traiter. Il faut distinguer entre deux grands modes de sélection.

Le premier, consiste à utiliser le concept d'événements ordonnés et donc de différer le traitement d'un événement jusqu'à ce qu'il corresponde à celui qui est attendu par l'une des chaînes en cours (sauf s'il s'agit d'un événement initiant une nouvelle chaîne). Notons toutefois que pour sélectionner un événement ou différer son traitement, celui-ci doit tout de même être analysé par le consommateur. Or, ces analyses sont effectuées sur les événements dans l'ordre de leur réception.

Le second, consiste à traiter les événements dans leur ordre de réception (ce qui risque de provoquer des incohérences). Ce mode est utilisé par la boucle d'événements standard qui retire les événements de sa file d'attente en mode premier entré premier sorti (First in, First out). Il est également utilisé par les objets « EventEmitter » de Node.js qui traitent les événements au fil de l'eau. Attention, notons toutefois que bien que ce soit le cas pour la version standard de Node.js, on peut retrouver dans les travaux de D. Bonetta sur les « Parallel Event Loop » [7] , une extension des objets « EventEmitter » via de nouvelles méthodes permettant de marquer les événements appartenant à une même chaîne. Les commandes associées aux événements d'une même chaîne ne sont alors lancées que si toutes les commandes précédentes ont terminées leur exécution. Précisons que bien que l'on remarque une nuance (c'est le traitement des commandes qui est différé et non celui des événements), ce mécanisme de sélection est bel et bien opéré via le concept d'événements ordonné. Ce modèle des

« Parallel Event Loop » semble toutefois encore aujourd'hui au stade théorique et il n'en existe pas (à notre connaissance) d'implantation.

Dans Antoid, nous exploitons la capacité qu'ont les acteurs à redéposer un message dans leur boîte aux lettres pour différer le traitement d'un événement.

Pour illustrer cette problématique, nous allons nous servir d'une chaîne d'événements correspondant au déplacement d'un personnage.

Dans Antoid, un personnage se déplace en passant de case en case. Un déplacement est possible sauf si la case de destination oppose un obstacle. Cet obstacle pouvant être un autre personnage ou un mur.

La chaîne d'événements est la suivante :

1. le personnage initie son déplacement ;
2. la case accueillant le personnage (en charge d'appliquer les lois physiques du jeu au personnage qu'elle accueille) vérifie que la case de destination est disponible pour un déplacement ;
3. la case de destination transmet à la case d'accueil les informations nécessaires et conserve son état jusqu'à ce que la chaîne d'événements se termine (en effet, la case d'accueil a besoin de la garantie que l'état de la case de destination reste le même tant que le déplacement n'est pas terminée. Car si ce n'était pas le cas, les informations données à la case d'accueil pourraient devenir obsolètes, donc inutilisables) ;
4. la case d'accueil traite les informations reçues. Si le déplacement est possible, la case d'accueil notifie le succès du déplacement à la case de destination et au personnage. Sinon, la case d'accueil notifie l'échec du déplacement à la case de destination et au personnage ;
5. si le déplacement est un succès la case de destination accueille le personnage et redevient disponible pour entamer une nouvelle chaîne d'événement concernant un déplacement. Sinon la case de destination redevient simplement disponible pour entamer une nouvelle chaîne d'événement concernant un déplacement ;

Précisons qu'un consommateur, comme une case (ou tout autre objet d'Antoid), traite les messages qu'il reçoit dès qu'il le peut, parallèlement aux autres consommateurs. En revanche, la situation dans laquelle les chaînes en cours se trouvent peut influencer la manière dont les messages sont traités, comme nous allons le voir dans les deux cas de figure suivants.

L'exemple est le suivant : Deux personnages, A et B tentent de se déplacer sur une même case C concomitamment.

Dans le premier cas, la case de destination traite les événements dans leur ordre de réception, ce qui produit la séquence suivante :

1. La case de destination C reçoit successivement les messages de vérification de la case A puis de la case B ;
2. la case C répond à la case A en lui transmettant les informations nécessaires et conserve son état (pour permettre la bonne conduite de la chaîne) ;
3. la case C répond au message de vérification de la case B en lui indiquant qu'elle est indisponible pour un déplacement (car elle n'a pas encore reçu de réponse de la part de la case A) ;
4. la case A reçoit successivement un message indiquant que le personnage A qu'elle accueille est victime d'un sort de paralysie puis le message d'information de la case C ;
5. la case A traite le premier message et indique au personnage A qu'il est désormais paralysé ;
6. la case A traite les informations de la case C. Cependant, comme le personnage A est paralysé, elle notifie le personnage A et la case C que le déplacement est impossible ;
7. la case B notifie au personnage B que le déplacement est un échec ;
8. la case C traite la notification de la case A et redevient disponible pour entamer une nouvelle chaîne d'événements concernant un déplacement.

Dans ce cas de figure, la case de destination traite nécessairement la vérification de la case B avant la notification d'échec de la case A, et cela quel que soit l'architecture utilisée.

Dans le deuxième cas, le traitement d'un événement peut être différé (le scénario change à partir de l'étape 3), cela produit la séquence :

3. La case de destination C n'étant pas disponible, elle diffère la vérification de la case B (ce message étant arrivé avant la notification de la case A, il est traité en premier) ;
4. La case C traite la notification de la case A et redevient disponible ;
5. La case de destination répond à la vérification de la case B qui avait été différée en lui envoyant les informations la concernant et conserve son état (pour permettre la bonne conduite de la chaîne) ;
6. La case B traite les informations et confirme que le déplacement est possible. Elle notifie donc le personnage B et la case de destination du succès du déplacement.
7. La case de destination accueille le personnage B et redevient disponible pour entamer une nouvelle chaîne d'événement concernant un déplacement.

Dans le premier cas, aucun des deux personnages ne réussit à se déplacer, ce qui crée une incohérence. En effet, le personnage B aurait du pouvoir se déplacer car sa case de destination est restée vide. Or, dans le deuxième cas, le personnage B réussit à se déplacer. Par conséquent, pour éviter ce type d'incohérence, il faut pouvoir différer le traitement d'un événement en cas de besoin.

Il est à noter que si un message attendu n'arrivait jamais, le consommateur en attente de ce message se retrouverait bloqué. Il serait toutefois possible de développer une sorte de compte à rebours via une horloge interne. Par exemple en décrémentant un compteur chaque fois que le consommateur différerait un message. De sorte qu'après qu'un certain nombre de messages aient été différés (le message attendu n'étant toujours pas arrivé), la chaîne d'événements en cours soit abandonnée.

Précisons aussi qu'il ne faut pas confondre incohérence et iniquité. Nous pourrions par exemple nous retrouver dans une situation où le joueur contrôlant le personnage B a envoyé sa commande plus vite que le joueur contrôlant le personnage A, mais qu'une fluctuation dans la vitesse de communication a inversé l'ordre d'arrivée des commandes. Dans ce cas de figure, le jeu n'est pas équitable pour les joueurs, mais il reste cohérent.

1.4.2 Répartition relative aux commandes

Dans cette sous-section, nous étudions comment les données et les opérations relatives aux commandes sont réparties. Chaque opération correspondant aux étapes de sélection, d'allocation et d'exécution applicables aux commandes.

Il est à noter que dans le cadre de ce mémoire le terme de commande peut correspondre soit à un ordre d'exécution comme c'est le cas pour les appels de méthode dans le paradigme objet, soit à une demande (qui peut être refusée) comme c'est le cas dans le cadre du paradigme acteur.

Étape de sélection

Cette sélection consiste à transformer l'événement sélectionné en une commande exécutable. Ce sont les consommateurs qui opèrent cette sélection. Les opérations et les données correspondantes sont donc réparties entre les consommateurs.

Le mécanisme réalisant cette sélection diffère d'un environnement à l'autre, dans le cadre de cette section nous nous limitons aux environnements que nous avons utilisés pour développer les deux version d'Antoid, soit Node.js pour la première et Erlang pour la deuxième.

Dans Node.js, ce sont les objets « EventEmitter », qui (contrairement à ce que leur nom indique), reçoivent les événements et les transforment en commandes avant de les déposer dans une file d'attente sous la forme d'un appel d'une fonction de rappel

(callback). Fonctions qui seront effectivement appelées par la boucle d'événements de Node.js lors de l'étape suivante. [2]

Pour ce faire, chaque objet « EventEmitter » dispose d'une liste de couples associant une forme de message et une fonction devant être appelée lorsque cette forme correspond à un événement reçu. Lorsqu'un message est reçu, l'objet « EventEmitter » parcourt sa liste à la recherche de formes correspondantes. Quand une forme est reconnue, l'appel de la fonction associée est déposée dans la file d'attente de la boucle d'événements. Une fois la liste parcourue, qu'une correspondance ait été trouvée ou non, le message est effacé.

Précisons qu'il est possible de retrouver plusieurs exemplaires d'une même forme dans une liste. Dans ce cas de figure, pour chaque forme reconnue, l'objet « EventEmitter » dépose l'appel de la fonction associée dans la file d'attente (ces fonctions pouvant être différentes les unes des autres ou identiques). Notons que l'ordre dans lequel ces formes sont reconnues a son importance, car les appels de fonctions associées sont déposés dans la file d'attente dans cet ordre.

Cette liste de formes d'événements reconnaissables est à rapprocher du concept d'événements significatifs et non-significatifs que nous avons décrit dans la section 1.2.1 consacrée aux architectures pilotées par les événements. En effet, un événement est significatif pour l'objet qui le reçoit si au moins une association correspondant à cet événement est dans sa liste.

Il est à noter que la liste de couples d'un objet peut être modifiée au cours de son existence via deux méthodes : « .on » et « .off » permettant d'ajouter et de retirer des associations et donc de rendre significatif ou non-significatif un événement pendant l'exécution. Un exemple pratique de cette forme d'adaptation est illustré dans la première version d'Antoid par les objets « EventEmitter » responsables du traitement des commandes des joueurs. En effet, à chaque fois qu'un joueur se connecte via le

2. La boucle d'événements de Node.js est elle aussi mal nommée. En effet, elle ne traite pas des événements, mais lance des fonctions de rappel (pour plus de détails, vous pouvez consulter la section 1.2.3).

serveur d'authentification du jeu, de nouvelles associations sont ajoutées aux objets pour capter les commandes du nouveau joueur alors qu'au contraire, lorsqu'un joueur se déconnecte, des associations doivent être retirées. Nous pourrions même aller plus loin dans l'utilisation de ce concept. Prenons l'exemple d'un personnage paralysé qui ne peut donc plus bouger. Plutôt que de traiter les événements en provenance du joueur concernant des actions que le personnage est incapable de réaliser, il serait possible d'ignorer ces commandes, car elles ne sont pas significatives tant que le personnage reste paralysé.

Dans Erlang, les acteurs sont représentés par des processus Erlang. Chaque processus, pour réaliser cette étape, a recours à une structure de programmation que l'on appelle une boucle de réception de messages (receive loop). Cette structure est composée d'un certain nombre de blocs, chacun associant une forme de message à une commande à exécuter lorsque cette forme correspond au message reçu (notons que cette commande peut correspondre à plusieurs opérations ou juste une, comme un simple envoi de message).

Quand un processus Erlang analyse un message de sa boîte aux lettres, il le compare aux formes des messages de chaque bloc jusqu'à trouver une correspondance. Lorsque c'est le cas, le processus Erlang exécute la commande associée à la forme reconnue. Notons que contrairement à Node.js, la recherche s'arrête dès qu'une correspondance est trouvée. Il ne peut donc résulter du traitement d'un message qu'une seule commande et non plusieurs. Si aucune correspondance n'est trouvée, le message est laissé dans la boîte aux lettres (son traitement est différé) et le prochain message est analysé.

De manière standard, il est impossible de modifier la structure de réception de messages une fois le processus Erlang lancé. Cependant, cette option est indispensable pour implémenter le concept d'événements ordonnés. En effet, il ne suffit pas que l'opérateur responsable de la sélection soit capable de différer le traitement d'un événement, il doit aussi pouvoir modifier (en cours d'exécution) les événements sélectionnables de sorte qu'il puisse s'adapter à l'évolution d'une chaîne en cours. Or,

c'est via l'ajout et le retrait de formes de messages reconnaissables que ce filtrage a lieu d'un cycle à l'autre. Dans cette optique nous avons donc développé un dispositif que nous avons appelé des « boucles de réception de messages préprogrammées » que nous décrivons en détail dans la section éponyme 3.2.2.

Notons qu'il n'est pas fait référence dans le paradigme acteur à la manière dont le mécanisme de sélection doit être implémenté. En effet, il est simplement indiqué qu'un acteur peut déterminer quoi faire du prochain message reçu.

Étape d'allocation

L'étape d'allocation consiste à déterminer à quelle entité confier l'exécution de la commande précédemment sélectionnée.

Pour commencer, il nous faut établir une distinction entre les niveaux conceptuel, logiciel et matériel.

D'une part, bien qu'il soit par exemple conceptuellement possible, de faire travailler concomitamment de nombreux objets (au sens du paradigme objet), il est physiquement impossible pour un microprocesseur ne possédant que quatre cœurs de réaliser plus de quatre opérations en parallèle. De plus, même si l'on dispose d'une architecture matérielle adéquate, telle qu'une grappe d'ordinateurs permettant de réaliser physiquement autant de tâches qu'il existe d'objets, encore faut-il que l'environnement logiciel soit capable d'exploiter cette architecture. Or à titre d'exemple, Node.js, bien qu'il soit capable d'exploiter les différents cœurs d'un microprocesseur est incapable d'allouer du travail à de multiples ordinateurs dans une grappe.

D'autre part, la manière dont le travail est alloué change en fonction du niveau où l'on se trouve. En effet, on retrouve deux grands modes d'allocation du travail : via un superviseur qui organise seul la distribution du travail auprès de travailleurs (voir figure 1.5) ou via la collaboration des travailleurs d'égal à égal ou « peer to peer » (voir figure 1.6). Dans le modèle supervisé, c'est un dispositif centralisé qui est en charge de l'allocation. Les opérations et les données correspondantes sont donc elles

aussi centralisées. Alors que dans le modèle d'égal à égal, ce sont les consommateurs qui s'organisent entre eux. Les opérations et les données correspondantes sont donc réparties entre eux.

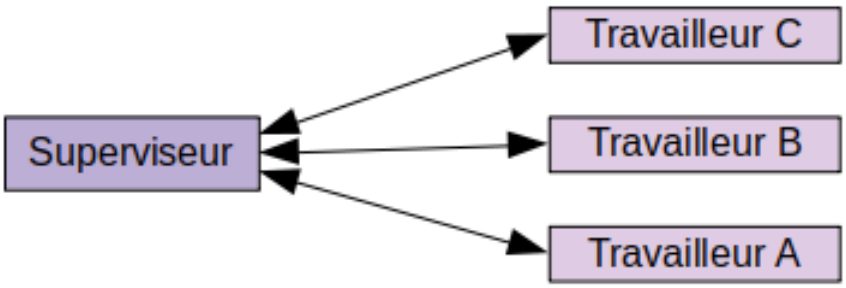

Figure 1.5 – Allocation via un superviseur

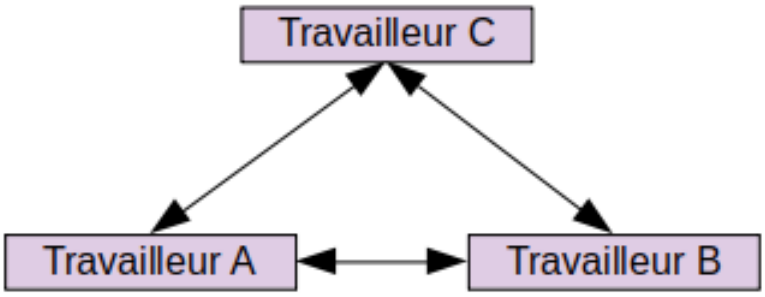

Figure 1.6 – Allocation d'égal à égal

Il est à noter que comme pour les modes de diffusion en flux poussé ou tiré, ces deux modèles d'organisation ont leurs avantages et leurs inconvénients. Il est par exemple plus simple dans le mode supervisé de répartir efficacement la charge (load balancing) entre les différents travailleurs, mais le superviseur peut devenir un goulot d'étranglement lorsque le flux de commandes devient trop important. À l'inverse, dans le mode d'égal à égal, le risque d'apparition de goulot d'étranglement est réduit, mais il devient plus difficile d'optimiser la répartition de la charge.

Pour illustrer ces deux modèles, nous allons encore une fois étudier Node.js et Erlang.

Node.js est basé sur le modèle supervisé. Il utilise comme superviseur sa (mal nommée) boucle d'événements. C'est donc elle qui alloue les travaux aux différents travailleurs.

Il est à noter que précédemment nous avions dit que toutes les commandes (correspondant à des appels de fonction de rappel) étaient regroupées dans une file d'attente, ce qui n'est pas tout à fait exact. En effet, il existe en réalité plusieurs files, chacune entreposant un type particulier de commande. On retrouve par exemple une file dédiée aux commandes associées à des comptes à rebours (encapsulées via la fonction « setTimeout »). Ces files sont parcourues dans un ordre particulier [31]. Ce mécanisme est d'ailleurs utilisé pour prioriser l'appel de certaines fonctions. Autre exemple, l'une de ces files stocke des commandes prioritaires (encapsulées via la fonction « setImmediate() ») que la boucle d'événements vide toujours avant les autres. Notons qu'il s'agit là du seul moyen offert par Node.js (de manière standard) pour changer l'ordre d'exécution des commandes, celui-ci est toutefois limité, car il ne permet pas de mettre en œuvre le concept d'événements partiellement ordonné. Les commandes priorisées sont en effet lancées dans leur ordre d'apparition dans cette file d'attente.

D'un point de vue logiciel, la boucle d'événements de Node.js peut allouer du travail auprès de différents travailleurs (workers) [32], chaque travailleur occupant un fil d'exécution (thread) indépendant. Par défaut, la boucle d'événements utilise l'algorithme « round-robin » pour répartir la charge de travail (load balancing), où les travailleurs reçoivent un travail à effectuer chacun leur tour.

D'un point de vue matériel, chaque travailleur est situé sur un des cœurs du microprocesseur. Cependant, il est impossible de disposer des travailleurs sur de multiples ordinateurs. Ce dernier point a contribué à l'abandon de Node.js pour le développement de notre moteur, car notre architecture matérielle ciblée est une grappe d'ordinateurs (pour plus de détails sur notre architecture matérielle, vous pouvez consulter la section 4.1).

La machine virtuelle d'Erlang, quant à elle, est basée à la fois sur le modèle d'égal à égal et sur le modèle supervisé.

Il est à noter que conceptuellement, dans le cadre d'Erlang, il existe deux manières de confier l'exécution d'une commande à un processus Erlang : en créant un nouveau processus (non-persistant) se terminant une fois la commande exécutée (son rôle se limitant à exécuter la commande) ou en envoyant un message à un processus persistant préexistant. Précisons cependant que, dans ce deuxième cas, il ne s'agit pas d'un appel de méthode distant (e.g un « Remote Procedure Call »), mais d'un message. Il est donc possible que cet autre processus, s'il ne reconnaît pas le message, refuse d'exécuter la commande, diffère son exécution ou encore n'exécute pas la commande de la manière attendue.

Comme Node.js, Erlang et sa machine virtuelle Beam permettent de distribuer le travail auprès de plusieurs cœurs via l'option « Symmetric MultiProcessing, SMP ». De manière standard, chaque cœur est occupé par une entité appelée un planificateur (« scheduler ») responsable de l'exécution d'un certain nombre de processus Erlang, chaque planificateur étant associé à une file d'attente entreposant les processus prêts à s'exécuter.

On retrouve deux mécanismes de répartition de la charge dans une machine virtuelle. Le plus simple consiste pour les planificateurs lorsqu'ils se retrouvent sans travail, à s'approprier une partie du travail d'un autre planificateur (task Stealing). Ce mécanisme, bien qu'il soit initié par le planificateur via l'appel de la fonction « try_steal_task » [4], est pris en charge de manière supervisée par la machine virtuelle qui va rechercher dans les files d'attentes des autres planificateurs un processus en attente d'exécution. Pour plus de détails au sujet du fonctionnement de la machine virtuelle d'Erlang, vous pouvez consulter le livre « The BEAM Book » de Erik Stenman [45].

Contrairement à Node.js, la machine virtuelle d'Erlang peut être distribuée dans une grappe d'ordinateurs, chaque machine virtuelle étant responsable de ses propres

planificateurs. Notons aussi que les machines virtuelles sont capables de s'allouer du travail d'égal à égal en passant par le réseau (de manière standard via le protocole TCP/IP). Prenons l'exemple d'un personnage qui se déplace dans le microcosme. Dans Antoid, chaque machine virtuelle héberge une partie des cases formant le microcosme et les personnages et les artefacts se trouvant sur ces cases. Contrairement aux cases, les personnages peuvent se déplacer dans le microcosme. Il est donc possible qu'ils se retrouvent au gré de leurs déplacements sur une case étant hébergée par une autre machine virtuelle. Or, dans ce cas de figure, c'est cette autre machine virtuelle qui doit prendre en charge ce personnage. Par conséquent, quand un personnage sort de la zone du microcosme hébergée par une machine virtuelle A pour se rendre dans une zone hébergée par une machine virtuelle B, la machine virtuelle A va confier la tâche d'héberger le personnage à la machine virtuelle B. Techniquement, en Erlang, cela revient pour la machine virtuelle A à supprimer le processus Erlang représentant le personnage et à demander à la machine virtuelle B de le recréer sur elle-même.

Il est à noter que nous apportons plus de détails au sujet d'Erlang dans le second chapitre de ce mémoire notamment en ce qui concerne les atouts du langages (qui ont orienté notre choix pour cette deuxième version d'Antoid), ses possibilités et sa syntaxe.

Étape d'exécution

Dans un système réparti où plusieurs commandes peuvent s'exécuter concomitamment, il peut arriver que plusieurs d'entre elles tentent de modifier une même variable, ce qui peut avoir pour effet de rendre cette variable incohérente.

Prenons l'exemple d'un tas de minerais auquel des personnages peuvent ajouter ou duquel ils peuvent retirer du minerai.

Techniquement, ce tas de minerais est une variable dont la valeur indique combien de minerais se trouvent actuellement dans le tas. Pour ajouter ou retirer une certaine quantité de minerais, les personnages ont recours à une commande qui mémorise la

valeur de la variable, ajoute ou retire la quantité voulue par le personnage à cette valeur en mémoire et remplace l'ancienne valeur de la variable par la nouvelle valeur.

Tant qu'un seul personnage (à la fois) modifie le tas de minerais, tout va bien, la valeur de la variable reste cohérente. Toutefois, si plusieurs personnages se mettent à modifier le tas de minerais en même temps, il existe un cas de figure où la valeur de cette variable devient incohérente.

En effet, imaginons que deux personnages ajoutent concomitamment du minerai à un tas qui en contient déjà dix. Le personnage A en ajoute deux et le personnage B quatre (ce qui devrait donner au final un tas de seize minerais).

Notre cas de figure est le suivant :

1. pour commencer, les deux commandes déclenchées par nos personnages mémorisent la valeur correspondant au nombre de minerais que contient le tas, soit dix (10) ;
2. ensuite, chaque commande y applique l'opération demandée par son personnage, la commande A se retrouve donc avec une nouvelle valeur indiquant douze (12) minerais et la commande B quatorze (14) ;
3. admettons maintenant, que la commande A soit la plus rapide, elle remplace la variable (qui indique 10 minerais) par sa nouvelle valeur pour que le tas compte désormais douze (12) minerais ;
4. la commande B remplace enfin la variable (qui indique 12 minerais) par sa nouvelle valeur pour que le tas compte désormais quatorze (14) minerais.

Ce que l'on remarque, c'est qu'alors que le tas aurait dû contenir seize (16) minerais, celui-ci n'en compte que quatorze (14). On peut donc affirmer que la valeur de la variable est bel et bien devenue incohérente.

Pour éviter ce problème, il est possible d'avoir recours au concept d'exclusion mutuelle via des outils comme les sémaphores qui permettent de verrouiller l'accès à une variable tant qu'une commande effectue des opérations sur celle-ci.

Dans notre exemple, cela veut dire que si la commande A avait été la plus rapide ; tant qu'elle n'aurait pas terminé d'effectuer ses opérations concernant la variable, (i.e. mémoriser sa valeur et la remplacer par une nouvelle valeur), la commande B n'aurait pas eu la capacité d'y accéder (i.e. ni de lire sa valeur, ni de la remplacer par une autre valeur).

Il est à noter que cette problématique n'existe pas dans le cadre de la deuxième version d'Antoid. En effet dans ce contexte, le tas de minerais prend la forme d'un acteur (un processus Erlang) auquel il faut envoyer un message pour lui demander d'ajouter ou de retirer du minerai. Or comme l'acteur est le seul capable d'agir sur son état, notre cas de figure ne pourra jamais apparaître. Notons aussi que cette solution est tout à fait applicable dans le cadre du paradigme objet via le concept d'encapsulation où un objet unique se voit déléguer la tâche d'ajouter ou retirer du minerai via l'appel de ses méthodes.

Chapitre 2

Langage de programmation Erlang et machine virtuelle Beam.

Au commencement de ce projet de maîtrise, la décision de rechercher un successeur à l'environnement Node.js a été prise. En effet, bien que Node.js ait rempli son rôle dans le développement d'une première version de notre moteur Antoid, nos exigences pour cette deuxième version ont évolué.

Premièrement, il était impératif que ce nouvel environnement soit capable d'implémenter de manière standard le paradigme acteur de Carl Hewitt [25]. Deuxièmement, étant donné que le contexte de notre expérience est celui d'un jeu massivement multijoueurs, il était important que cet environnement soit capable de mettre en scène un très grand nombre d'objets d'Antoid (cases, personnages et artefacts) en interaction. Troisièmement, ce nouvel environnement devait être capable de s'adapter à l'architecture matérielle que nous avions choisi, c'est-à-dire une grappe d'ordinateurs. En d'autres termes, l'environnement devait être apte à distribuer Antoid entre plusieurs ordinateurs reliés ensemble dans un réseau. Enfin, Antoid étant un jeu vidéo, un moyen de communiquer facilement avec les joueurs devait être disponible.

Parmi les environnements répondant à ces critères, celui qui a retenu notre attention est le langage de programmation Erlang [19] et sa machine virtuelle Beam

(Bogdan/Björn's Erlang Abstract Machine). Notamment, car bien que ce soit un langage éprouvé et utilisé dans de nombreux domaines dont celui du jeu vidéo, ce n'est pas un langage que nous avons rencontré au cours de notre cursus académique. Il est aussi à noter que notre démarche n'a pas été de tester différents environnements dans l'optique de sélectionner le plus performant, mais simplement d'en trouver un qui nous permette d'expérimenter nos algorithmes distribués.

Ajoutons que consacrer une part conséquente de ce projet de maîtrise au choix d'un environnement n'aurait pas été judicieux, car d'une part, nous ne voulions exploiter qu'une petite partie de ce qu'Erlang et Beam ont à proposer (ce que nous décrivons dans ce chapitre) et d'autre part nous ne souhaitions pas capitaliser sur ce langage. En effet, un de nos objectifs sur le long terme est de développer notre propre machine virtuelle légère dans le langage de programmation C (Beam étant d'ailleurs elle aussi développée en C).

2.1 Atouts d'Erlang

Avant d'aborder cette section, il est important de noter que bien que nous utilisons généralement le terme d'Erlang pour désigner les différentes fonctionnalités de notre environnement logiciel, c'est avant tout par souci de simplification. En effet, comme nous l'avons dit, nous n'utilisons qu'une petite partie d'Erlang et de Beam, or cette partie correspond à des fonctionnalités de la machine virtuelle Beam. Il nous aurait par exemple été tout à fait possible d'utiliser d'autres langages basés sur Beam tel qu'Elixir en lieu et place d'Erlang pour développer cette deuxième version d'Antoid.

2.1.1 Processus légers

Beam offre un gestionnaire de processus légers [1]. Ces processus légers (que nous appelons des processus Erlang) sont créés et pris en charge directement par la machine virtuelle, de sorte qu'elle se montre capable de maintenir un grand nombre de processus Erlang en interaction.

En effet, Beam, dans sa version actuelle permet, par défaut d'atteindre jusqu'à 32 768 processus Erlang. Ce nombre peut toutefois être dépassé pour atteindre plusieurs centaines de milliers de processus Erlang pourvu que la machine virtuelle ait accès à suffisamment de mémoire. À titre d'exemple, nous sommes parvenu dans le cadre de nos expériences à maintenir 141 913 processus Erlang sur un seul Raspberry PI (RPI) ne disposant pourtant que de 1Go de mémoire vive. Chaque objet d'Antoid (les cases, les personnages et leurs artefacts) étant représenté par un processus Erlang.

2.1.2 Messagerie inter-processus universelle

Outre son gestionnaire de processus, Beam offre un gestionnaire de messagerie permettant aux processus Erlang de communiquer entre eux via l'émission et la réception de messages. Chaque processus Erlang est rattaché à une boîte aux lettres dans laquelle sont entreposés tous les messages lui étant destinés (nous revenons plus en détail sur ces boîtes aux lettres dans la section 1.3.1).

Il est à noter, que les communications, du point de vue d'un processus Erlang, se déroulent de manière transparente que son interlocuteur soit situé sur le même nœud (i.e. la même machine virtuelle), sur un autre nœud du même processeur ou même sur un nœud situé sur un autre ordinateur. Ceci est un atout dans la mesure où pour augmenter le nombre d'objets d'Antoid notre stratégie consiste simplement à ajouter de nouveaux ordinateurs à notre grappe.

Naturellement, techniquement, ces communications ne sont pas toutes gérées de la même façon par Beam. Le cas le plus simple et le plus rapide est celui d'une communication entre deux processus Erlang situés sur un même nœud. Dans ce cas de figure, le gestionnaire de messagerie du nœud achemine directement le message de l'émetteur vers la boîte aux lettres de son destinataire. Dans le cas où une communication implique deux nœuds différents (même situés sur un même processeur), les deux machines virtuelles ont recours au protocole de communication TCP/IP. Précisons toutefois qu'il ne s'agit là que d'une option par défaut et qu'il est par exemple possible de choisir un autre protocole comme UDP (plus rapide, mais sujet à des pertes de messages) ou même d'en définir un nouveau. Dans le cadre d'Antoid, nous

avons fait le choix de conserver le protocole TCP/IP pour réduire le risque de perte de messages.

2.1.3 Ports Erlang

Dans un jeu vidéo, il est impératif de pouvoir communiquer avec l'extérieur et plus particulièrement avec les joueurs via leur interface utilisateur. Dans cette optique, Erlang offre les «Ports Erlang» qui offrent un moyen de communication entre un programme Erlang et un programme écrit en C ou C++.

Du point de vu d'Erlang, un port est composé de deux éléments : un programme externe écrit en C/C++ et un processus Erlang qui sert d'interface avec ce programme externe. Le processus Erlang et le programme externe communiquent de manière bidirectionnelle via des messages binaires.

Dans la version actuelle d'Antoid, nous n'utilisons pas encore ces ports Erlang. En effet, dans le cadre de cette maîtrise, nous nous sommes concentrés sur les personnages non-joueurs. Il est toutefois prévu d'utiliser les ports Erlang pour interfacer le moteur de jeu à un serveur Node.js (tiré de la première version d'Antoid) chargé de gérer les joueurs. Notons toutefois que nous avons pu nous assurer durant notre projet de maîtrise qu'il est bel et bien possible pour notre moteur de communiquer avec l'extérieur via un port Erlang. Pour plus de détails concernant ces ports Erlang, vous pouvez consulter la documentation d'Erlang à ce sujet [34].

2.2 Programmation en Erlang

Erlang est un langage multi-paradigmes couvrant des aspects fonctionnel, concurrent et événementiel.

Précisons bien que nous ne faisons ici qu'un bref tour d'horizon des spécificités d'Erlang qui seront utiles à la compréhension de certains concepts présentés dans le

chapitre 3 [1].

2.2.1 Processus Erlang et fonctions

À la base de la programmation en Erlang, nous retrouvons le processus Erlang. Chaque processus Erlang lorsqu'il est créé est associé à une fonction à exécuter. Quand l'exécution de la fonction se termine, le processus Erlang est détruit. Ces fonctions sont composées d'une séquence d'instructions et de structures de programmation standards (alternative, itérative) Notons toutefois qu'en Erlang, seul le concept de récursivité est disponible pour réaliser des itérations (sauf cas particulier de la boucle de réception de messages qui est une structure de programmation standard d'Erlang sur laquelle nous revenons plus tard). De plus, contrairement à des langages tel que C ou Java, les variables sont non mutables.

Il est à noter qu'Erlang possède aussi quelques subtilités au niveau de sa syntaxe qui le différencie des langages plus classiques. Par exemple, une variable en Erlang commence impérativement par une majuscule et les commentaires par le signe de pourcentage (%). Un autre point important est celui de la ponctuation. En effet, tandis que la plupart des langages utilisent un symbole unique, Erlang utilise les trois signes suivants dans sa syntaxe :

1. la virgule (,) -> que l'on peut assimiler à un « et ». Elles servent à séparer d'une part, les paramètres dans une fonction ou une liste par exemple et d'autre part les instructions d'une même clause ;
2. le point virgule (;) -> que l'on peut assimiler à un « ou ». Il marque la fin d'une clause ;
3. le point (.)-> qui marque la fin d'un énoncé, comme une fonction ou une structure de programmation ;

Nous illustrons le fonctionnement de cette ponctuation dans le programme 2.1 représentant une fonction qui évalue deux chiffres avant d'afficher et retourner le

1. Précisons que dans le cadre du développement de cette deuxième version d'Antoid, nous avons principalement eu recours aux deux ouvrages suivants : [2], [24]

plus grand. Notons qu'il n'existe pas d'instruction pour retourner le résultat d'une fonction, celle-ci retournant toujours le résultat de la dernière expression évaluée.

programme 2.1 – Ponctuation dans Erlang

```
wich_is_greater(X, Y) ->

    if
        X > Y ->
            display(X),
            X;

        true -> %% fonctionne comme un 'else'
            display(Y),
            Y       %% la derniere clause ne se termine jamais
                    %% par un point-virgule
    end.
```

Créer un processus Erlang, passe par la commande «spawn», qui prend en argument le nom d'une fonction et une liste des paramètres à passer en argument. Ainsi, si nous voulons exécuter une fonction « bonjour » avec les paramètre «1,2,3» nous aurions le code du programme 2.2.

programme 2.2 – Création de processus Erlang via la commande spawn

```
spawn(bonjour,[1,2,3]).
```

Un processus Erlang peut aussi correspondre à un enchaînement de fonctions, chaque fonction appelant la fonction suivante, juste avant de s'achever. De cette manière, le processus Erlang s'arrêtera lorsque la dernière fonction aura terminé son exécution. Cet enchaînement peut toutefois parfaitement ne pas avoir de fin ; rendant le processus Erlang persistant, sauf en cas d'accident.

Il est à noter que cet enchaînement ininterrompu d'appels de fonctions ne produit pas un empilement de fonctions dans la pile d'exécution. En effet, dans Erlang, lorsque la dernière instruction d'une fonction correspond à l'appel d'une fonction (qui peut être la même ou pas), cela correspond à une récursion terminale (tail-recursion). Plus précisément, la fonction terminant son exécution est retirée de la pile d'exécution pour laisser la place à la fonction appelée évitant ainsi une surconsommation de la mémoire disponible. Cela n'est d'ailleurs pas sans rappeler le concept de continuation.

Le programme 2.3 présente un enchaînement de fonctions se passant en argument un paramètre.

programme 2.3 – Processus Erlang : persistance des données

```
fonctionA(Parametre1) ->
        operations,
        fonctionB(Parametre1).
fonctionB(Parametre1) ->
        operations,
        fonctionC(Parametre1).
fonctionC(Parametre1) ->
        ...
```

Il est à noter qu'en Erlang, le type de paramètre attendu par une fonction n'est jamais défini. Il est par exemple possible d'appeler plusieurs fois une même fonction tout en lui passant parfois un entier et d'autres fois une chaîne de caractères. Précisons toutefois que Erlang est un langage fortement typé, une fonction ne convertira donc jamais un entier en chaîne de caractères comme peut le faire un langage tel que Javascript par exemple.

Dans cette version d'Antoid, chaque objet d'Antoid est représenté par un processus persistant et chaque fonction représente une itération de son existence. Or, une donnée ne persiste que tant que la fonction persiste. Pour conserver ou faire évoluer son état, un objet d'Antoid doit donc avoir recours au passage de paramètres en argument

lors de chaque nouvel appel de fonction. Pour plus de détails à ce sujet, vous pouvez consulter la section 3.2.2.

2.2.2 Communication entre processus Erlang

Les processus Erlang ne peuvent envoyer de messages qu'à des processus dont ils connaissent l'identifiant que l'on qualifie ici de PID (Process IDentifier). Ce PID, s'il n'est pas communiqué via un message n'est connu que par le créateur du processus et le processus lui-même.

Le programme 2.4 présente un processus Erlang envoyant son PID via un message à un autre processus Erlang. Précisons que le point d'exclamation représente la commande d'envoi de message et la commande «self()» retourne au processus Erlang son propre PID.

programme 2.4 – Emission de message Erlang

```
PID_Destinataire ! {self()}.
```

Précisons aussi que le PID est déterminé par la machine virtuelle et non le développeur et qu'un objet d'Antoid ne conserve pas son PID d'une expérience à l'autre. Ainsi, pour reconnaître nos objets d'Antoid d'une expérience sur l'autre, nous leur avons assigné un autre identifiant qui lui reste fixe.

Pour retirer les messages contenus dans sa boîte aux lettres, un processus Erlang utilise une boucle de réception de messages (receive loop). Cette boucle est une structure de programmation illustrée par le programme 2.5.

programme 2.5 – La boucle de réception de messages

```
receive
    {Forme de message 1} ->
        Operations;
    ...;
    {Forme de message N} ->
        Operations
end.
```

La structure présente un nombre variable de blocs correspondant chacun à une forme qu'un message pourrait avoir (un « Message Pattern ») associée à une commande à lancer (une « Linked Command »). La boucle se termine lorsqu'un message est reconnu. Si la boucle ne reconnaît aucun des messages de la boîte aux lettres, celle-ci se mettra en attente de nouveaux messages et ce jusqu'à ce qu'un message corresponde.

Techniquement, une forme de message en Erlang correspond à un n-uplet de paramètres. Par exemple, si nous souhaitons reconnaître un message contenant les données 1 et 2 dans cet ordre nous aurions le programme 2.6.

programme 2.6 – Reconnaître une forme avec la boucle de réception de messages

```
receive
    {1,2} ->
        Operations;
end.
```

Techniquement, le gestionnaire de messagerie de la machine virtuelle d'Erlang (Beam) fonctionne de la manière suivante pour l'envoi d'un message :

1. l'objet d'Antoid voulant émettre un message enregistre ce message dans son espace mémoire ;
2. l'émetteur demande ensuite l'accès exclusif à la boîte aux lettres du destinataire ;

3. une fois l'accès accordé, l'émetteur enregistre dans la boîte aux lettres, un pointeur correspondant à l'adresse où est mémorisé le message ;
4. enfin, une fois le pointeur enregistré, l'accès est déverrouillé.

Le mécanisme de retrait, fonctionne de la manière suivante :

1. l'objet d'Antoid demande l'accès au verrou de sa boîte aux lettres ;
2. une fois l'accès accordé, il prend connaissance du pointeur dans l'optique de lire le message ;
3. enfin, l'accès est déverrouillé.

Précisons aussi que le dépôt et le retrait d'un message ne peuvent pas s'effectuer concomitamment et que les messages ne sont enregistrés qu'en un seul exemplaire.

2.2.3 Banc d'essai

Nous présentons ici un petit programme Erlang ayant pour objectif de comparer la vitesse à laquelle deux processus s'échangent des messages lorsque les deux processus sont :

1. situés sur un même nœud ;
2. situés sur deux nœuds différents sur la même machine ;
3. situés sur deux nœuds différents sur deux machines différentes reliées via un câble Ethernet ;
4. situés sur deux nœuds différents sur deux machines différentes reliées via WiFi.

Ce banc d'essai présente un test de « ping/pong ». Un processus A envoie un message « ping » à un processus B qui répond par un message « pong » et cela un certain nombre de fois. À chaque échange, le processus A envoi un message à un processus C afin de l'informer du délai entre l'émission et la réception du dernier échange. À la fin de l'expérience, le processus C sauvegarde toutes les informations reçues dans un fichier.

Le code pour le processus A est présenté dans le programme 2.7. Ce processus envoie un message « ping » à un processus B. À chaque fois qu'un message « pong » est reçu, un nouveau message « ping » est envoyé au processus B. Ce processus envoie aussi un message contenant les délais de chaque ping/pong à un processus C. Le processus A envoie 10 000 messages en tout.

programme 2.7 – Le processus A

```
%% PongProcess : adresse du processus B
%% SaveProcess : adresse du processus C
%% Number : nombre iteration
%% TS1,TS2,TS3 : les timestamps du message pour calculer le delai
% en megasecondes, secondes et microsecondes

start(PongProcess,SaveProcess,Number) ->

  PongProcess ! (ping,self(),Number), % envoi du premier ping
  loop(PongProcess,SaveProcess,Number+1). % lancement de la boucle

loop(PongProcess,SaveProcess,Number) ->

  receive

  {pong,10001,TS1,TS2,TS3} -> % reception du dernier pong,
                              % fin du banc d'essai

    SaveProcess ! {write}, % demande ecriture des delais (fichier)
    PongProcess ! {terminate}, % demande au processus B de finir
    exit(normal); % termine le processus A

  {pong,PongNumber,TS1,TS2,TS3} -> % reception d'un pong

    {TS4,TS5,TS6} = os:timestamp(), % generation du timestamp
    % envoi les timestamps du ping/pong %
    SaveProcess ! {save,[{PongNumber,TS1,TS2,TS3,TS4,TS5,TS6}]},
    PongProcess ! {ping,self(),Number}, % envoi d'un ping
    % lancement de la prochaine iteration %
    loop(PongProcess,SaveProcess,Number+1);

  end.
```

Le code pour le processus B est présenté dans le programme 2.8. Ce processus renvoi un message « pong » aux processus qui lui envoient des messages « ping ».

programme 2.8 – Le processus B

```
%% Emiter : adresse du processus A
%% Number : nombre iteration
%% TS1, TS2, TS3 : les timestamps du message pour calculer le delai

start() ->

  loop(). % lancement de la boucle

loop() ->

  receive

  {ping, Emiter, Number} -> % reception d'un ping

    {TS1, TS2, TS3} = os:timestamp(), % generation du timestamp
    Emiter ! {pong, Number, TS1, TS2, TS3}, % envoi d'un pong
    loop(); % lancement de la prochaine iteration

  {terminate} -> % reception du message de fin

    exit(normal) % termine le processus B

  end.
```

Le code pour le processus C est présenté dans le programme 2.9. Ce processus enregistre les messages qu'il reçoit dans une liste en vue de les écrire dans un fichier quand le banc d'essai est terminé.

programme 2.9 – Le processus C

```
%% List : liste des ping/pong et de leurs delais
%% Event : un ping/pong et son delai

start() ->

  List = [], % creation de la liste (vide)
  loop(list). % lancement de la boucle

loop(List) ->

  receive

  {save,Event} -> % reception d'un ping/pong et son delai

    NewList = List ++ Event, % ajouter dans la liste
    loop(NewList); % lancement de la prochaine iteration

  {write} -> % reception du message de fin

    % ecrit dans le fichier bench %
    file:write_file("bench",
      lists:flatten(io_lib:format("~p",[List])),[append]),
    exit(normal); termine le processus C

  end.
```

Résultats du banc d'essai

Pour l'analyse de ces résultats, nous prenons la vitesse moyenne de l'envoi d'un message entre deux processus sur un même nœud (la plus rapide) comme unité de comparaison.

1. deux processus situés sur deux nœuds différents sur la même machine. La communication entre les deux processus prend en moyenne 3 fois plus de temps ;
2. deux processus situés sur deux nœuds différents sur deux machines différentes reliées via câble Ethernet. La communication entre les deux processus prend en moyenne 7 fois plus de temps.
3. deux processus situés sur deux nœuds différents sur deux machines différentes reliées via WiFi. La communication entre les deux processus prend en moyenne 30 fois plus de temps.

Chapitre 3

Distribution d'Antoid

Notre projet consiste à créer un moteur de jeu expérimental : Antoid. Actuellement, il s'agit uniquement d'en construire de solides fondations, de sorte qu'il soit possible de le distribuer efficacement dans une grappe d'ordinateurs dont la taille augmente progressivement.

Nous exposons dans ce chapitre notre solution, consistant via le passage du paradigme objet au paradigme acteur à :

— lutter contre l'apparition de goulots d'étranglement en distribuant la boucle d'événements d'Antoid de manière à ce que chaque objet d'Antoid possède la sienne propre ;
— maintenir la cohérence du système, aujourd'hui distribué, en dotant chaque objet d'Antoid de la capacité à définir comment traiter le prochain événement dont il sera informé sans perte d'information.

Précisons que bien que le chapitre 2 de ce mémoire, soit consacré à Erlang, il nous arrive dans ce chapitre, d'illustrer certains aspects conceptuels avec des éléments techniques tirés d'Erlang.

3.1 Du paradigme objet au paradigme acteur

Dans sa première version, Antoid exploite le paradigme objet via le langage de programmation Javascript. Aujourd'hui, pour sa deuxième version, Antoid exploite le paradigme acteur défini par Carl Hewitt [25] via le langage de programmation Erlang et sa machine virtuelle Beam.

Dans cette section, nous présentons l'évolution des objets d'Antoid du paradigme objet vers le paradigme acteur en mettant l'accent sur trois capacités que nous considérons comme essentielles : la capacité à persister, à moduler l'enchaînement de ses actions et à créer de nouveaux objets d'Antoid.

3.1.1 Persistance

`Sans persistance, pas de modulation de l'enchaînement des actions`

Dans la première version d'Antoid, où nous exploitons le paradigme objet, chaque objet d'Antoid est un objet Javascript dont la persistance est assurée par la persistance de ses valeurs d'attributs.

Dans la seconde version d'Antoid, où nous exploitons le paradigme acteur, chaque objet d'Antoid est représenté par un acteur. Cependant, techniquement, nous utilisons le paradigme fonctionnel via Erlang, où chaque objet d'Antoid est représenté par un enchaînement ininterrompu de fonctions, chaque fonction appelant la prochaine fonction avant de se terminer. C'est grâce à l'enchaînement ininterrompu de différentes fonctions que nous appelons « itérations », que l'objet d'Antoid persiste (nous détaillons le déroulement d'une itération type dans la section 3.2.1

Notons que contrairement à la première version d'Antoid, les objets d'Antoid ne possèdent plus d'attributs dans lesquels stocker leur état. C'est pourquoi, dans cette seconde version, nous utilisons les paramètres des fonctions qui les représentent pour

faire persister leur état (pour plus de détails à ce sujet, vous pouvez consulter la section 3.2.2).

Il est à noter que pour moduler son action, un objet d'Antoid doit réactualiser son état d'une itération sur l'autre. Cette capacité à persister via l'enchaînement ininterrompu de fonctions est donc incontournable.

3.1.2 Modulation de l'enchaînement des actions

Dans la deuxième version d'Antoid, nous avons opté pour une architecture entièrement décentralisée, où chaque objet d'Antoid est responsable du maintien de sa propre cohérence ; l'idée étant que si l'état de tous les objets d'Antoid reste cohérent, l'état global du jeu reste lui aussi cohérent.

Pour rendre cela possible, nous avons recours au concept d'événements ordonné (que nous détaillons dans la section 1.4.1), via la mise en place dans chaque objet d'Antoid de chaînes d'événements dont le rôle est d'imposer un ordre de traitement des événements évitant aux objets d'Antoid d'entrer dans un état incohérent. Cependant, pour respecter ces chaînes d'événements, les objets d'Antoid doivent pouvoir moduler l'enchaînement de leurs actions, ou dans notre contexte, pouvoir sélectionner quels événements traiter d'itération en itération (chaque événement menant à une action différente).

Dans le paradigme acteur, cette capacité fait référence au fait qu'un acteur peut définir comment réagir au prochain message qu'il reçoit. Ainsi, lorsqu'un message correspond à un événement n'étant pas en accord avec une chaîne d'événements, il peut décider de différer son traitement. Précisons toutefois que le concept de chaînes d'événement n'existe pas de manière standard en Erlang, par conséquent, nous avons développé notre propre outil que nous avons appelé les « boucles de réception de messages préprogrammées » que nous détaillons dans la section éponyme 3.2.2.

Il nous faut aussi noter que bien que nous ne l'ayons pas réalisé pour notre première version d'Antoid, Node.js permet (de manière standard) la mise en place de cette ca-

pacité. En effet, les objets « EventEmitter » (que nous détaillons dans la section 1.2.3) disposent de méthodes offrant des services de communication et de traitement d'événements (la méthode .emit) d'une part, et de mise en œuvre de chaînes d'événements (les méthodes .on et .off) d'autre part. En revanche, définir quoi faire avec le prochain message signifie également qu'il doit être possible de différer son traitement ; chose impossible pour ces objets de manière standard. Ainsi, un message n'étant pas en accord avec une chaîne d'événement en cours n'aurait pas été différé mais perdu.

3.1.3 Création

Que nous soyons dans le cadre du paradigme objet ou du paradigme acteur, les objets comme les acteurs ont la capacité de créer de nouveaux objets ou de nouveaux acteurs [1].

Nous devons toutefois préciser que contrairement aux acteurs qui ont tous la capacité de créer de nouveaux acteurs, tous les objets d'Antoid n'ont pas la capacité de créer de nouveaux objets d'Antoid. En effet, cette capacité est exclusive aux cases formant l'environnement d'Antoid. Ainsi, quand un personnage souhaite construire un mur, il initie bel et bien cette action de jeu, mais c'est une case qui crée le mur sur elle-même. Cette capacité pour les cases à créer de nouveaux objets d'Antoid est fondamentale puisqu'elle est à la base de notre algorithme de création de microcosme à partir d'une case initiale unique (pour plus de détails à ce sujet, vous pouvez consulter la section 3.3.1).

Il est aussi à noter que les objets d'Antoid ne créaient pas de nouveaux objets dans la première version d'Antoid. En effet, toutes les cases étaient créées au lancement du jeu et la création des personnages n'était pas prise en charge par l'environnement d'Antoid mais par un serveur Node.js.

1. Précisons que dans Javascript la création de nouveaux objets passe par le concept de prototype et non de classe.

3.2 Nouveaux objets d'Antoid

Dans cette nouvelle version d'Antoid, nous avons cherché via le passage du paradigme objet au paradigme acteur à localiser dans chaque objet d'Antoid le traitement des événements qui le concernent. Traitement qui était dans la version précédente d'Antoid, pris en charge d'une part par les objets « EventEmitter » de Node.js et d'autre part par la boucle d'événements de Node.js. Pour ce faire, chaque objet d'Antoid exploite désormais une boucle de messages, correspondant techniquement à la boucle de réception de messages d'Erlang. Rappelons que dans un contexte acteur, le terme de boucle de messages (message loop) est équivalent à celui de boucle d'événements (event loop).

Cette boucle de messages couvre les trois premières étapes sur cinq d'une itération type d'un objet d'Antoid. Soit la réception de messages, la reconnaissance de formes des messages reçus et la sélection de commandes à exécuter correspondant aux messages reconnus.

Les deux étapes suivantes, l'exécution de commandes et la réactualisation de l'état de l'objet d'Antoid sont prises en charge directement par l'objet d'Antoid (dans l'ancienne comme dans la nouvelle version). Ainsi, en distribuant la boucle de messages, l'ensemble de ces cinq étapes est désormais pris en charge individuellement par l'objet d'Antoid.

3.2.1 Itération type de la boucle d'Antoid

Comme nous pouvons le voir dans la figure 3.1, chaque itération comporte en tout cinq étapes : la réception de messages, la reconnaissance de formes de messages, la sélection de commandes , l'exécution de commandes et enfin la réactualisation de l'état de l'objet d'Antoid.

Réception de messages

Cette étape de réception de messages consiste pour un objet d'Antoid à retirer le message le plus ancien contenu dans sa boîte aux lettres et à le transmettre à l'étape

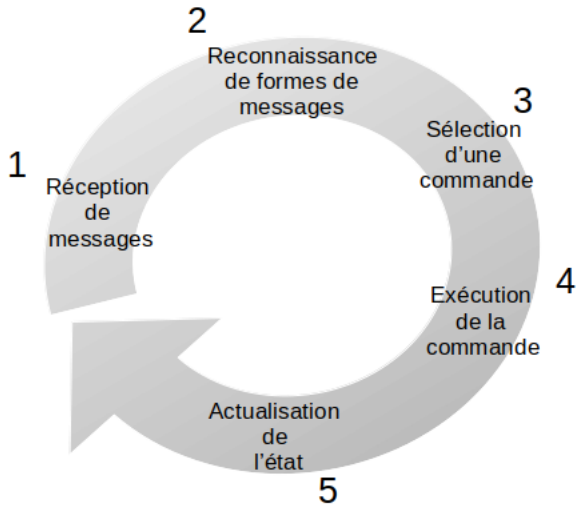

Figure 3.1 – Iteration Type

suivante.

Reconnaissance de formes de messages

La reconnaissance de formes de messages consiste à vérifier que le message transmis par l'étape précédente peut être traité lors de cette itération. La vérification s'effectue en comparant le message reçu aux formes de messages contenues dans une liste. Précisons que, pour cette étape, nous utilisons un moteur de reconnaissance de formes (pattern matching), en l'occurrence, celui d'Erlang.

Dans le cas où le message reçu correspond bel et bien à l'une des formes de messages présentes dans cette liste, celui-ci est transmis à l'étape suivante. Dans le cas contraire, le traitement du message est différé.

Ce mécanisme, consistant à différer le traitement des messages, permet aux objets d'Antoid de découpler l'ordre dans lequel ils reçoivent les messages de l'ordre dans lequel ils les traitent, tandis qu'il en va différemment dans le modèle objet par exemple où l'ordre de traitement des messages (i.e. appels de méthode) est le même que leur ordre de réception.

Il est à noter qu'en Erlang, lorsqu'un message est différé, celui-ci est placé dans une file d'attente jusqu'à ce qu'un message soit reconnu et transmis à l'étape de sélection de commandes. À ce moment là, tous les messages contenus dans la file d'attente sont replacés dans la boîte aux lettres dans leur ordre d'origine (devant les messages plus récents). Par exemple, si le premier message reçu lors d'une itération est différé, lorsqu'il est replacé dans la boîte aux lettres à l'itération suivante, il reprend la position de tête et sera donc de nouveau le premier message retiré.

Cette particularité permet aux objets d'Antoid de sélectionner les événements d'un même type dans leur ordre de réception. Ce mécanisme permet donc par exemple à une case de toujours traiter la demande de déplacement arrivée en premier. Précisons aussi qu'en Erlang, cette file d'attente est appelée une « Save Queue ».

Sélection de commandes

La sélection d'une commande consiste à faire correspondre le message transmis par l'étape précédente à une commande à lancer.

À cette fin, l'objet d'Antoid exploite une liste de couples, chaque couple associant un message à une commande à lancer en cas d'occurrence de ce message. Nous détaillons le fonctionnement de ces listes dans l'état de l'art à la section 1.2

Exécution de commandes

Dès que la commande a été sélectionnée, l'objet d'Antoid débute son exécution.

Précisons que ces commandes diffèrent selon qu'il s'agit d'un personnage ou d'une des cases composant le microcosme. En effet, pour un personnage, il s'agit toujours de ce que nous appelons une action de jeu, comme un déplacement ou la construction d'un mur, alors que pour une case, il s'agit soit de commandes de relais de messages, soit de commandes d'exécution des actions de jeu initiées par les personnages, sans oublier l'application des lois physiques et la diffusion des informations visuelles (pour plus de détails sur ces commandes, voir la section 3.3 consacrée aux spécifications des objets d'Antoid).

Il est à noter que l'exécution d'une commande peut inclure l'émission d'un ou plusieurs messages.

Réactualisation de l'état de l'objet d'Antoid

Cette étape permet à un objet d'Antoid de réactualiser son état de manière à moduler son action d'une itération sur l'autre.

Dans notre contexte, le terme d'action a deux acceptions : la première fait référence aux actions qu'effectue un objet d'Antoid au sein d'une chaîne d'événements, alors que la seconde fait référence à une chaîne d'événements dans son intégralité, ce qui correspond à ce que nous appelons une action de jeu comme par exemple le déplacement d'un personnage.

Cette modulation a pour but d'une part d'enchaîner les actions dans un ordre respectant la chaîne d'événements en cours (un objet d'Antoid ne peut enchaîner les actions dans n'importe quel ordre au risque de provoquer une incohérence) et d'autre part d'adapter le comportement de l'objet d'Antoid en fonction de la situation. Un personnage pourrait par exemple choisir de changer la direction de ses déplacements s'il rencontre un mur lui barrant la route.

Pour plus de précisions sur les différents éléments constituant l'état des objets d'Antoid, vous pouvez consulter la section 3.2.2. Le mécanisme de modulation est étudié dans la section 3.1.2.

3.2.2 État des objets d'Antoid

L'état d'un objet d'Antoid comprend les paramètres, qui font l'objet d'une réactualisation lors de la cinquième étape de l'itération type, mais également le message reçu durant l'itération en cours.

Tandis que la réactualisation de ces paramètres a pour seul rôle de moduler l'action de l'objet d'Antoid, les messages reçus ont également pour rôle de l'informer sur les variations de son environnement.

Techniquement, chaque itération est matérialisée par une fonction. Ainsi, quand une itération, avant de s'achever lance une nouvelle itération, c'est en fait une fonction qui appelle une nouvelle fonction tout en lui passant en argument les paramètres définis lors de l'étape de réactualisation de l'état de l'objet d'Antoid. (Nous voyons à la section 3.1.1 que cet enchaînement de fonctions est aussi ce qui permet aux objets d'Antoid de persister).

La réactualisation de l'état ne consiste pas seulement à réactualiser les valeurs, mais aussi à supprimer ou ajouter certains éléments d'information. C'est par exemple le cas lors de la création du microcosme. En effet, les cases utilisent un certain nombre de compteurs leur permettant de former un réseau maillé d'une taille particulière. Une fois le réseau formé, ces compteurs ne sont plus utiles, ils sont donc retirés de l'état des cases dans leur prochaine itération.

Cette restructuration est rendue possible via l'utilisation de listes comme structures de données pour contenir les différents paramètres (le nombre d'éléments qu'une liste peut contenir pouvant varier).

Différentes catégories de données

Avant de détailler les différentes catégories de données que l'on peut retrouver dans les paramètres des objets d'Antoid, nous voulons préciser que la structure des messages a été standardisé. Un message prend la forme d'un n-uplet. Ce n-uplet est divisé en deux parties, une partie commune et une partie spécifique à chaque catégorie de message. Notons aussi, que l'ordre dans lequel les éléments de chaque n-uplet sont agencés est significatif. Par exemple, l'identifiant de la catégorie du message est toujours situé en première position.

Le programme 3.1 présente un exemple de message correspondant à la demande de déplacement d'un personnage à sa case d'accueil.

programme 3.1 – Exemple de message utilisé dans Antoid V2

```
{Character_Move, CharacterId, Direction, MoveCounter, MoveTS}

% Character_Move : identifiant de la categorie "deplacements"
% CharacterId : identifiant du personnage faisant la demande
% Direction : direction du deplacement
% MoveCounter : identifiant du deplacement
% MoveTS : horodatage (timestamp) de l'emission du message
```

Les données correspondant aux paramètres des objets d'Antoid appartiennent à quatre grandes catégories :

1. les données utilisées pour la reconnaissance de formes de messages.

   Ces données sont utilisées lors de l'étape de reconnaissance de formes de messages. Elles correspondent à des expressions adoptant une forme analogue aux messages qu'elles servent à reconnaître, ce sont donc des n-uplets.

   Ces expressions sont intégrées dans des structures que nous appelons des boucles de réception de messages préprogrammées que nous décrivons plus loin dans cette section.

   Pour effectuer la comparaison entre les messages reçus et ces expressions, nous utilisons le moteur de reconnaissance de formes d'Erlang. Pour qu'un message soit reconnu, sa structure et celle de l'expression utilisée doivent respecter les conditions suivantes :

   (a) le nombre d'éléments doit être le même ;

   (b) l'ordre dans lequel les éléments sont placés doit être le même ;

(c) en cas de recherche de correspondance exacte, la valeur de chaque élément doit être la même.

Il est possible de rechercher une correspondance partielle. Cela se fait en laissant en blanc chaque élément dont la valeur ne doit pas être prédéfinie. Dans ce cas, l'élément correspondant dans le message peut prendre n'importe quelle valeur.

Il est à noter que, bien qu'Erlang soit muni d'un module permettant d'utiliser des expressions regulières (regular expression), celles-ci sont inutilisables dans le cadre de la reconnaissance de formes de messages.

2. Les données correspondant au couplage entre les messages et les commandes.

Ces données sont utilisées lors de l'étape de sélection de commandes. Elles correspondent à une liste de couples, chaque couple associant un message à une commande à lancer en cas d'occurrence de ce message.

Comme les expressions utilisées pour reconnaître les formes de messages, ces données sont intégrées aux boucles de réception de messages préprogrammées sur lesquelles nous revenons dans cette section.

3. Les données correspondant à des paramètres complémentaires.

Ces données peuvent prendre plusieurs formes, comme des compteurs ou des indicateurs. Elles contribuent à la modulation de l'exécution des commandes.

Par exemple lors de la création du microcosme un compteur est utilisé pour indiquer aux cases dans quel ordinateur de la grappe d'ordinateurs elle devra créer sa fille.

4. Les données correspondant à des événements passés conservés en mémoire.

Comme les messages (ou les expressions utilisées pour les filtrer), ces données prennent la forme de n-uplets. Elles sont actuellement utilisées pour conserver des informations visuelles comme par exemple la présence d'un personnage sur une case distante.

Boucles de réception de messages préprogrammées

Les objets d'Antoid exploitent deux catégories de données pour moduler l'enchaînement de leurs actions : les expressions servant à reconnaître les messages et les associations entre commandes et messages reconnus. C'est en variant les combinaisons de ces deux catégories de données qu'un objet d'Antoid module l'enchaînement de ses actions.

Techniquement, les trois premières étapes de l'itération type d'un objet d'Antoid sont exécutées par la boucle de réception de messages d'Erlang. Cette boucle de réception de messages (receive loop) est une structure de programmation spéciale dont on peut voir la syntaxe dans le programme 3.2.

À chacune de ses itérations, la boucle de réception de messages retire le premier message de la boîte aux lettres qu'elle compare alors aux différentes formes de messages contenues dans sa structure. Si aucune des formes de messages ne correspond, le traitement du message est différé et la boucle de réception de message commence une nouvelle itération. Dès qu'une correspondance est détectée, la commande associée à la forme de message reconnue est sélectionnée et exécutée. Une fois l'exécution de la commande terminée, le programme sort de la structure.

Précisons que si la boîte aux lettres est vide, la boucle de réception de messages se met alors en attente de nouveaux messages. Précisons aussi que bien que nous ne l'utilisons pas dans Antoid, il est possible de définir un temps d'attente maximal après lequel le programme sort de la structure.

programme 3.2 – Boucle de réception de message

```
receive % reception du message

  FormeDeMessage A -> % recherche de correspondance
    CommandeAssociee A ; % lancee si la forme A correspond

  FormeDeMessage B -> % recherche de correspondance
    CommandeAssociee B ; % lancee si la forme B correspond

    ...

end. % si aucune correspondance, retirer le prochain message
     % sinon sortir de la structure
```

La combinaison de blocs associant ce que nous avons appelé une « Forme De Message » et une « Commande Associée », intègre à la fois, notre liste de formes de messages ainsi que notre liste d'associations entre commandes et messages. À chacune de ces combinaisons de blocs correspond donc une modalité d'action d'un objet d'Antoid (un de ses cas d'utilisation). Techniquement, un objet d'Antoid module l'enchaînement de ses actions en modulant les combinaisons de blocs, les différentes combinaisons correspondant à ce que nous appelons les boucles de réception de messages préprogrammées.

Rappelons qu'une itération type est matérialisée par une fonction qui appelle une nouvelle fonction avant de s'achever et que pour réactualiser son état, un objet d'Antoid passe en argument de cette nouvelle fonction les paramètres qu'il souhaite conserver pour sa prochaine itération. Cependant, aujourd'hui, les boucles préprogrammées ne peuvent pas être passées en argument. En effet, elles sont directement intégrées dans le code de la fonction appelée. L'état réactualisé se retrouve donc en partie dans les arguments et en partie dans le code de la fonction appelée. Par conséquent, pour permettre aux objets d'Antoid de sélectionner une boucle préprogrammée, nous

avons finalement créé plusieurs fonctions, chaque fonction intégrant le code d'une des boucles préprogrammées. Ainsi, pour réactualiser son état, l'objet d'Antoid sélectionne d'abord la fonction intégrant la bonne boucle préprogrammée avant de lui passer en argument le reste de son état.

Pour illustrer ce concept, nous allons utiliser les cases dans le contexte de la diffusion d'informations visuelles. Notons cependant que ce que nous présentons ici n'est qu'un exemple très simplifié, pour plus de détails sur le concept d'information visuelle et leur diffusion dans le microcosme d'Antoid, vous pouvez consulter les spécifications d'Antoid à la section 3.3.1.

Lorsqu'une case reçoit un message de la part d'une de ses voisines incorporant une information visuelle (comme l'apparition d'un personnage sur une autre case), celle-ci agit différement en fonction de ce qui se trouve sur elle. Ainsi, si la case accueille un mur, elle met fin à la diffusion, car il est impossible de voir à travers un mur. Si la case accueille un personnage, celui-ci doit être informé, elle lui transmet donc cette information visuelle avant de continuer la diffusion vers d'autres cases. Enfin, si la case ne possède aucune voisine, elle continue simplement la diffusion vers d'autres cases. Techniquement, chacune de ces trois situations correspond à une boucle préprogrammée particulière (illustré dans le programme 3.3). À chaque itération, la case sélectionne la boucle correspondant à sa situation, ce qui correspond ici au type d'objet d'Antoid qu'elle accueille.

Il faut toutefois noter, que dans un langage fonctionnel, les appels répétés de fonctions différentes peuvent saturer la pile d'exécution ; chaque fonction appelée s'empilant sur la fonction appelante.

programme 3.3 – Les différentes boucles préprogrammées pour la diffusion d'information visuelles

```
boucle_preprogrammee_A() -> % la case accueille un mur

receive

  information_visuelle ->
    CommandeAssociee A ; % arret de la diffusion

end.

boucle_preprogrammee_B() -> % la case accueille un personnage

receive

  information_visuelle ->
    CommandeAssociee B ; % informe le personnage
                         % continue la diffusion

end.

boucle_preprogrammee_C() -> % la case n'accueille rien

receive

  information_visuelle ->
    CommandeAssociee C ; % continue la diffusion

end.
```

Cette problématique n'existe toutefois pas avec Erlang. En effet, grâce au mécanisme de récursion terminale [2] d'Erlang, si l'appel de la nouvelle fonction correspond à la dernière instruction de la fonction appelante, alors la fonction appelante sera retirée de la pile d'exécution avant que la fonction appelée n'y soit ajoutée. Ainsi, étant donné que la dernière chose que fait un objet d'Antoid correspond à appeler une nouvelle fonction, la pile d'exécution ne peut être saturée.

3.3 Spécifications des objets d'Antoid

Dans cette section, nous présentons les spécifications de nos objets d'Antoid que sont les cases, les personnages et les murs.

Avant de commencer, il nous faut préciser que notre objectif pour cette deuxième version d'Antoid n'a pas été de développer un jeu complet mais bien d'expérimenter des algorithmes distribués rarement appliqués au domaine du jeu vidéo massivement multijoueurs.

Dans sa version actuelle, Antoid met en scène des personnages non joueurs évoluant dans un monde virtuel que nous appelons microcosme. Ces personnages peuvent effectuer les actions de jeu suivantes : se déplacer, construire et détruire des murs ainsi que percevoir visuellement ce qui les entoure.

Notons, qu'outre les spécificités appartenant à chaque type d'objet d'Antoid, tous ont une perception partielle de leur environnement. En d'autres termes, aucun objet d'Antoid n'a une vision globale de ce qui se passe dans le jeu et ce à tout moment.

Ce concept de perception partielle est associé à celui de zone d'effet des événements. En effet, quand un événement se produit, il n'affecte qu'une certaine zone du microcosme et ne peut donc être perçu que par les objets d'Antoid situés dans cette zone d'effet (nous détaillons ce concept dans la sous-section 3.3.1).

2. Ce mécanisme porte toutefois mal son nom. En effet, il peut tout à fait s'agir de fonctions différentes et pas seulement une fonction unique s'appelant elle-même.

3.3.1 Spécifications des cases

Une étape impérative lors de la conception d'un monde virtuel est la définition des lois physiques auxquelles vont être assujettis tout ceux qui vont évoluer au sein de ce monde. Ces lois pouvant par exemple indiquer que les personnages ne peuvent ni passer ni voir à travers un mur.

L'entité responsable de faire respecter ces lois et donc par extension de faire en sorte que les interactions se produisant dans le monde virtuel restent cohérente est le moteur physique du jeu (physics engine).

Dans Antoid, ce moteur physique est distribué dans les cases composant l'environnement que nous appelons « microcosme ».

Dans cette section nous présentons comment nos choix conceptuels ont influencés la manière dont les cases gèrent les lois physique d'Antoid.

Avant de commencer, précisons qu'il ne faut pas confondre les moteurs physiques adaptés aux simulations scientifiques et ceux adaptés aux jeux vidéo. En effet, les premiers cherchent à produire des simulations aussi proche de la réalité que possible, ces moteurs sont donc très exigeant en terme de ressources. Les jeux vidéo quant à eux ont des contraintes strictes en terme de performances, ils doivent avant tout être jouables. C'est pourquoi leurs moteurs physiques ont plutôt recours à des approximations, comme c'est le cas pour Antoid.

Exécution des actions de jeu

Dans Antoid, tous les personnages et murs sont localisés sur les cases, chaque case pouvant accueillir un personnage ou un mur à la fois.

Techniquement, lorsqu'un personnage tente de réaliser une action de jeu (déplacement, construction ou destruction de murs), celui-ci transmet à la case qui l'accueille un message lui indiquant ce qu'il souhaite faire. Par exemple, pour entreprendre un déplacement, le personnage va indiquer dans son message qu'il souhaite se déplacer

dans une direction donnée. Une fois ce message transmis, le travail de la case revient dans un premier temps à déterminer quels seront les effets de cette action de jeu en fonction de la situation et des lois physiques du jeu en vigueur. Dans un second temps, la case exécute ces effets.

Pour que le déplacement d'un personnage puisse se réaliser, sa route ne doit pas contenir d'obstacle (un personnage ou un mur). Ainsi, dans ce cas de figure, la case va vérifier qu'aucun obstacle n'entrave la route du personnage et en fonction des informations obtenues, effectuer le déplacement ou non.

Il faut toutefois noter que certaines actions de jeu ne peuvent pas être prises en charge par une seule case. En effet, les cases n'ont pas accès aux informations détenues par les autres cases et ne peuvent pas non plus agir sur l'état de ces autres cases. Par conséquent, dans certains cas de figure, plusieurs cases vont devoir collaborer pour déterminer quels effets va avoir une action de jeu mais aussi pour exécuter cette action de jeu.

Le déplacement d'un personnage est un de ces exemple. En effet, cette action de jeu consiste pour le personnage à passer d'une case à une autre. Or la case accueillant le personnage ne sait pas si la case visée par le personnage est disponible pour un déplacement. Par conséquent, les deux cases vont devoir échanger des informations pour s'assurer que le déplacement est possible et si cela est le cas, se coordonner pour que le personnage quitte sa case d'accueil afin d'aboutir sur sa case d'arrivée.

Il faut aussi noter qu'en pratique, pour l'essentiel, les personnages lancent des actions de jeu et modifient leur état. Ainsi, en définitive, outre la fonction de moteur physique, les cases forment aussi l'environnement d'exécution (runtime) d'Antoid.

Ce rôle explique le nom de « microcosme » que nous avons donné au monde permanent d'Antoid. En effet, c'est une référence à S. Papert qui, dans son ouvrage « Mindstorms » (Jaillissement de l'esprit) [33], définit un concept analogue dont le rôle est de permettre de se familiariser avec des lois inaccessibles autrement que dans un monde virtuel.

Zone d'effet des événements

Certains événements ont pour particularité d'affecter tous les objets d'Antoid se trouvant dans une zone plus ou moins grande du microcosme que nous appelons sa « zone d'effet ». Aujourd'hui, cette zone d'effet prend toujours la forme d'un cercle avec à son centre la case ayant accueilli l'événement (le diamètre du cercle pouvant varier d'un type d'événement à l'autre).

Un exemple d'événement est celui de l'apparition d'un personnage sur une case. Dans ce cas de figure, le personnage devient visible pour tous les autres personnages suffisamment proches (c'est-à-dire ceux situés dans la zone d'effet de l'événement).

Il faut toutefois noter que cette zone d'effet peut parfois être réduite par certains éléments du jeu. Parmi ceux-ci, nous retrouvons la loi de notre moteur physique indiquant que les personnages sont incapables de voir à travers un mur. Ainsi, si un mur s'interpose entre la case où le personnage est apparu et une autre case, cette autre case ne fera pas partie de la zone d'effet de l'événement.

Nous illustrons ce phénomène dans la figure 3.2 où un événement visible se produit au centre du microcosme. Le personnage A (en bas) est hors de la zone d'effet, il ne voit donc pas l'événement. Le personnage B (à gauche) se trouve derrière un mur qui bloque son champ de vision, il ne voit donc pas l'événement non plus. Le personnage C (à droite) en revanche est situé dans la zone d'effet et rien ne bloque son champ de vision, il voit donc l'événement.

Dans Antoid, tous les messages, quel que soit l'émetteur et le récepteur, transitent obligatoirement par les cases. Le mode de cet acheminement est étroitement lié à l'architecture du microcosme, qui adopte la topologie d'un réseau maillé de cases. Une case peut donc communiquer directement soit avec le personnage ou le mur qu'elle accueille soit avec les cases avec lesquelles elle est connectée que nous appelons ses voisines.

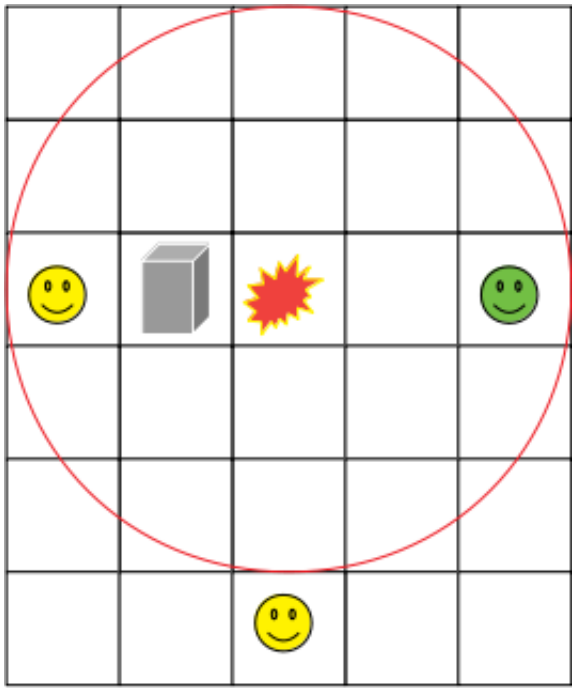

Figure 3.2 – Zone d'effet d'un événement visible

Ainsi, pour qu'un personnage puisse percevoir un événement se produisant au loin, le message incorporant cette information événementielle devra voyager de case en case jusqu'à ce que la case qui accueille le personnage le reçoive et le lui transmette.

Cette manière d'acheminer les messages est à la base de nos algorithmes diffusant les effets d'un événement dans sa zone d'effet. En effet, lorsqu'une case est informée d'un événement (en recevant un message) ayant une zone d'effet s'étendant sur plusieurs cases, celle-ci va transmettre aux cases voisines impactées (toujours en fonction de la situation et des lois physiques en vigueur) un message incorporant les informations relatives à cet événement. Ces cases voisines vont à leur tour informer leurs propres voisines et ce jusqu'à ce que la diffusion ait couverte la zone d'effet de l'événement.

Précisons qu'une case n'informe ses voisines qu'après avoir appliqué les effets de cet événement sur elle-même, ce qui, dans le cas d'un événement visible, correspond (entre autres) à informer le personnage qu'elle accueille (si elle accueille bel et bien

un personnage) de ce qu'il voit en lui transmettant un message.

Il est à noter que bien que ce rôle des cases puisse faire penser au concept de routage, il n'en est rien. En effet, contrairement aux routeurs, les cases traitent systématiquement les événements dont elles sont informées.

Outre l'application des effets d'un événement sur elle-même, chaque case, dans le cadre de la diffusion des informations événementielles à ses voisines altère le message qu'elle a reçu de telle manière qu'il indique la distance restante à parcourir dans le microcosme avant que les limites de la zone d'effet de l'événement ne soient atteintes.

Prémisses d'une gestion de l'information visuelle

Dans un jeu vidéo, il est impératif que les personnages puissent percevoir ce qui se trouve dans leur environnement, que ce soit d'autres personnages ou des obstacles comme des murs. Le moyen de perception le plus communément utilisé dans les jeux est celui de la vision. C'est pourquoi, pour cette deuxième version d'Antoid, nous avons fait le choix de travailler à l'intégration du concept d'information visuelle. Notons toutefois que ces algorithmes de gestion de l'information visuelle sont aisément transposables en algorithmes de gestion de l'information sonore ou de toute autre catégorie d'information liée à la perception des personnages.

En toute logique, puisque ce sont les cases qui sont en charge de l'application des effets d'un événement sur elles-mêmes et de la diffusion des informations événementielles, ce sont elles qui sont en charge de l'apparition, de la disparition et de la diffusion des informations visuelles. Nous insistons cependant sur le fait que notre objectif n'est pas de rechercher un maximum de réalisme dans l'imitation d'une information visuelle mais simplement d'informer les personnages de l'occurrence d'événements dans leur champ de vision.

Dans notre version actuelle d'Antoid, nous avons travaillé sur les mécanismes de diffusion et de mémorisation des informations visuelles mais nous réservons à une version ultérieure l'utilisation de cette information visuelle par les personnages pour

moduler l'enchaînement de leurs actions (en modifiant par exemple leur trajectoire avant de rencontrer un obstacle).

Nous avons exploité la capacité des cases à communiquer de voisine à voisine, dans l'optique de diffuser l'information visuelle depuis la case où s'est produit un événement visible vers ses cases voisines afin de couvrir la zone d'effet de cet événement. Actuellement, nous traitons les informations visuelles liées au déplacement des personnages ainsi que celles liées à la construction et la destruction de murs.

Les lois physiques qui s'appliquent sont les suivantes : un personnage ne peut voir passé une certaine distance et il est incapable de voir à travers un mur.

Un personnage peut donc voir apparaître ou disparaître personnages et murs d'une case à condition que cette case ne soit pas trop éloignée de celle qui l'accueille, qu'aucun mur ne s'interpose et enfin que sa case d'accueil et la case où l'événement visible se produit soient alignées l'une par rapport à l'autre. Notons que cette première version de l'algorithme ne prend pas les diagonales en compte ; un personnage n'est donc pas capable de percevoir tout ce qui se trouve autour de lui. La figure 3.3 illustre le champ de vision d'un personnage (en vert) lorsque qu'il peut voir jusqu'à deux cases de distance.

Avant de poursuivre, nous voulons montrer en quoi notre concept de traitement local des événements peut influencer la manière dont nous pensons nos algorithmes.

Avant toute chose, il faut préciser que la case accueillant quelque chose de visible (e.g. un personnage) ne diffuse pas continuellement les informations visuelles concernant l'objet d'Antoid qu'elle accueille. En effet, les informations visuelles sont uniquement diffusées lors de l'occurrence d'un événement d'apparition ou de disparition. Or de cette manière, tous les personnages qui rentrent dans la zone d'effet de l'événement après la diffusion sont incapables de voir ce qui se trouve sur la case, ce qui est indésirable.

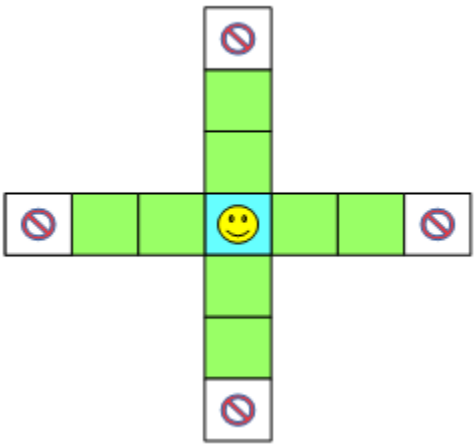

Figure 3.3 – Champs de vision d'un personnage

Pour remédier à ce problème sans nécessiter de diffuser continuellement cette information (ce qui serait très exigeant en terme de ressources), nous avons fait en sorte que chaque case située dans la zone d'effet de ces événements conserve en mémoire les informations visuelles concernant les objet d'Antoid se trouvant sur les autres cases. Ainsi, lorsqu'un personnage arrive sur une case, celle-ci lui transmet les informations visuelles qu'elle conserve en mémoire.

Précisons que lorsqu'un événement de disparition est reçu par une case, cela a pour effet de lui faire effacer l'information visuelle concernant ce qui se trouvait à cet endroit (on ne veut effectivement pas qu'un personnage puisse voir quelque chose qui n'existe plus).

La gestion des informations visuelles concernant les murs est légèrement plus complexe. En effet, d'après nos lois physiques, lorsqu'un mur apparaît, toutes les informations visuelles concernant ce qui se trouve derrière ce mur ne sont plus visibles. C'est pourquoi lorsqu'une case reçoit une information visuelle concernant l'apparition d'un mur, celle-ci va effacer de sa mémoire toutes les informations visuelles en rapport avec ce qui se trouve derrière le nouveau mur.

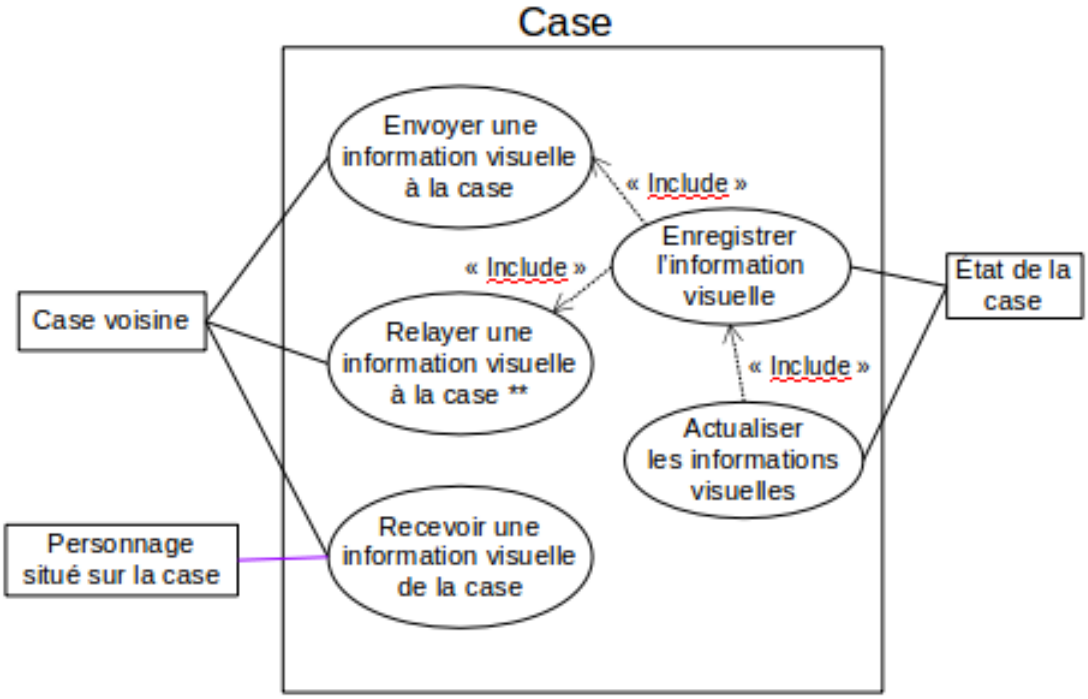

Figure 3.4 – Spécification des cases liées à la gestion de l'information visuelle

De plus, tant qu'un mur est situé sur une case, celle-ci bloque la diffusion des informations visuelles concernant ce qui se trouve derrière le mur à ses voisines. Lorsque celui-ci est détruit, tout ce qui se trouve derrière le mur redevient visible. Dans ce cas de figure, la case d'où le mur a été détruit va diffuser toutes les informations visuelles dont elle bloquait la diffusion jusqu'alors.

Les figures 3.4 et 3.5 présentent les diagrammes des cas d'utilisation des cases et des personnages en lien avec la gestion de l'information visuelle. Notons aussi que nous détaillons les algorithmes de diffusion de l'information visuelle dans l'annexe C.

Création d'un microcosme

Il existe deux phases qui gouvernent la création d'un microcosme : une première phase correspondant à son initialisation et une deuxième, correspondant à son extension (qui peut se dérouler en plusieurs étapes).

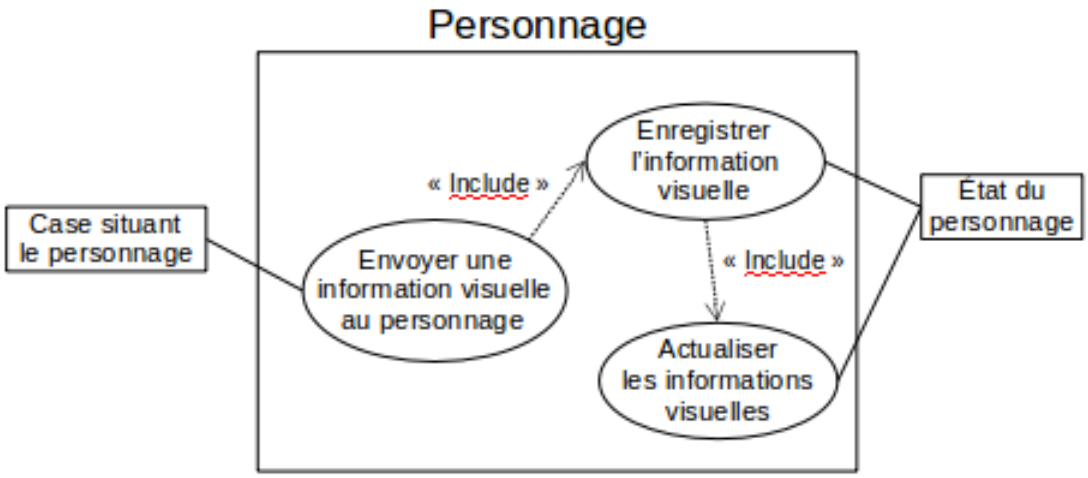

Figure 3.5 – Spécification des personnages liées à la gestion de l'information visuelle

Dans la phase d'initialisation, tout commence par la création d'une case appelée « case initiale ». Cette case, qui est le premier objet d'Antoid à voir le jour au lancement du jeu, est créée via une commande lancée depuis la console Erlang. Cette case, intègre le programme et les données qui vont définir la future forme du microcosme, comme sa taille par exemple.

Chaque case ne peut créer qu'une case et une seule durant son existence que nous appelons sa fille. C'est l'enchaînement de création de cases en partant de la case initiale qui va de fil en aiguille créer un microcosme complet, chaque case transmettant à sa fille son programme et les données nécessaires à la poursuite de la création du microcosme.

Il est à noter que dans le cadre de cette deuxième version d'Antoid, nous avons fait le choix de donner une forme de rectangle à notre microcosme, chaque case ayant donc de deux à quatre voisines. Précisons aussi qu'il est possible de définir la largeur et la longueur (en nombre de cases) à l'initialisation du microcosme en paramétrant la case initiale. Un exemple de microcosme d'une largeur de 2 cases et d'une longueur de 3 cases est illustré dans la figure 3.6.

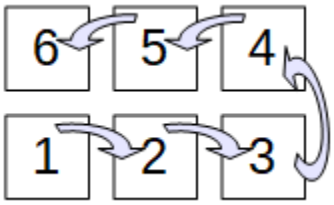

Figure 3.6 – Creation d'un microcosme

La création d'un microcosme ne comprend pas seulement la création de nouvelles cases, elle comprend aussi la formation du réseau maillé via la recherche de voisines.

Comme nous l'avons indiqué précédemment, une case ne peut interagir directement qu'avec ses voisines. Or à sa création, une case ne connaît que la case qui l'a créé (que nous appelons sa mère). Par la suite, une fois qu'elle aura créé elle-même une case, elle la connaîtra de facto comme sa fille. Toutefois, pour découvrir ses autres voisines (si elles existent), la case a recours à un algorithme de recherche de voisines.

Cet algorithme consiste pour la case mère à rechercher l'identifiant de la case située en dessous de sa fille. Pour ce faire, la mère interroge sa propre voisine située en dessous d'elle qui connaît la case recherchée. Une fois l'information reçue, la case mère peut alors transmettre l'information à sa fille qui à son tours transmettra son identifiant à la case située en dessous d'elle. Nous illustrons cette situation dans la figure 3.7.

À titre d'exemple, dans le cadre de notre microcosme illustré sur la figure 3.6, l'enchaînement effectué pour raccorder la case 6 et la case 1 est le suivant :

1. la mère de la case 6 (la case 5) demande à sa voisine située en dessous d'elle (la case 2) l'adresse de la case 1 ;
2. la case 2 connaissant l'adresse de la case 1, elle transmet cette information à la case 5 ;
3. une fois l'information reçue, la case 5 peut à son tour transmettre l'information à sa fille (la case 6) ;

4. la case 6 reçoit l'information et informe la case 1 qu'elle est sa voisine située au dessus d'elle ;
5. la case 1 reçoit l'information raccordant ainsi le réseau maillé.

Précisons que dans la version actuelle d'Antoid, les cases situées sur les diagonales ne sont pas voisines, la case 5 et la case 1 ne sont donc pas voisines. La case 5 n'ayant pas besoin de connaître l'adresse de la case 1 elle va donc « oublier » cette information dès qu'elle l'aura transmise à sa fille.

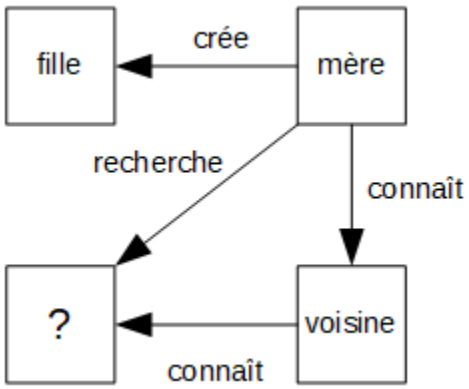

Figure 3.7 – Recherche de voisine

Si le nombre de personnages et de murs augmente radicalement, l'espace dans lequel ils évoluent doit s'étendre. Dans Antoid, cela se traduit simplement par l'augmentation du nombre de case et cela correspond à la phase d'extension du microcosme.

Pour étendre le microcosme, nous ajoutons de nouvelles machines à notre grappe d'ordinateurs. Toutefois, nous ne voulons pas que cela demande d'arrêter le jeu et donc casse le principe de monde permanent.

Pour ce faire nous avons doté notre microcosme de certaines caractéristiques :

1. il doit, bien sûr, être capable de s'étendre ;
2. il ne doit pas être nécessaire de modifier le code pour augmenter sa taille ;
3. il ne doit pas être nécessaire d'arrêter le jeu.

Pour démarrer l'extension du microcosme, nous envoyons (actuellement via la console Erlang) un message à la dernière case créée (celle qui n'a pas encore de fille). Cette case crée donc sa case fille en la dotant des caractéristiques d'une case de départ pour cette extension. Cette extension étant créée via le même algorithme utilisé pour initialiser le microcosme.

Actuellement, le microcosme s'entend toujours dans la même direction ce qui est illustré par la figure 3.8, où la dernière case du microcosme de la figure 3.6 (la case 6) initie la création de 3 cases supplémentaires.

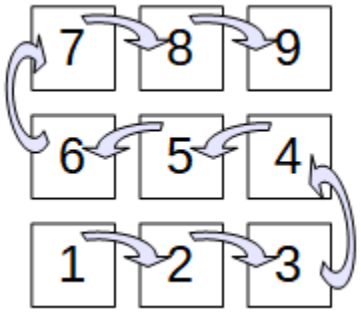

Figure 3.8 – Extension d'un microcosme

La figure 3.9 présente un diagramme de cas d'utilisation des cases en lien avec la création de microcosme.

3.3.2 Spécification des personnages

Dans Antoid, le rôle essentiel des personnages est de lancer des actions de jeu. Aujourd'hui, il existe trois sortes d'actions de jeu que les personnages peuvent entreprendre : se déplacer et construire ou détruire des murs. Précisons cependant, que la construction et la destruction sont réservées à des personnages spéciaux que nous appelons des Trolls.

Bien que ce soit les personnages qui initient les actions de jeu, c'est finalement l'environnement via les cases, qui prend en charge leur exécution, afin d'y appliquer

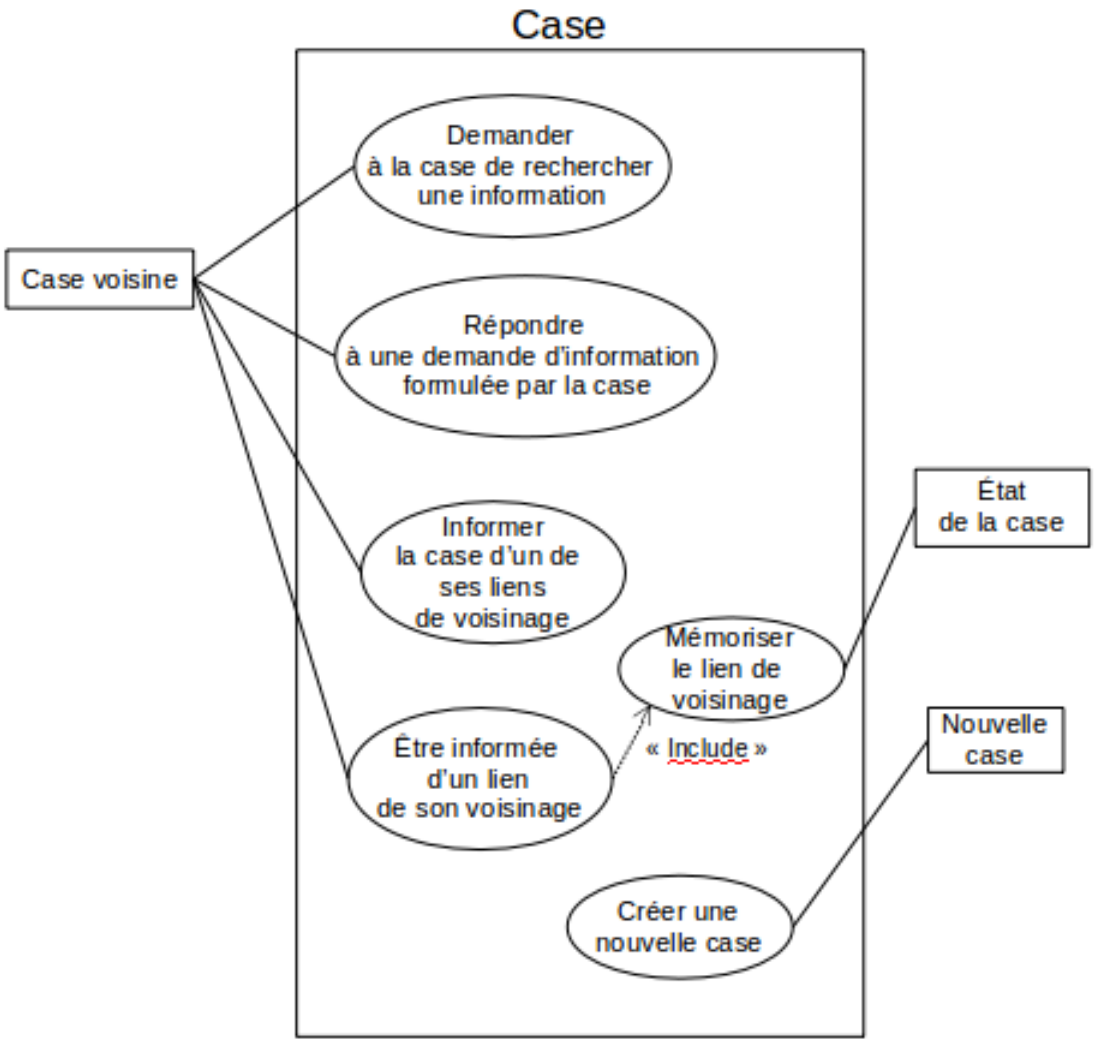

Figure 3.9 – Spécification des cases liées à la construction du microcosme

les lois physiques qui y sont associées et pour diffuser l'information visuelle relative à leurs effets.

Pour un personnage, lancer une action de jeu correspond à envoyer un message à la case sur laquelle il se trouve ; ces messages, contenant toutes les informations nécessaires à son exécution par les cases.

Notons que certaines informations sont présentes dans tous les types de messages comme le type d'action de jeu ou l'identifiant de l'émetteur alors que d'autres ne sont nécessaires que pour des messages spécifiques comme la direction dans laquelle un personnage souhaite se déplacer par exemple.

Déplacement des personnages

L'action de jeu liée au déplacement des personnages, consiste à leur permettre de se mouvoir de case en case dans le microcosme.

Il est à noter, que le microcosme pouvant potentiellement être distribué sur un nombre quelconque d'ordinateurs (de nœuds), les personnages peuvent migrer d'ordinateur en ordinateur de façon transparente, puisque ce sont les cases qui exécutent les actions de jeu, déplacement compris. Il est également à noter que ce sont les cases qui, une fois le microcosme initial formé et avant de lancer l'instance d'Antoid, créent les personnages (non joueurs), conformément aux conditions initiales qui leurs sont transmises. Il est par exemple possible de paramétrer la case initiale de manière à ce qu'un personnage soit créé toutes les 3 cases comme illustré à la figure 3.10.

Dans cette version d'Antoid, une case ne peut accueillir qu'un personnage ou un mur à la fois. Le déplacement d'un personnage est donc possible, sauf si la case de destination accueille déjà un personnage ou un mur (un personnage ne peut pas non plus sortir des limites du microcosme).

Aujourd'hui, les personnages peuvent se déplacer dans quatre directions différentes : le nord, le sud, l'est et l'ouest. Le personnage change la direction vers laquelle il essaie

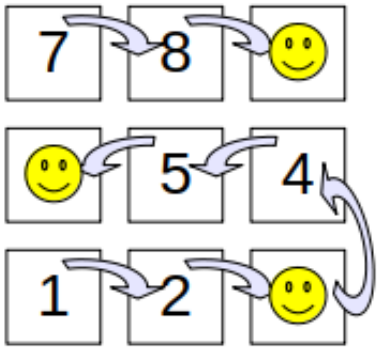

Figure 3.10 – Création des personnages non joueurs dans le microcosme

de se déplacer quand il est bloqué ; dans cette version d'Antoid, les personnages répètent inlassablement une même séquence de changement de direction.

Les figures 3.11 et 3.12 présentent des diagrammes des cas d'utilisation des cases et des personnages en lien avec le déplacement.

3.3.3 Spécification des murs

Les murs sont le seul exemplaire d'artefact disponible dans cette version d'Antoid. Comme les personnages, ils évoluent sur les cases formant le microcosme. Ils ont pour fonction de bloquer les déplacements des personnages et les informations visuelles.

Notons que bien que les murs ne possèdent pas aujourd'hui d'action de jeu à lancer, ils ne sont pas pour autant des objets d'Antoid passifs. En effet, au même titre que les personnages, ils sont capable de communiquer avec les cases qui les accueillent, de traiter les événements qu'ils reçoivent et de modifier leur propre état. Il serait donc conceptuellement possible dans une version ultérieure d'Antoid de doter les murs de la capacité de se déplacer par exemple.

Construction et destruction de murs

Cette action de jeu, a pour particularité de ne pouvoir être entreprise que par un type de personnage particulier que nous appelons Troll. Comme tous les personnages,

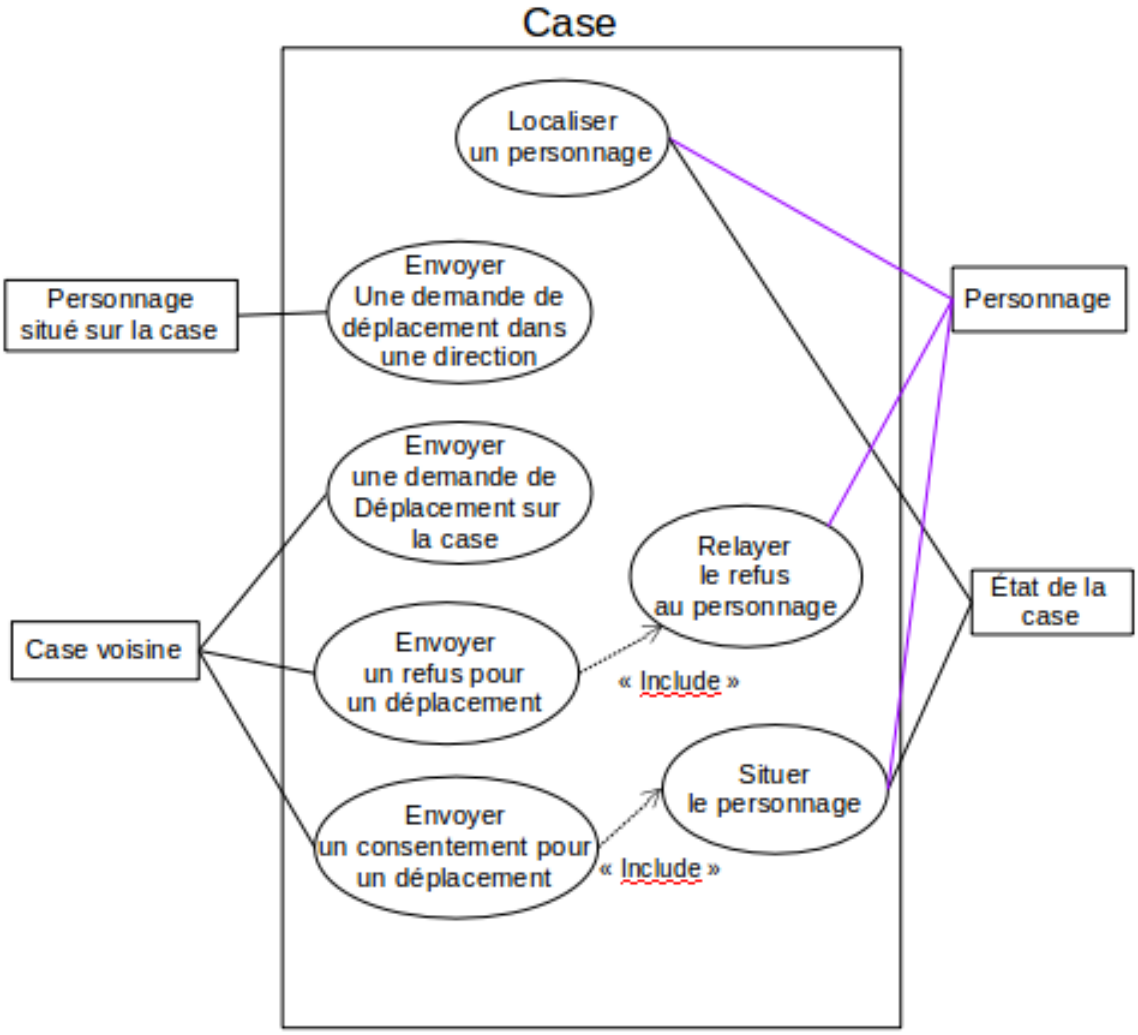

Figure 3.11 – Spécification des cases liées au déplacement des personnages

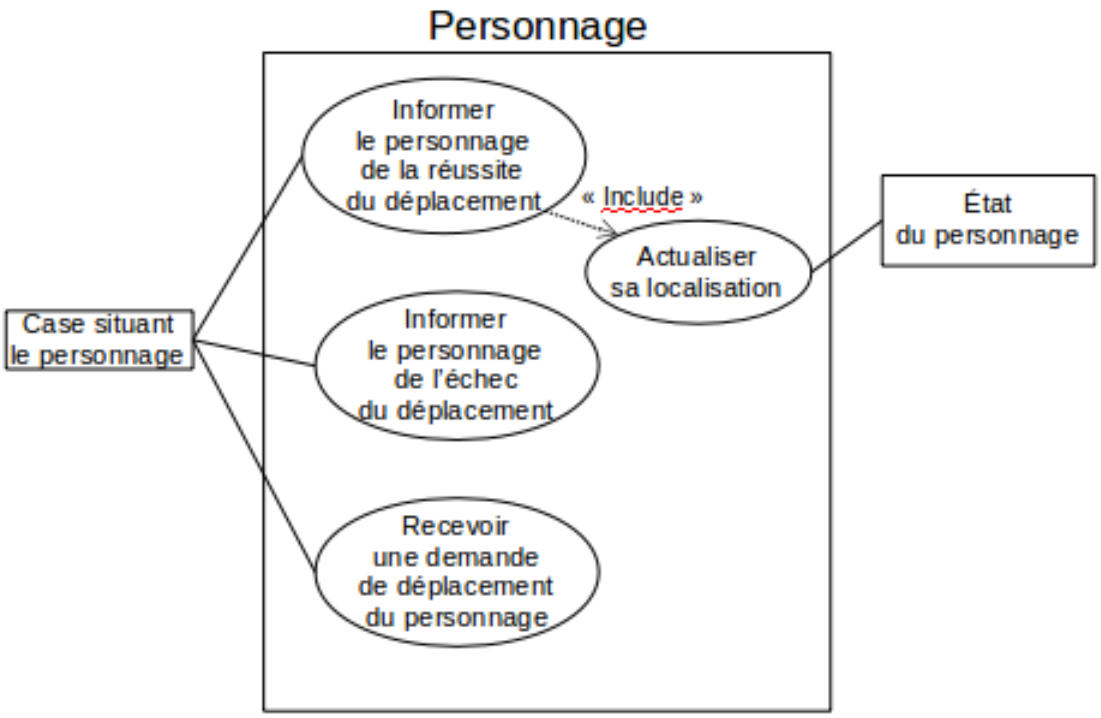

Figure 3.12 – Spécification des personnages liées au déplacement des personnages

ils peuvent se déplacer de case en case, mais ils ont en plus la capacité de construire et de détruire des murs.

L'action de jeu liée à la construction ou à la destruction de murs, consiste à construire un mur sur une case ou à détruire un mur présent sur une case. Les cases créent, suppriment et localisent les murs en fonction des tentatives des Trolls dont elles sont informées. Un Troll peut lancer la construction ou la destruction d'un mur sur une des cases voisines à celle qui l'accueille.

Son comportement est le suivant : il tentera toujours de construire un mur après avoir parcouru un certain nombre de cases et il tentera toujours de détruire un mur bloquant son passage après avoir dû contourner des murs un certain nombre de fois. Par exemple, il serait possible de paramétrer un Troll de telle manière qu'il construise un mur tous les six déplacements et détruise le prochain mur sur sa route après avoir été bloqué trois fois.

Les figures 3.13 et 3.14 présentent des diagrammes de cas d'utilisation des cases et des Trolls en liens avec la construction et la destruction de murs.

Notons que pour plus de détails au sujet des actions de jeu disponibles dans Antoid, il est possible de consulter l'annexe C.

3.4 Conclusion du chapitre

Dans ce chapitre, nous avons présenté le modèle que nous avons conçu et implanté en vu de verifier notre hypothèse qui postule que grâce à cette nouvelle version de notre moteur, il serait possible d'accroître massivement le nombre d'entités en interaction et donc d'événements, simplement en ajoutant de nouveaux nœuds à une grappe d'ordinateur. Cela sans interrompre le jeu et sans faire s'effondrer les performances.

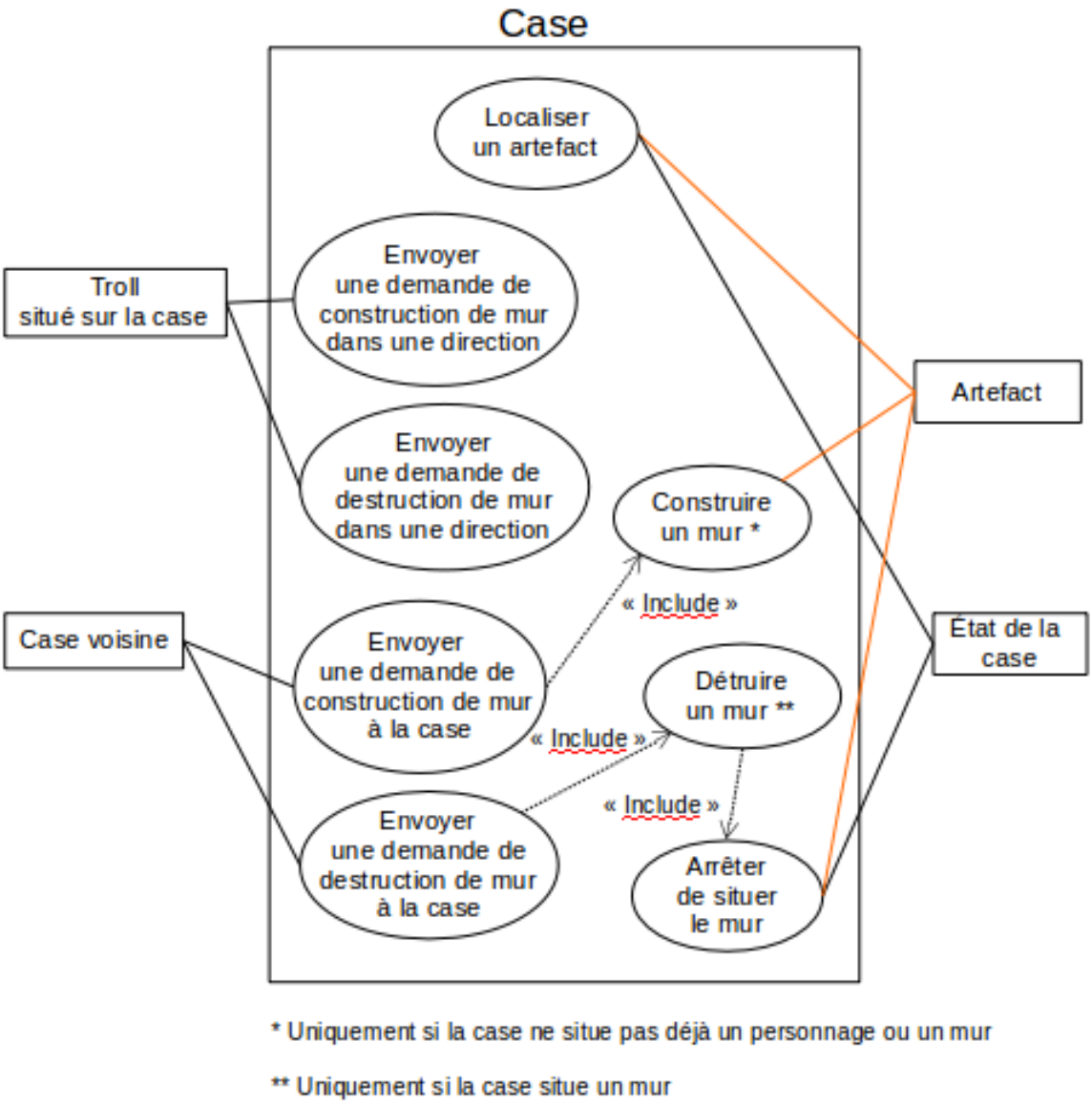

Figure 3.13 – Spécification des cases liées à la construction et la destruction de murs

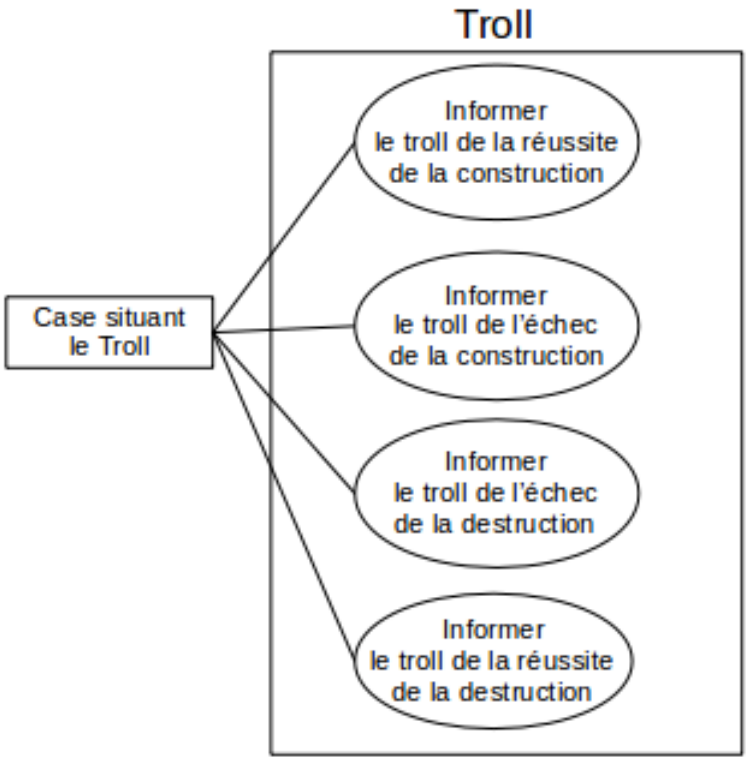

Figure 3.14 – Spécification des trolls liées à la construction et la destruction de murs

Chapitre 4

Expérience

En introduction de ce mémoire, nous avons énoncé notre objectif d'augmenter la capacité d'Antoid à supporter de plus en plus d'objets d'Antoid en interaction (les personnages, leurs artefacts et bien sûr les cases) et donc d'augmenter sa capacité à traiter de plus en plus d'événements. Cela, tout en conservant impérativement un système cohérent, résilient (par système résilient, nous entendons que celui-ci doit rester opérationnel même s'il subit une baisse radicale dans ses performances) et, autant que possible, performant. Pour réaliser cet objectif, nous avons cherché à rendre Antoid apte à une mise à l'échelle.

Le concept de mise à l'échelle, possède deux acceptions différentes :

1. une mise à l'échelle verticale consistant à ajouter de nouvelles ressources à un nœud, via par exemple l'ajout de mémoire vive ;
2. une mise à l'échelle horizontale permettant d'améliorer les capacités d'une architecture en ajoutant de nouveaux nœuds comme dans une grappe d'ordinateurs.

Dans Antoid, la mise à l'échelle est principalement horizontale. C'est en effet via l'ajout de nouveaux ordinateurs (des Raspberry PI) dans notre grappe d'ordinateurs que nous comptons augmenter la capacité de notre architecture à traiter de plus en plus d'événements. Nous pouvons toutefois noter, que nous avons eu recours à la mise

à l'échelle verticale lorsque nous avons remplacé le système d'exploitation « Raspbian » par « Raspbian Lite » (plus léger), ceci afin d'augmenter la quantité de mémoire vive disponible.

Avant d'entrer dans le vif du sujet, il est à noter que nous avons réalisé une validation des fonctionnalités d'Antoid, afin de vérifier que notre moteur soit bel et bien capable d'effectuer notre expérience. Vous pouvez retrouver celle-ci dans l'annexe A

4.1 La grappe d'ordinateurs : Olifan

Pour effectuer nos expériences, nous avons monté une grappe d'ordinateurs personnelle composée de Raspberry PI que nous appelons Olifan, en hommage au superordinateur de l'Université de Sherbrooke : le Mammouth.

Le fait d'utiliser notre propre plateforme physique pour nos expériences a été très avantageux, car elle nous a permis d'obtenir rapidement des informations sur l'exécution d'Antoid dans un environnement distribué. Notons aussi que pour le reste de cette section, nous utilisons l'abréviation « RPI » pour désigner un Raspberry PI.

4.1.1 Environnement matériel

Nos expériences ont été réalisées sur des « Raspberry PI modèle 3 B » [36], équipés de processeurs quatre cœurs ARM Cortex-A53 cadencés à 1,2GHz. Ils sont munis de 1GB de mémoire vive cadencé à 900MHz. Ils sont aussi munis d'un port Ethernet 10/100 MB/s que nous utilisons pour interconnecter les différents RPI via un commutateur réseau (switch).

Nous avons aussi ajouté à chaque RPI une carte sd de classe 10 U1 capable d'écrire à une vitesse de 10MB/s.

Le choix d'utiliser des RPI pour réaliser ces expériences, est en grande partie lié au fait qu'ils sont peu coûteux, ce qui permet d'en acquérir un certain nombre et donc d'expérimenter la distribution d'Antoid sur une grappe de calcul personnelle. Les RPI

jouissent aussi d'une grande communauté ce qui est un avantage en cas de problèmes techniques.

4.1.2 Environnement logiciel

Chaque RPI utilise le système d'exploitation Raspbian Lite [37] qui est une version allégée de Raspbian. Raspbian Lite, que nous utilisons dans sa version 4.14 est un système d'exploitation libre basé sur Débian GNU/Linux qui est adapté pour fonctionner sur un RPI.

En ce qui concerne Erlang [19], nous utilisons sa version 21.0, tandis que le stockage et l'analyse des données post-expérience est fait via le logiciel de base de données PostgreSQL [35] dans sa version 10.5.

4.2 Le protocole expérimental

Dans le cadre de nos expériences, nous avons cherché à mesurer les performances de notre implémentation d'Antoid sur notre grappe d'ordinateurs Olifan qui contient aujourd'hui quatre RPI.

Les mesures que nous avons effectuées au cours de ces expériences peuvent être divisées en deux catégories :

1. La première catégorie, concerne les capacités de notre plateforme. Ces mesures ont pour objectif de déterminer jusqu'à quel point Antoid et Olifan sont capables de supporter une mise à l'échelle du nombre d'objets d'Antoid et par extension d'événements.

   Ayant quatre RPI à notre disposition, nous avons pu effectuer quatre séries d'expériences. La première série consiste à produire une suite d'expériences sur un RPI, où chaque nouvelle expérience produit plus d'événements que la précédente. Cela jusqu'à ce qu'Antoid atteigne ses limites et cesse de fonctionner

(i.e. ne réponde plus). La seconde série s'effectue sur deux RPI, la troisième sur trois et la dernière sur quatre RPI.

Ces mesures vont notamment nous permettre de déterminer les capacités de mise à l'échelle horizontale d'Antoid en observant l'évolution de la limite du nombre d'événements que peut traiter Antoid lorsque de nouvelles ressources sont ajoutées à Olifan.

2. La seconde catégorie de mesures porte sur les performances de notre plateforme en terme de temps de traitement. En effet, rappelons qu'Antoid est un jeu vidéo voué à accueillir des joueurs ; il est donc nécessaire que les temps de traitements soient stables et suffisamment courts pour ne pas altérer l'expérience de jeu. Ces mesures, consistent à étudier la moyenne du temps de traitement des événements dans Antoid et plus particulièrement, l'évolution de cette moyenne en fonction du nombre d'événements traités et de la quantité de machines disponibles dans la grappe d'ordinateurs.

   Comme pour la première catégorie de mesures, nous avons procédé à quatre séries d'expériences. Chaque série consistant à reprendre toutes les expériences déjà produites pour la catégorie précédente. Ainsi, la première série d'expérience consiste à reprendre toutes les expériences réalisées avec un RPI, la deuxième celles sur deux RPI et ainsi de suite jusqu'à quatre RPI.

   Ces mesures vont nous permettre d'une part d'observer jusqu'à quel point les performances restent stables lorsque le nombre d'événements augmente et d'autre part, si les performances d'Antoid sont améliorées par l'ajout de nouvelles ressources dans Olifan.

4.2.1 Enregistrement des informations nécessaires aux mesures

Dans le cadre de nos expériences, toutes les informations nécessaires à nos mesures sont sauvegardées par des processus Erlang permanents que nous appelons des

« processus d'écriture ». Chaque processus d'écriture est associé à une case du microcosme, chaque case ayant créé son propre processus d'écriture pendant la formation du microcosme (il existe donc autant de processus d'écriture qu'il existe de cases).

L'enregistrement des informations se déroule de la manière suivante :

Lorsqu'une case termine le traitement d'un événement, elle envoie les informations liées à cet événement à son processus d'écriture via un message. Le processus d'écriture conserve ces messages dans sa boîte aux lettres et ce jusqu'à ce que l'expérience soit achevée. Une fois l'expérience terminée, tous les processus d'écriture de toutes les cases écrivent ces informations dans des fichiers. Les informations contenues dans ces fichiers sont alors intégrées dans notre base de données pour analyse.

Dans une version antérieure d'Antoid, la phase d'écriture dans les fichiers se déroulait pendant l'expérience et non après. Or, comme l'écriture dans des fichiers ne fait pas partie de ce que nous appelons des actions de jeu (comme un déplacement), l'expérience s'en retrouvait biaisée. Dans notre version actuelle d'Antoid, ce biais a été réduit et se limite désormais à la transmission des messages et l'espace qu'ils occupent dans les boîtes aux lettres des processus d'écriture. En outre, afin de réduire un peu plus ce biais, nous nous sommes limités aux informations liées aux déplacements des personnages dans le microcosme. Précisons toutefois que même si les informations liées aux événements de construction et de destruction de murs ou les informations visuelles ne sont pas transmises en vue d'être écrites, ces événements ont bel et bien lieu durant nos expériences et sont traités par nos objets d'Antoid. Notons par ailleurs que l'évolution des temps de traitement pour ces événements est équivalente à celle des temps de traitement pour les déplacements des personnages.

Il est à noter que le fait pour la case de transmettre des informations liées aux événements qu'elle traite pourrait ne pas être considéré comme un biais. En effet, Une fonctionnalité que nous pouvons retrouver dans un certain nombre de jeux massivement multijoueurs, consiste à communiquer aux joueurs qui le souhaitent, des informations complémentaires sur ce qui se passe dans leur environnement qui sont éventuelle-

ment consultables à des fins statistiques. Cette fonctionnalité, n'existe toutefois pas aujourd'hui dans Antoid et reste donc en définitive un biais pour nos expériences.

Notons que nous détaillons dans l'annexe D la manière dont les données produites par notre expérience sont intégrées à notre base de données et comment ces données sont utilisées une fois enregistrées.

4.2.2 Réalisation des expériences

Outre les cases, les personnages, les artefacts ou les processus d'écriture, nous utilisons, dans le cadre de nos expériences, un dernier type de processus que nous appelons « l'expérimentateur ». Cet expérimentateur, contrairement à tous nos autres processus Erlang, n'est pas hébergé par Olifan, mais par une machine externe reliée à Olifan par réseau local. Son rôle consiste notamment à créer la case initiale du microcosme et à lancer la phase d'écriture une fois l'expérience achevée.

Déroulement d'une expérience

Toutes nos expériences suivent le même schéma. Celui-ci peut être divisé en trois grandes phases :

1. La première phase, consiste à mettre en place l'environnement dans laquelle l'expérience va se produire. Cette phase comprend elle-même plusieurs étapes :
   (a) les RPI nécessaires à l'expérience, ainsi que la machine qui hébergera le processus Erlang expérimentateur sont alimentés et branchés au réseau local. Notons qu'aujourd'hui, Antoid impose aux adresses IP des différents RPI de se suivre. Par exemple, si le premier RPI possède l'adresse «192.168.0.1» le deuxième RPI devra avoir l'adresse «192.168.0.2». Précisons toutefois, que la machine hébergeant l'expérimentateur peut prendre n'importe quelle adresse ;
   (b) un nœud Erlang est démarré sur chaque RPI ainsi que sur la machine qui hébergera l'expérimentateur. Chaque nœud est identifié par l'adresse IP de l'ordinateur sur lequel il est créé ;

(c) le processus Erlang expérimentateur est créé sur la machine externe ;

(d) la case initiale encapsulant les conditions initiales de l'expérience est créée via l'expérimentateur sur le nœud Erlang situé sur le premier RPI. Cela a pour effet de lancer la création du microcosme et des personnages prévus sur le nombre de RPI requis ;

(e) une fois que le microcosme et les personnages sont en place, l'expérience est automatiquement lancée.

2. La deuxième phase consiste à produire l'expérience. Dans cette étape, les personnages sont « lâchés » dans le microcosme où ils effectuent les différentes actions de jeu à leur disposition. L'expérience se termine lorsque chaque personnage a effectué un certain nombre de déplacements (nombre qui aura été fixé dans les conditions initiales). Précisons que les personnages ayant terminé de se déplacer en premier ne sont pas supprimés ou retirés du microcosme, ils restent simplement immobiles à leur dernière position.
3. La troisième et dernière phase est la phase d'écriture, elle consiste à intégrer les différentes informations dans notre base de données afin de procéder à des analyses.

Configuration des expériences

Dans Antoid, configurer une expérience revient à paramétrer la case initiale du microcosme. Ces paramètres contiennent des informations telles que le nombre de cases et de personnages (dont des Trolls) qui devront être créés, le nombre de nœuds disponibles pour la distribution du microcosme, le nombre de déplacements que chaque personnage devra effectuer avant de s'arrêter, la distance maximale de propagation des informations visuelles et toutes les autres informations nécessaires au bon déroulement de l'expérience.

Pour les mesures que nous souhaitons effectuer lors de ces expériences, il a été question de faire varier deux paramètres :

1. le nombre de RPI sur lesquels doit se produire l'expérience ;
2. le nombre d'événements se produisant durant l'expérience.

Le nombre d'événements qui se produira pendant l'expérience n'est toutefois pas un paramètre à proprement parler. En effet, dans cette version d'Antoid, le nombre d'événements est fonction de deux choses : du nombre de personnages actifs dans le microcosme et du nombre de déplacements que les personnages vont effectuer, les déplacements étant les actions de jeu qui vont déclencher tous les autres événements. En effet, c'est après s'être déplacé un certain nombre de fois qu'un Troll tente de construire un mur et c'est après avoir plusieurs fois échoué à se déplacer à cause de murs qu'un Troll tentera de détruire le prochain sur sa route. En outre, tous ces événements provoquent bien entendu la diffusion d'informations visuelles (pour plus de précision vous pouvez consulter les specifications d'Antoid dans le chapitre 3 de ce mémoire : 3.3). Ainsi, pour faire varier le nombre d'événements, nous avons choisi de fixer pour chaque personnage, un nombre maximal de déplacements et d'augmenter progressivement le nombre de personnages d'une expérience à l'autre.

Notons toutefois, qu'augmenter le nombre de personnages ne suffit pas. Car, si le nombre de personnages augmente le nombre de cases pour les accueillir doit lui aussi augmenter.

Trois règles ont donc été mises en place pour toutes nos expériences afin de maintenir une certaine linéarité :

1. le nombre de 324 personnages a été déterminé pour l'expérience initiale ;
2. à chaque nouvelle expérience ce nombre initial de personnages doit être ajouté au nombre de personnages présents dans l'expérience précédente. Par exemple, la deuxième expérience contiendra 648 personnages ;
3. le microcosme devra contenir 36 cases par personnages. Ainsi, lors de la première expérience nous avons 11 664 cases.

Tous les autres paramètres sont fixes et n'ont pas été modifié d'une expérience sur l'autre. Ils sont consultables sur le tableau suivant : 4.1.

Paramètres fixes	*Valeur*
Nombre de personnage initial (et ajouté à chaque expérience)	324 personnages
Nombre de cases par personnage	36 cases par personnage
Nombre de Trolls par personnage	1 Troll pour 8 personnages
Nombre de déplacements (personnages et Trolls)	324 déplacements
Nombre de déplacement avant une construction (Trolls)	6 déplacements
Nombre de déplacement bloqués par un mur avant une destruction (Trolls)	3 blocages
Taille d'une frontière (en nombre de cases)	108 cases
Distance maximale de diffusion des informations visuelles (en nombre de cases)	5 cases

Tableau 4.1 – Paramètres fixes

Justification de ces paramètres

1. 324 personnages initiaux : L'intérêt du nombre 324 pour les personnages est double. En effet, il est divisible par 2, 3 et 4 et permet donc de répartir les personnages équitablement entre les machines quelque soit le nombre de Raspberry PI dans la grappe tout en étant assez petit pour permettre de réaliser un bon nombre d'itérations pour chaque série ;
2. 36 cases par personnages : permet d'espacer les personnages de manière à ce qu'ils ne se bloquent pas les uns les autres sans arrêt ;
3. 1 Troll pour 8 personnages : seuls les trolls peuvent construire des murs. En limitant le nombre de trolls, on limite le nombre de murs dans le microcosme évitant ainsi que les personnages se retrouvent bloqués.

Avant de passer à la section suivante, nous souhaitons apporter des précisions au sujet du concept de frontière entre RPI. Lorsque le microcosme est distribué entre plusieurs RPI, plusieurs cases hébergées sur un RPI se retrouvent avec une case voisine hébergée sur un autre RPI. Les cases dans ce cas de figure forment ce que nous appelons une « frontière ».

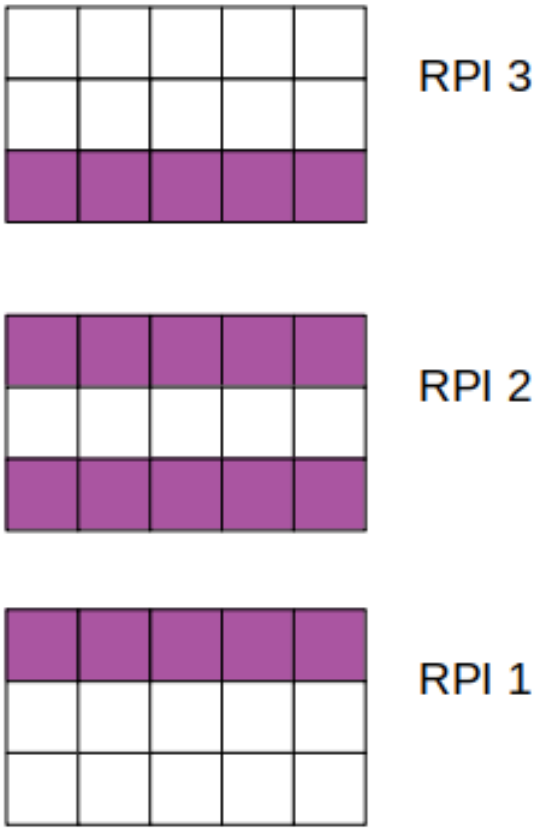

Figure 4.1 – Frontières entre RPI

Dans le cadre de nos expériences, nous avons établi qu'une frontière est toujours composées de 108 cases. Cependant, le nombre de frontières peut varier en fonction du nombre de RPI présent dans notre grappe d'ordinateur. En effet, si le microcosme est distribué sur deux RPI, il n'existe que deux frontières, une dans chaque RPI. Maintenant, si le nombre de RPI passe à trois, nous nous retrouvons avec quatre frontière, un RPI faisant la jonction entre les deux autres RPI (nous illustrons ce concept sur la figure 4.1, où les cases appartenant à une frontière sont en surbrillance).

Il est à noter que dans le cadre de nos expériences, augmenter la taille des frontières pourrait influencer le nombre de migration entre les RPI (c'est-à-dire des déplacements commençant sur un RPI et se terminant sur un autre).

La non-reproductibilité des expériences

Une expérience est dite reproductible si il est possible pour les mêmes conditions initiale d'aboutir au même déroulement et aux mêmes résultats. Pour s'assurer de posséder cette propriété, certains systèmes comme le jeu de la vie [21] par exemple imposent un fonctionnement au tour par tour, s'émancipant de cette manière des aléas tels que les variations dans la planification des tâches par le système d'exploitation, de la fréquence d'horloge du microprocesseur, de la vitesse du réseau ou même d'un papillon s'introduisant dans la machine.

Dans le cadre d'Antoid, nous avons fait le choix de laisser ces aléas influencer le déroulement de nos expériences. Ainsi, même en conservant des conditions initiales identiques pour plusieurs expériences, le déroulement et les résultats de celles-ci n'ont presque aucune chance d'être identiques. En effet, il serait parfaitement possible qu'un déplacement de personnage traité légèrement plus rapidement dans une expérience engendre un effet papillon sur les actions de tous les autres personnages. Il convient toutefois de noter que bien que nos expériences ne soient pas reproductibles, les résultats demeurent suffisamment proches d'une expérience sur l'autre et restent donc valides pour une analyse.

Notons par ailleurs que nous ne considérons pas cette non reproductibilité des expériences comme un désavantage. En effet, rappelons qu'Antoid est un jeu vidéo dans lequel il est prévu d'accueillir des joueurs. Ces joueurs apporteront leur lot d'imprédictibilité qu'Antoid devra être en capacité de gérer. Il est donc tout à fait pertinent pour ces expériences desquelles les joueurs sont absents d'être sujettes à de l'imprédictibilité.

4.3 Résultats de l'expérience

Dans cette section, nous faisons l'analyse des résultats de notre expérience qui compte 60 itérations comme nous pouvons le constater dans le tableau 4.2. Nous ne présentons pas toutefois la totalité de ceux-ci, pour la totalité de nos résultats, vous pouvez consulter l'annexe B où ils sont exposés.

1 RPI	6 expériences	1944 personnages	629 856 déplacements	69 984 cases
2 RPI	12 expériences	3888 personnages	1 259 712 déplacements	139 968 cases
3 RPI	18 expériences	5832 personnages	1 889 568 déplacements	209 952 cases
4 RPI	24 expériences	7776 personnages	2 519 424 déplacements	279 936 cases

Tableau 4.2 – Résumé des expériences

4.3.1 L'évolution de la limite d'événements traitables

Dans un premier temps, nous avons cherché à savoir jusqu'à quel point Antoid et Olifan sont capables de supporter une mise à l'échelle du nombre d'objets d'Antoid (et d'événements). Pour répondre à cette question, nous pouvons consulter le tableau 4.2 qui résume pour chaque combinaison de RPI le nombre de personnages maximal qu'il a été possible d'héberger dans Olifan. Nous pouvons relever que l'évolution de la limite du nombre de personnages est linéaire. En effet, la capacité maximale de Olifan pour Antoid observée sur un RPI a doublé sur deux RPI, triplé sur trois et quadruplé sur quatre. En revanche, nous devons noter que lors de ces expériences, les déplacements des personnages étaient en très grande majorité des déplacements se déroulant dans un même RPI. En effet, même dans les expériences sur quatre RPI, le nombre de déplacements où un personnage migre d'un RPI à un autre ne dépasse jamais les 1 % quelque soit le nombre de personnages. Cela est dû au faible nombre de cases faisant la jonction entre les différents RPI. Rappelons d'ailleurs que dans un jeu vidéo massivement multijoueurs on veut que le nombre de migration soit le plus bas possible, le temps de communication entre deux processus situés sur deux machines étant plus lent que sur une même machine.

4.3.2 L'évolution des performances

Dans un second temps, nous avons évalué l'évolution des performances d'Antoid sur Olifan. Nous pouvons voir sur la figure 4.2 (graphique) montrant l'évolution de la moyenne du temps de traitement des déplacements, que le temps moyen de traitement d'un déplacement reste relativement stable d'une expérience à l'autre. Effectivement, l'ordre de grandeur reste de l'ordre du centième de seconde.

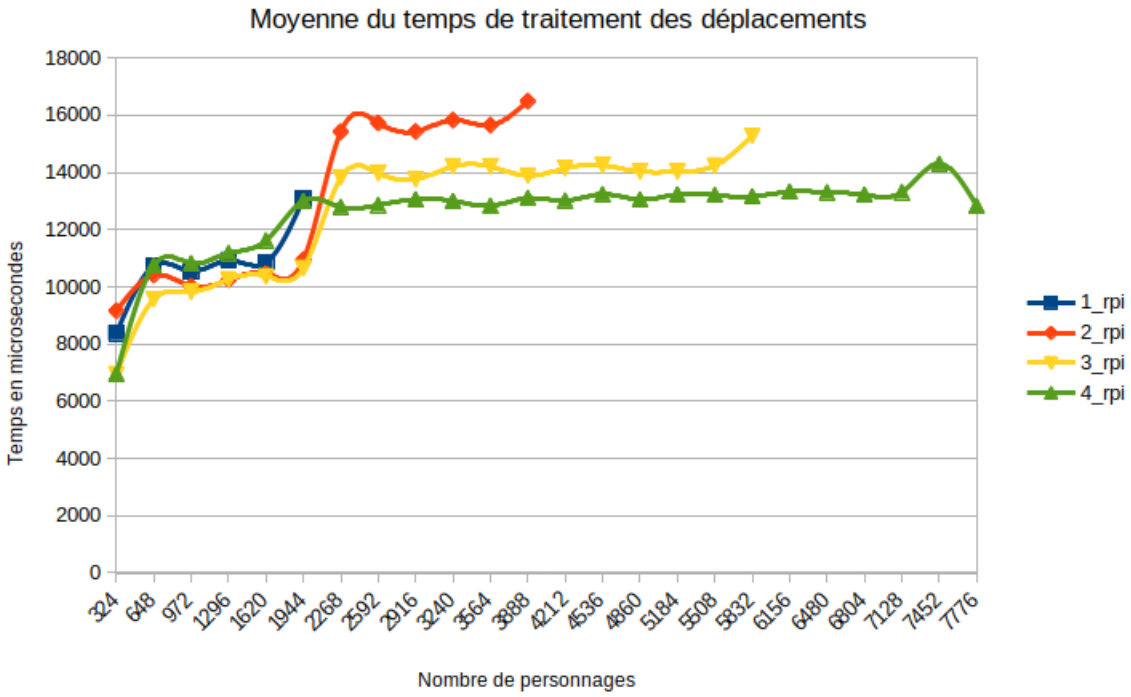

Figure 4.2 – Moyenne du temps de traitement des déplacements

Il est aussi à noter que même lorsque Antoid et Olifan hébergent le nombre maximal de personnages pour chaque combinaison de RPI, la moyenne conserve sa stabilité. Comme on peut le voir sur le tabeau 4.3, l'augmentation maximale peut être observée entre les deux dernières itérations produites sur un RPI, où la moyenne augmente de 10 853 micro-secondes à 13 077 micro-seconde. Nous pouvons même observer une évolution inverse sur quatre RPI, où l'itération contenant le nombre maximal de personnages affiche un temps de traitement moyen de 12 447 micro-secondes qui est inférieur à l'itération précédente qui affiche un temps moyen de 13 942 micro-secondes.

Il faut cependant noter que lorsque l'on étudie l'écart-type sur la figure 4.3 (graphique), nous observons une différence beaucoup plus marquée entre les deux dernières itérations de chaque combinaison de RPI. Nous n'observons en revanche de forte augmentation qu'entre les deux dernières itérations pour chaque combinaison de RPI. Toutes les autres itérations démontrent que l'écart-type reste lui aussi relativement

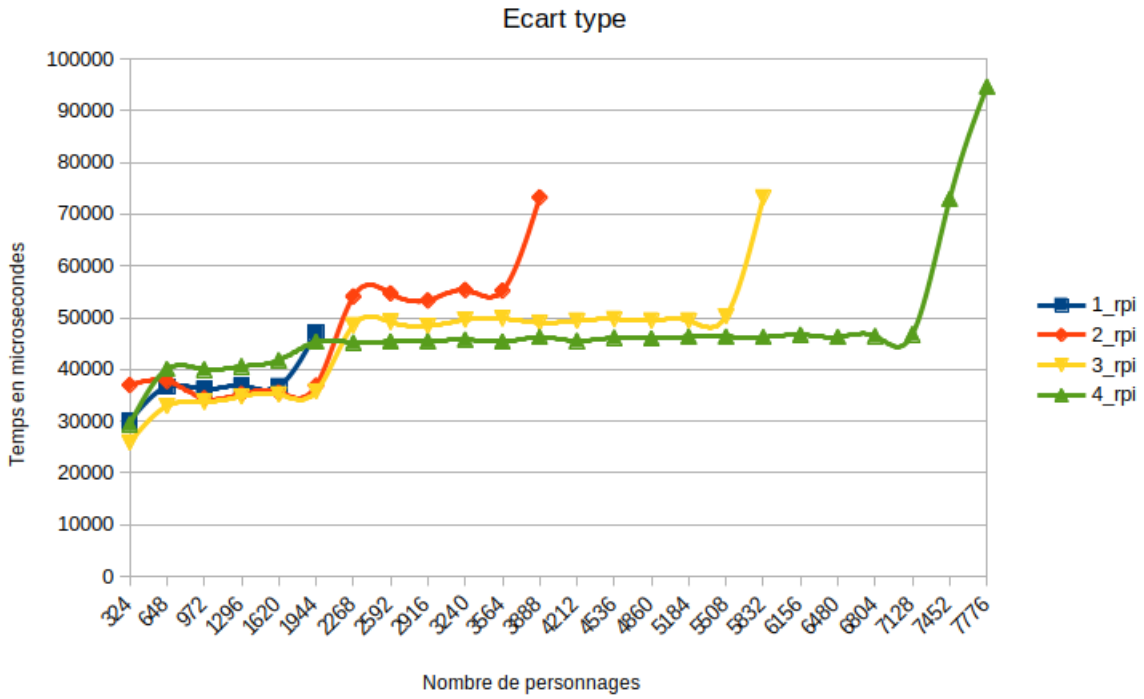

Figure 4.3 – Écart-type du temps de traitement des déplacements

stable d'une expérience à l'autre.

En poursuivant notre étude, nous avons finalement observé que cette différence qui apparaît lors des dernières itérations de chaque combinaison de RPI était due aux migrations (les déplacements s'effectuant d'un RPI à un autre). En effet, comme nous pouvons le constater sur le tableau 4.3, la moyenne et l'écart-type pour les déplacements internes à un noeud restent stables, alors que la moyenne et l'écart-type des migrations augmentent très fortement lorsque la limite d'une combinaison de RPI (en nombre de personnages) est atteinte, comme on peut le voir sur le tableau 4.4.

À titre d'exemple, le temps maximal qu'a pris le traitement d'un déplacement est de presque 11 secondes, lors de l'itération accueillant 7776 personnages sur 4 RPI. Alors que le temps maximal pour l'expérience n'accueillant que 6804 personnages ne dépassait pas les 0,7 secondes. Nous ne connaissons pas encore les raisons de ce

Moyenne					Écart-type				
Nb PNJ	1 RPI	2 RPI	3 RPI	4 RPI	Nb PNJ	1 RPI	2 RPI	3 RPI	4 RPI
1296	10931	10232	10247	11163	1296	36786	35226	34735	40586
1620	10853	10461	10373	11590	1620	36739	35500	35180	41694
1944	13077	10948	10656	12997	1944	47114	36832	35729	45398
3240		15830	14211	12985	3240		55352	49592	45719
3564		15657	14204	12839	3564		55215	49839	45397
3888		16031	13885	13106	3888		57417	49021	46218
5184			14020	13229	5184			49409	46310
5508			14222	13204	5508			50161	46333
5832			15051	13155	5832			68612	46221
7128				13306	7128				46747
7452				13942	7452				59806
7776				12447	7776				80950

Tableau 4.3 – Moyenne et écart-type pour les déplacements internes à un noeud

Moyenne				Écart-type			
Nb PNJ	2 RPI	3 RPI	4 RPI	Nb PNJ	2 RPI	3 RPI	4 RPI
1296	14841	15018	14707	1296	43222	39652	40102
1620	14219	9661	20075	1620	37217	27133	55291
1944	10967	10474	17176	1944	31858	29627	43740
3240	12958	16224	22082	3240	30043	48923	63438
3564	12093	15753	12704	3564	35355	45024	37987
3888	972177	16304	12468	3888	1833533	46670	35648
5184		18709	18411	5184		64906	61102
5508		32001	17281	5508		108237	52848
5832		328968	17737	5832		893023	48886
7128			21100	7128			79407
7452			442127	7452			1367657
7776			523789	7776			1719164

Tableau 4.4 – Moyenne et écart-type pour les migrations

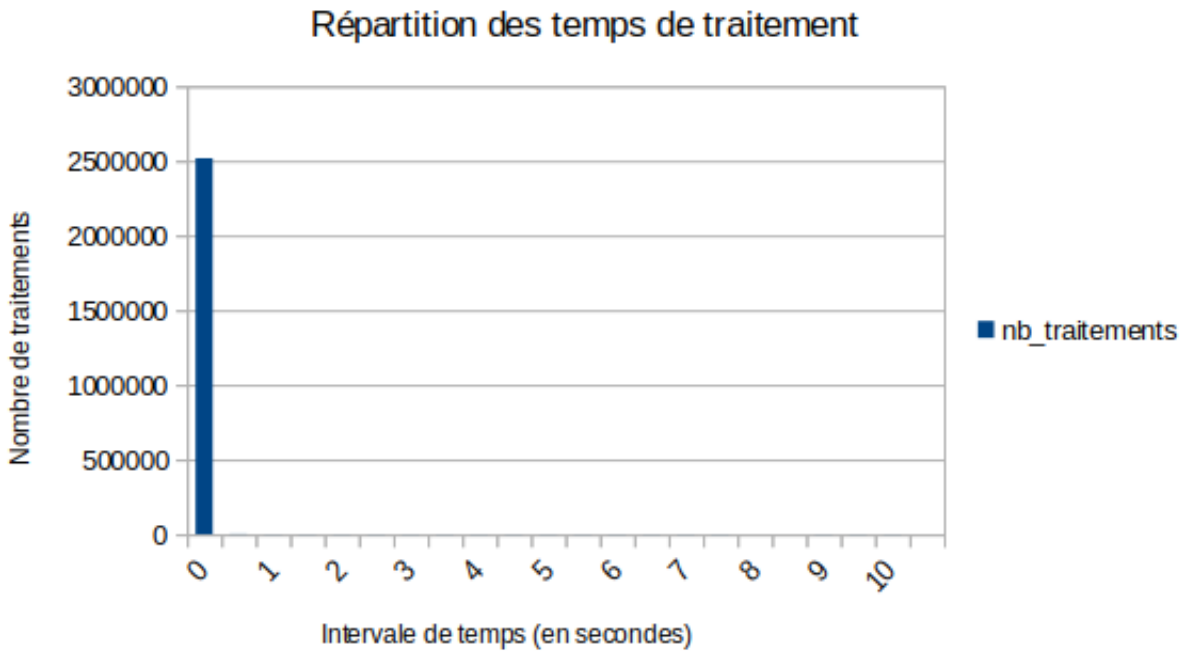

Figure 4.4 – Répartition des temps de traitements Expérience 60 (4 RPI, 7776 personnages)

phénomène. Il est toutefois possible que cela soit dû au fait que le temps de traitement nécessaire par le processus responsable du déplacement dépasse le temps maximal que peut lui accorder la machine virtuelle d'Erlang et qu'il ait donc besoin de reprendre le contrôle plus d'une fois pour terminer son traitement, mais cela reste une hypothèse à vérifier.

Il faut cependant noter que ce problème ne devrait pas se poser tant qu'un nouveau RPI est ajouté à la grappe d'ordinateurs Olifan lorsqu'une limite est atteinte. Ajoutons par ailleurs, que bien que certains traitements dépassent largement les temps « acceptables » pour les joueurs, ceux-ci sont particulièrement rares et ne représentent qu'un très faible pourcentage. En effet, comme nous pouvons le voir dans les figures 4.4 et 4.5 (graphiques), même dans l'expérience où le nombre de traitement dépassant les 0,5 secondes est le plus élevé, celui-ci reste négligeable comparé au nombre total de traitements (0.08%).

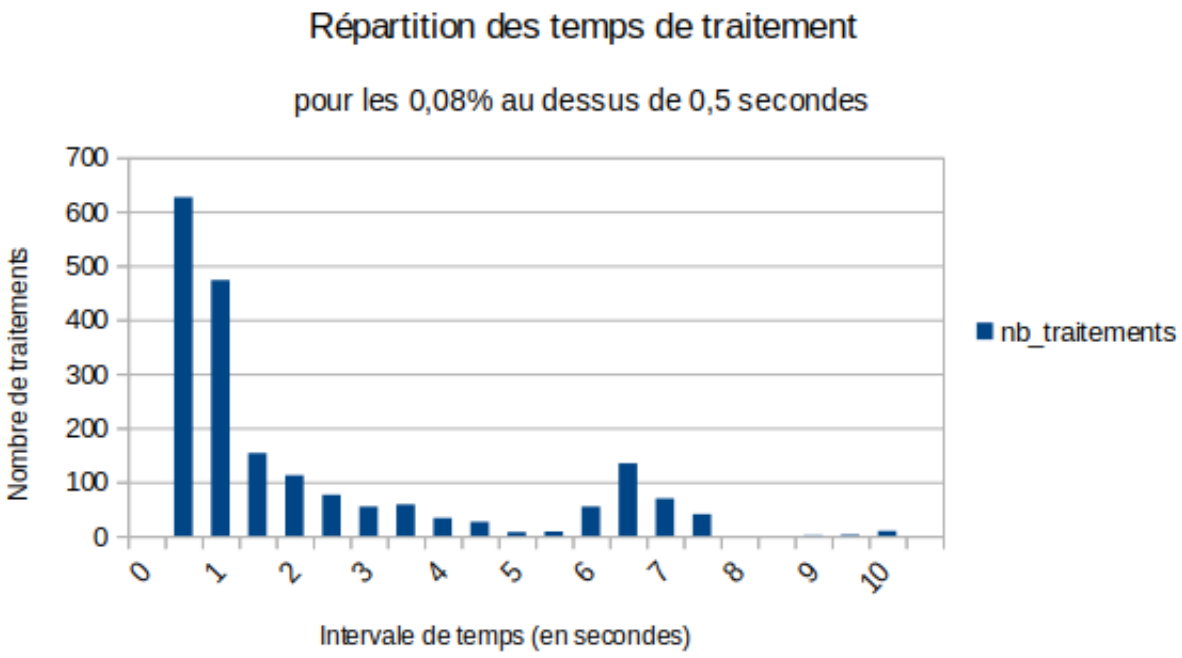

Figure 4.5 – Répartition des temps de traitements Expérience 60 (4 RPI, 7776 personnages) pour les traitements dépassant les 0.5 secondes (0.08% du total)

4.3.3 Post-analyse

À la lumière de ces résultats, il nous semble pertinent de dire que notre objectif visant à permettre l'augmentation du nombre de personnages, d'artefacts, et bien sûr de cases via l'ajout de nouveaux ordinateurs est rempli. De plus, comme nous avons pu le voir à travers ces résultats, Antoid permet une augmentation linéaire du nombre d'événements traités. Nous devons toutefois noter, que ces expériences ont été effectuées sur un nombre limité de nœuds. Il serait donc tout à fait pertinent pour des expériences futures d'augmenter la taille de notre grappe d'ordinateurs Olifan.

Ajoutons que, compte tenu des performances limitées qu'offrent des Raspberry PI et du fait que le code implémentant notre modèle n'ait pas été développé avec l'idée d'optimiser les temps de traitement ni leur stabilité, les performances qui ont été démontrées par Antoid nous semblent particulièrement prometteuses dans le cadre d'un jeu vidéo massivement multijoueurs.

Conclusion

Notre démarche au cours de ce projet de maîtrise a été d'expérimenter des algorithmes distribués rarement utilisés dans le domaine du jeu vidéo massivement multijoueurs. Cela afin d'étudier la possibilité de développer ce type de jeux (généralement onéreux) à faible coût.

Notre hypothèse pour ces algorithmes partait de l'idée qu'un événement dans un jeu vidéo ayant une zone d'effet limitée, il était possible, en transformant l'environnement du jeu en un maillage de cases (chacune devenant un Acteur doté de sa propre boucle d'événements) et en utilisant un algorithme de diffusion capable d'opérer dans ce maillage, d'augmenter le nombre de personnages, d'artefacts (eux aussi des acteurs) et de cases en interaction simplement en ajoutant de nouveaux ordinateurs à une grappe d'ordinateurs, sans interrompre le jeu et sans nuire aux performances.

Pour mettre en œuvre ces algorithmes, nous avons développé une nouvelle version de notre moteur de jeu Antoid via le langage de programmation Erlang et monté une grappe d'ordinateur appelée Olifan.

Ensuite, pour étudier notre hypothèse, nous avons conçu une expérience consistant à mesurer l'évolution des délais de déplacements des personnages non-joueurs dans l'univers du jeu lorsque l'on augmente progressivement le nombre de personnages et de cases d'une part, puis le nombre d'ordinateurs d'autre part.

Au final, l'expérience qui a compris 60 itérations, a impliqué jusqu'à 279936 cases, 7776 personnages et quatre Raspberry Pi, a montré des résultats d'une grande stabilité

(en moyenne) pour les délais de déplacements.

En outre, il est à noter que notre nouveau mécanisme permettant aux objets d'Antoid de contrôler l'enchaînement de leurs actions a entrainé une amélioration par rapport à notre première version d'Antoid. Tout en maintenant un niveau de cohérence équivalent, il a permis d'éviter aux objets d'Antoid d'ignorer et perdre des événements en différant leur traitement.

Ces résultats étant encourageants, il nous semble tout à fait pertinent de poursuivre dans cette direction.

Aujourd'hui, Antoid ne peut pas encore accueillir de joueurs. Pour remédier à cela, notre prochaine étape consiste à interfacer notre nouveau moteur de jeu avec notre serveur Node.js que nous avons décidé de conserver et de dédier exclusivement à la gestion des joueurs.

Cependant, afin de ne pas limiter la quantité de joueurs qu'Antoid peut supporter, nous envisageons d'expérimenter l'interfaçage de plus d'un serveur Node.js en utilisant les ports Erlang (que nous avons présentés dans le chapitre 2 de ce mémoire). En effet, le rôle de ces serveurs se limitant à acheminer les messages entre le moteur Antoid et les programmes clients des joueurs, il n'est pas nécessaire qu'ils communiquent entre eux. De cette manière l'incapacité de Node.js à être distribué dans une grappe d'ordinateurs ne pose pas de problème.

Dans un avenir plus lointain, nous souhaiterions développer dans le langage de programmation C, une machine virtuelle allégée reprenant les seules parties de la machine virtuelle d'Erlang Beam que nous utilisons. c'est-à-dire principalement un gestionnaire de processus légers et un gestionnaire de messagerie universelle. En effet, de très nombreuses fonctionnalités de Beam nous sont inutiles dans le cadre d'Antoid. Or, notre objectif à long terme serait de pousser le concept de distribution plus loin en visant des processeurs légers.

Annexe A

Validation des fonctionnalités d'Antoid

Dans cette annexe, nous présentons plusieurs tests montrant le bon fonctionnement d'Antoid. Pour valider nos tests, nous utilisons les informations relatives à des événements intégrées à notre base de données. Pour plus de précisions sur le fonctionnement des actions de jeu que nous présentons, vous pouvez vous référer aux spécifications des objets d'Antoid donnés à la section 3.3

A.1 La construction du microcosme

La création du microcosme, doit pouvoir se faire à partir de la création d'une case initiale et se distribuer sur un nombre quelconque de nœuds Erlang.

Dans ces tests, nous avons procédé à la création de deux microcosmes. D'abord sur un seul nœud puis sur quatre nœuds (ce qui correspond au nombre de nœud maximum pour notre expériences).

Les deux microcosmes sont composés de 400 cases. Ainsi, lors du premier test, toutes ces cases devront se trouver sur le même nœud alors que dans le second test, chaque nœud ne devra héberger que 100 cases.

NbLocations	NodeName
400	1

Tableau A.1 – Test : construction d'un microcosme non distribué

NbLocations	NodeName
100	1
100	2
100	3
100	4

Tableau A.2 – Test : construction d'un microcosme distribué

Pour valider ces tests, nous allons utiliser l'événement correspondant à la création d'une case. Cet événement est transmis par une case après sa création. Il comprend entre autre l'identifiant de la case et l'identifiant du nœud dans lequel elle se trouve.

Pour que le premier test soit valide, nous devons obtenir 400 de ces événements et évidemment le même identifiant pour le nœud hébergeant les cases. Pour que le second test soit valide, chacun des quatre identifiants de nœud doit être associé à 100 événements sur les 400.

A.1.1 Résultats des tests

Au terme de ces tests, nous pouvons dire que la création du microcosme et sa distribution dans plusieurs nœuds fonctionnent. En effet, comme nous pouvons le voir sur le tableau A.1, nous obtenons bel et bien 400 événements sur notre nœud correspondant donc aux 400 cases qu'il est supposé héberger. De même, sur le tableau A.2, nous pouvons voir que pour chaque identifiant de nœud (1, 2, 3 et 4), nous obtenons 100 événements, montrant bien que chaque nœud héberge un quart du microcosme.

A.2 Le déplacements des personnages

Les personnages doivent être capable de se déplacer dans le microcosme en passant de la case où ils se trouvent vers une case voisine à celle ci. Les déplacements

peuvent échouer si le personnage tente de se rendre sur une case déjà occupée par un autre personnage, un mur ou encore si le personnage tente de sortir des limites du microcosme.

Pour ces tests, nous avons créé un microcosme distribué sur quatre nœuds dans lequel plusieurs personnages vont pouvoir se déplacer.

Pour valider ces tests, nous utilisons l'événement correspondant à un déplacement de personnage. Il est transmis par la case sur laquelle se trouve le personnage lorsqu'il initie un déplacement. Les informations qui nous intéressent ici, sont les identifiants des nœuds sur lesquels se trouvent la case de départ et la case d'arrivée ainsi que le résultat du déplacement qui peut être :

1. Une réussite, dont le code dans nos résultats est le chiffre 0.
2. Un échec car bloqué par un mur, dont le code dans nos résultats est le chiffre 1.
3. Un échec car bloqué par un autre personnage, dont le code dans nos résultats est le chiffre 2.
4. Un échec car le personnage est sur le bord du microcosme, dont le code dans nos résultats est le chiffre 3.

Pour que ces tests soient validés, nous devons donc obtenir des événements concernant des déplacements dans un même nœud et des événements concernant des migrations d'un nœud à l'autre. Ces tests doivent montrer des réussites et les différentes formes d'échec.

Il est à noter que comme le nombre de déplacement par personnage est fixe, le nombre de déplacements (échecs comme réussites) est donc prévisible. Pour ces tests, nous avons créé 2592 personnages devant chacun effectuer 324 tentatives de déplacements. Ainsi, nous devons obtenir 839 808 déplacements en tout pour valider ce test.

NbEvents	Result
780884	0
20028	1
32786	2
6110	3

Tableau A.3 – Test : déplacements des personnages

NbEvents	Result
4400	0
14	1
202	2

Tableau A.4 – Test : migration des personnages d'un noeud à l'autre

A.2.1 Résultats des tests

Au terme de ces tests, nous pouvons affirmer que le déplacement des personnages est fonctionnel. Nous pouvons en effet voir sur le tableau A.3 que nous obtenons tous les types de déplacements possibles. La migration des personnages est elle aussi fonctionnelle comme le montre le tableau A.4 qui présente aussi tous les cas de déplacements possibles. Enfin, si nous additionnons le nombre d'événement du tableau A.3, nous arrivons bel et bien à 839 808 déplacements en tout. Ceci valide nos tests.

A.3 La construction et la destruction de murs

Les murs sont des artefacts que les trolls (des personnages spéciaux) doivent être en capacité de construire ou détruire sur une des cases voisines de la case sur laquelle ils se trouvent. Les murs doivent aussi bloquer les déplacements vers la case sur laquelle ils se trouvent.

Pour ces tests, nous avons créé un microcosme distribué sur quatre nœuds contenant quelque centaines de personnages parmi lesquels, des trolls qui auront la possibilité de construire et détruire des murs.

Pour valider ce test, nous utilisons trois types d'événements :

A.3. La construction et la destruction de murs

1. Les constructions de murs.
2. Les destructions de murs.
3. Les déplacements des personnages.

Comme pour l'événement concernant le déplacement des personnages, les événements concernant la construction et la destruction de murs sont transmis par la case sur laquelle se trouve le troll ayant initié la tentative de construction ou de destruction. Les informations qui nous intéressent sont les identifiants des nœuds hébergeant la case où est situé le troll, la case ciblée par la construction ou la destruction ainsi que le résultat qui peut être :

1. la construction :
   (a) Une réussite, dont le code dans nos résultats est le chiffre 0.
   (b) Un échec car bloqué par un mur, dont le code dans nos résultats est le chiffre 1.
   (c) Un échec car bloqué par un autre personnage, dont le code dans nos résultats est le chiffre 2.
   (d) Un échec car le troll tente de construire un mur hors des limites du microcosme, dont le code dans nos résultats est le chiffre 3.
2. la destruction :
   (a) Une réussite, dont le code dans nos résultats est le chiffre 0.
   (b) Un échec car il n'y a pas de mur sur la case voisine, dont le code dans nos résultats est le chiffre 1.

Pour que ces tests soient validés, nous devons donc obtenir des événements concernant la construction et la destruction de murs. Nous cherchons aussi à obtenir des événements concernant une collaboration entre deux nœuds différents pour tous ces événements.

NbEvents	Result
887	0
524	1
135	2
100	3

Tableau A.5 – Test : construction de murs

NbEvents	Result
382	0
2	1

Tableau A.6 – Test : destruction de murs

Notons que contrairement aux déplacements, le nombre de constructions ou de destructions n'est pas fixe. Une construction est en effet initiée par un troll uniquement lorsqu'il a parcouru une certaine distance (en nombre de cases) et une destruction n'est initié que si le troll s'est retrouvé bloqué un certain nombre de fois par un mur lors de ses déplacements. Ainsi, compte tenu du fait que nous ne pouvons prévoir quelle distance va parcourir un troll (qui se retrouvera très probablement bloqué à plusieurs reprises), ni combien de murs il rencontrera, nous ne pouvons pas prévoir le nombre d'événements de constructions ou de destructions.

A.3.1 Résultats des tests

Au terme de ces tests, nous pouvons affirmer que la construction et la destruction des murs fonctionnent. Nous pouvons en effet voir sur les tableaux A.5 et A.6, que nous obtenons tous les cas de constructions et de destructions disponibles.

Nous pouvons aussi constater dans les tableaux A.7 et A.8 que des constructions et des destructions incluant deux nœuds sont aussi possibles.

A.4 Les informations visuelles

Lorsqu'un déplacement, une construction ou une destruction a lieu, les informations visuelles liées à ces événements doivent être diffusées depuis la case où cet

NbEvents	Result
6	0
7	1
3	2

Tableau A.7 – Test : construction de murs distribuée

NbEvents	Result
7	0
1	1

Tableau A.8 – Test : destruction de murs distribuée

événement s'est produit vers ses cases voisines. Cette diffusion doit s'arrêter lorsque la distance maximale de diffusion est atteinte, lorsqu'un mur bloque la diffusion ou lorsque la diffusion a atteint un bord du microcosme. En pratique une information visuelle concerne l'apparition ou la disparition d'un objet d'Antoid sur une case.

Pour ces tests, nous avons créé un microcosme distribué sur quatre nœuds contenant une centaines de personnages et trolls afin qu'ils puissent se déplacer, construire et détruire des murs.

Pour valider ces tests, nous utilisons l'événement correspondant à la diffusion d'information visuelle. Il est transmis par les cases lorsque celles-ci diffusent une information visuelle dans une direction ou qu'elle arrête la diffusion de l'information visuelle.

Les cas possibles de diffusion sont les suivants :

1. L'information visuelle est diffusée dans une direction, le code dans nos résultats est le chiffre 0.
2. La diffusion a atteint sa distance maximale, le code dans nos résultats est le chiffre 1.
3. La diffusion est bloquée par un mur, le code dans nos résultats est le chiffre 2.
4. La diffusion s'arrête car elle a atteint le bord du microcosme, le code dans nos résultats est le chiffre 3.

NbEvents	Result
299779	0
69117	1
2751	2
8504	3

Tableau A.9 – Test : diffusion d'information visuelles d'apparition

NbEvents	Result
298955	0
68920	1
2747	2
8474	3

Tableau A.10 – Test : diffusion d'information visuelles de disparition

Pour que ces tests soient validés, nous devons donc obtenir d'une part, des événements concernant la diffusion d'informations visuelles concernant l'apparition de personnages ou de murs, et d'autre part, la disparition de personnages et de murs. Nous devons aussi obtenir des résultats montrant la diffusion de ces informations visuelles d'un nœud à l'autre.

A.4.1 Résultats des tests

Comme nous pouvons le voir sur les tableaux A.9 et A.10, nous obtenons bel et bien les différents types de résultats concernant l'apparition ou la disparition de personnages et de murs.

De plus, comme nous pouvons le voir sur les tableaux A.11 et A.12, les informations visuelles sont bel et bien capables de passer d'un nœud à l'autre.

NbEvents	Result
18674	0
5012	1
51	2

Tableau A.11 – Test : diffusion d'information visuelles distribuée d'apparition

NbEvents	Result
18610	0
5006	1
50	2

Tableau A.12 – Test : diffusion d'information visuelles distribuée de disparition

Annexe B

Résultats complets de l'expérience

Moyenne					Écart-type				
Nb PNJ	1 RPI	2 RPI	3 RPI	4 RPI	Nb PNJ	1 RPI	2 RPI	3 RPI	4 RPI
324	8378	9162	6950	6946	324	30027	36954	25877	29390
648	10734	10409	9569	10755	648	36643	37721	32901	40032
972	10544	10025	9829	10830	972	36198	34399	33699	39990
1296	10931	10241	10264	11163	1296	36786	35244	34755	40584
1620	10853	10466	10371	11590	1620	36739	35503	35156	41756
1944	13077	10948	10655	12997	1944	47114	36825	35713	45394
2268		15427	13821	12794	2268		54092	48519	45168
2592		15728	13976	12859	2592		54691	49167	45416
2916		15433	13757	13063	2916		53302	48379	45457
3240		15828	14215	12985	3240		55340	49591	45763
3564		15654	14206	12839	3564		55202	49833	45381
3888		16494	13888	13106	3888		73248	49018	46199
4212			14146	13017	4212			49405	45522
4536			14247	13238	4536			49742	46130
4860			14029	13055	4860			49432	45983
5184			14025	13229	5184			49429	46330
5508			14240	13204	5508			50255	46342
5832			15282	13155	5832			73234	46224
6156				13329	6156				46695
6480				13305	6480				46257
6804				13223	6804				46442
7128				13313	7128				46787
7452				14305	7452				72900
7776				12830	7776				94638

Tableau B.1 – Moyenne et écart-type pour tous les déplacements

Moyenne					Écart-type				
Nb PNJ	1 RPI	2 RPI	3 RPI	4 RPI	Nb PNJ	1 RPI	2 RPI	3 RPI	4 RPI
324	8378	9125	6839	6773	324	30027	36905	25583	28882
648	10734	10405	9575	10734	648	36643	37748	32974	40090
972	10544	10023	9826	10799	972	36198	34412	33730	39983
1296	10931	10232	10247	11163	1296	36786	35226	34735	40586
1620	10853	10461	10373	11590	1620	36739	35500	35180	41694
1944	13077	10948	10656	12997	1944	47114	36832	35729	45398
2268		15433	13809	12782	2268		54113	48508	45172
2592		15730	13980	12858	2592		54707	49195	45442
2916		15434	13754	13048	2916		53316	48388	45406
3240		15830	14211	12985	3240		55352	49592	45719
3564		15657	14204	12839	3564		55215	49839	45397
3888		16031	13885	13106	3888		57417	49021	46218
4212			14144	13019	4212			49414	45537
4536			14242	13230	4536			49736	46123
4860			14026	13044	4860			49436	45943
5184			14020	13229	5184			49409	46310
5508			14222	13204	5508			50161	46333
5832			15051	13155	5832			68612	46221
6156				13323	6156				46678
6480				13305	6480				46262
6804				13217	6804				46411
7128				13306	7128				46747
7452				13942	7452				59806
7776				12447	7776				80950

Tableau B.2 – Moyenne et écart-type pour les déplacements internes à un noeud

Moyenne				Écart-type			
Nb PNJ	2 RPI	3 RPI	4 RPI	Nb PNJ	2 RPI	3 RPI	4 RPI
324	13629	15658	16049	324	42109	42217	48284
648	11747	8816	12508	648	28979	21899	34993
972	10563	10386	15175	972	30143	27088	40799
1296	14841	15018	14707	1296	43222	39652	40102
1620	14219	9661	20075	1620	37217	27133	55291
1944	10967	10474	17176	1944	31858	29627	43740
2268	9269	19473	16645	2268	21891	53418	43490
2592	13399	12129	13136	2592	37584	33024	35725
2916	13714	15636	18395	2916	34236	42120	60928
3240	12958	16224	22082	3240	30043	48923	63438
3564	12093	15753	12704	3564	35355	45024	37987
3888	972177	16304	12468	3888	1833533	46670	35648
4212		16545	12006	4212		39583	39019
4536		18976	19610	4536		54292	50716
4860		17170	21621	4860		44021	69344
5184		18709	18411	5184		64906	61102
5508		32001	17281	5508		108237	52848
5832		328968	17737	5832		893023	48886
6156			18913	6156			58894
6480			14064	6480			41584
6804			18811	6804			68071
7128			21100	7128			79407
7452			442127	7452			1367657
7776			523789	7776			1719164

Tableau B.3 – Moyenne et écart-type pour les migrations

Annexe C

Spécifications détaillées des actions de jeu

Cette annexe est une extension de la troisième section du chapitre deux de ce mémoire concernant les spécifications des différentes actions de jeu disponibles dans Antoid, que sont les déplacements des personnages, la construction et la destruction de murs ainsi que la diffusion d'informations visuelles dans le microcosme. Pour chacune de ces actions, nous présentons les données utilisées lors de cette action par les objets d'Antoid impliqués, ainsi que les différentes manière dont une action peut se dérouler. Chaque cas est aussi illustré par un diagramme de séquence.

C.1 Déplacements des personnages

C.1.1 Les données liées au déplacement

Quand un personnage arrive sur une case, il mémorise l'identifiant de cette case et la case mémorise l'identifiant du personnage. Cet identifiant est divisé en deux parties :

1. L'identifiant généré par Erlang qui permet aux différents objets d'Antoid de communiquer entre eux via le système de messagerie de la machine virtuelle Beam.

2. L'identifiant généré par Antoid, servant principalement à reconnaître un objet d'Antoid d'une expérience à l'autre (l'identifiant généré par Erlang ne restant pas le même d'une expérience à l'autre).

Pour la case, cet identifiant sert à déterminer si un personnage se trouve sur elle et le cas échéant, à communiquer avec ce personnage. Pour le personnage, cet identifiant sert à communiquer avec la case sur laquelle il se trouve afin d'initier des actions de jeu dont les déplacements.

Pour déterminer dans quelle direction se déplacer, un personnage conserve dans ses paramètres internes une liste contenant les différentes directions dans lesquelles il peut tenter de se rendre (pour rappel, l'est, l'ouest, le nord ou le sud).

Dans la version actuelle d'Antoid, le personnage se déplace dans une direction tant qu'il ne rencontre pas d'obstacle. Si le personnage se retrouve bloqué, il prend la prochaine direction de sa liste jusqu'à ce qu'il ne soit plus bloqué et reprend ses déplacements.

Prenons par exemple, la liste suivante : [Est, Ouest, Nord, Sud]. Le personnage va d'abord tenter de se déplacer vers l'Est. S'il se retrouve bloqué, il tentera alors de se déplacer vers l'Ouest et ainsi de suite jusqu'à ce qu'il ait épuisé toutes les directions. Quand son déplacement vers le Sud sera bloqué, il reviendra à la direction Est.

C.1.2 Le processus de déplacement

Un déplacement est initié par un personnage. Il transmet un message à la case qui le localise, indiquant la direction dans laquelle il souhaite aller. S'il existe bel et bien une case dans cette direction, la case sur laquelle se trouve le personnage lui transmet la demande de déplacement de son personnage. Les deux cases vont alors collaborer pour déterminer les conséquences de cette tentative de déplacement.

Si les cases déterminent que le déplacement est possible, la case de départ du personnage cesse de le localiser (en effaçant l'identifiant du personnage) pour permettre à

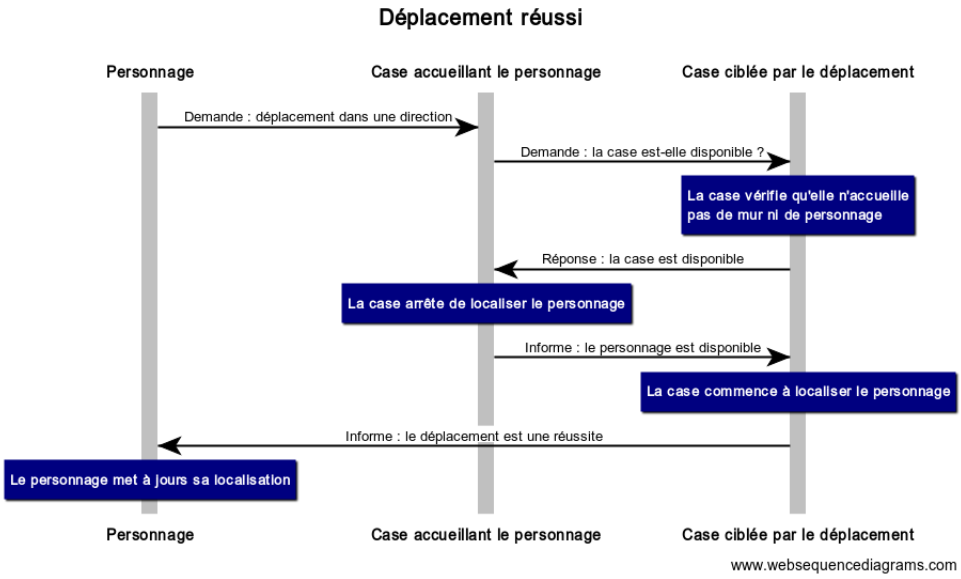

Figure C.1 – Déplacement réussi

la case d'arrivée de commencer à localiser le personnage (en mémorisant l'identifiant du personnage). La case d'arrivée informe alors le personnage que son déplacement a été un succès. Dans le cas où le déplacement est impossible, la case de départ du personnage l'informe que son déplacement est un échec.

Dans les prochaines sous sections, nous présentons les différents cas de figures pouvant se produire dans le cadre des déplacements de personnages. Chaque cas est accompagné d'un diagramme de séquence illustrant la collaboration entre les différents objets d'Antoid.

Un déplacement réussi

Ce cas est illustré à la figure C.1. Ici, la case ciblée par le déplacement est libre. La case de départ arrête donc de localiser le personnage et la case d'arrivée commence à le localiser.

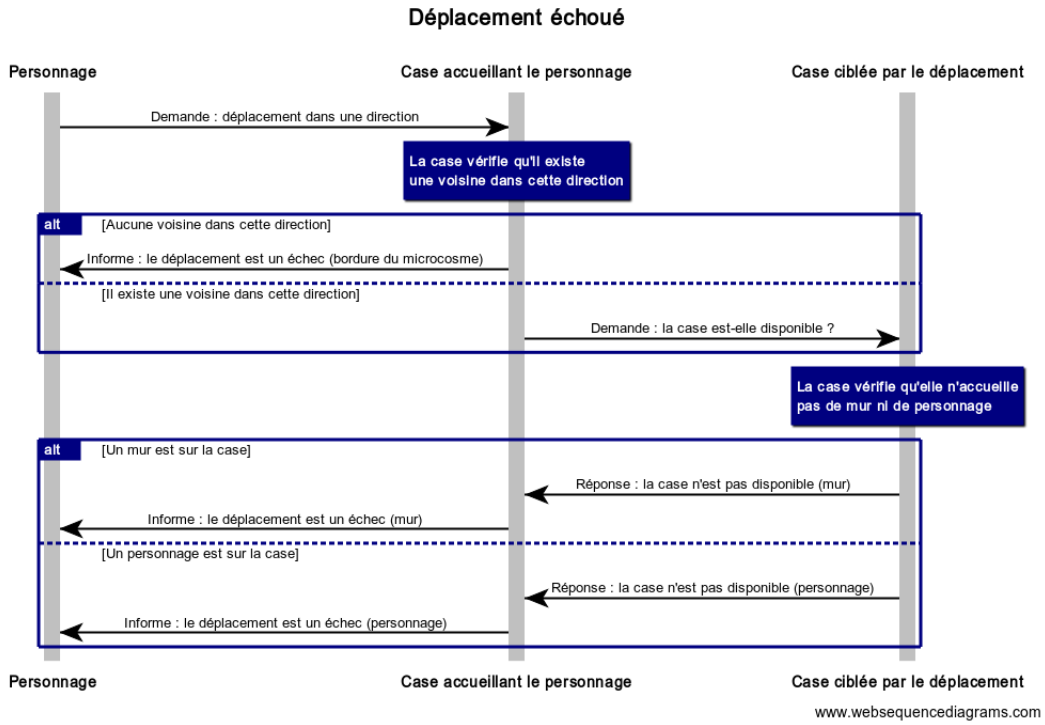

Figure C.2 – Déplacement raté

Échec d'un déplacement

Ce cas est illustré à la figure C.2. Ici, le personnage reste sur sa case. Un déplacement peut échouer dans les cas suivants :

1. Il n'existe pas de case dans la direction où souhaite se déplacer le personnage.
2. La case ciblée par le déplacement est déjà occupée par un personnage.
3. La case ciblée par le déplacement est déjà occupée par un mur.

Une case convoitée par plusieurs personnages

Ce cas est illustré à la figure C.3. Ici, plusieurs personnages souhaitent se déplacer sur une même case, en même temps.

Si nous admettons que la case est libre, c'est la demande étant arrivée en premier qui verra son déplacement aboutir. Toutes les autres demandes seront différées jus-

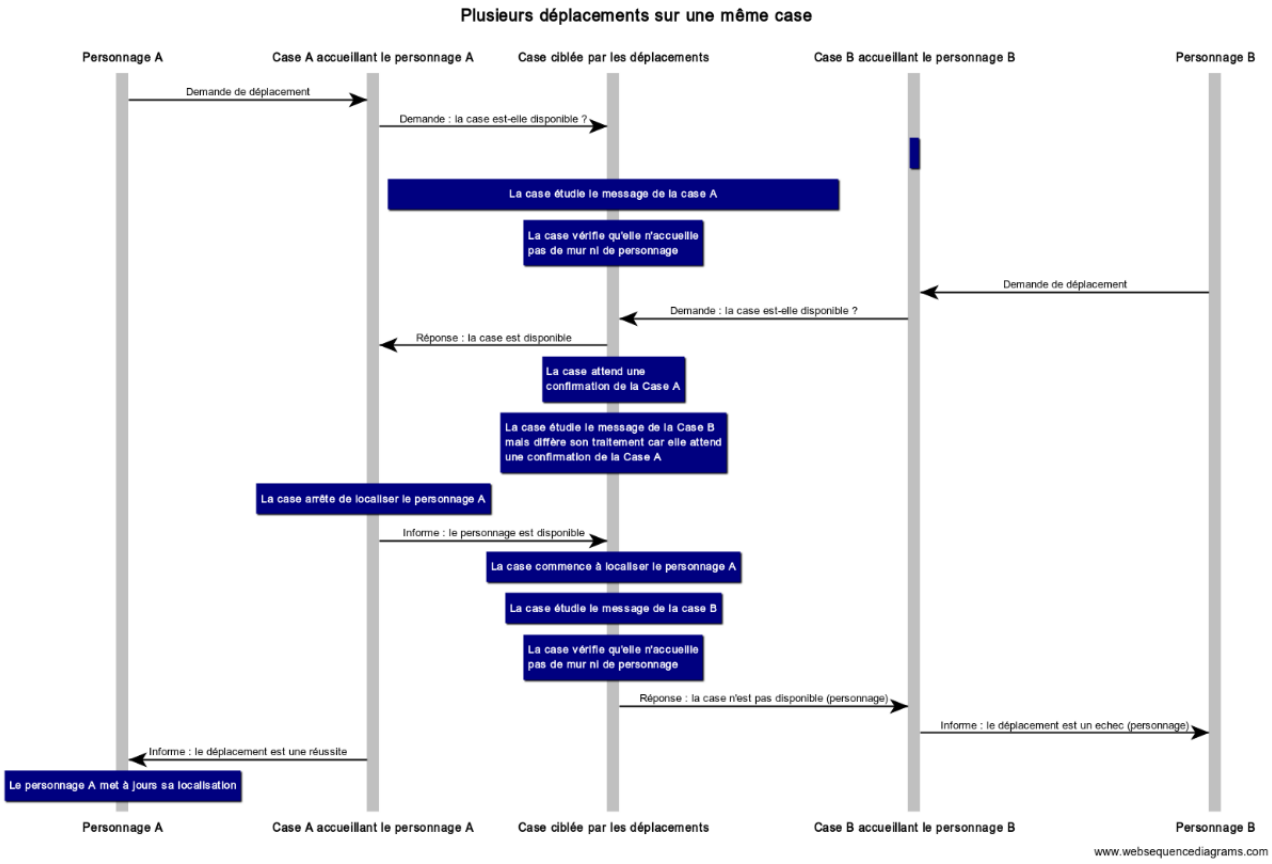

Figure C.3 – Plusieurs déplacements sur une même case

qu'à ce que le déplacement soit totalement terminé ou que le personnage annule son déplacement.

Notons que les personnages ne peuvent pas encore annuler leurs déplacements dans cette version d'Antoid, donc les messages différés ne recevront jamais de réponse positive.

C.2 Construction de murs

Les murs sont des artefacts pouvant être construits par des personnages spéciaux que nous appelons des trolls.

C.2.1 Les données liées à la construction de murs

Comme pour les autres personnages, quand un troll arrive sur une case il mémorise l'identifiant de cette case et la case mémorise l'identifiant du troll. Au même titre que les déplacements, la construction de mur est une action de jeu. C'est donc en envoyant un message à la case qu'il occupe que le troll peut initier une construction de mur.

La direction dans laquelle les trolls tentent de construire leurs murs est déterminée à la création du troll. Ainsi, si cette direction est « l'Est », le troll tentera toujours de construire ses murs sur la case voisine située à l'Est.

La cadence à laquelle un troll tente de construire un mur est fonction du nombre de déplacements réussis effectué, qui est paramétré à la création du troll. Ainsi, si ce nombre est à trois, le troll tentera de construire un mur après avoir réussi son troisième déplacement. Une fois la tentative de construction achevée (qu'elle soit une réussite ou un échec), le troll attendra de nouveau d'avoir réussi trois déplacements avant de retenter une construction de mur.

Rappelons que ce sont les cases et non les trolls qui créent les murs, le troll ne fait qu'initier la construction. De plus, les murs comme tous les objets d'Antoid, possèdent un identifiant.

Une construction réussie

Ce cas est illustré à la figure C.4. Ici, la case ciblée par la construction est libre. Elle va donc créer un mur sur son emplacement.

Échec d'une construction

Ce cas est illustré à la figure C.5. Ici, aucun mur n'est construit. Cela peut se produire dans les trois cas suivant :

1. Il n'existe pas de case dans la direction où le troll souhaite construire un mur.
2. La case ciblée par la construction est déjà occupée par un personnage.
3. La case ciblée par la construction est déjà occupée par un mur.

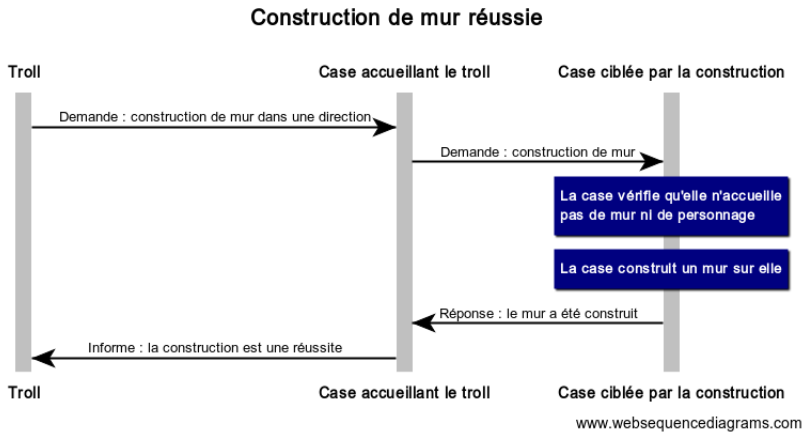

Figure C.4 – Construction réussie

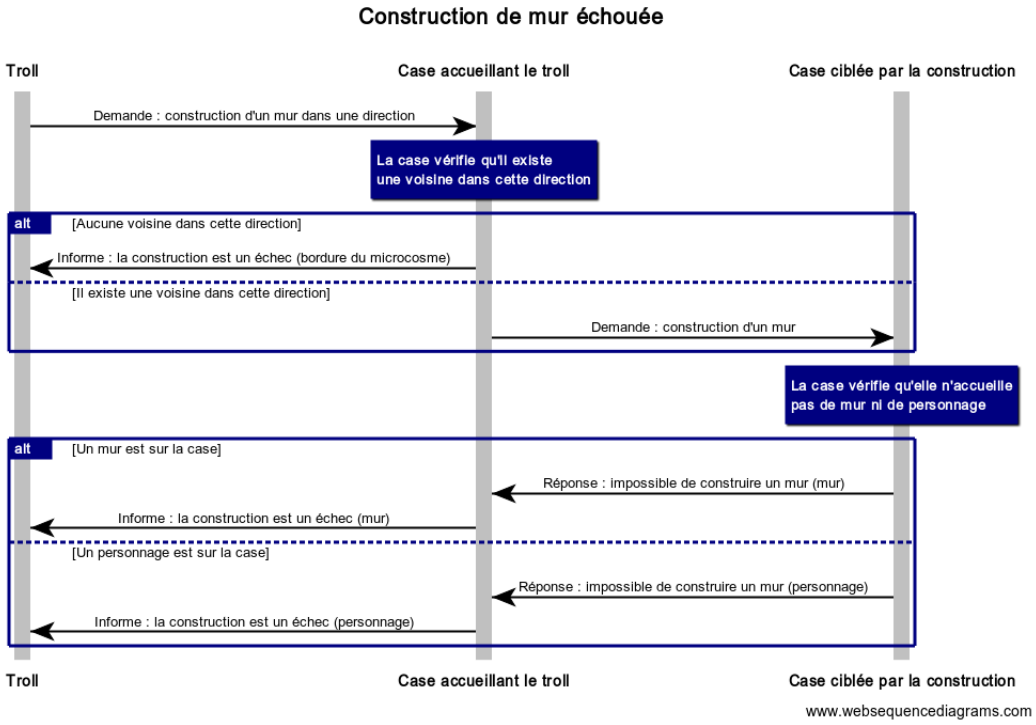

Figure C.5 – Construction échouée

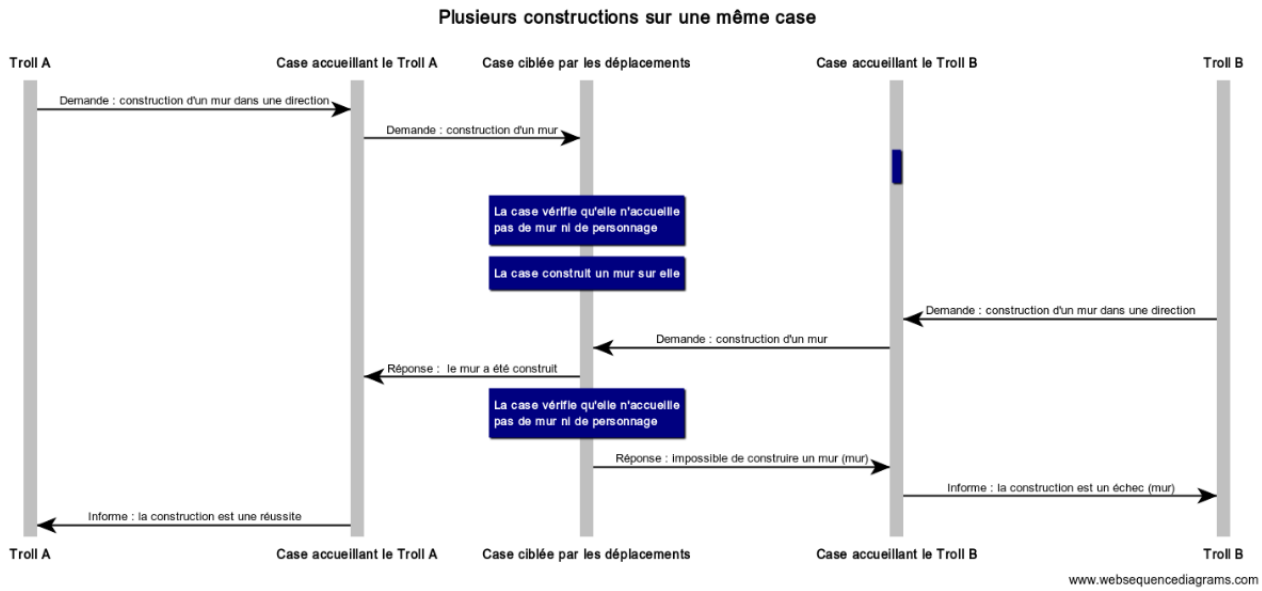

Figure C.6 – Plusieurs constructions sur une même case

Plusieurs trolls tentent de construire un mur sur une même case

Dans ce cas de figure C.6, plusieurs trolls souhaitent construire un mur sur une même case en même temps.

Si nous admettons que la case est libre, c'est la demande étant arrivée en premier qui verra sa construction aboutir. Dès cette première demande traitée, la case va créer un mur sur son emplacement. Toutes les autres demandes arrivées après recevront donc une réponse négative.

C.3 Destruction de murs

Comme pour la construction, la destruction de murs est une action de jeu réservée aux trolls.

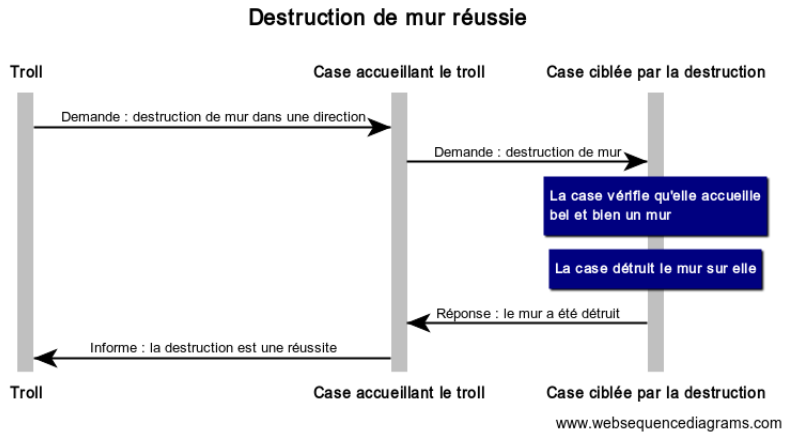

Figure C.7 – Destruction réussie

C.3.1 Les données liées à la destruction de murs

La destruction de murs, fonctionne comme toutes les autres actions de jeu. Elle est initiée en envoyant un message à la case que le troll occupe.

La fréquence d'utilisation de cette action, est déterminée par un nombre limite de fois qu'un troll « accepte » d'être bloqué dans ses déplacements par un mur. Ainsi, quand cette limite est atteinte, le troll tentera de détruire le mur sur son chemin.

Comme pour la construction de murs, ce sont les cases et non les trolls qui détruisent les murs, le troll ne fait qu'initier la destruction.

Une destruction réussie

Ce cas est illustré à la figure C.7. Ici, la case ciblée par la construction est occupée par un mur. Elle va donc détruire le mur sur son emplacement.

Échec d'une destruction

Ce cas est illustré à la figure C.8. Ici, il n'y a pas de mur à détruire sur la case. Cela peut se produire quand plusieurs trolls tentent de détruire un même mur où le troll le plus lent échoue dans sa tentative.

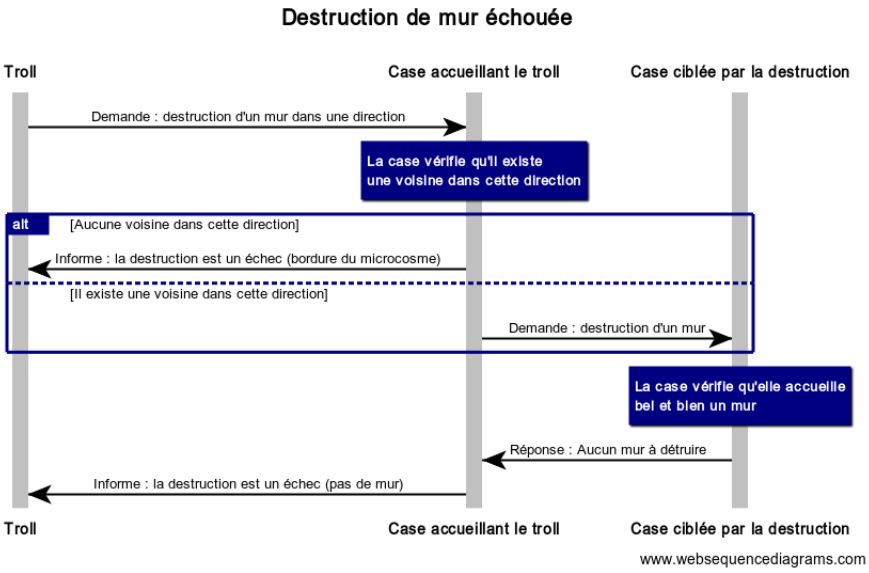

Figure C.8 – Destruction échouée

C.4 Diffusion d'informations visuelles

C.4.1 Les données liées à la diffusion d'information visuelles

Chaque case conserve dans sa mémoire une liste des objets d'Antoid qu'elle voit, à quelle distance ils se trouvent et dans quelle direction.

La case ajoute des éléments à cette liste lorsqu'elle reçoit des informations visuelles concernant l'apparition d'un objet et retire des éléments lorsqu'elle reçoit des informations concernant la disparition d'un objet. Précisons aussi que les cases ne constituent pas des objets visibles dans Antoid.

Lorsqu'elles sont diffusées, les informations visuelles encapsulent les cinq informations suivantes :

1. la direction d'où vient l'information visuelle ;
2. la distance parcourue par l'information visuelle ;
3. le type d'information visuelle, qui peut être soit l'apparition soit la disparition d'un objet d'Antoid ;

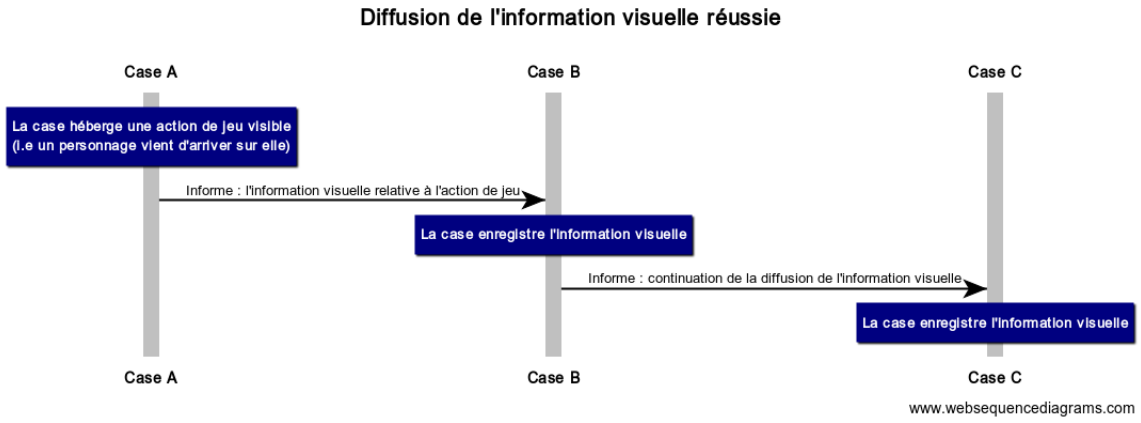

Figure C.9 – Diffusion d'information visuelle

4. le type d'objet d'Antoid qui apparaît ou disparaît (aujourd'hui un personnage ou un mur) ;
5. si le type d'objet d'Antoid est occultant ou non.

Une diffusion sur plusieurs cases

Ce cas est illustré à la figure C.9. Ici, tant que la diffusion n'est pas bloquée, les cases continuent à transmettre l'information visuelle à la voisine située dans la bonne direction.

Interruption de la diffusion

Ce cas est illustré à la figure C.10. Ici, la diffusion de l'information visuelle s'interrompt. Cela peut arriver dans les cas suivant :

1. il n'existe pas de case à qui transmettre l'information visuelle ;
2. la distance maximale de la diffusion est atteinte ;
3. un mur sur une case bloque la diffusion vers la case suivante.

L'apparition d'un mur

Ce cas est illustré à la figure C.11. Ici, un mur est construit sur une case (il apparaît) et ce qui se trouve derrière le mur n'est plus visible. Ainsi, lorsqu'une case

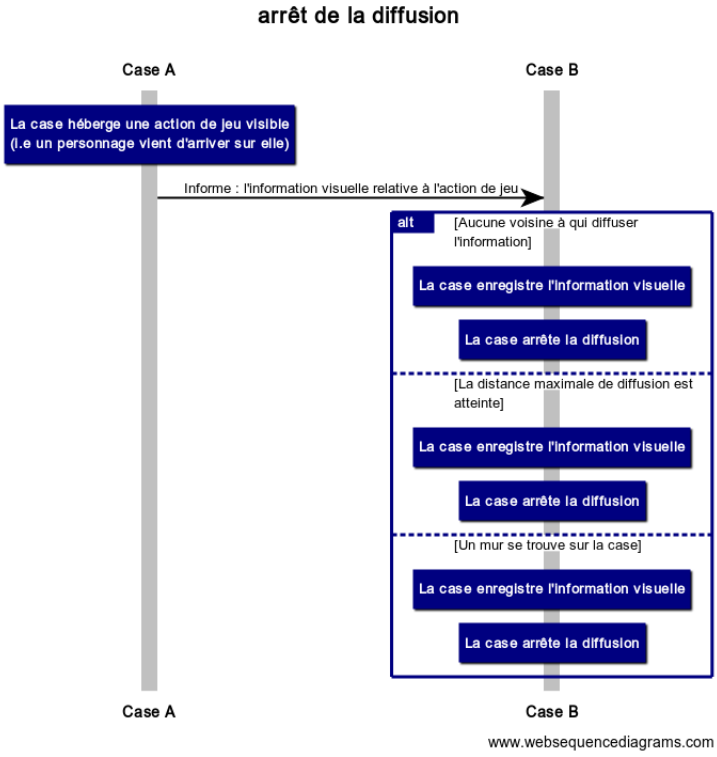

Figure C.10 – Arrêt de la diffusion d'information visuelle

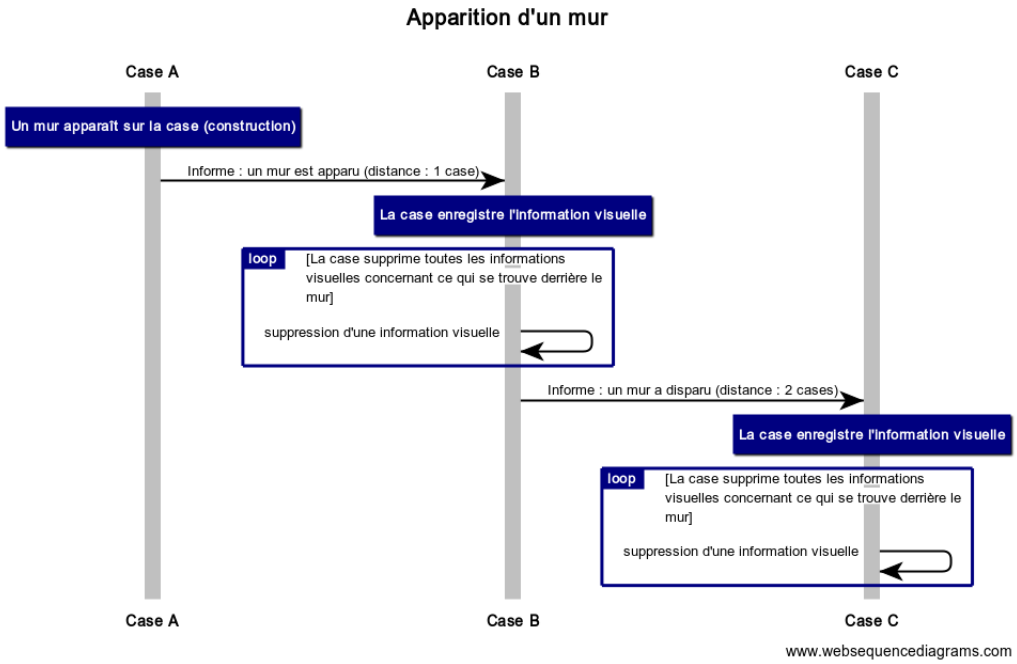

Figure C.11 – Apparition d'un mur

reçoit une information visuelle concernant l'apparition d'un mur sur une case, elle supprime toutes les informations concernant ce qui se trouve derrière ce mur.

La disparition d'un mur

Ce cas est illustré à la figure C.12. Ici, un mur est détruit (il disparaît) et les objets d'Antoid qui se trouvaient derrière ce mur redeviennent visible. La case dont le mur a été détruit diffuse alors toutes ces informations visuelles à ses voisines en plus de celle concernant la disparition du mur.

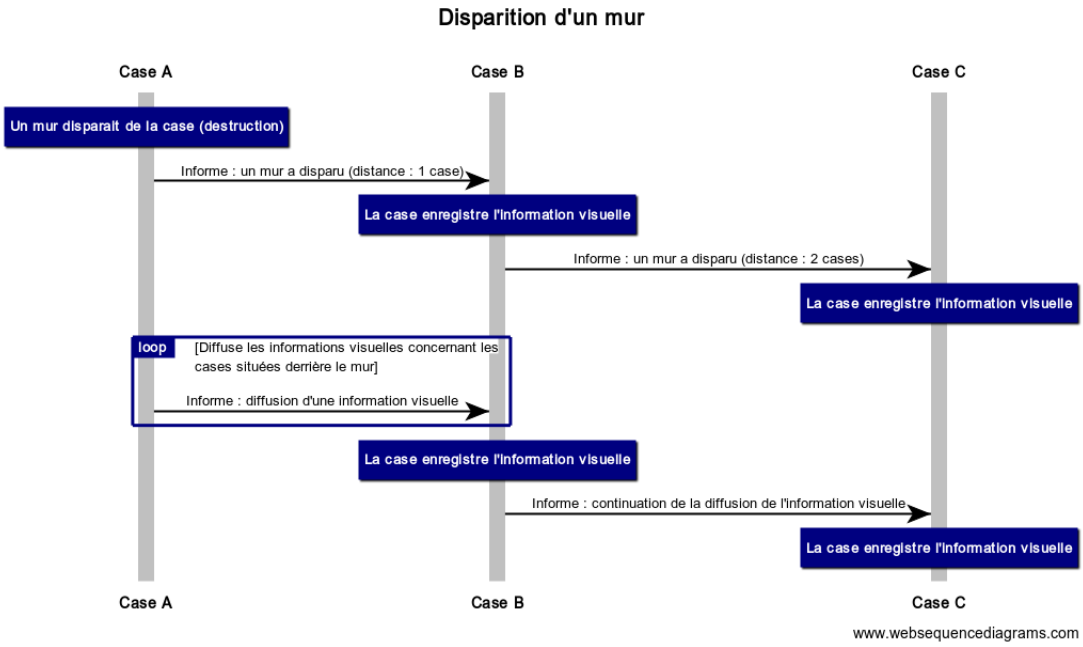

Figure C.12 – Disparition d'un mur

Annexe D

Méthodologie de traitement des données issues des expériences

Cette annexe présente la manière dont les données produites par nos expériences sont intégrées à notre base de données et comment ces données sont utilisées une fois enregistrées.

D.1 Traitement des fichiers

Au terme d'une expérience, chacun des processus d'écriture (que nous avons décrit dans le chapitre 4 de ce mémoire) écrit dans des fichiers textes au format csv (Comma Separated Values), les différents événements incorporés dans les messages qu'il a reçu de la case à laquelle il est associé. Notons que si l'expérience a été produite sur plus d'un RPI, ces fichiers sont répartis entre les différents RPI.

Un fichier enregistre les données correspondant à un type d'événement particulier, interprété par un type d'objet d'Antoid particulier. Par exemple, un événement peut correspondre à un déplacement et être interprété par une case en fonction des informations événementielles qu'elle perçoit.

Pour différencier les fichiers les uns des autres, nous appliquons une convention sur leurs noms. Chaque nom de fichier est composé de deux codes, le premier est

une référence qui correspond au type de l'événement et le deuxième correspond à l'identifiant de l'objet d'Antoid qui enregistre les données.

Pour illustrer, prenons l'exemple d'un fichier stockant les événements correspondant au déplacement d'un personnage.

Le code de ce type d'événement est : «MOVE». Maintenant, si nous supposons que l'identifiant de la case ayant enregistré l'événement est «350», nous obtiendrons un fichier nommé «MOVE350».

D.2 Intégration dans la base de données

Pour organiser les données des fichiers, nous les intégrons dans la base de données relationnelle PostgreSQL. La figure D.1 illustre cette procédure.

À la fin d'une expérience, nous utilisons un script shell illustré dans le programme D.1 (bash) [20] pour réunir dans un dossier unique les fichiers dispersés dans les différents nœuds. Cela facilite leur intégration dans la base de données.

programme D.1 – Script réunissant les fichiers dispersés sur plusieurs noeuds

```
for var1 in 1 $1
do
  scp -r 192.168.0.$var1:/home/Antoid/Experience
         /home/Antoid/Experience
done
```

Rappelons que l'adresse IP des nœuds est normalisée et commence à l'adresse 192.168.0.1 (par exemple le nœud numéro 3 possède l'adresse 192.168.0.3)

Nous utilisons alors une fonction PL/pgSQL (procedural langage/PostgreSQL) pour l'intégration des fichiers dans les différentes tables.

Les données sont organisées de la même manière que les fichiers. Chaque table reprend donc l'identifiant de chaque type d'événement, comme «MOVE».

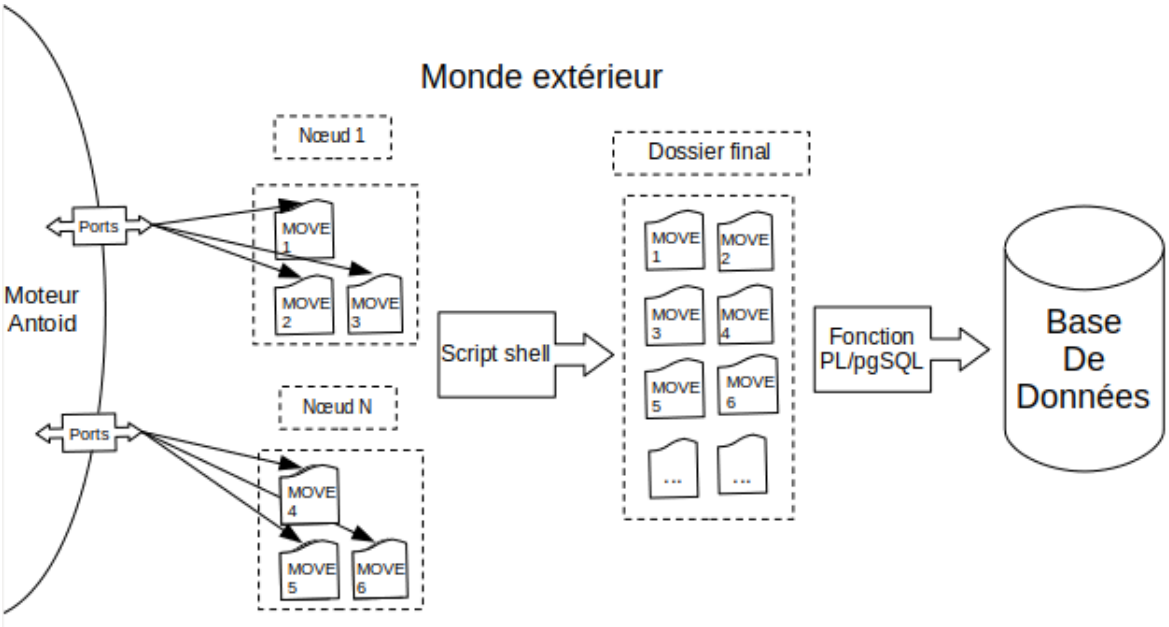

Figure D.1 – Schema illustrant la procédure de création et de stockage des données

D.3 Différents usages des données

Hormis l'usage évident de cette base de données à des fins statistiques, nous avons aussi utilisé ces données dans deux autres cas de figures.

Tout d'abord, à des fins de débogage. En effet, la combinaison de tous ces événements associée aux requêtes SQL nous a offert un outil puissant pour repérer les incohérences dans notre programme. Cet outil s'est montré très efficace, surtout dans le cadre de notre style de programmation local. Il est ainsi possible de suivre le comportement de chaque objet d'Antoid ou au contraire d'étudier le comportement global du programme.

Par exemple, nous pouvions nous assurer que chaque personnage effectuait bel et bien le nombre de déplacements prévu par l'expérience via la requête présenté dans le programme D.2.

programme D.2 – Exemple de requête SQL utilisée sur des données Antoid

```
SELECT Experience, CharacterId, COUNT(Move) Compte
FROM MOVE
WHERE Compte <> ValeurAttendue
GROUP BY Experience, CharacterId
ORDER BY Experience, CharacterId
```

Un autre usage de ces données a été d'utiliser les événements de déplacement dans le cadre du développement d'un prototype de restitution graphique développé en C et exploitant la bibliothèque SDL2 [12].

Nous avons utilisé ces données dans le cadre de nos tests afin de simuler un flux de données qu'Antoid enverrait à un joueur.

Notons que ce prototype permet aussi de produire la cinématique de tous événements s'étant produit entre deux jalons chronologique. Il est par exemple possible de reproduire la génération du microcosme de la case initiale au microcosme complet.

Bibliographie

[1] J. Armstrong, « Concurrency Oriented Programming in Erlang, » *Invited talk, FFG*, 2003.

[2] J. Armstrong, *Programming Erlang : software for a concurrent world*. Pragmatic Bookshelf, 2013.

[3] R. Balicer, « Modeling Infectious Diseases Dissemination Through Online Role-Playing Games, » *Epidemiology (Cambridge, Mass.)*, vol. 18, pp. 260–1, 2007.

[4] « Beam source code, » (page consultée le 2019-11-15). Disponible à https://github.com/erlang/otp/blob/master/erts/emulator/beam/erl_process.c

[5] « Berlin Interpretation of rogue-like genre, » 2008. Disponible à http://www.roguebasin.com/index.php?title=Berlin_Interpretation

[6] H. Baker et C. Hewitt, « Laws for communicating parallel processes, » *MIT Artificial Intelligence Laboratory Working Papers, WP-134A*, 1977.

[7] D. Bonetta, « The parallel event loop model and runtime, » Thèse de doctorat, Università della Svizzera italiana, 2014.

[8] S. Brookes, « Communicating parallel processes, » dans *Millenium Perspectives in Computer Science, Proceedings of the Oxford-Microsoft Symposium in honour of Professor Sir Antony Hoare, edited by Jim Davies, Bill Roscoe, and Jim Woodcock, Palgrave Publishers*, 2000.

[9] K. M. Chandy, M. Charpentier, et A. Capponi, « Towards a theory of events, » dans *Proceedings of the 2007 inaugural international conference on Distributed event-based systems*. ACM Press, 2007, p. 180.

[10] W. community, « HTML5 specifications, » (page consultée le 2019-11-15). Disponible à https://html.spec.whatwg.org/multipage/introduction.html#introduction

[11] K. M. Chandy et W. Schulte, « What is event driven architecture (EDA) and why does it matter ? » *Retrieved May*, vol. 20, p. 2009, 2007.

[12] S. developpers, « Simple DirectMedia Layer, » (page consultée le 2019-11-15). Disponible à https://www.libsdl.org/index.php

[13] E. O. Developers, « StacklessIO, » 2008. Disponible à http://www.eveonline.com/devblog.asp?a=blog&bid=584

[14] E. O. Developers, « bloodbath of B-R5RB, » 2014. Disponible à https://www.eveonline.com/fr/article/the-bloodbath-of-b-r5rb

[15] E. O. Developers, « TRANQUILITY TECH III, » 2015. Disponible à https://www.eveonline.com/fr/article/tranquility-tech-3

[16] T. K. Das, G. Singh, A. Mitchell, P. S. Kumar, et K. McGee, « NetEffect : a network architecture for large-scale multi-user virtual worlds, » dans *Proceedings of the ACM symposium on Virtual reality software and technology*. Citeseer, 1997, pp. 157–163.

[17] K. Elphinstone et G. Heiser, « From L3 to seL4 what have we learnt in 20 years of L4 microkernels ? » dans *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 133–150.

[18] O. Etzion, P. Niblett, et D. C. Luckham, *Event processing in action*. Manning Greenwich, 2011.

[19] « Erlang, » (page consultée le 2019-11-15). Disponible à https://www.erlang.org/

[20] F. S. Foundation, « GNU Bash, » (page consultée le 2019-11-15). Disponible à https://www.gnu.org/software/bash/

[21] M. Games, « The fantastic combinations of John Conway's new solitaire game "life" by Martin Gardner, » *Scientific American*, vol. 223, pp. 120–123, 1970.

[22] I. Greif, « Semantics of communicating parallel processes. » Thèse de doctorat, Massachusetts Institute of Technology, 1975.

[23] D. Grey, « Screw the Berlin Interpretation ! » 2013. Disponible à http://www.gamesofgrey.com/blog/?p=403

[24] F. Hebert, *Learn you some Erlang for great good ! : a beginner's guide.* No Starch Press, 2013.

[25] C. Hewitt, « Actor model of computation : scalable robust information systems, » *arXiv preprint arXiv :1008.1459*, 2010.

[26] M. Hori, T. Iseri, K. Fujikawa, S. Shimojo, et H. Miyahara, « Scalability issues of dynamic space management for multiple-server networked virtual environments, » dans *2001 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (IEEE Cat. No.01CH37233)*, vol. 1, Aug 2001, pp. 200–203 vol.1.

[27] C. A. R. Hoare, « Communicating sequential processes, » *Comm. ACM 21 (1978) 666-677*, 1978.

[28] D. Kushner, « Engineering Everquest, » *IEEE Spectrum Magazine*, 2005.

[29] I. Millington et J. Funge, *Artificial Intelligence for Games, Second Edition*, 2nd édition. Morgan Kaufmann, septembre 2009.

[30] « Node.js, » (page consultée le 2019-11-15). Disponible à https://nodejs.org/fr/

[31] « Node.js Documentation : The Node.js Event Loop, » (page consultée le 2019-11-15). Disponible à https://nodejs.org/de/docs/guides/event-loop-timers-and-nexttick/

[32] « Node.js Documentation : Worker Threads, » (page consultée le 2019-11-15). Disponible à https://nodejs.org/api/worker_threads.html#worker_threads_worker_threads

[33] S. Papert, *Mindstorms : children, computers, and powerful ideas.* New York : Basic Books, 1980.

[34] « Erlang Documentation : Ports and Port Drivers, » (page consultée le 2019-11-15). Disponible à http://erlang.org/doc/reference_manual/ports.html

[35] « PostgreSQL, » (page consultée le 2019-11-15). Disponible à https://www.postgresql.org/

[36] « RASPBERRY PI 3 : SPECS, BENCHMARKS & TESTING, » (page consultée le 2019-11-15). Disponible à https://www.raspberrypi.org/products/raspberry-pi-3-model-b/

[37] « Raspbian, » (page consultée le 2019-11-15). Disponible à https://www.raspbian.org/FrontPage

[38] K. Simpson, *You Don't Know JS : Scope & Closures*. O'Reilly Media, 2014. Disponible à https://github.com/getify/You-Dont-Know-JS/blob/master/scope%20&%20closures/README.md#you-dont-know-js-scope--closures

[39] K. Simpson, *You Don't Know JS : this & Object Prototypes*. O'Reilly Media, 2014. Disponible à https://github.com/getify/You-Dont-Know-JS/blob/master/this%20&%20object%20prototypes/README.md#you-dont-know-js-this--object-prototypes

[40] K. Simpson, *You Don't Know JS : Async & Performance*. O'Reilly Media, 2015. Disponible à https://github.com/getify/You-Dont-Know-JS/blob/master/async%20&%20performance/README.md#you-dont-know-js-async--performance

[41] K. Simpson, *You Don't Know JS : ES6 & Beyond*. O'Reilly Media, 2015. Disponible à https://github.com/getify/You-Dont-Know-JS/blob/master/es6%20&%20beyond/README.md#you-dont-know-js-es6--beyond

[42] K. Simpson, *You Don't Know JS : Types & Grammar*. O'Reilly Media, 2015. Disponible à https://github.com/getify/You-Dont-Know-JS/blob/master/types%20&%20grammar/README.md#you-dont-know-js-types--grammar

[43] K. Simpson, *You Don't Know JS : Up & Going*. O'Reilly Media, 2015. Disponible à https://github.com/getify/You-Dont-Know-JS/blob/master/up%20&%20going/README.md#you-dont-know-js-up--going

[44] M. v. Steen et A. S. Tanenbaum, *Distributed systems*. Pearson Education, 2017.

[45] E. Stenman, « The BEAM Book, » 2017. Disponible à http://erlang.org/doc/reference_manual/ports.html

[46] Tolkiendil, « Site internet proposant de télécharger Moria, » (page consultée le 2019-11-15). Disponible à http://www.tolkiendil.com/telechargements/jeux/moria_2

[47] « V8 a JavaScript engine, » (page consultée le 2019-11-15). Disponible à https://v8.dev/

[48] B. D. Vleeschauwer, B. V. D. Bossche, T. Verdickt, F. D. Turck, B. Dhoedt, et P. Demeester, « Dynamic microcell assignment for massively multiplayer online gaming, » dans *Proceedings of the 4th Workshop on Network and System Support for Games, NETGAMES 2005, Hawthorne, New York, USA, October 10-11, 2005*, 2005, pp. 1–7. Disponible à https://doi.org/10.1145/1103599.1103611