TABLE DES MATIÈRES

Page

LISTE DES TABLEAUX

Page

LISTE DES FIGURES

Page

LISTE DES ALGORITHMES

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

CPU	Central Processing Unit
FPS	Frames per second OU First Person Shooter
GPU	Graphical Processing Unit
IaaS	Infrastructure as a Service
LS	Lightweight Scaling
MLIT	Main Loop Iteration Time
MMOG	Massively Multiplayer Online Game
MOBB	Monolithic Oriented Bouncing Balls
OS	Operating System
PaaS	Platform as a Service
PM	Physical Machine
QoS	Quality of Service
RAM	Random Access Memory
RTT	Round Trip Time
SaaS	Software as a Service
SOBB	Substrates Oriented Bouncing Balls
TSP	Time Spent with Physics
TSR	Time Spent with Rendering
UML	Unified Modeling Language
VM	Virtual Machine

INTRODUCTION

Depuis ses débuts en 1970, l'industrie du jeu vidéo a connu une effervescence considérable. Estimée en 2010 à plus de 25 milliards de dollars américains d'après Gallagher (2011), président de l'ESA (*Entertainment Software Association*), celle-ci a dépassé le chiffre d'affaires du cinéma dans de nombreux pays développés. Aujourd'hui, les jeux vidéo se sont diversifiés et ciblent un public toujours plus large, allant des joueurs occasionnels (*casual players*) aux plus passionnés. Initialement conçus pour les bornes d'arcades puis les consoles portables, les consoles de salon et les ordinateurs, les jeux vidéo sont désormais disponibles sur un plus large éventail de terminaux, comme les téléphones intelligents et les tablettes. De nos jours, les jeux vidéo ne se limitent plus au divertissement, ils servent également dans le domaine de l'apprentissage.

Depuis 2002, un nouveau paradigme nommé *cloud gaming* (Shea *et al.*, 2013) révolutionne la façon dont les jeux vidéo sont distribués. Plutôt que de les exécuter sur le terminal du joueur, ceux-ci sont entièrement hébergés sur des serveurs de traitement distants. Le terminal du joueur désormais qualifié de *thin* ou *dumb client* ne fait que transmettre les commandes aux serveurs et reçoit en retour un flux vidéo contenant la représentation graphique et sonore du jeu.

Le principe du *cloud gaming* est d'ailleurs lui-même basé sur le paradigme du *cloud computing*. Ce dernier a connu un essor considérable avec la montée en puissance des composants informatiques (associée à une baisse des coûts) ainsi qu'à la démocratisation d'Internet haut débit. Le *cloud computing* permet d'accéder de manière simplifiée à différents types de services informatiques, allant d'une infrastructure virtualisée pour l'IaaS à des logiciels prêts à utiliser dans le cas du SaaS, en passant par des environnements de développement autogérés (PaaS).

Tout comme la visio-conférence, les jeux vidéo sont des applications ayant des contraintes temporelles. L'effet des actions entreprises par les joueurs doit se manifester le plus

rapidement possible à l'écran, pour conserver une ergonomie satisfaisante. De ce fait, à ce jour, la grande majorité des études liées au *cloud gaming* ont présenté des techniques visant l'amélioration de la qualité de service, fortement pénalisée par la latence d'Internet. Malheureusement, peu de travaux ont tenté de proposer des architectures innovantes. Actuellement, la grande majorité des jeux vidéo sont monolithiques : ils sont compilés et constitués d'un binaire unique représenté par une seule instance d'un processus (contenant un ou plusieurs *threads*) dans un système d'exploitation.

Objectifs de la recherche

L'objectif principal de ce projet est de spécifier et de valider une nouvelle architecture virtualisée et orientée services permettant la création de jeux vidéo à partir de briques logicielles spécialisées. Notre architecture inclut les composants suivants : un modèle d'affaires défini par le paradigme des *substrates* (expliqué plus loin), des entités fonctionnelles ainsi qu'un algorithme de gestion de ressources permettant d'assurer une qualité de service prédéfinie.

Méthodologie de travail et contributions

Pour atteindre chacun de ces objectifs, nous proposons d'utiliser le paradigme des *substrates*. Implémenté avec succès par Belqasmi *et al.* (2011) dans le cas des serveurs vocaux interactifs (IVR, *Interactive Voice Response*), l'approche des *substrates* consiste à construire une application à partir de plusieurs petits modules dotés d'interfaces et spécialisés pour un ensemble de tâches données. Les *substrates* sont fournis par des *substrates providers* dans une architecture hébergée dans le *cloud* et orientée services. Ils sont destinés à être loués par des *service providers*, pour satisfaire les besoins de clients finaux, qui dans notre cas sont des joueurs. Lorsque des *service providers* composent des *substrates* pour construire une application, on parle alors de composition.

Dans le cas des infrastructures de virtualisation de jeux vidéo, notre paradigme sera présenté à travers une architecture orientée services, en détaillant les entités nécessaires et les étapes permettant la création d'une composition à partir de *substrates*. Puis à travers un prototype de jeu orienté *substrates*, nous tenterons de prouver la viabilité de notre approche en le comparant à un jeu conçu avec une architecture traditionnelle, c'est à dire monolithique.

De plus, nous proposons de définir et valider un algorithme de gestion de ressources pour l'architecture présentée dans le chapitre deux. Ce dernier, également qualifié d'algorithme de dimensionnement dynamique des ressources, devra tenir compte du temps de réponse d'une application et prendre les décisions nécessaires pour qu'elle s'exécute dans des conditions adéquates. Ainsi, en cas de sous approvisionnement, c'est-à-dire si le temps de réponse dépasse une limite maximale dégradant l'expérience utilisateur, il devra tenter d'allouer des ressources pour y mettre un terme. En cas de sur approvisionnement, c'est-à-dire si le temps de réponse est suffisamment bas, il devra procéder à une libération des ressources, dans le but de réduire les dépenses. Outre la nécessité de conserver une qualité de service, cet algorithme devra assurer la consolidation des ressources, c'est-à-dire minimiser au maximum le nombre de serveurs actifs, dans un but écologique.

Ainsi, les principales étapes pour la réalisation de ce projet sont :

- la réalisation d'une analyse exhaustive des travaux de recherche actuels sur les architectures orientées service et virtualisées, appliquées aux jeux vidéo;
- la spécification et la validation partielle du modèle d'affaires décrivant les entités impliquées dans la virtualisation des plateformes de jeux vidéo ainsi que leurs relations/interactions. Cet objectif est atteint tout d'abord par la présentation d'une infrastructure virtualisée orientée services, puis par le développement d'un prototype de jeu réel, que l'on a soumis à un banc de test visant à mesurer la performance de notre approche sur plusieurs niveaux;
- la définition et la validation d'un algorithme de dimensionnement dynamique de ressources visant à maintenir une qualité de service des jeux vidéo hébergés;
- l'analyse de l'efficacité de l'algorithme proposé.

Enfin, nos deux contributions vont faire l'objet de publications. Notre première contribution, à savoir l'architecture de virtualisation de jeux vidéo orientée services, a déjà fait l'objet d'un article de conférence et demeure en attente d'acceptation, pour l'ICC 2014 qui aura lieu à Sidney. La seconde, relative à l'algorithme de dimensionnement dynamique de ressources prendra la forme d'un article de journal.

Organisation du rapport

Ce mémoire est organisé en cinq chapitres. Le premier présente un état de l'art des concepts théoriques, incluant le *cloud computing*, les technologies du jeu vidéo (incluant le *cloud gaming*) ainsi qu'une revue de littérature sur la gestion dynamique des ressources au sein des infrastructures infonuagiques. Le second chapitre traite de l'architecture orientée services pour l'hébergement de jeux vidéo que nous proposons et introduit la notion de *substrates* sous forme d'un nouveau modèle d'affaires (*business model*). Le troisième chapitre se penche sur la conception d'un prototype de jeu orienté *substrates*, qui sera comparé à travers plusieurs scénarios de tests de performances, à son homologue monolithique. Le quatrième chapitre décrit notre algorithme de gestion dynamique des ressources, permettant de garantir une qualité de service dans le cas de l'infrastructure présentée dans le chapitre deux. Le cinquième chapitre présentera les résultats de notre algorithme lorsqu'il est soumis à différents scénarios de tests, pour en étudier l'efficacité. Enfin, la conclusion et les travaux futurs reviendront sur les contributions de ce mémoire.

CHAPITRE 1

REVUE DE LITTERATURE

1.1 Présentation du *cloud computing*

1.1.1 Définition

Pour définir le paradigme du *cloud computing* (ou infonuage), Buyya *et al.* (2011 p.3) établissent une comparaison habile avec une prise électrique :

- on peut l'utiliser à la demande à n'importe quel moment et sans contrainte de durée;
- la puissance qu'elle délivre s'adapte aux appareils branchés et donc à nos besoins;
- on ne se soucie pas de la façon dont l'énergie qu'elle nous procure nous est acheminée ni de la façon dont elle est produite;
- on est facturé proportionnellement à notre consommation.

La prise électrique est donc un service et constitue une forme d'abstraction de l'infrastructure possédée par un fournisseur. Le principe du *cloud computing* est très similaire et vise à simplifier de la même manière l'accès à des ressources informatiques telles que de la puissance de calcul ou encore de la capacité de stockage. D'après Buyya *et al.* (2011, p.4), le *cloud computing* peut également être vu comme un réseau informatique distribué composé de serveurs virtualisés, dont les ressources peuvent être approvisionnées de manière dynamique et reposant sur un contrat de service entre un fournisseur et un client, le SLA (*Service Level Agreement*). Le mot *cloud* (nuage) quant à lui est une métaphore faisant référence à l'abstraction des ressources pouvant être utilisées par les particuliers, organisations et entreprises depuis n'importe quel endroit.

Buyya *et al.* (2011, p.4) fournit également d'autres définitions. Il cite un rapport de la compagnie McKinsey qui explique que le *cloud computing* est essentiellement un service basé sur des ressources informatiques (calcul, stockage et réseau) où la gestion du matériel est hautement abstraite pour le client et où les capacités de l'infrastructure sont très

élastiques. Au final, le client se retrouve essentiellement avec des dépenses d'exploitation (OPEX) par opposition aux dépenses d'investissement (APEX). Enfin, Buyya *et al.* (2011 p.4) citent comme référence la définition du NIST (*National Institute of Standards and Technology*) qui explique que le *cloud computing* est un modèle de type *pay-per-use* permettant d'accéder à la demande, via un réseau, à des ressources informatiques (réseau, serveurs, stockage, applications) pouvant être rapidement approvisionnées.

Pour résumer, nous retiendrons que le *cloud computing* offre :

- un service de ressources informatiques élastique et facturé à l'usage;
- l'illusion de ressources infinies faisant l'objet d'une abstraction vis-à-vis du client;
- une interface graphique (pour les utilisateurs ayant des compétences techniques modestes) ou une API (pour les développeurs), facilitant la gestion des services offerts aux clients;
- la garantie d'une qualité de service définie par un SLA.

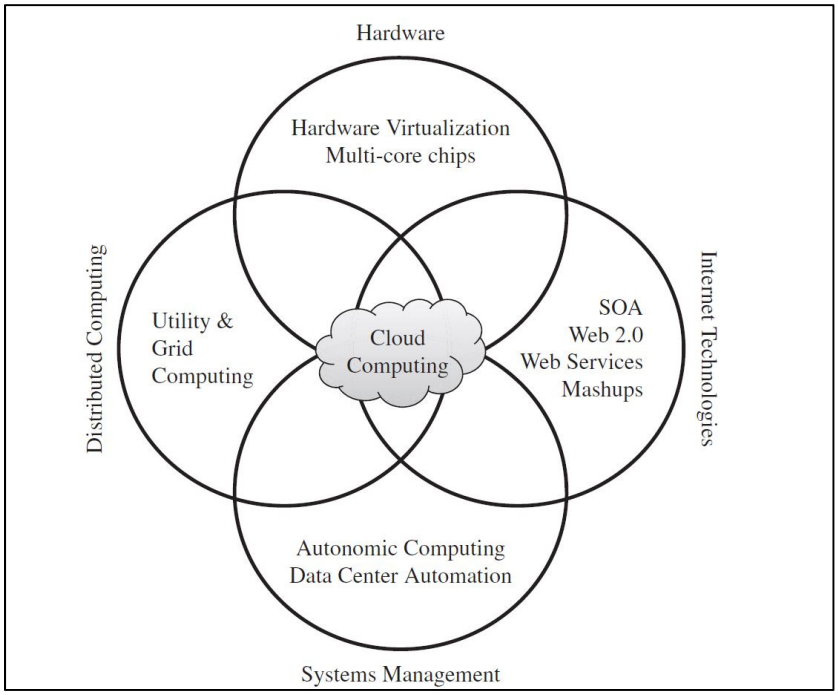


Figure 1.1 Technologies composant le *cloud computing*
Tirée de Buyya *et al.* (2011, p. 6)

D'autre part, comme le montre la figure 1.1, le *cloud computing* peut être vu historiquement comme une convergence de plusieurs technologies, à savoir :

- la virtualisation, décrite dans la partie 1.2;
- l'informatique distribuée, qui consiste à répartir le traitement d'une tâche sur un réseau de plusieurs machines physiques;
- la gestion automatisée des systèmes, qui définit des principes permettant à une architecture informatique de devenir autonome : celle-ci optimise automatiquement l'utilisation de ses propres ressources, détecte, évite et répare les pannes;
- les architectures orientées services, dont l'émergence a permis de faire communiquer plusieurs applications différentes par le biais d'un protocole commun. Le standard le plus en vogue actuellement est le critère de conception REST (*Representational state transfer*) qui se base sur le protocole HTTP, permettant de définir efficacement des interfaces de communications web.

1.1.2 Modèle en couches du *cloud computing*

Le *cloud computing* est défini par un modèle en couches comportant trois niveaux : le SaaS (*Software as a Service*), le PaaS (*Platform as a Service*) et l'IaaS (*Infrastructure as a Service*). La figure 1.2 décrit succinctement ce modèle en couches.

Le niveau SaaS correspond à une application (web souvent) utilisée par un client final à travers une interface graphique. La responsabilité du client se limite à la gestion des données entrées dans son application, tandis que le fournisseur de SaaS gère tout le reste (infrastructure, mises à jour de l'application, etc.). Dropbox (service de stockage et partage de fichiers en ligne), Microsoft Office 365 (suite bureautique en ligne), SAP (connue pour son progiciel de gestion intégré), Google Apps (incluant le service de messagerie Gmail et d'autres outils) sont des exemples de SaaS.

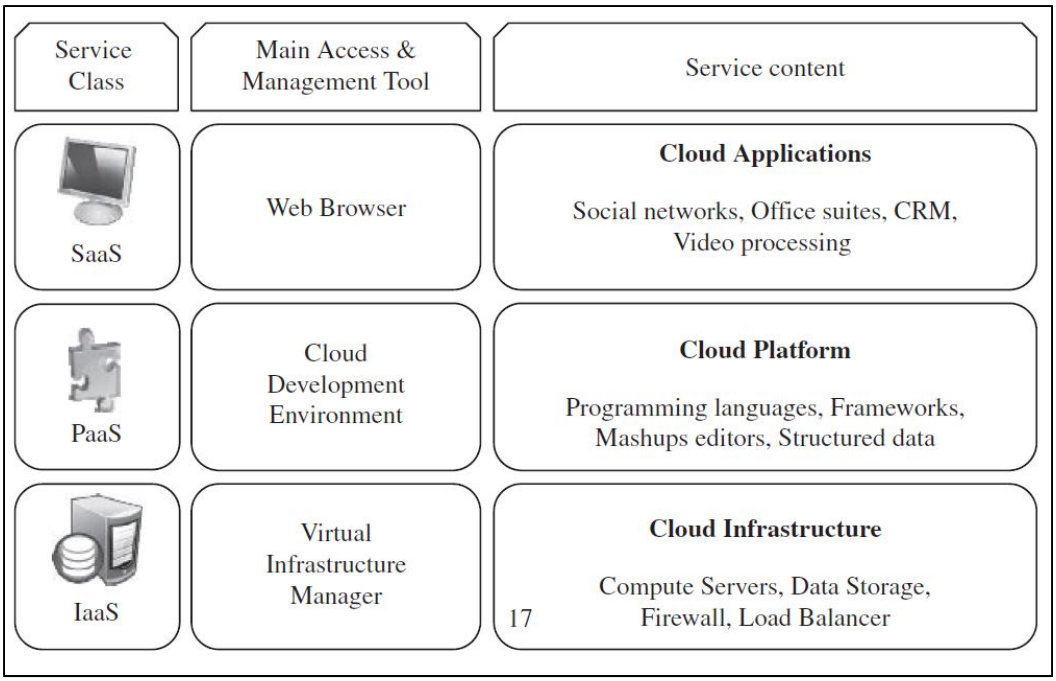


Figure 1.2 Modèle en couches du *cloud computing*
Tirée de Buyya *et al.* (2011)

Le niveau PaaS définit des interfaces de programmation (APIs) et offre un environnement de développement et de déploiement d'applications, permettant aux développeurs de s'affranchir des considérations de bas niveau. Le client est donc responsable de l'installation et de la gestion des applications qu'il déploie, tandis que le fournisseur de PaaS prend en charge la gestion de l'infrastructure, l'automatisation des ressources à allouer, les systèmes d'exploitation virtualisés ainsi que des programmes permettant la prise en charge du PaaS. Google App Engine (supportant le langage Python et Java) et Microsoft Azure (supportant certains langages de la famille .NET) sont des exemples de PaaS.

Enfin, à la base de ce modèle en couche, se trouve le niveau IaaS, qui offre généralement la possibilité de louer des machines virtuelles, que le client peut ajuster à ses besoins. Ce dernier est alors responsable de l'installation des systèmes d'exploitation et des applications, tandis que le fournisseur d'IaaS s'assure de la gestion des machines virtuelles, ainsi que du matériel sous-jacent. Amazon EC2, RackSpace Cloud et GoGrid sont des exemples de fournisseurs d'IaaS.

1.1.3 Différents types de *cloud computing*

Le *cloud computing* peut être déployé selon quatre modèles : public, privé, hybride et communautaire.

Dans le cas public, l'infrastructure est détenue par un organisme qui offre des services aussi bien aux particuliers qu'aux entreprises. Dans le cas privé, l'infrastructure est utilisée exclusivement par un organisme et peut être située dans ses locaux (*on premises*) ou ailleurs (*off premises*). Un *cloud* est dit communautaire lorsque celui-ci a été créé pour servir une cause, un but commun et peut être destiné à un ou plusieurs organismes. Enfin, un *cloud* hybride combine plusieurs types (public, privé et communautaire).

1.1.4 Avantages et inconvénients

D'après Sosinsky (2011), les avantages du *cloud computing* sont nombreux. Tout d'abord, la présence d'un service à la demande permet aux clients d'être autonomes et d'activer des machines virtuelles sans passer par un tiers. Ces derniers bénéficient également d'une élasticité qui, dans le cas des offres d'IaaS, par exemple, leur permet d'adapter la quantité des ressources allouées à leurs besoins. L'usage de chaque type de ressource (processeur, mémoire, espace disque, etc.) est d'ailleurs automatiquement mesuré séparément par le fournisseur de *cloud*, ce qui permet de facturer la consommation au juste prix.

Comme les infrastructures sont maintenues par des fournisseurs spécialisés qui améliorent continuellement leur efficacité et offrent leurs services à un nombre croissant de clients, les prix sont de plus en plus compétitifs. Parallèlement, les clients peuvent externaliser partiellement leur service informatique pour en assouplir la gestion. Ils s'affranchissent également d'une importante part des coûts liés à la gouvernance des technologies de l'information, en raison de l'absence d'infrastructure et d'employés pour les gérer.

Les offres de *cloud computing* sont toujours accompagnées d'une qualité de service définie par un SLA. Ces clauses sont plus facilement garanties par les fournisseurs grâce à leur

expérience et leur gestion avancée de la haute disponibilité, à travers les techniques de basculement (*failover*) en cas de panne et au balancement de charge (*load balancing*).

Dans le cas du SaaS, les clients bénéficient automatiquement des dernières mises à jour de leurs logiciels. De la même manière pour le PaaS, les environnements d'hébergement des applications sont automatiquement mis à jour.

Néanmoins, le *cloud* comporte aussi des inconvénients. De nombreuses entreprises ont envisagé la migration de leurs applications vers une infrastructure infonuagique, mais cette étape de transition est souvent périlleuse.

Dans le cas du SaaS, les applications sont certes conçues pour être personnalisées mais cela n'atteint pas le degré de personnalisation des logiciels développés sur mesure pour les besoins spécifiques d'une entreprise. Par exemple, dans le cas d'un progiciel de gestion intégré (ou ERP en anglais) offert en tant que SaaS, si celui-ci n'inclut pas une fonctionnalité requise pour une entreprise, seul le fournisseur de SaaS peut accepter de l'intégrer, s'il en a la possibilité. Cela est d'autant plus difficile avec des logiciels propriétaires lorsqu'ils ne sont pas conçus de façon modulaire.

Toutefois, comme le souligne Buyya *et al.* (2011), le défaut majeur du *cloud* est incontestablement la sécurité des informations hébergées. Certes, les clients ont un contrat qui définit des règles de confidentialité avec les fournisseurs, mais cela n'est pas suffisant. C'est d'ailleurs une des raisons qui a poussé la création de *cloud* privés. Ce problème est d'autant plus vrai lorsque les informations sont réparties sur plusieurs centres de traitement de données (*datacenters*) localisés dans des pays différents. Chaque gouvernement a des lois différentes vis-à-vis du respect de la confidentialité et cela peut être problématique pour les entreprises possédant des données privées.

1.2 Introduction à la virtualisation

Comme nous l'avons vu dans les sections précédentes, la virtualisation est un des piliers ayant permis l'avènement du *cloud computing*. Dans ce chapitre nous présenterons cette technologie en nous appuyant sur l'ouvrage de Von Hagen (2008).

1.2.1 Définition

La virtualisation offre la possibilité d'exécuter des applications ou des machines (hébergeant des systèmes d'exploitation) dans des conteneurs logiques, tout en offrant une abstraction des architectures et ressources physiques sous-jacentes. Depuis quelques années, la montée en puissance des composants informatiques et notamment la démocratisation des processeurs multi-cœurs ont été les principaux facteurs ayant donné un intérêt à la virtualisation.

Il existe plusieurs types de virtualisation. La virtualisation d'applications désigne la compilation d'un code vers un autre code indépendant du système exploitation. Généralement qualifié de *byte code*, celui-ci est proche d'un code machine nécessitant un interpréteur. C'est le cas de l'implémentation actuelle du langage Java qui fait appel à une machine virtuelle. L'intérêt principal est de pouvoir exécuter une application sur n'importe quel environnement sans recompilation nécessaire : les phases de déploiement sont alors simplifiées et la portabilité du code est assurée.

D'autre part, il y a la virtualisation des machines également appelée virtualisation de serveurs. C'est celle qui nous intéresse le plus puisque c'est la technologie maîtresse permettant la concrétisation de la couche IaaS du modèle infonuagique. Elle correspond à l'exécution d'une ou plusieurs machines virtuelles (dites invitées), chacune ayant son propre système d'exploitation, sur un autre système d'exploitation parent (dit hôte). Les machines virtuelles invitées sont isolées et utilisent, selon la politique d'accès et l'allocation fournie par le parent, des ressources virtualisées. Le système d'exploitation parent prend souvent la forme d'une petite machine virtuelle exécutée directement au-dessus des ressources physiques, que l'on appelle l'hyperviseur. Son objectif est d'orchestrer les requêtes entre les

machines virtuelles invitées et le matériel. Enfin, il existe d'autres formes de virtualisation (telles que les réseaux virtualisés) mais cela va au-delà du contexte de ce mémoire.

1.2.2 Techniques de virtualisation de machines

Les hyperviseurs sont classés en deux catégories principales :

- type un ou *bare metal*, ils s'éxecutent directement sur le matériel;
- type deux ou hébergés, ils fonctionnent par-dessus un système d'exploitation conventionnel. C'est le cas par exemple des hyperviseurs VMWare Workstation, Oracle VirtualBox et Microsoft Hyper-V (ce dernier étant également disponible en type un).

Parmi les hyperviseurs de type un, il y a :

- ceux qui offrent une virtualisation dite complète (*full virtualisation*). En émulant si nécessaire certaines fonctionnalités du matériel sous-jacent, ils permettent ainsi de virtualiser des systèmes d'exploitations non modifiés. On retrouve cette approche dans l'hyperviseur VMWare ESXi;
- ceux qui offrent une paravirtualisation, très similaire à la virtualisation complète, à la différence qu'elle nécessite un système d'exploitation invité modifié au niveau du noyau. Ces modifications lui permettent de communiquer directement avec l'hyperviseur et de s'affranchir de certains coûts de virtualisation (également appelé *overheads*). Cette approche peut donc offrir de meilleures performances en termes de temps de traitement. Cette technique est utilisée par l'hyperviseur Xen.

D'autre part, il existe un type de virtualisation de machines au niveau du noyau, ne nécessitant pas d'hyperviseur : c'est le cas de KVM (*Kernel Virtual Machine*) et d'UML (*User Mode Linux*). Cette fois-ci, on exécute une version spécifique du noyau du système d'exploitation associé à une machine virtuelle, en tant que processus dans le domaine utilisateur.

Enfin tous ces types de virtualisation de serveurs peuvent tirer profit d'une virtualisation dite matérielle, intégrée notamment dans les processeurs Intel (via les extensions de la technologie VT) ou AMD (via la technologie AMD-V). Elle offre des accès privilégiés et protégés entre une machine virtuelle et le matériel. Elle permet également de limiter la dégradation des performances liée aux coûts de virtualisation.

1.2.3 Avantages

Avec la montée en puissance des processeurs d'aujourd'hui, mais aussi en fonction des tâches qu'elles accomplissent, les infrastructures informatiques non virtualisées peuvent souffrir de sur-approvisionnement (ou sous-utilisation). En effet, d'après Beloglazov *et al.* (2012), la plupart du temps, les serveurs sont utilisés de 10 à 50% de leur capacité. D'autre part, un serveur en marche mais inoccupé consomme en moyenne 70% de sa puissance maximale. La virtualisation est la solution idéale à ce problème puisqu'elle permet de mutualiser les ressources en réalisant ce qu'on appelle de la consolidation : une même machine physique dotée d'un hyperviseur accueille plusieurs machines virtuelles. Ainsi, les machines physiques allumées peuvent être minimisées et demeurent pleinement exploitées. La consolidation contribue à réduire la consommation électrique et donc les dépenses pour les fournisseurs de *cloud*.

D'autre part, les environnements virtualisés desservis par un réseau local offrent une technique extrêmement utile nommée migration, permettant de déplacer une machine virtuelle d'une machine physique vers une autre. De nombreux progrès ces dernières années ont permis d'effectuer ces migrations en quelque secondes, avec une interruption temporaire de la machine virtualisée allant de quelques millisecondes à quelques secondes. En général, le contenu de l'image du système invité (pouvant être très volumineux) n'est pas copié : s'il s'agit d'un disque virtuel existant sur un espace de stockage partagé en réseau (SAN ou *Storage Area Network*), celui-ci est démonté de la machine physique source puis remonté sur la destination. Durant la migration, le contenu de la mémoire vive est également déplacé de la source vers la destination. Selon la tâche accomplie, l'état des programmes en cours

d'exécution évolue à une certaine fréquence et cause à travers des écritures dans la mémoire vive des modifications fréquentes sur certaines zones, tandis que d'autres demeurent intactes. De ce fait une brève mise en pause de la machine est nécessaire pour copier définitivement les zones modifiées, qualifiées de *dirty pages*, avant de résumer l'activité de la machine virtuelle.

La migration prend tout son sens au niveau IaaS, que ce soit pour effectuer de la consolidation ou pour allouer des ressources à une machine virtuelle ayant atteint la limite fixée par son hôte.

La virtualisation offre également de nombreuses solutions aux exigences du *cloud computing* : simplification de la gestion des ressources et de la supervision, élasticité, support facilité pour les vieilles applications nécessitant certaines architectures matérielles (pouvant être émulées) ou systèmes d'exploitation dépassés, simplification du déploiement à la demande de nouvelles machines, etc.

1.2.4 Inconvénients

Malheureusement, la virtualisation n'est pas exempte de défauts. L'un des principaux est lié à la consolidation de serveurs : malgré l'isolation promise, plus on augmente la charge sur une machine physique, plus on augmente ses chances de tomber en panne. Ainsi, plusieurs machines virtuelles hébergées sont concernées si une machine physique cède. Il existe plusieurs solutions à ce problème : utiliser un système de supervision pouvant prendre des décisions basées sur des mesures de taux d'utilisation et répliquer les machines virtuelles sur plusieurs machines physiques. On parle alors de redondance.

Enfin, notamment dans le cadre de ce mémoire, la virtualisation d'un système d'exploitation pour héberger un certain type d'applications peut occasionner une dégradation des performances, comparé au cas natif, c'est-à-dire sans virtualisation. C'est notamment le cas des processeurs graphiques dont le support par les hyperviseurs varie fortement. Par exemple

avec Workstation 9.0, VMWare a récemment réussi à améliorer cette prise en charge, tandis que Virtualbox d'Oracle peine à l'utiliser efficacement.

1.3 Introduction technique aux jeux vidéo

Gregory (2009, p. 9) définit rigoureusement un jeu vidéo comme une simulation :

- interactive, car elle nécessite l'intervention du joueur;
- temps réel, en raison des contraintes de temps définies selon les règles du jeu. Le plus souvent, l'état du jeu -dépendant d'un temps virtuel- devra donner l'illusion d'évoluer au même rythme que notre temps réel;
- basée sur des agents, car de nombreuses entités (ou agents), comme par exemple les personnages virtuels, peuvent être amenées à interagir entre elles, ou avec le joueur.

Dans cette partie nous allons passer en revue les différentes catégories de jeux vidéo, puis nous introduirons leur principe de fonctionnement.

1.3.1 Les différentes catégories

Depuis 1970, avec la naissance des jeux vidéo sur les bornes d'arcade, puis leur apparition sur consoles (de salon ou portables) sans oublier leur distribution sur ordinateurs, les catégories de jeux vidéo se sont énormément diversifiées aussi bien grâce à la créativité des concepteurs de jeux vidéo qu'aux progrès techniques (puissance des cartes graphiques, processeurs, contrôleurs, Internet haut-débit, etc.). Mais comment catégorise-t-on les jeux vidéo? La classification des jeux vidéo est faite selon plusieurs critères et fait essentiellement appel au bon sens. Un jeu vidéo peut en fait appartenir à une ou plusieurs catégories simultanément, et il n'est pas rare de trouver de nombreux jeux hybrides. On distingue toutefois plusieurs modes de classification que l'on peut combiner. En voici une liste non exhaustive :

- selon le thème, les émotions, le type d'action et les objectifs que le jeu cherche à susciter chez le joueur : jeux d'aventure, de stratégie, de course automobile, d'épouvante/horreur, de rôle (RPG ou *Role Playing Game*), de tir à la première personne (FPS ou *First Person Shooter*), de réflexion (énigmes, puzzles, etc.), jeux éducatifs, jeux sérieux, simulateurs professionnels (pour l'armée), etc;

- selon les contrôleurs utilisés : plutôt que d'utiliser une souris, un clavier ou une manette, un jeu peut faire appel à des contrôleurs plus spécifiques qui améliorent l'ergonomie et l'expérience du joueur, comme les simulateurs de vols professionnels. Il en est de même pour la réalité augmentée (usage de caméras, capteurs), le jeu sur terrain réel avec téléphone intelligent doté d'un GPS, etc. De nos jours, les contrôleurs évolués se sont considérablement démocratisés. C'est le cas par exemple de ceux dont le but est de susciter le mouvement physique chez le joueur, comme la Wiimote de Nintendo utilisant accéléromètres et capteurs gyroscopiques pour la retranscription de mouvements, le Wiifit pour faire du sport chez soi, ou encore le Kinect de Microsoft pour la détection et la reconnaissance de mouvement et d'image avec caméra. Cela permet de créer une interaction plus intéressante et novatrice avec le monde virtuel;

- nombre de joueurs dans une partie : En général, la plupart des jeux multijoueurs se jouent entre 2 et 64 joueurs, tandis que les MMOGs (*Massively Multiplayer Online Games*) peuvent héberger plusieurs milliers de personnes simultanément. On parle de MMORPG pour désigner un RPG (*Role Playing Game*) massivement multi-joueurs et MMOFPS pour désigner un jeu de tir à la première personne massivement multi-joueurs.

Le but de ce bref aperçu est donc avant tout de montrer que les jeux vidéo peuvent appartenir simultanément à plusieurs catégories.

1.3.2 Principe de fonctionnement

1.3.2.1 Une organisation modulaire

Depuis quelques années, les jeux vidéo sont devenus des programmes si complexes qu'ils nécessitent d'être divisés en modules, eux-mêmes regroupant des sous-modules. Cette séparation est tout d'abord cruciale pour les programmeurs afin de conserver un code compréhensible et clair. Cela permet durant les phases de conception puis de développement d'ajouter de nouvelles fonctionnalités ou de corriger les *bugs* qui apparaissent. D'un autre côté, cette approche modulaire est indispensable pour pouvoir réutiliser le code d'un jeu à un autre et éviter de réinventer la roue. L'industrie du jeu vidéo connaît une concurrence féroce et cela lui permet de minimiser la durée des cycles de développement.

Comme l'explique Gregory (2009), dans un jeu vidéo rigoureusement programmé et à l'architecture bien organisée, on peut séparer le code en deux parties :

- le code spécifique au jeu, qui comme son nom l'indique permet de programmer la logique du jeu, c'est-à-dire de déterminer le comportement de chaque composant du jeu. Ce code n'est pas réutilisable et fait qu'un jeu diffère d'un autre via ses règles et son contenu. Plus récemment, ce code tend à être remplacé par une base de données contenant les propriétés des objets du jeu (personnages, véhicules, etc.), le scénario, etc. Il s'agit du paradigme des *data-driven game engines*;

- le code du moteur de jeu, qui est une sorte de pilier pour le précèdent : il permet de gérer des tâches génériques comme le rendu graphique, la gestion du son, l'animation des modèles en trois dimensions, la simulation physique des objets et la détection des collisions, la communication avec les réseaux, etc. Ce code est hautement réutilisable d'un jeu à l'autre.

En dehors du code, un jeu vidéo contient également d'autres éléments rarement réutilisables nommés *assets*. Il s'agit des sons, des modèles en trois dimensions, des images, des

cinématiques, etc. Les *assets* sont majoritairement créés par des artistes, et non des personnes au profil technique.

L'annexe I donne un aperçu assez complet mais non exhaustif des modules qui peuvent être contenus dans un moteur de jeu. On s'aperçoit évidemment que les jeux vidéo actuels sont des systèmes extrêmement sophistiqués. Cela est d'autant plus vrai lorsque l'on s'intéresse à l'interaction entre ces modules. Toutefois, dans cette partie, on se contentera de définir succinctement quelques modules importants :

- le moteur de rendu ou *rendering engine* (détaillé dans la partie 1.3.3) : Probablement le plus complexe et le plus imposant, le moteur de rendu permet de générer une image à partir de la description d'une scène virtuelle. La plupart des modèles en 3D (arbres, personnages, véhicules, etc.) sont des fichiers de données structurées qui définissent un maillage de points connectés par des arrêtes. Sur ces modèles, on peut appliquer des textures (images, pour simuler par exemple la couleur de la peau, du métal etc.). Le *rendering engine* permet de calculer en outre les effets liés à l'éclairage, l'ombrage et peut appliquer des effets spéciaux (flou de l'objectif, etc.);

- le moteur physique ou *physics engine* (détaillé dans la partie 1.3.4) : D'un côté, il permet de détecter les collisions entre les objets. Cela est indispensable pour empêcher par exemple qu'une voiture ne s'enfonce dans le sol, ou encore interdire à un personnage de passer à travers un mur. D'un autre côté, il permet de simuler de manière réaliste le comportement d'objets soumis à des lois physiques, incluant forces et contraintes;

- le module d'animation : Il permet de modifier la forme d'un modèle en 3D. C'est souvent le cas de l'animation squeletale d'un être vivant, la déformation de la tôle d'une voiture suite à une collision, l'animation d'un visage pour exprimer différentes émotions (tristesse, sourire, colère, etc.). Ce module est souvent intimement lié au moteur physique, car tous deux ont une influence sur la cinétique et la morphologie des objets;

- le gestionnaire de contrôleurs : il permet de capturer les commandes saisies par un utilisateur et les traduit en évènements dans le but d'être interprétés par la logique du jeu. Il coordonne ainsi les évènements générés par les mouvements de la souris, la pression des boutons du clavier, etc;

- le module de son : il permet de coordonner et d'activer l'ensemble des sons d'un jeu. Il gère les effets spéciaux, les musiques, les voix, etc;

- le moteur réseau : indispensable dans le cas des jeux multi-joueurs, c'est une couche d'abstraction qui gère la réplication des objets partagés et la compensation de la latence.

1.3.2.2 La boucle principale

Comme l'indique Gregory (2009), la boucle principale est le cœur d'un jeu vidéo. Elle coordonne l'exécution des modules et permet de rafraîchir l'état du jeu, lui permettant ainsi de s'exécuter en temps réel. Dans l'industrie du jeu vidéo, on l'appelle également *main loop* ou *game loop*.

Une boucle principale simple peut se résumer aux actions suivantes :

1) capturer les commandes du joueur (touches appuyées, etc.);
2) exécuter la logique du jeu pour mettre à jour son état : mettre à jour les positions des objets, exécuter l'intelligence artificielle, gérer les collisions via le moteur physique, etc.;
3) redessiner la scène à l'écran via le moteur de rendu;
4) répéter tant que le jeu n'est pas terminé.

D'autres exemples plus sophistiqués de boucles de jeu sont proposés dans l'annexe II. Enfin, tout comme un dessin animé donne l'illusion d'une animation grâce à une succession d'images, pour que l'animation d'un jeu paraisse fluide, le nombre minimum d'images par secondes (ou frames par seconde) est de 30. Cette fréquence minimum de 30 Hz correspond à une période maximum de 1/30 = 33.33 ms. De ce fait, le temps d'une itération de la boucle

principale, c'est-à-dire la durée pour effectuer les trois premières étapes, ne doit pas dépasser 33.33 ms. Si cette durée est dépassée, l'animation paraîtra saccadée, le jeu aura l'air de fonctionner au ralenti et l'expérience utilisateur sera décevante.

1.3.3 Le moteur de rendu

Le moteur de rendu est le module le plus important dans un jeu vidéo. Il permet de générer une image à partir de la description d'une scène virtuelle en trois dimensions.

Comme l'explique Gregory (2009, p. 400), la composition d'une scène virtuelle en trois dimensions peut se résumer à :

- des surfaces en trois dimensions, représentées sous une certaine forme mathématique et possédant des propriétés de base comme les vecteurs de position, de rotation et d'échelle;
- une caméra virtuelle, qui jouera le rôle de point de vue de la scène;
- des sources de lumières, pour éclairer la scène, sans quoi les surfaces filmées par la caméra virtuelle ne seraient pas visibles;
- des matériaux, associés aux surfaces pour leur conférer un aspect particulier.

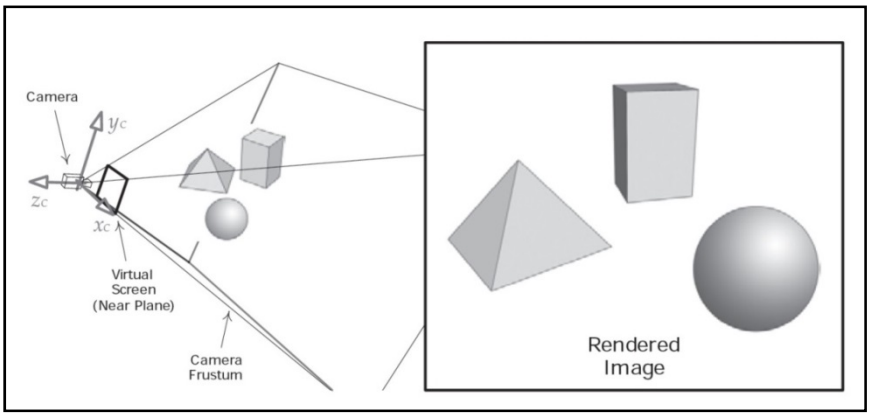


Figure 1.3 Exemple d'une scène en trois dimensions
Tirée de Gregory (2009, p. 401)

L'objectif final du moteur de rendu sera de produire une image de cette scène, donc de calculer la couleur de chaque pixel, comme le montre la figure 1.3.

1.3.4 Le moteur physique

Un moteur physique dans un jeu vidéo permet de simuler de manière réaliste (ou approchée) le comportement d'objets soumis aux lois de la mécanique. Il inclut également un gestionnaire de collisions, qui permet de détecter des contacts, puis de prendre les décisions nécessaires (on parle de résolution). Sans gestion des collisions, on ne pourrait empêcher des personnages de passer à travers les murs, éviter qu'une voiture ne s'enfonce dans le sol, etc. Le moteur physique est donc indispensable pour éviter des comportements irréalistes.

Depuis quelques années, Gregory (2009) indique que les moteurs physiques permettent de gérer une sous-branche de la mécanique nommée *rigid body dynamics*. Celle-ci repose principalement sur :

- le fait que les *rigid bodies* soient, comme leur nom l'indique, des corps indéformables;
- la seconde loi de Newton qui stipule que la force est proportionnelle à l'accélération et la masse, selon la formule :

$$\boldsymbol{F}(t) = m\boldsymbol{a}(t) = m\frac{d^2\boldsymbol{x}(t)}{dt^2} \tag{1.1}$$

Gregory (2009, p. 596) fournit une liste pertinente mais non exhaustive de tâches qu'un moteur physique peut accomplir :

- détecter les collisions entre les objets dynamiques (mobiles) et le monde statique;
- simuler le mouvement des corps soumis à la gravité et aux autres forces;
- simuler des systèmes masse-ressort;
- simuler des environnements et objets destructibles. Exemples : bâtiments après l'impact d'un projectile, tôle d'une voiture après un crash, etc.;
- permettre à des personnages virtuels de soulever de manière réaliste des objets avec leurs mains. En général, cette tâche est très rarement utilisée dans les jeux vidéo à cause de sa complexité, mais demeure intéressante à l'avenir;
- simuler de manière réaliste le comportement d'un véhicule ayant des suspensions et dont les pneus sont soumis (entre autres) à des forces de friction;

- simuler la chute des poupées de chiffons (*rag dolls*). C'est le cas par exemple d'un personnage qui tombe de manière réaliste après avoir été assommé;
- simuler les tissus légers, comme par exemple les habits d'un personnage, le mouvement d'un drapeau soumis à son propre poids et à la force du vent;
- simuler la mécanique des fluides, comme la propagation de la fumée, l'occupation d'un liquide versé dans un récipient, etc.;
- et bien d'autres encore.

1.3.5 Jeux en ligne multi-joueurs

Contrairement aux jeux dits solo (*single player*), les jeux multi-joueurs font participer plusieurs personnes simultanément. Lorsque cette participation est faite à travers un réseau étendu tel Internet, on parle alors de jeu en ligne multi-joueurs (*online multiplayer game*).

Techniquement parlant, le défi est de maintenir pour chaque joueur une vue consistante du monde virtuel dans lequel il évolue. Les différents objets et actions doivent être synchronisés pour que le jeu demeure cohérent et équitable.

1.3.5.1 Architecture

Steed *et al.* (2009, p. 138) détaille les différentes approches utilisées par les moteurs de jeu pour interconnecter plusieurs joueurs en même temps. Nous ne retiendrons que les plus populaires : l'architecture pair à pair avec maître et l'architecture client-serveur.

L'architecture pair à pair avec maître, comme le montre la figure 1.4, est une extension du mode pair à pair simple. Dans cette approche distribuée, chaque hôte a pour responsabilité d'envoyer les mises à jour de son propre état aux autres. A travers la désignation –arbitraire– d'un maître, l'avantage de l'approche pair à pair avec maître est de permettre aux hôtes de rejoindre une partie en ne connaissant que les coordonnées du maître. Généralement, il s'agit

de la paire adresse IP et port. Dès qu'un hôte rejoint une partie, le maître lui fournit les coordonnées des participants déjà présents afin d'établir les communications nécessaires.

Steed *et al.* (2009, p. 131) explique que dans une architecture pair à pair, on distingue deux catégories d'objets dynamiques, c'est-à-dire ceux dont les propriétés peuvent être modifiées durant la partie comme les personnages, véhicules, armes, etc. :

- les objets dits locaux sont ceux sur lesquels un hôte a une autorité. Par exemple, dans un jeu de voitures, un hôte a autorité sur la position de son propre véhicule;
- les objets dits distants sont des objets sur lesquels un hôte donné n'a pas autorité. Par exemple, dans un jeu de voitures, un hôte n'a pas autorité sur la position des véhicules des concurrents.

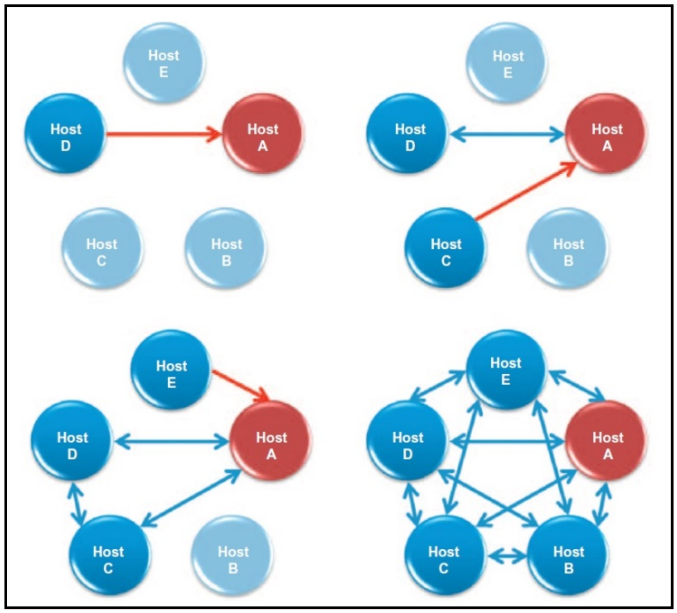


Figure 1.4 Architecture pair à pair avec maître
Tirée de Steed *et al.* (2009)

Le problème avec l'approche pair à pair est que son implémentation est beaucoup plus complexe lorsqu'il s'agit de maintenir une vue consistante du jeu pour chaque participant. En effet, il arrive souvent que des conflits soient générés par les actions effectuées (ex : collision

d'une voiture avec une autre), et leur résolution n'est pas triviale avec une architecture décentralisée.

L'architecture client-serveur, comme le montre la figure 1.5, est une approche centralisée. Le serveur est la seule entité à avoir autorité sur tous les objets du jeu. Les clients ne possèdent que des copies. Lorsqu'un client effectue une action, il la transmet immédiatement au serveur et simule localement une prédiction du résultat sur l'état actuel de sa propre copie. Le serveur effectue sa propre simulation en tenant compte de l'ensemble des actions effectuées par les hôtes, puis leur renvoie un résultat cohérent. Les clients mettent alors à jour leur prédiction en fonction de la réponse du serveur, qui est prioritaire. Pour résumer, les clients ne donnent pas un ordre lorsqu'ils effectuent une action, ils demandent l'autorisation d'en effectuer une et c'est le serveur qui décide des conséquences de ces actions.

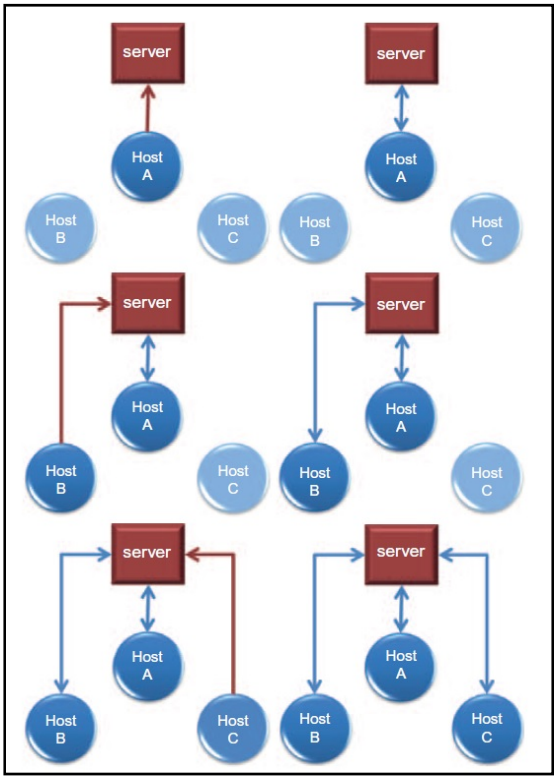


Figure 1.5 Architecture client-serveur
Tirée de Steed *et al.* (2009)

L'approche client-serveur facilite énormément la mise en place d'une synchronisation de l'état du jeu : elle permet une meilleure consistance, puisque les conflits potentiels générés par les actions des participants sont résolus par le serveur. Elle offre également une solution efficace à la triche puisque le serveur peut choisir de transmettre ou non les positions des autres joueurs. C'est par exemple le cas dans un jeu de guerre, où l'espionnage fournit un avantage stratégique majeur.

1.3.5.2 Problèmes associés à la latence

La latence liée au réseau est un problème majeur dans les jeux vidéo multijoueurs. Steed *et al.* (2009, p. 329), indique qu'un jeu en ligne demeure jouable dans de bonnes conditions lorsque la latence client-serveur est inférieure à 150 ms. Les jeux les plus touchés par l'augmentation sont ceux qui impliquent des actions dont la réalisation doit être immédiate pour conserver une certaine précision : jeux de tir à la première personne, jeux de voitures, etc. A l'inverse, un jeu au tour par tour comme les échecs peut tolérer une latence très élevée. Un des cas, surnommé « problème du joueur à l'épreuve du feu » est détaillé par Steed *et al.* (2009, p. 358) à travers la figure 1.6:

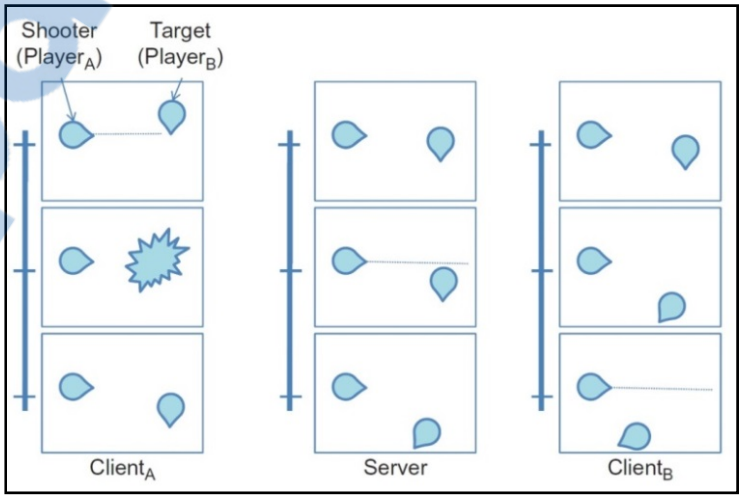


Figure 1.6 Problème du joueur à l'épreuve du feu
Tirée de Steed *et al.* (2009)

Dans cette figure, chaque colonne représente un point de vue différent et illustre trois images successives. Au départ, on constate que du point de vue du client A, celui-ci tire et réussi à détruire B d'après sa propre simulation locale. Mais à cause de la latence induite par le réseau, l'action du tir arrive en retard au serveur. Comme ce dernier est autoritaire sur l'état de tous les objets, il répond au client A en indiquant que B s'est déplacé (B semble avoir évité le tir de justesse). A rectifie donc le résultat de sa simulation en ressuscitant B à la position reçue par le serveur. Ce cas de figure est évidemment frustrant pour le joueur A puisque le jeu n'est ni cohérent ni équitable. Ce cas est considéré comme une dégradation de la consistance du jeu due à la latence.

1.4 Présentation du *cloud gaming*

1.4.1 Principe

A l'origine, la distribution des jeux vidéo se faisait à travers un support de stockage physique (cartouche, disques, etc.) vendu en magasin. Avec la démocratisation d'Internet et des connexions à hauts débits, la distribution électronique permettant de télécharger des jeux a connu une grande effervescence. Mais depuis quelques années, dans le même esprit que la vidéo à la demande, le *cloud gaming* est apparu. Ce paradigme de jeu à la demande se caractérise par une approche technique et un modèle d'affaires radicalement différents.

Lorsqu'un utilisateur final souscrit à une offre de *cloud gaming* et souhaite jouer, celui-ci n'installe pas les binaires et autres fichiers du jeu sur son terminal (ordinateur, console, tablette, *smartphone*, etc.). En effet, il télécharge uniquement un client logiciel léger. Celui-ci va se connecter aux serveurs du fournisseur et créer une session de jeu. Durant la phase de service, les commandes (c'est-à-dire les touches appuyées par le joueur) sont envoyées via Internet vers les serveurs du fournisseur, hébergeant le jeu. Ces actions sont interprétées et tout le traitement est effectué à distance. En retour, ces serveurs transmettent un flux vidéo compressé contenant le résultat visuel et sonore du jeu. La figure 1.7 synthétise ce processus.

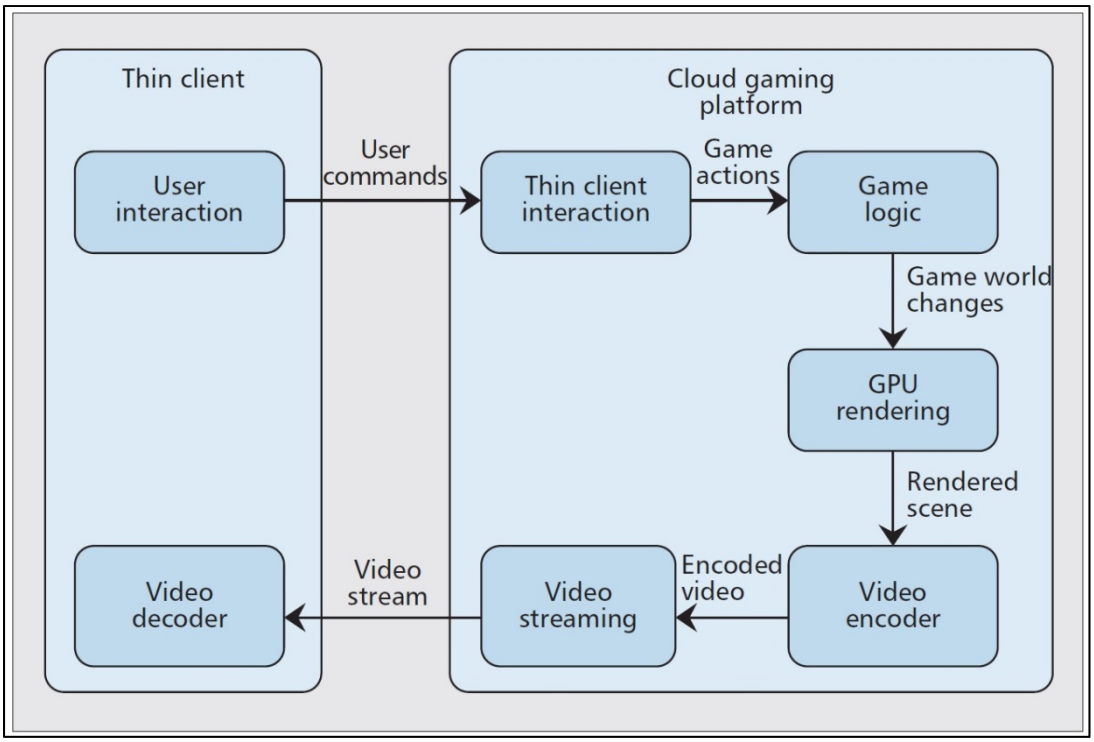


Figure 1.7 Principe de fonctionnement du *cloud gaming*
Tirée de Shea *et al.* (2013)

De nos jours, les fournisseurs actuels sont G-Cluster, OnLive, Gaikai (racheté par Sony en 2012) et Kalydo.

1.4.2 Avantages et inconvénients

Le principal avantage pour les compagnies d'édition et de développement de jeux vidéo est l'abolition du piratage. Puisque les jeux ne sont plus fournis et installés chez le client final, ceux-ci n'ont aucun risque d'être partagés de manière illégale. Actuellement, différentes mesures existent pour limiter le piratage des jeux vidéo : numéros de série, mécanismes de vérification d'authenticité, DRM ou *Digital Rights Management*, etc. Celles-ci sont toutefois inefficaces puisque de nombreux pirates (qualifiés de *crackers*) prennent plaisir à faire sauter ces protections et fournissent leurs solutions au grand public.

Du côté des utilisateurs finaux, l'un des avantages majeurs est la possibilité de jouer à des jeux exigeants sur des terminaux peu puissants comme les ordinateurs d'entrée de gamme ou les terminaux portables *low profile* à faible consommation. Contrairement aux joueurs sur console de jeux, dont les capacités de traitement sont fixées pour chaque modèle, les joueurs sur ordinateurs doivent mettre à jour leur configuration en modifiant processeur, carte graphique, quantité de mémoire vive afin de pouvoir jouer aux jeux vidéo les plus récents (et souvent les plus exigeants) dans des conditions optimales. Le *cloud gaming* est une solution intéressante puisque ce problème de mise à jour du matériel revient désormais au fournisseur et peut être optimisé.

Du côté des inconvénients cette fois-ci, le problème majeur du *cloud gaming* est la latence induite par l'envoi des commandes et la réception du flux vidéo. Plus concrètement, lorsqu'un joueur effectue une action (exemple : tourner à droite dans un jeu de voiture), sa réalisation visuelle doit être la plus immédiate possible. Sinon, ce retard perceptible au-delà de 200 ms d'après Shea *et al.* (2013) devient désagréable et nuit à l'expérience utilisateur. Pour pallier à ce problème, les fournisseurs de *cloud gaming* tentent d'avoir des *datacenters* géographiquement proches de chaque zone contenant les joueurs, afin de réduire au maximum les délais de communication.

Concernant les utilisateurs finaux, ceux-ci doivent en permanence être connectés à Internet pour pouvoir jouer. D'autre part, alors que le marché de l'occasion existait à travers la revente des jeux acquis sur un support physique (CD, DVD, etc.), le modèle d'affaires du *cloud gaming* ne le permet plus.

Enfin, il demeure un aspect écologique qui mérite d'être étudié. Aucune étude à ce jour ne s'est penchée sur cette problématique, mais il serait intéressant de comparer le coût énergétique d'utilisation des jeux dans l'état actuel (donc installés sur les terminaux) et dans les conditions du *cloud gaming* (nécessitant une connexion permanente, des serveurs de traitement et des terminaux légers).

1.4.3 Travaux antérieurs liés au *cloud gaming*

A ce jour, peu de travaux se sont penchés sur l'amélioration de l'architecture du *cloud gaming*. En effet, la plupart se concentrent sur l'optimisation de la qualité de service.

Barboza *et al*. (2010) présente une approche très simple permettant à un fournisseur de *cloud gaming* d'offrir un service de jeux vidéo à la demande. Il détaille ainsi l'architecture infonuagique qui permet d'automatiser l'approvisionnement de machines virtuelles, le chargement et l'exécution d'un jeu lorsqu'un utilisateur final se connecte et souhaite commencer une partie.

D'autre part, les travaux de Zhao *et al.* (2012) sont probablement les plus intéressants et se rapprochent le plus des objectifs de ce mémoire. Dans leur article, ils commencent d'abord par classer les terminaux des utilisateurs finaux selon trois catégories :

- catégorie 1 : le client possède un faible CPU et un faible GPU;
- catégorie 2 : le client possède un bon CPU mais un faible GPU;
- catégorie 3 : le client possède un faible CPU mais un bon GPU.

Selon les catégories, on aura différents cas de figure à traiter côté serveur :

- *remote-update* : on envoie au client le résultat de la simulation physique, de l'intelligence artificielle et de l'animation. C'est le cas de la catégorie 3;
- *remote-render* : on envoie le résultat du traitement graphique (image du jeu). C'est le cas de la catégorie 2;
- la combinaison des deux : *remote-update* & *remote-render*, qui est le cas de la catégorie 1.

Côté architecture, l'article de Zhao *et al.* (2012) explique qu'actuellement le *cloud gaming* suit le modèle d'une machine virtuelle pour chaque utilisateur final. Il innove en expliquant qu'une approche faisant appel à des modules de moteurs de jeu virtualisés et distribués sur plusieurs machines est une approche plus fine, permettant d'équilibrer l'usage des ressources.

Les performances de leurs prototypes indiquent même une amélioration dans la qualité de service grâce à une réduction des temps de traitement. Les auteurs n'ont malheureusement pas expliqué d'où venait cette amélioration et se sont contentés d'en faire le constat dans leur article. Mais d'après leur architecture, l'hypothèse que l'on peut émettre concernant cette efficacité est que chaque module, lorsqu'exécuté en parallèle avec les autres, peut dans le meilleur des cas (mais pas forcément le plus écologique) être hébergé sur une machine physique ne contenant qu'une machine virtuelle. En d'autres termes, on se retrouve avec un jeu exécuté sur au moins deux machines physiques. Si l'*overhead* de virtualisation et de communication (via le réseau local) entre les modules s'avère négligeable, les performances sont indéniablement meilleures. Pour conclure, si les performances mesurées par Zhao *et al.* (2012) sont meilleures, le rapport performances/consommation énergétique qui est absent ne l'est probablement pas.

D'autre part, les travaux de Nae *et al.* (2009) indiquent cette fois-ci qu'héberger un jeu vidéo dans le *cloud* est périlleux. Leur article ne traite pas de *cloud gaming* mais étudie l'impact de l'hébergement de programmes serveurs de MMOGs dans l'infonuage. Ils constatent que cela peut avoir un effet négatif sur les performances, à cause de *l'overhead* lié à la virtualisation. Ils insistent toutefois sur le fait qu'il est impératif de bien choisir les politiques de virtualisation et les mécanismes de réservation de ressources pour arriver à de bons résultats. Malgré leurs réserves, ils demeurent optimistes pour l'avenir de la virtualisation de jeux vidéo, persuadés que cette technologie évoluera pour mieux cerner les besoins spécifiques de ce genre d'applications temps réel.

Enfin, le projet décrit par Glinka *et al*. (2009) vise à faciliter le développement et le déploiement de ROIA (*RealTime Online Interactive Application*) en tirant profit d'une infrastructure distribuée. Le projet se décompose en deux parties :

- la phase de développement d'un jeu est facilitée par un *middleware* nommé RTF (*Real Time Framework*), visant à simplifier la création d'applications temps réel et hautement interactives sur Internet, tout en assurant un dimensionnement automatique : les

ressources sont ajoutées de manière transparente tout en maintenant un niveau de QoS constant. RTF offre une multitude de fonctionnalités pour implémenter certaines tâches de bas niveau complexes, que l'on retrouve dans les MMOGs : réplication des objets, gestion des zones (différents endroits géographiques d'un jeu qui peuvent être gérés par plusieurs serveurs différents), traitement distribué, etc. Enfin, RTF offre également une collection d'autres modules réutilisables pour effectuer certaines tâches spécifiques (chat, audio, etc.);

- la phase de gestion est permise par un outil nommé HMI (*Hoster Management Interface*), qui permet la création, le contrôle et la supervision de chaque instance d'un service.

Dans sa conclusion, cet article fait également une référence intéressante aux architectures orientées services avec des contraintes temps réel. Il explique toutefois que les ROIA ne se prêtent pas toujours à une décomposition aisée en petits services autonomes et insiste surtout sur le fait que le paradigme du RSOA (*Realtime Service Oriented Architecture*) se base souvent sur une métrique nommée WCET (*Worst Case Execution Time*), très peu adaptée à des services ayant des temps de traitement imprévisibles et hautement variables.

1.5 Gestion dynamique de ressources en environnement infonuagique

La gestion dynamique des ressources en environnement infonuagique permet d'exploiter au mieux une infrastructure informatique distribuée. Les principales fonctionnalités sont :

- assurer l'élasticité en gérant l'allocation et la libération des ressources, en fonction des besoins des machines virtuelles. Cette tâche peut avoir recours à la migration de machines virtuelles, dans le cas où une machine physique ne possède pas assez de ressources pour subvenir aux besoins d'une machine virtuelle qu'elle héberge;
- maintenir une consolidation efficace, c'est-à-dire minimiser le nombre de machines physiques allumées afin de réduire la consommation d'énergie et donc les coûts.

1.5.1 Présentation du problème du *bin packing*

Dans cette partie, nous nous penchons sur le problème du *bin packing,* qui est un prérequis indispensable à la mise en place d'une consolidation efficace.

1.5.1.1 Définition

Comme l'explique Skiena (2011, p. 595), supposons que l'on ait *n* objets de tailles variées et *m* conteneurs de capacités différentes. Le but est de stocker tous les objets en utilisant le minimum de conteneurs. En faisant le parallèle avec la consolidation, on comprend aisément que nous cherchons à regrouper les machines virtuelles pour minimiser le nombre de machines physiques allumées.

Malheureusement, ce problème est NP-complet. Il est donc plus intéressant de trouver des heuristiques. Par chance, ces dernières s'avèrent plutôt efficaces.

Dans notre cas, nous aurons besoin d'un *bin packing* dit multidimensionnel, où chaque objet est décrit par plusieurs dimensions ou paramètres. C'est effectivement le cas des machines virtuelles dont les ressources sont multiples, à savoir la quantité allouée de cœurs CPU, de mémoire, d'espace disque, etc.

Enfin, on peut classifier les algorithmes de *bin packing* en deux catégories : *off-line*, c'est-à-dire que l'on connait à l'avance tous les objets à disposer dans les conteneurs, et *on-line* (notre cas), c'est-à-dire que l'on reçoit progressivement de nouveaux objets et qu'on les place au fur et à mesure qu'ils arrivent. Le cas *on-line* est plus compliqué et peut être optimisé avec une relocalisation des objets. En effet, à chaque arrivée d'un nouvel objet, il faut vérifier si la disposition actuelle des objets déjà assignés aux conteneurs va mener à une situation optimale. Or, on peut être amené à déplacer à nouveau les premiers objets pour obtenir une disposition encore plus économe dans les conteneurs.

1.5.1.2 Heuristiques

On suppose que tous les conteneurs ont la même taille. Parmi les heuristiques, on distingue :

- *next fit* (NF) : on numérote arbitrairement les conteneurs. On choisit un objet au hasard et on essaie de le placer dans le conteneur. Si on n'arrive pas à disposer un objet dans un conteneur, on passe au conteneur suivant;
- *first fit* (FF) : on numérote arbitrairement les conteneurs. On choisit un objet au hasard et on essaie de le placer dans le conteneur ayant le plus petit indice;
- *first fit decreasing* (FFD) : identique au *first fit*, à la différence que les objets sont triés par ordre décroissant en fonction de leur taille. On commence par insérer les plus volumineux;
- *best fit* (BF) : on choisit un objet au hasard et on essaie de le placer dans le conteneur le plus rempli pouvant accueillir l'objet. Cet algorithme existe également en *best fit decreasing* (BFD).

Bhatia *et al.* (2009) expliquent que la complexité temporelle du *next fit* est de $O(n)$, tandis que celle du *first fit*, *first fit decreasing*, *best fit* et *best fit decreasing* est de $O(n\log(n))$. Le *next fit*, en raison de sa simplicité est donc le plus rapide pour trouver une solution. Toutefois, si l'on s'intéresse au nombre de conteneurs utilisés au pire des cas, par rapport au nombre de conteneurs utilisés dans la disposition la plus optimale :

- *next fit* atteint le double de conteneurs utilisés;
- *best fit* et *first fit* atteignent 1.7 fois le nombre de conteneurs;
- *best fit decreasing* et *first fit decreasing* sont les plus efficaces et atteignent 11/9, soit environ 1.22 fois le nombre de conteneurs utilisés.

1.5.1.3 Lien avec la consolidation

Si l'on assimile les conteneurs à des machines physiques et les machines virtuelles aux objets, on constate que la résolution du problème de *bin packing* nous permet de disposer efficacement les machines virtuelles au sein des machines physiques.

Aussi, parmi les heuristiques présentées précédemment, nous avons vu que le FFD et le BFD offrent la consolidation la plus compacte dans le pire des cas. Comme les machines physiques et les machines virtuelles sont caractérisées par plusieurs paramètres (taux d'utilisation CPU, taux de mémoire vive utilisé, taux d'espace disque utilisé etc.) et doivent être triées (comme le requièrent FFD ou BFD), il est nécessaire de trouver une solution pour synthétiser leur taille de manière globale, sans quoi on ne pourrait les comparer. Dans la littérature, on parle également de résoudre le problème du *vector bin packing* ou *multidimensional bin packing*. Pour cela, Wood *et al.* (2007) proposent une approche très simple, qui consiste à faire le produit des taux d'utilisation. Ainsi, le taux global d'utilisation d'une machine physique ou virtuelle, désigné par u_{global}, peut-être synthétisé par la formule :

$$u_{global} = u_{CPU} \times u_{RAM} \times u_{DISK} \tag{1.2}$$

Comme le résultat u_{global} est un scalaire, il est possible d'utiliser la version unidimensionnelle d'une heuristique de *bin packing* pour résoudre des problèmes multidimensionnels.

1.5.2 Travaux antérieurs sur les approches orientées vers le dimensionnement dynamique

Parmi les techniques permettant d'automatiser l'élasticité au sein d'une infrastructure infonuagique, nous nous pencherons tout d'abord sur l'approche proposée par Rui *et al.* (2012), qui servira de base pour l'algorithme proposé dans ce mémoire.

Dans cet article, les auteurs partent du constat que les stratégies de dimensionnement actuelles offertes par les fournisseurs de *cloud* ne permettent pas un ajustement précis des ressources. En effet, la plupart des solutions commerciales consistent à instancier une machine virtuelle supplémentaire lorsque les performances d'une application hébergée ne sont pas satisfaisantes. Cela est certes efficace pour les applications compatibles, comme les serveurs web, mais certaines ne sont pas capables de tirer profit de cette technique. L'article introduit donc une mise à l'échelle avec une granularité plus fine en permettant de reconfigurer les ressources allouées aux machines virtuelles. Ces ressources peuvent être la

quantité de cœurs de processeurs, de mémoire, de bande passante, etc. Les décisions liées à cette reconfiguration sont prises par un algorithme nommé *Lightweight Scaling* ou LS. Son nom reflète sa vocation puisqu'il se veut simple à implémenter et relativement léger à exécuter. Il est ainsi capable de prendre des décisions à partir de deux métriques, pour déclencher une reconfiguration des ressources allouées : le temps de réponse d'une application et le taux d'utilisation de chaque ressource par chaque machine virtuelle hébergeant un des tiers de l'application. Les auteurs ont choisi pour leur banc de test d'appliquer l'algorithme à une application multi-tiers hébergeant un site de commerce électronique. Ici, chaque tiers est constitué d'au moins un processus. Chaque processus est hébergé par une seule machine virtuelle.

L'algorithme étant expliqué dans l'article, nous nous contenterons dans cette partie de décrire son fonctionnement global à travers une vue simplifiée de l'algorithme. Ainsi, définissons tout d'abord les notations suivantes :

Tableau 1.1 Paramètres simplifiés du LS
Adapté de Rui *et al.* (2012)

$\mathbf{t_0}$	Temps de réponse de l'application mesuré
$\mathbf{t_{su}}$	Seuil supérieur du temps de réponse de l'application pour tenter un approvisionnement en ressources (*scale up*)
$\mathbf{t_{sd}}$	Seuil inférieur du temps de réponse de l'application pour tenter une libération de ressources (*scale down*)
$\mathbf{u_{su}}$	Seuil supérieur du taux d'utilisation d'une ressource par un processus pour être candidat à un *scale up*
$\mathbf{u_{sd}}$	Seuil inférieur du taux d'utilisation d'une ressource par un processus pour être candidat à un *scale down*

Algorithme 1.1 LS (Lightweight Scaling Algorithm)
Adapté de Rui *et al.* (2012)

```
1.  while( l'application fonctionne ) :
2.          Mesurer t0
3.          if( t0 > tsu ) :
4.                  LSU()
5.          else if( t0 < tsd ) :
6.                  LSD()
```

Algorithme 1.2 LSU (Lightweight Scaling Up Algorithm)
Adapté de Rui *et al.* (2012)

```
1.  Self_healing()
2.  if( t0 > tsu ) :
3.          Resource_Level_Scaling_Up()
4.  while(t0 > tsu) :
5.          VM_Level_Scaling_Up()
6.          Resource_Level_Scaling_Up()
```

L'algorithme 1.1 est le point d'entrée. Cette fonction agit de la sorte :

1) si le temps de réponse de l'application mesuré n'est pas satisfaisant, on appelle la fonction LSU qui tentera d'augmenter la quantité de ressources allouées à l'application. On cherche ainsi à mettre un terme au sous-approvisionnement pour améliorer les performances;
2) si le temps de réponse mesuré est suffisamment bas, on appelle la fonction LSD qui tentera de réduire la quantité de ressources allouées à l'application. Quitte à avoir un niveau de performances plus faible mais dans la zone de tolérance, on évite ainsi le sur-approvisionnement.

Dans le premier cas, un appel à la fonction LSU engendre :

1) le *Self Healing*, une procédure qui consiste à rééquilibrer la répartition des ressources entre les différents processus d'une même application. Voici son fonctionnement :
   a) on génère une liste L_{sh} de paires de toutes les permutations p1 et p2, p1 et p2 étant deux processus de la même application, dont les machines virtuelles sont hébergées sur la même machine physique;

b) si p1 possède pour une ressource donnée un taux d'utilisation inférieur à u_{sd}, celui-ci est candidat pour donner une unité de cette ressource à un autre processus dont le taux d'utilisation dépasse u_{su}. On rajoute alors la paire (p1,p2) à L_{sh};
c) on choisit une paire dans L_{sh}, on équilibre les ressources, puis on voit l'effet sur t_0;
d) on répète la procédure depuis a) tant que L_{sh} n'est pas vide et que t_0 n'est pas entré dans la zone de tolérance.

2) si le *Self Healing* ne suffit pas à améliorer le temps de réponse, on procède à un *Resource Level Scaling Up*. Ici le principe consiste à :
   a) déterminer une liste L_{rlsu} de tous les processus qui dépassent pour une ressource donnée le seuil u_{su} ;
   b) sélectionner un processus p_0 selon un critère. Par exemple, celui qui du point de vue des ressources coûte le moins cher à augmenter;
   c) augmenter les ressources allouées à p_0 d'une unité et observer l'effet sur t_0 ;
   d) répéter la procédure depuis a) jusqu'à ce que la liste L_{rlsu} soit vide et que t_0 entre dans la zone de tolérance.

3) si le *Resource Level Scaling Up* échoue dans l'amélioration du temps de réponse, la dernière technique avec la plus faible priorité s'appelle *VM Level Scaling Up*. Elle consiste à déterminer selon un critère de sélection le tiers de l'application à approvisionner, et lui instancier une machine virtuelle. Cette technique est ensuite suivie à nouveau du *Resource Level Scaling Up* jusqu'à ce que le temps de réponse de l'application soit corrigé. Toutefois, nous verrons à partir du second chapitre que le *VM Level Scaling Up* n'est pas compatible avec notre implémentation de jeu vidéo orienté *substrates*, nous ne l'inclurons pas dans la conception de notre algorithme. En effet, le *VM Level Scaling Up* suppose que chaque *substrate* est capable de subdiviser son travail pour le déléguer à plusieurs processus connectés qui coopèrent. Chaque *substrate* aurait alors un processus dit maître qui devra gérer l'instanciation et la destruction de processus dits esclaves. Ce scénario emprunté à l'informatique distribuée est très intéressant, mais il

demeure dans les directions futures de ce mémoire en raison des efforts considérables pour l'implémenter.

Enfin, les résultats fournis par cet article indiquent que comparé aux techniques offertes par Amazon EC2 ou RightScale, le LS affiche de meilleures performances en termes de :

- temps nécessaire pour réaliser l'élasticité;
- quantité de machines physiques utilisées;
- coût de revient pour le fournisseur de *cloud*.

Parmi les autres travaux, l'article de Verma *et al.* (2008) est l'un des premiers visant à résoudre des problèmes d'optimisation entre la consommation énergétique et la gestion des performances dans un environnement virtualisé. Les auteurs optent pour une stratégie centralisée permettant de déterminer le compromis coût-performance. Malheureusement, dans ce mémoire nous ne pourrons pas prendre en compte la modélisation de l'énergie à cause des capacités limitées de notre simulateur (cf. cinquième chapitre).

Par ailleurs, Gueyoung *et al.* (2010) proposent un *framework* d'élasticité très sophistiqué et holistique, puisqu'il prend simultanément en compte trois contraintes : optimiser les coûts d'infrastructure, les performances des applications hébergées, et les coûts temporaires ayant lieu durant les procédures d'élasticité. Ces derniers sont essentiellement liés aux coûts de migrations des machines virtuelles, comme l'utilisation accrue des processeurs des machines physiques de source et de destination ou encore l'utilisation supplémentaire de la bande passante du réseau. Bien qu'il complète efficacement les travaux de Verma *et al.* (2008) en rajoutant une contrainte d'optimisation liée à la consolidation, une des raisons pour lesquelles cet article n'a pas servi de base pour ce mémoire est son approche pour déterminer les coûts temporaires qui nécessite une phase d'apprentissage. Dans notre cas, nous avons préféré conserver un algorithme agnostique aux applications supervisées.

De manière très spécifique cette fois-ci, Mills *et al.* (2011) étudient le problème de l'allocation initiale des machines virtuelles dans un environnement infonuagique. Pour un *cloud* donné, ils distinguent dans leur politique d'allocation deux niveaux : tout d'abord le

cluster à choisir (trois stratégies proposées), puis la machine physique dans ce cluster (6 stratégies proposées). Cela donne un total de 18 combinaisons possibles d'algorithmes pour satisfaire les demandes d'instanciation de machines virtuelles. Ces algorithmes sont soit des approches aléatoires soit des heuristiques permettant de résoudre le problème du *bin packing*. L'article conclut que la stratégie de sélection du *cluster* affecte sensiblement les coûts du fournisseur de *cloud* tandis que dans le cas du choix de la machine physique, aucune stratégie ne se démarque suffisamment. Comme nous le verrons dans le quatrième chapitre, nos conditions seront plus simples puisqu'un seul *cluster* sera considéré.

Shekhar *et al.* (2008) se penchent sur l'étude d'une consolidation écologique dans une infrastructure infonuagique. Ce travail établit des relations entre consommation d'énergie, taux d'utilisation des ressources et performance des applications. Il révèle qu'il existe un compromis entre ces trois objectifs. En effet, la consommation d'énergie est simultanément influencée par les performances et le taux d'utilisation : il existe un point optimal dans cette tendance dont la courbe est en forme de U. Ceci se comprend lorsque l'on considère les deux cas extrêmes. Dans le cas du sur-approvisionnement, le taux d'utilisation d'une ressource est faible, l'énergie consommée par une machine physique à l'état inoccupé n'est pas amortie, du coup la consommation énergétique globale est importante. A l'inverse, en cas de sous-approvisionnement, le taux d'utilisation d'une ressource dépasse un certain seuil de saturation et la consommation énergétique augmente à cause de la dégradation des performances qui implique des temps de traitement plus importants. Cet article est intéressant puisqu'il inclut les coûts de migration (tout comme Gueyoung *et al.* (2010)) impliqués par les phases de consolidation.

Beloglazov *et al.* (2012) étudient de manière approfondie la façon la plus optimale pour détecter qu'une machine physique est saturée, dans le but de déclencher une migration pour alléger sa charge et répondre à un objectif de QoS global. L'article montre entre autres que pour améliorer la qualité de la consolidation, il faut maximiser le temps moyen entre chaque migration, qui pour rappel implique un coût.

Dans un deuxième article, Beloglazov *et al.* (2012) décrivent une heuristique pour une consolidation qui optimise la consommation d'énergie et les performances des applications. Les auteurs précisent encore une fois qu'une politique de consolidation agressive (minimiser à tout prix la quantité de machines physiques allumées) mène à une dégradation des performances des applications hébergées. L'algorithme présenté étudie l'historique des taux d'utilisations des machines virtuelles et prend des décisions de manière adaptive.

Shrivastava *et al.* (2011) présentent une stratégie de migration des machines virtuelles qui considère la topologie des applications multi-tiers pour prendre des décisions adéquates. Cette approche dite consciente analyse les interactions dues aux dépendances entre machines virtuelles d'une même application, ainsi que la topologie du réseau sous-jacent pour offrir une consolidation plus efficace. Le bénéfice apporté se situe essentiellement au niveau de l'économie de bande passante, comparé aux approches classiques de migration.

Enfin, Yang *et al.* (2011) définissent une stratégie de migration qui dépend de plusieurs niveaux de charge pour chaque machine virtuelle donnée : zone de charge légère, zone optimale, zone à risque et zone de saturation. A chaque zone est associée une stratégie différente. L'article combine cela à un algorithme permettant de prédire les moments de transition d'une zone à une autre et prend les décisions nécessaires, à savoir migrer ou non.

Jusqu'à présent, nous n'avons vu que des architectures de dimensionnement centralisées. Beltran *et al.* (2012) proposent une approche distribuée pour répondre au besoin de l'élasticité, nommée AMAS (*Automatic MAchine Scaling*). La vocation principale de cette stratégie est de pouvoir faire face à la taille et la complexité grandissantes des infrastructures infonuagiques. Le plus intéressant dans ce travail est de comprendre les entités présentes au niveau de chaque machine physique, les protocoles de communication et les algorithmes mis en œuvre afin de pouvoir prendre des décisions de manière décentralisée. Néanmoins, parmi les faiblesses de cet article, l'implémentation proposée se limite à une forte granularité : seul l'ajout ou le retrait d'une machine virtuelle sont considérés. De même, les auteurs précisent qu'ils ne se penchent pas sur le cas des applications multi-tiers où plusieurs processus

travaillent de concert pour achever une tâche donnée. Pour résumer, bien que l'approche décentralisée présentée par *Beltran et al.* (2012) soit intéressante, les limitations actuelles sont handicapantes pour les besoins de l'architecture proposée dans ce mémoire.

De façon similaire, Yazir *et al.* (2010) proposent une architecture décentralisée pour une gestion dynamique des ressources. Les auteurs estiment encore une fois que les approches centralisées ne sont efficaces que pour les *datacenters* de taille relativement petite. Il faut donc se tourner vers une stratégie distribuée pour éviter les problèmes de mise à l'échelle. L'architecture générale est assez similaire aux travaux de Beltran *et al.* (2012) puisque chaque machine physique héberge un programme de gestion qui communique avec ses pairs pour effectuer le dimensionnement. Toutefois, les protocoles, les algorithmes et les machines à états sont différents. L'article présente une approche intéressante lorsqu'un des programmes de gestion souhaite se débarrasser d'une machine virtuelle via une migration en cas de saturation de la machine physique qui l'héberge. Répondant au nom de *three way handshake*, cette technique sera présentée dans le cinquième chapitre en tant que direction future. Une autre idée intéressante est présentée dans cet article : il s'agit de PROMETHEE, une méthode permettant de prendre des décisions basées sur la pondération de plusieurs critères. Concernant les résultats, les auteurs montrent qu'il existe certes des cas où les architectures centralisées offrent de meilleures performances, puisque les décisions sont prises en fonction d'une vue globale, mais que l'approche décentralisée s'avère plus bénéfique lorsque la taille des infrastructures infonuagiques augmente.

1.6 Conclusion du chapitre

Dans ce chapitre, nous avons fait un tour d'horizon sur le principe de fonctionnement et les défis du *cloud computing* et des jeux vidéo, puis nous nous sommes penchés sur les avancées du *cloud gaming* et de la gestion de ressources en environnements infonuagiques.

Dans le chapitre suivant, nous allons décrire notre modèle d'affaires à travers la description détaillée de notre architecture orientée services, permettant la prise en charge des *substrates* virtualisés.

CHAPITRE 2

ARCHITECTURE PROPOSÉE

Dans ce chapitre, nous allons présenter notre modèle d'affaires à travers notre architecture orientée services ainsi que le concept des *substrates*. Nous montrerons comment les modules de notre architecture coopèrent pour assurer le bon déroulement de chaque phase de fonctionnement : publication et découverte des *substrates*, puis composition, réservation et enfin activation des compositions. Nous nous pencherons également sur la description de notre modèle d'information qui illustre l'organisation interne de la base de données principale de notre architecture.

2.1 Objectifs

L'architecture que nous proposons est en partie inspirée de celles présentées par Zhao *et al.* (2012) et Glinka *et al*. (2009). Elle emprunte à Zhao *et al.* (2012) l'aspect distribué d'un jeu vidéo où chaque module peut être exécuté sur une machine virtuelle, mais elle se rapproche aussi des travaux de Glinka *et al*. (2009) pour son orientation services et par sa capacité à déployer facilement des sessions de jeux à travers des interfaces de gestion. Néanmoins la nôtre apporte les fonctionnalités suivantes :

- les modules d'un jeu donné sont assemblés pour former une composition, qui est simplement un service viable, c'est-à-dire un ensemble de modules compatibles. Il est impératif que cette opération d'assemblage soit facilitée à des personnes avec des connaissances techniques modestes : les *service providers*. Ceux-ci peuvent ainsi réserver et demander le déploiement d'une session de jeu pour un certain nombre de joueurs, que l'on qualifiera d'utilisateurs finaux (*end users*);
- l'architecture vise à promouvoir au maximum la réutilisation des modules de jeux vidéo. Ainsi un moteur de rendu peut être réutilisé d'un jeu à l'autre, si le concepteur d'un jeu le permet;

- pour un jeu donné, si plusieurs modules compatibles remplissant la même fonction sont offerts, on peut en choisir un explicitement ou demander à notre architecture de trouver automatiquement la meilleure combinaison selon une stratégie spécifique (meilleur prix pour un niveau de QoS donné, par exemple);
- l'architecture reprend le modèle d'affaires proposé par Belqasmi *et al.* (2011), où les modules de toute application sont nommés *substrates* et sont offerts par des fournisseurs nommés *substrates providers*. Les *substrates* sont alors sélectionnés (ou composés) par les *service providers*, réservés pour une période donnée, puis activés pour démarrer la phase de service.

Concernant les *substrates*, on en distingue deux catégories. Il y a les *cores*, qui ne sont pas réutilisables et qui intègrent l'ensemble des propriétés unique à un jeu donné, à savoir la logique et les règles ainsi que les *assets* (textures, images, sons, modèles 3D, etc.). Ils nécessitent en revanche la seconde catégorie nommée *dependencies* qui correspond à des modules plus ou moins spécialisés et hautement réutilisables. Par exemple, on peut avoir un *core* pour un jeu de voiture contenant la logique du jeu et les *assets* (modèles 3D des voitures, textures, musiques, sons et effets spéciaux, etc.) faisant appel à des modules spécialisés comme un moteur de rendu ou un moteur physique.

Enfin, pour conclure sur les différences entre l'architecture traditionnelle des jeux vidéo (qui est monolithique) et notre approche orientée *substrates*, on retiendra que dans le cas monolithique, des développeurs expérimentés assemblent des modules (réutilisables ou non) pour construire un jeu. Développer un autre jeu pourra occasionner l'utilisation de ces mêmes modules, mais dans tous les cas il nécessitera l'intervention de programmeurs expérimentés. Or, avec notre paradigme des *susbtrates*, les développeurs définissent durant la conception du jeu une compatibilité entre des modules (les *dependencies* et un *core*), pour permettre l'assemblage de ces derniers à la volée par des personnes au profil technique beaucoup plus modeste. En effet, ce sera le cas des techniciens employés par les *service providers*.

2.2 Présentation des entités et des phases

Dans cette partie, nous allons décrire notre architecture permettant de supporter le paradigme des *substrates*. Nous allons dans un premier temps voir un aperçu global, puis nous nous pencherons sur l'interaction entre ses différentes entités, pour chaque phase.

2.2.1 Vue générale

La figure 2.1 offre une vue globale de l'architecture. Les phases sont représentées à travers des flèches de couleurs différentes, tout comme les niveaux qui englobent les entités.

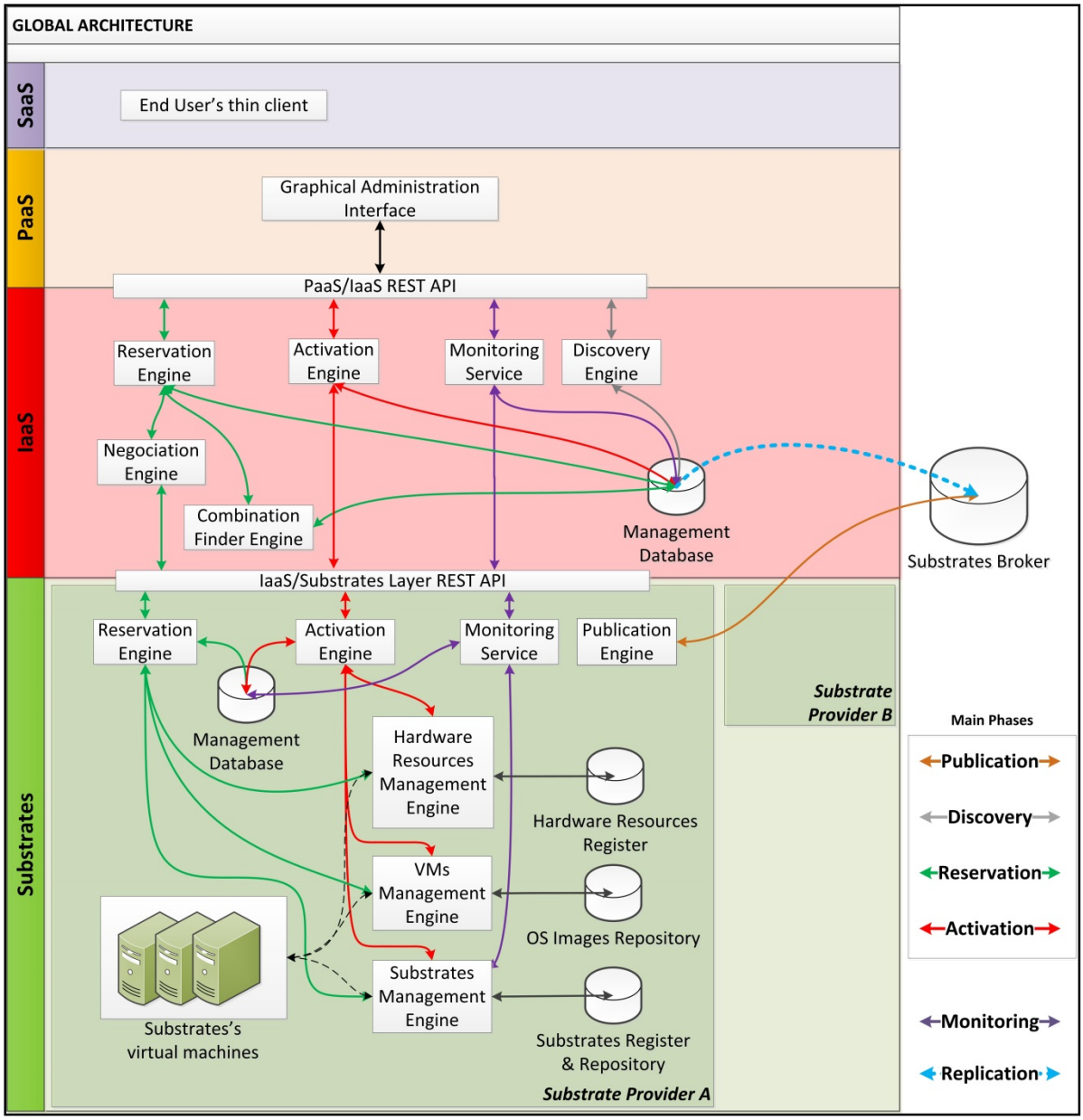


Figure 2.1 Architecture globale

Comme certaines entités peuvent avoir un rôle différent selon la phase, nous estimons qu'il est plus judicieux de décrire les phases en premier.

2.2.2 Description des phases

2.2.2.1 Phase de publication des *substrates*

A l'origine, notre infrastructure est vide et ne contient aucun *substrate*. La phase de publication est donc nécessaire et permet aux *substrates providers* d'inscrire leurs *substrates* afin de les rendre visibles et disponibles aux *service providers*.

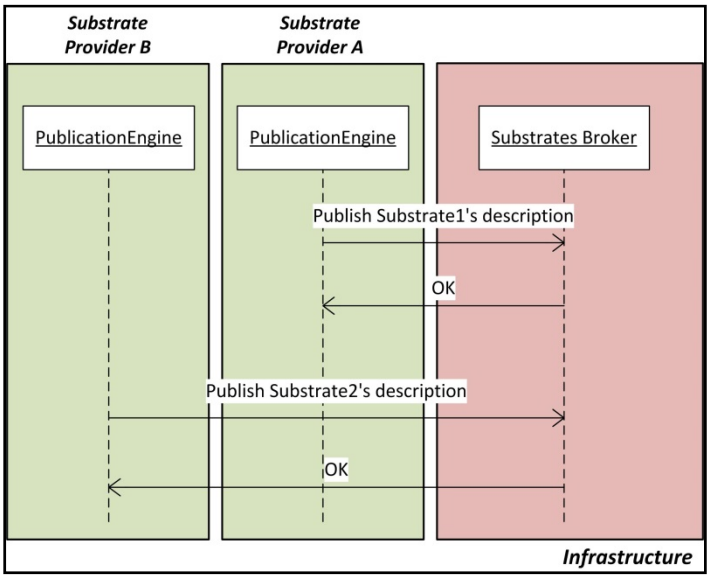


Figure 2.2 Phase de publication

La publication d'un *substrate* se fait par l'envoi d'une description, permettant de définir plusieurs propriétés que nous verrons plus loin dans le modèle d'information. Il s'agit par exemple du type (*core* ou *dependency*), de la configuration matérielle requise (CPU, RAM, etc.), de la compatibilité avec les autres *substrates*, etc. Une description est vérifiée puis envoyée par le *Publication Engine* de chaque *Substrates Provider,* à destination du *Substrates Broker*, une base de données centrale. La phase de publication est la seule où un accès direct est réalisé sur le *Substrates Broker*. A l'avenir, seule la *Management Database*, possédée par chaque fournisseur d'IaaS, sera sollicitée pour obtenir des informations sur les *substrates*. Cette dernière se synchronise automatiquement sur le *Substrates Broker* et joue le rôle de cache.

2.2.2.2 Phase de découverte des *substrates*

Suite à la phase de publication, notre infrastructure contient désormais des jeux sous forme de *substrates*. Durant la phase de découverte, un *service provider* qui dispose d'un outil de composition (désigné par GUI) va pouvoir mettre à jour la liste des *substrates* qui lui sont proposés.

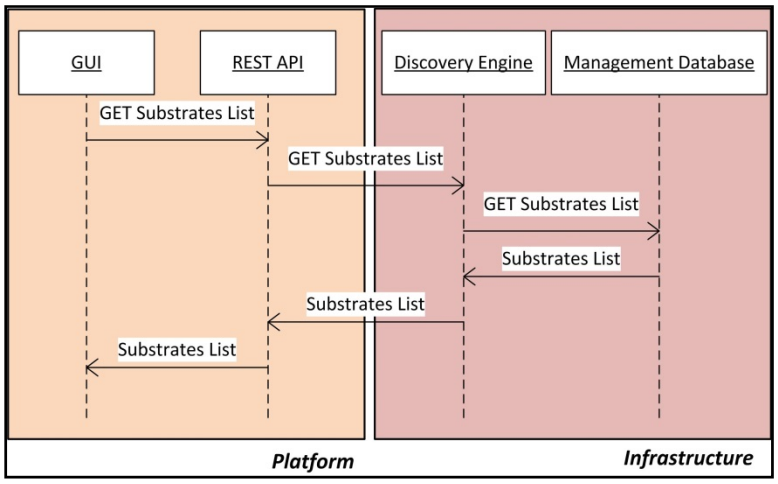


Figure 2.3 Phase de découverte

La GUI communique avec un module de découverte nommé *Discovery Engine*, via l'intermédiaire d'interfaces REST (d'où l'entité REST API). Le *Discovery Engine* effectue des requêtes sur la *Management Database*, afin de pouvoir récupérer la liste des *substrates* disponibles.

2.2.2.3 Phase de négociation d'une composition

Dès qu'un *service provider* sélectionne un ensemble de *substrates* compatibles et réalise donc une composition, il va devoir enregistrer et réserver cette dernière pour une période donnée. La GUI va s'occuper de cette tâche et transmettra la requête à l'infrastructure. Cette dernière va tenter de trouver une solution, c'est-à-dire une combinaison de *substrates providers* qui peuvent subvenir aux besoins du *service provider* pour héberger chaque

substrate inclus dans la composition. Cela passe donc par l'acceptation de contraintes imposées par le *service provider* : coût maximal, qualité de service choisie, substrates sélectionnés, etc. Durant cette phase de négociation, chaque *substrates provider* va tenter de réaliser ce qu'on appelle une réservation, en attendant un signal d'activation.

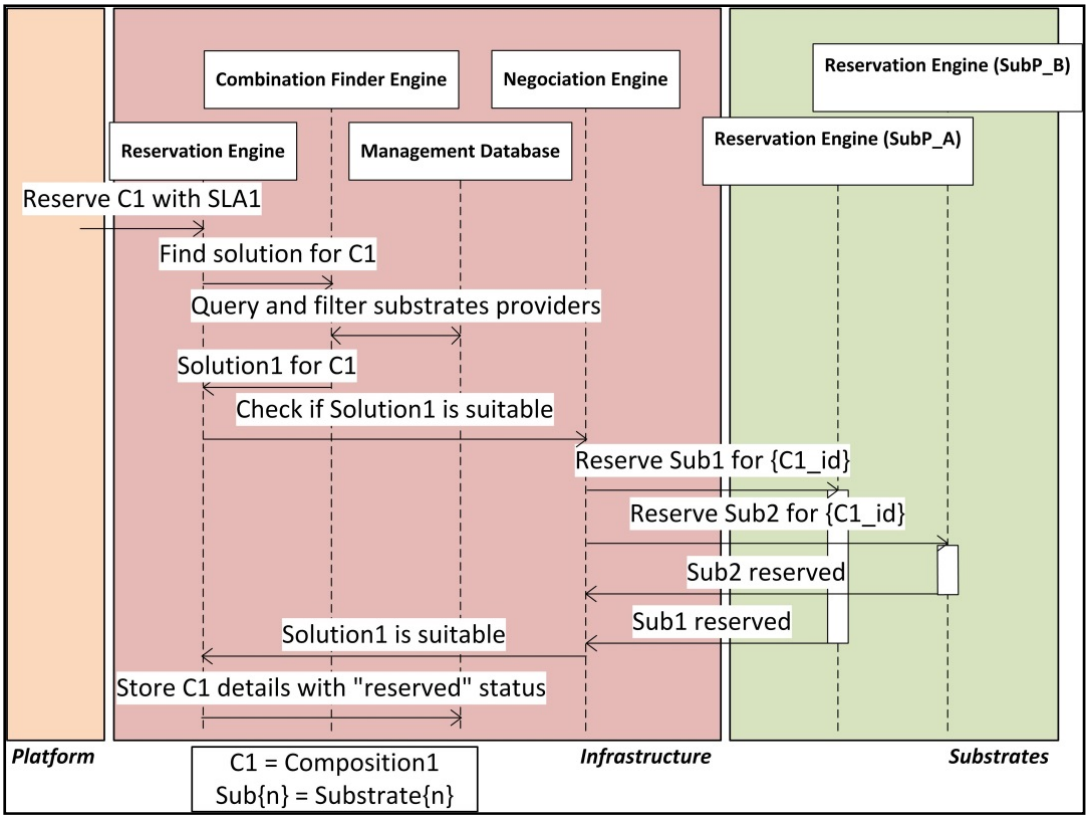


Figure 2.4 Phase de négociation (vue globale)

Une proposition d'architecture interne pour les *substrates provider*, est décrite dans la figure 2.5. La demande de réservation provenant du *Negociation Engine* de l'infrastructure est traitée en premier par le *Reservation Engine* du *substrates provider*. Celui-ci va devoir vérifier pour une période donnée que :

1) les ressources matérielles sont disponibles, via le *Hardware Resources Management Engine*;
2) les images des systèmes d'exploitation pour héberger le ou les *substrates* sont disponibles, via le *Virtual Machine Management Engine*;

3) les binaires du ou des *substrates* sont disponibles, via le *Substrates Management Engine*.

Dès que cette vérification est complétée, chaque *substrates provider* mémorise une réservation dans sa *Management Database* et répond au *Negociation Engine* de l'infrastrucure avec une réponse positive.

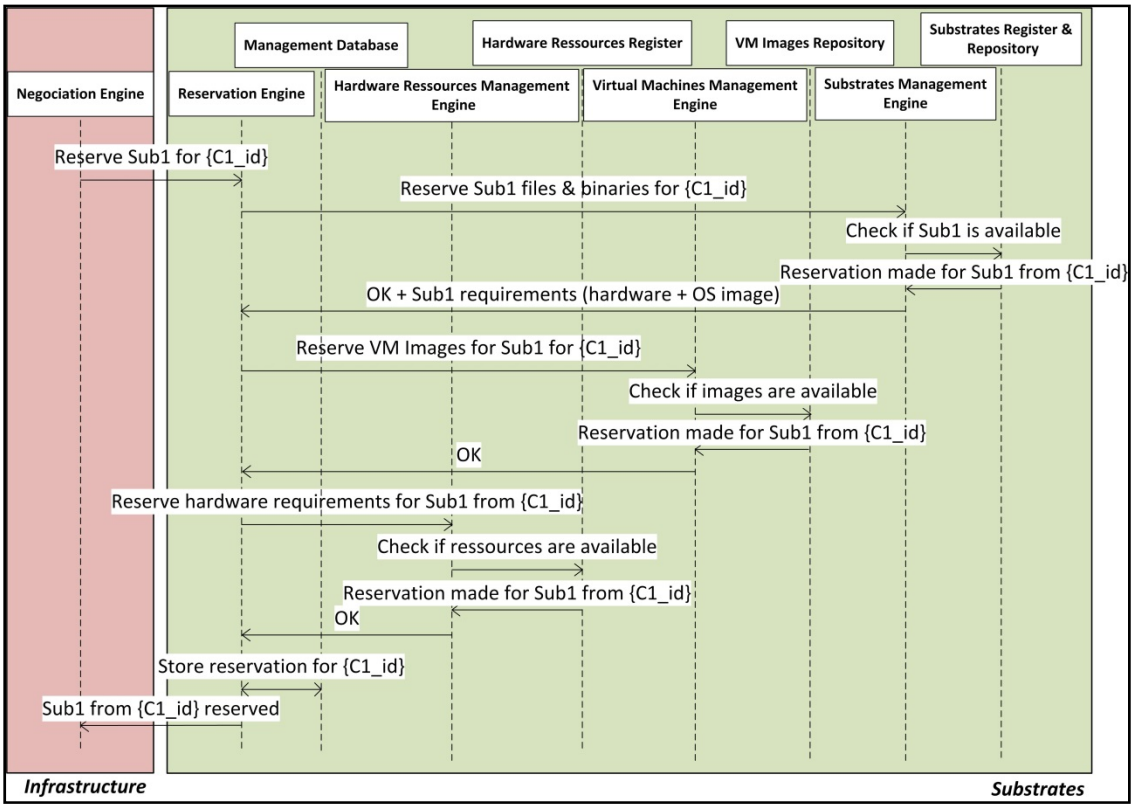


Figure 2.5 Phase de négociation (vue à l'intérieur d'un *substrates provider*)

2.2.2.4 Phase d'activation d'une composition

Lorsqu'un *service provider* a réussi à enregistrer et réserver une composition, il peut l'activer pour la rendre opérationnelle. C'est durant cette phase que les machines virtuelles seront lancées, chargées avec les *substrates* et que la composition deviendra effective. L'idée est de simplement demander d'activer au niveau de chaque *substrates provider* la réservation faite durant la phase de négociation. La séparation de la phase de négociation et d'activation offre une meilleure flexibilité, indispensable aux *substrates providers* : cela leur permet de traiter facilement plusieurs requêtes de différents *service providers*, ce qui en soit représente une tâche complexe. Parfois, il se peut qu'une réservation n'aboutisse pas à une activation. Dans ce cas il est préférable de pouvoir l'annuler. La figure 2.6 illustre les séquences aboutissant à une demande d'activation. Dans cet exemple, nous avons considéré une composition constituée de deux *substrates*, sub1 et sub2, où sub1 joue le rôle de *core* et sub2 de *dependency*. De ce fait, l'*Activation Engine* fournit à sub1 les coordonnées de sub2, afin qu'ils entrent en communication. Enfin, la figure 2.7 montre les séquences d'activation à l'intérieur de l'infrastructure des *substrates providers*.

Figure 2.6 Phase d'activation (vue globale)

Comme le montre la figure 2.7, durant la phase d'activation, chaque *substrates provider* peut entamer une phase de test avant d'entrer dans la phase de service. Cette phase permet de s'assurer que les *substrates* fonctionnent correctement et que les performances sont celles définies par le SLA.

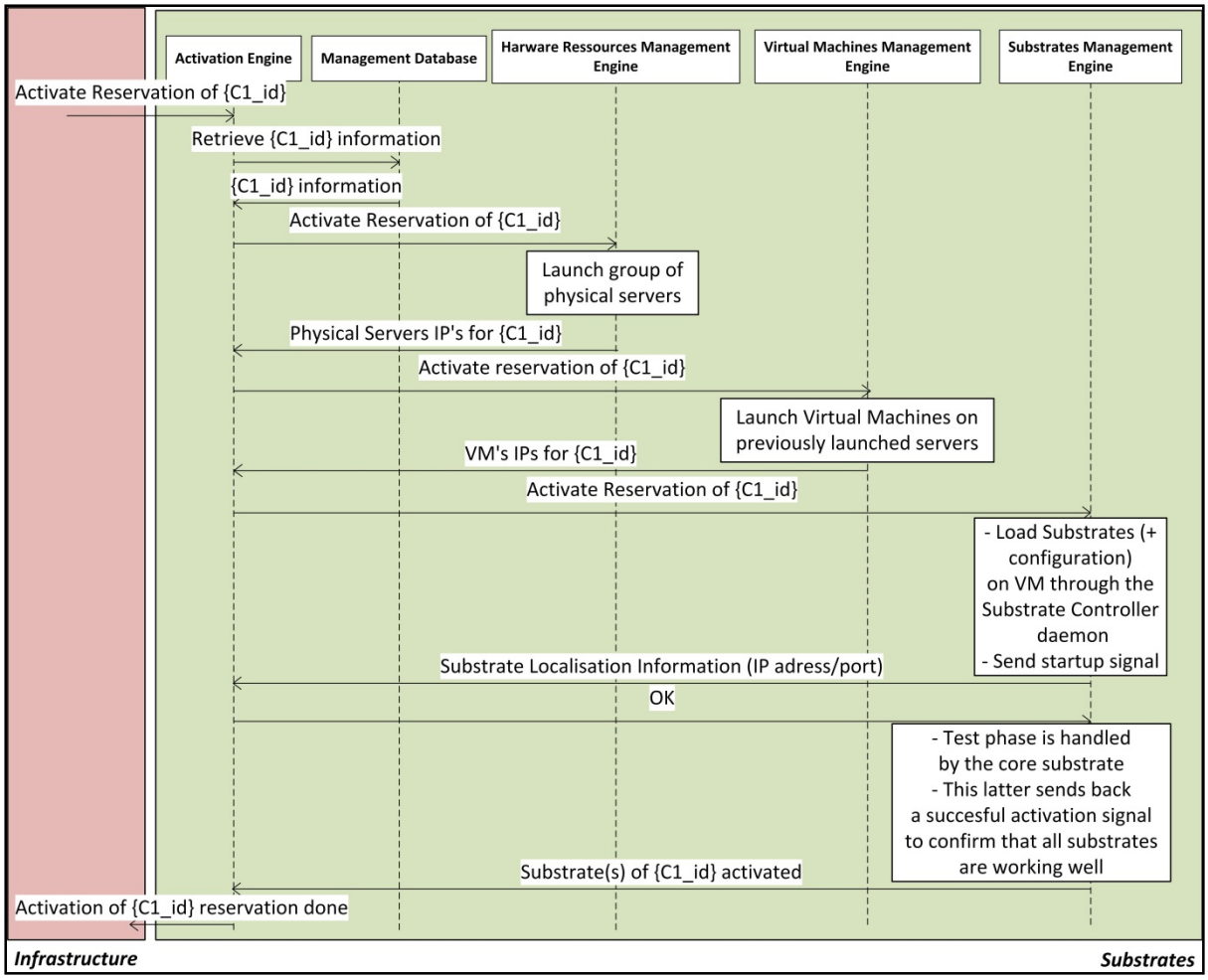


Figure 2.7 Phase d'activation (vue à l'intérieur d'un *substrates provider*)

2.2.2.5 Phase de service d'une composition

La phase de service correspond à l'utilisation de la composition par un ou plusieurs clients finaux, ayant signé un contrat avec le *service provider.*

2.3 Modèle d'information

Dans cette partie, nous allons décrire l'organisation interne de notre *Management Database*, afin de voir comment celle-ci permet de gérer les différentes phases de l'architecture.

Le diagramme de la figure 2.8 utilise le modèle relationnel pour décrire notre modèle d'information. Plutôt que d'expliquer le rôle de chaque table, nous estimons qu'il est préférable d'utiliser un exemple pédagogique (figure 2.9), permettant de comprendre son principe de fonctionnement.

2.3.1 Diagramme Entité-Association

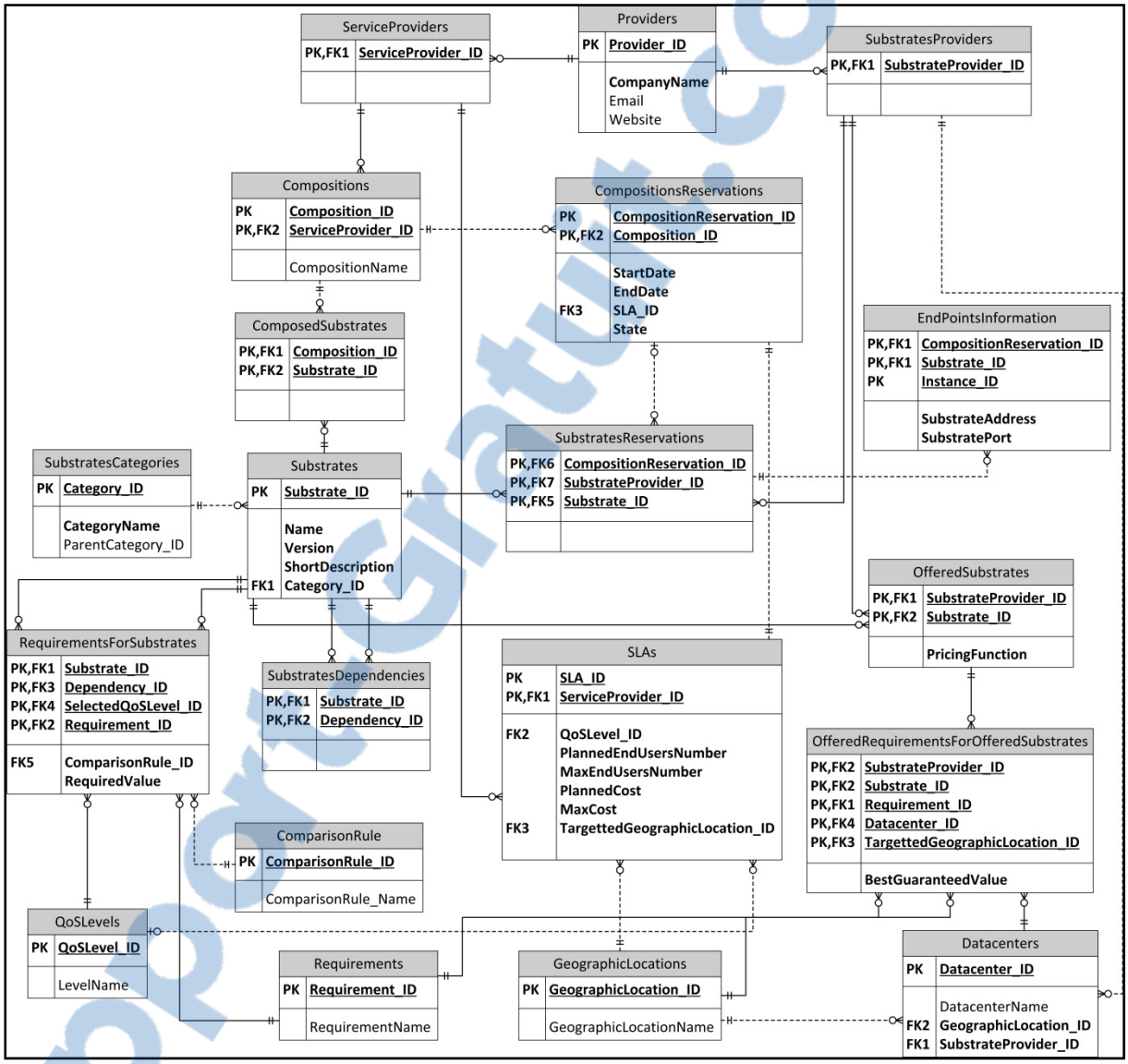
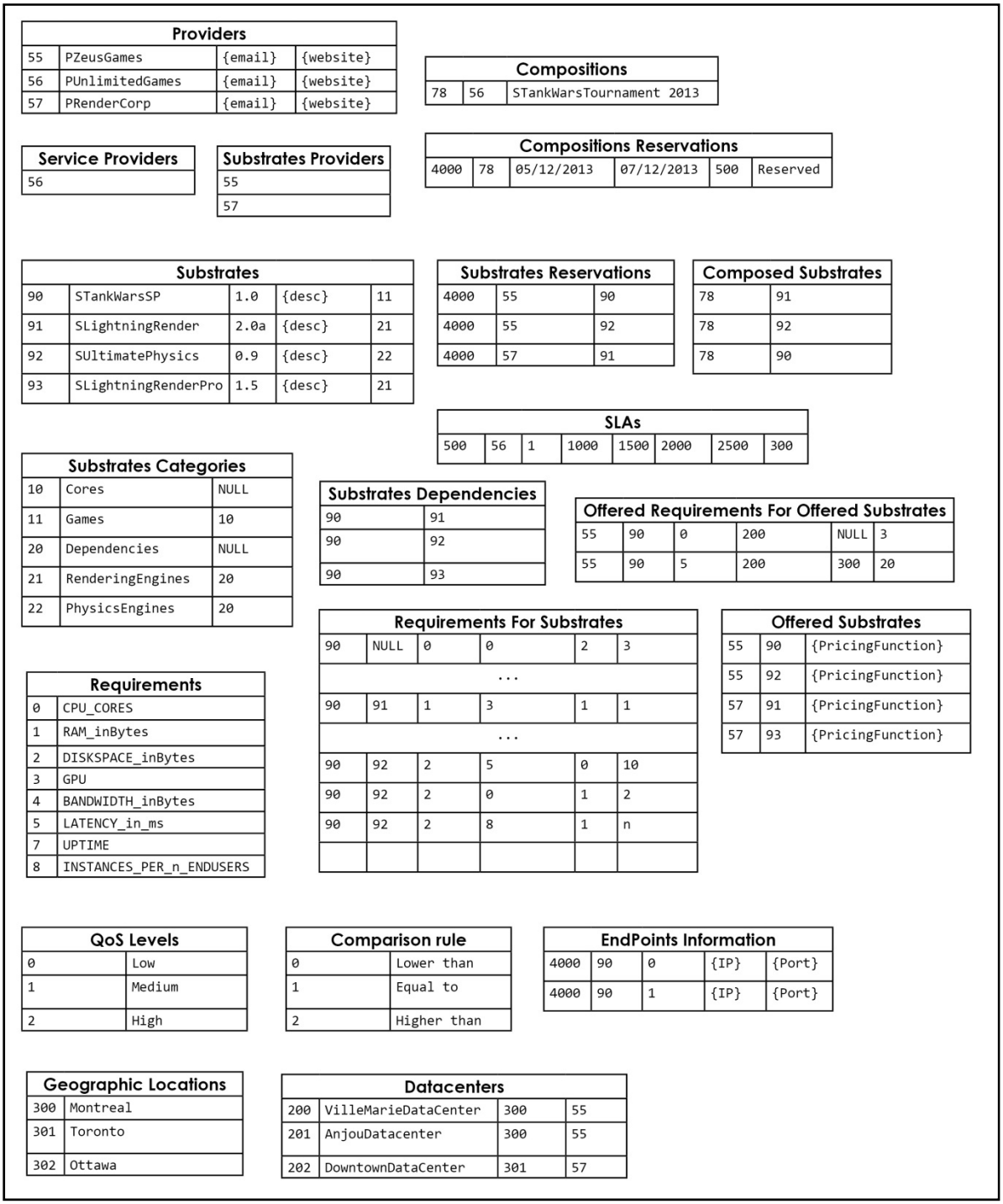


Figure 2.8 Modèle d'information de l'architecture

2.3.2 Exemple de démonstration



Providers

55	PZeusGames	{email}	{website}
56	PUnlimitedGames	{email}	{website}
57	PRenderCorp	{email}	{website}

Compositions

78	56	STankWarsTournament 2013

Service Providers

56

Substrates Providers

55
57

Compositions Reservations

4000	78	05/12/2013	07/12/2013	500	Reserved

Substrates

90	STankWarsSP	1.0	{desc}	11
91	SLightningRender	2.0a	{desc}	21
92	SUltimatePhysics	0.9	{desc}	22
93	SLightningRenderPro	1.5	{desc}	21

Substrates Reservations

4000	55	90
4000	55	92
4000	57	91

Composed Substrates

78	91
78	92
78	90

SLAs

500	56	1	1000	1500	2000	2500	300

Substrates Categories

10	Cores	NULL
11	Games	10
20	Dependencies	NULL
21	RenderingEngines	20
22	PhysicsEngines	20

Substrates Dependencies

90	91
90	92
90	93

Offered Requirements For Offered Substrates

55	90	0	200	NULL	3
55	90	5	200	300	20

Requirements For Substrates

90	NULL	0	0	2	3
...					
90	91	1	3	1	1
...					
90	92	2	5	0	10
90	92	2	0	1	2
90	92	2	8	1	n

Offered Substrates

55	90	{PricingFunction}
55	92	{PricingFunction}
57	91	{PricingFunction}
57	93	{PricingFunction}

Requirements

0	CPU_CORES
1	RAM_inBytes
2	DISKSPACE_inBytes
3	GPU
4	BANDWIDTH_inBytes
5	LATENCY_in_ms
7	UPTIME
8	INSTANCES_PER_n_ENDUSERS

QoS Levels

0	Low
1	Medium
2	High

Comparison rule

0	Lower than
1	Equal to
2	Higher than

EndPoints Information

4000	90	0	{IP}	{Port}
4000	90	1	{IP}	{Port}

Geographic Locations

300	Montreal
301	Toronto
302	Ottawa

Datacenters

200	VilleMarieDataCenter	300	55
201	AnjouDatacenter	300	55
202	DowntownDataCenter	301	57

Figure 2.9 Exemple de contenu pour le modèle d'information

L'exemple illustré par la figure 2.9 correspond au contenu factice d'une base de données employant notre modèle d'information. Il permet de mieux comprendre l'intérêt de chaque table et justifie les relations entre chaque champ.

Commençons par la table des *Providers*. Celle-ci indique en premier lieu que nous avons un fournisseur dont l'identifiant unique est 55 et dont le nom de la compagnie est *ZeusGames* (l'email et le site web ont été omis). Ce fournisseur est en fait un *substrates provider* puisqu'il est listé dans la table des *Substrates Providers*, mais rien ne l'empêche de devenir un *service provider* : il suffit pour cela de le rajouter à la table des *Service Providers*.

La table *Substrates Categories* permet de classer les *substrates* selon leur catégorie. Ainsi, la première entrée nous indique qu'il existe une catégorie nommée *Cores* identifiée par le numéro 10. Sa troisième colonne dont la valeur est NULL signifie en fait que *Cores* n'a pas de catégorie parente (la figure 2.8 indique bien que la troisième colonne désigne *ParentCategory_ID*). En revanche, la catégorie des jeux (*Games*) identifiée par 11 possède une catégorie parente qui est 10, soit *Cores*. L'idée est qu'en fait, les *Cores* peuvent inclure plusieurs catégories de logicielles, dont les jeux. En effet, nous pouvons imaginer des *Cores* pour d'autres applications, telles que la visio-conférence.

Dans la table des *Substrates*, la première entrée nous apprend qu'il existe un *substrate* nommé *TanksWarsSP* (un jeu de tank *Single Player*). Son id est 90 et sa version actuelle est 1.0. Le dernier chiffre 11, comme l'indique notre modèle d'information fait référence à la catégorie *Games*, comme l'indique la table *Substrates Categories*. La seconde entrée indique la présence d'un *substrate* dont l'id est 91, dont le nom est *Lightning Render* et dont la catégorie est 21. Ainsi, c'est un moteur de rendu puisque 21 correspond à la catégorie *Rendering Engines* d'après la table *Substrates Categories*.

La table *Substrates Dependencies* comporte trois entrées et indique que le *substrate* 90 (*TanksWarsSP*) est compatible avec trois *substrates* différents : 91 (*Lightning Render*), 92 (*Ultimate Physics*) et 93 (*Lightning Render Pro*). Ainsi pour pouvoir composer un service

viable articulé autour du jeu *TankWarsSP*, il faudra choisir au moins un *physics engine* (92, *Ultimate Physics* le seul proposé) mais faire un choix entre les deux *rendering engines* (91 et 93) proposés. Ceci est une règle implicite non indiquée par le modèle d'information : si un *substrate* dépend de deux *substrates* de la même catégorie, il faut en choisir un seul.

La table *Requirements* liste les exigences matérielles nécessaires pour héberger chaque *substrate*. Celle-ci ne doit pas être confondue avec la table *Requirements For Substrates* qui liste cette fois-ci les valeurs exigées par chaque *substrate* pour chaque exigence. Ainsi, la première entrée 90,NULL,0,0,2,3 indique que

1) le *substrate* 90 (TankWarsSP);
2) vis-à-vis de lui-même (deuxième champ à NULL);
3) pour le niveau de QoS 0 (*Low* d'après la table *QoSLevels*);
4) pour l'exigence 0 (CPU_CORES d'après la table *Requirements*);
5) nécessite une valeur supérieure (car 2 signifie *Higher Than* dans la table *Comparison Rule*);
6) à trois (cœurs de processeur, comme l'indiquait le quatrième point).

La deuxième entrée indique de même une exigence imposée par le *substrate* 90 vis-à-vis du *substrate* 91 (*Lightning Render*).

Dans la table *Compositions*, l'unique entrée indique qu'il existe une composition dont l'id est 78, effectuée par le *service provider* 56 portant le nom *TankWarsTournament2013*. Cette composition contient trois *substrates* listés dans la table *Composed Substrates*, et fait l'objet d'une unique réservation dans la table *Compositions Reservations*. Cette réservation indique l'utilisation d'un SLA dont l'id est 500, décrit dans la table SLAs. L'unique SLA indique que :

- une composition doit fonctionner avec le niveau 1 de QoS (*Medium* d'après la table *QoS Levels*);
- 1000 *end-users* sont prévus mais cela peut varier jusqu'à 1500;
- le coût souhaité peut varier entre 2000 et 2500;

- la localisation géographique (300, soit *Montréal* d'après la table *Geographic Location*) des *end-users* est indiquée dans le but de fournir un service avec la latence la plus faible.

La table *Offered Substrates* indique les *substrates* offerts par chaque *Substrates Provider*. Elle intègre également un champ permettant de calculer le coût en fonction :

- des ressources consommées durant la durée de réservation;
- du coût induit par la licence du *substrate*;
- etc.

La table *Offered Requirements for Offered Substrates* indique quelles sont les meilleures exigences offertes par les *substrates providers* pour chaque *substrate*. Certaines exigences comme la latence (id n°5 dans la table *Requirements*) peuvent dépendre de la localisation géographique des *end-users*, d'où la présence d'un champ *DataCenter_ID* qui va de pair avec la destination cible *TargettedGeographicLocation_ID*. Autrement dit, le *substrates provider* s'engage à fournir une latence avec une limite maximum pour une destination donnée.

La table *Substrates Reservations* indique pour chaque réservation quels sont les *substrates* réservés (à ne pas confondre avec la table *Composed Substrates* qui indique uniquement les modèles des *substrates* sélectionnés pour la même composition), ainsi que les *substrates providers* concernés par leur hébergement.

Enfin, la table *EndPoints Information* permet de stocker la localisation de chaque instance d'un *substrate* en cours d'exécution afin de permettre la communication. Dans notre exemple, l'approche retenue est très simple : elle consiste à stocker l'adresse IP et le port de chaque *substrate*.

2.4 Conclusion du chapitre

Dans ce chapitre, nous avons présenté une architecture permettant de gérer des jeux virtualisés orientés *substrates*. Dans le prochain chapitre, nous chercherons à étudier les conséquences d'une telle approche distribuée à travers une implémentation réelle de jeu vidéo. Nous allons donc présenter un prototype basé sur le concept des *substrates,* puis nous étudierons ses performances, que l'on cherchera à comparer à son homologue monolithique.

CHAPITRE 3

PROTOTYPE TEMPS RÉEL

3.1 Objectifs

Dans ce chapitre, nous expliquons tout d'abord notre approche pour la conception d'un prototype de jeu vidéo, développé selon deux paradigmes. La première version dite monolithique suivra le paradigme traditionnel, où tous les modules résident au sein d'un même processus, tandis que la seconde reposera sur notre approche orientée *substrates*, où les modules seront distribués entre plusieurs machines virtuelles. A travers divers scénarios de tests, nous chercherons à étudier l'impact des paradigmes sur les performances du prototype, dans l'objectif de comparer leur efficacité.

3.2 Définition du principe d'un *benchmark*

Développer un jeu vidéo est très souvent un travail très exigent et qui requiert de l'expérience. Selon la complexité du jeu et la finition recherchée, les objectifs fixés dans un cahier de charge peuvent rapidement augmenter le temps de développement.

Pour économiser du temps, nous avons choisi de développer un *benchmark* plutôt qu'un jeu vidéo. Un *benchmark* est une application qui utilise la grande majorité des modules d'un jeu vidéo, mais qui n'offre pas d'interaction avec un joueur. Son but est de s'exécuter en temps réel et relever les performances de son propre fonctionnement. Il existe de nombreux *benchmarks*, le plus célèbre étant 3DMark (Futuremark, 2013). Son objectif est de permettre aux joueurs d'obtenir un score synthétisant les capacités de leurs ordinateurs.

Par ailleurs, il est important de noter que nos prototypes valideront uniquement une des trois étapes du *workflow* du *cloud gaming*. Pour rappel, ce dernier peut être décomposé simplement en trois étapes qui s'exécutent en parallèle:

1) transmission des commandes du joueur vers les serveurs de traitement, via Internet;

2) interprétation des commandes, mise à jour de l'état du jeu et génération du rendu au niveau des serveurs de traitement contenant le jeu;
3) transmission du résultat audio-visuel dans un flux vidéo, des serveurs de traitement vers le terminal du joueur.

Si nous avions voulu développer un prototype utilisant les trois étapes, la première aurait demandé des efforts considérables, car elle nécessite le développement d'un client doté d'une intelligence artificielle (pour remplacer la présence d'un joueur humain). La troisième étape est moins difficile mais faute de temps, nous n'avons pas trouvé une librairie de *streaming* en C++ bien documentée, permettant d'encoder une suite d'images en un flux vidéo pour le transmettre sur un réseau.

Toutefois, le fait de n'avoir pas pris en compte les étapes 1 et 3 n'est pas problématique. En effet, comparé au paradigme monolithique, le paradigme des *substrates* n'a aucun effet sur les étapes 1 et 3. Il modifie uniquement l'étape 2 puisqu'il touche au fonctionnement interne d'un jeu vidéo. Pour résumer, la latence apportée par les étapes 1 et 3 impacteront de la même façon le paradigme monolithique et le paradigme des *substrates*.

3.3 Principe de fonctionnement

Notre *benchmark*, baptisé *Bouncing Balls*, consiste à simuler la chute de plusieurs balles sur un sol. Il utilise trois modules :

- un *core*, contenant la logique de l'application;
- un moteur de rendu (*rendering engine*), chargé d'effectuer le rendu de la scène;
- un moteur physique (*physics engine*), chargé d'effectuer les tâches liées à la physique. Dans notre cas il s'agira de simuler les balles soumises à leur propre poids.

Évidemment, pour comparer l'approche orientée *substrates* à l'approche traditionnelle qui est monolithique, nous avons conçu deux version de *Bouncing Balls* : SOBB (*Substrates*

Oriented Bouncing Balls) et MOBB (*Monolithic Oriented Bouncing Balls*). Dans le cas de SOBB, les trois modules précédemment cités sont implémentés sous forme de *substrates*.

Pour obtenir les meilleures performances, les deux versions du prototype ont été programmées en C++, langage de bas niveau, utilisé par la grande majorité de développeurs de jeux (notamment commerciaux). Contrairement à Java ou encore Python, il offre moins de facilités et s'avère par moments plus difficile d'usage : pas de *garbage collector*, pas de *duck typing*, etc. Toutefois, il permet de générer des exécutables très rapides.

Pour accélérer et simplifier la phase de développement, nous avons utilisé de nombreuses bibliothèques offertes par le *framework* C++ libre Qt 4 (Qt Project, 2013). D'autre part, notre moteur de rendu est basé sur un autre nommé Irrlicht Engine (Irrlicht Engine, 2013), également développé en C++. Du côté du moteur physique, nous n'avons pas utilisé de moteur physique existant étant donné la simplicité de notre prototype. Nous nous sommes contentés de résoudre l'équation différentielle (1.2) pour simuler la chute des balles soumises à la gravité en utilisant la méthode d'Euler améliorée (également appelée méthode de Heun). Dans notre cas, nous avons considéré un système physique idéal sans perte d'énergie, où les balles rebondissent à l'infini. Enfin, la figure 3.1 donne un aperçu du prototype.

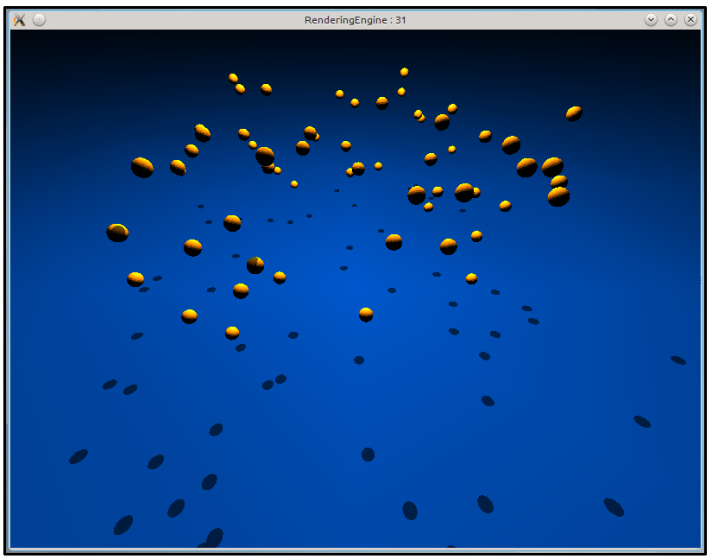


Figure 3.1 Capture d'écran du prototype

3.3.1 *Workflow* de l'application

La figure 3.2 présente les étapes suivies pour effectuer une itération de notre *workflow* :

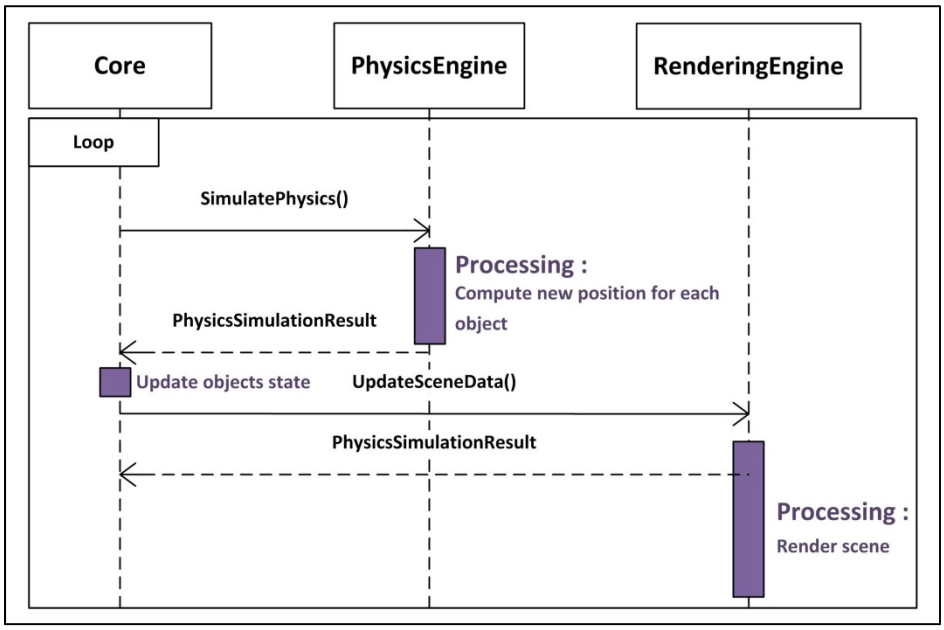


Figure 3.2 *Workflow* de l'application

Ainsi, on constate que :

1) le *core* commence par envoyer une commande nommée *SimulatePhysics* au moteur physique;
2) le moteur physique calcule la future position des balles pour un intervalle de temps donné;
3) le moteur physique répond au *core* avec le résultat de la simulation. Ce dernier contient les nouvelles coordonnées des balles;
4) le *core* envoie un message au moteur de rendu (*UpdateSceneData*), lui indiquant de générer une nouvelle image. Ce message contient également les instructions permettant de mettre à jour les objets. Dans notre cas, il s'agit évidemment des nouvelles coordonnées des balles;
5) s'il s'agit de la première itération du *workflow* (numéro 0), le moteur de rendu met à jour les propriétés de ses objets grâce au contenu de la requête *UpdateSceneData*, répond au

core par *SceneDataUpdated* et commence à générer immédiatement un rendu de la scène. S'il s'agit d'une itération n, avec n > 0, le moteur de rendu doit d'abord terminer de générer l'image n-1, si ce n'est pas encore le cas. Puis il met à jour son état grâce au contenu de la requête *UpdateSceneData*, répond par *SceneDataUpdated* puis commence à générer l'image de l'itération n.

3.3.2 Architecture du prototype monolithique (MOBB)

L'architecture de MOBB est la plus simple des deux prototypes. Les trois modules (*core*, moteur de rendu et moteur physique) sont implémentés sous forme de processus légers, ou *threads*. Ces derniers communiquent entre eux via des files d'attentes à fil sécurisé (*thread safe queues*), pour éviter les problèmes de concurrence. Comme la communication entre deux modules doit être bidirectionnelle, deux files d'attentes sont nécessaires à chaque fois. Dans notre cas, nous avons utilisé la bibliothèque *Intel Thread Building Blocks*, qui nous fournit une implémentation efficace de file d'attente à fil sécurisé. Enfin, MOBB se résume à un processus. La figure 3.3 illustre son architecture.

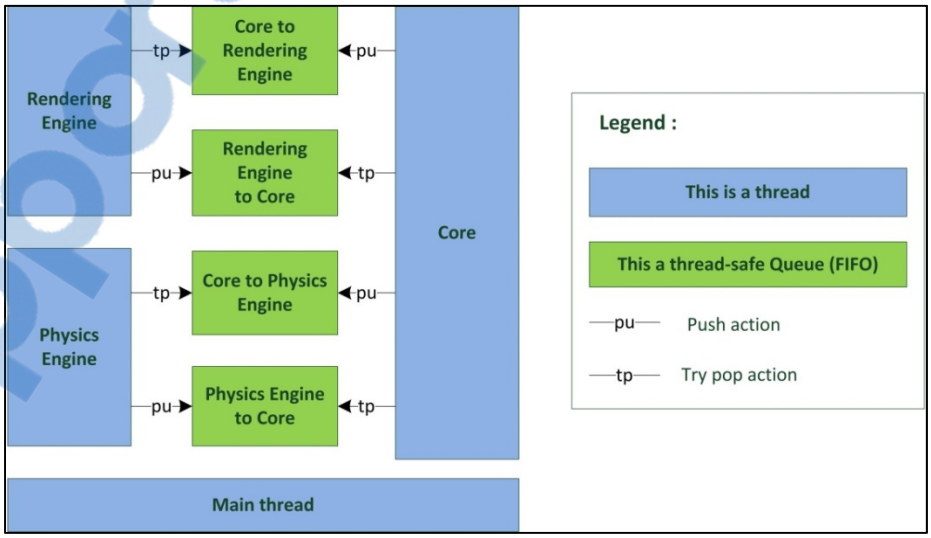


Figure 3.3 Architecture du prototype monolithique

3.3.3 Architecture du prototype orienté *substrates* (SOBB)

L'architecture de SOBB est un peu plus complexe. En effet, les trois modules du *benchmark* doivent être capables de communiquer via un réseau. Ainsi, chaque module communique d'abord avec un intermédiaire nommé *connector*, chargé de la transmission et de la réception des requêtes via le réseau. Plutôt que d'utiliser le protocole TCP, nous avons fait le choix d'utiliser enet (enet, 2013), une petite bibliothèque offrant de la fiabilité par-dessus le protocole UDP. Conçue à la base pour les jeux vidéo multi-joueurs, elle permet d'améliorer les performances en réduisant sensiblement la latence des messages transmis. Cette fois-ci, chacun des trois modules est implémenté sous forme de processus, eux-mêmes contenant plusieurs processus légers. SOBB est donc constitué de trois processus. La figure 3.4 illustre cette architecture.

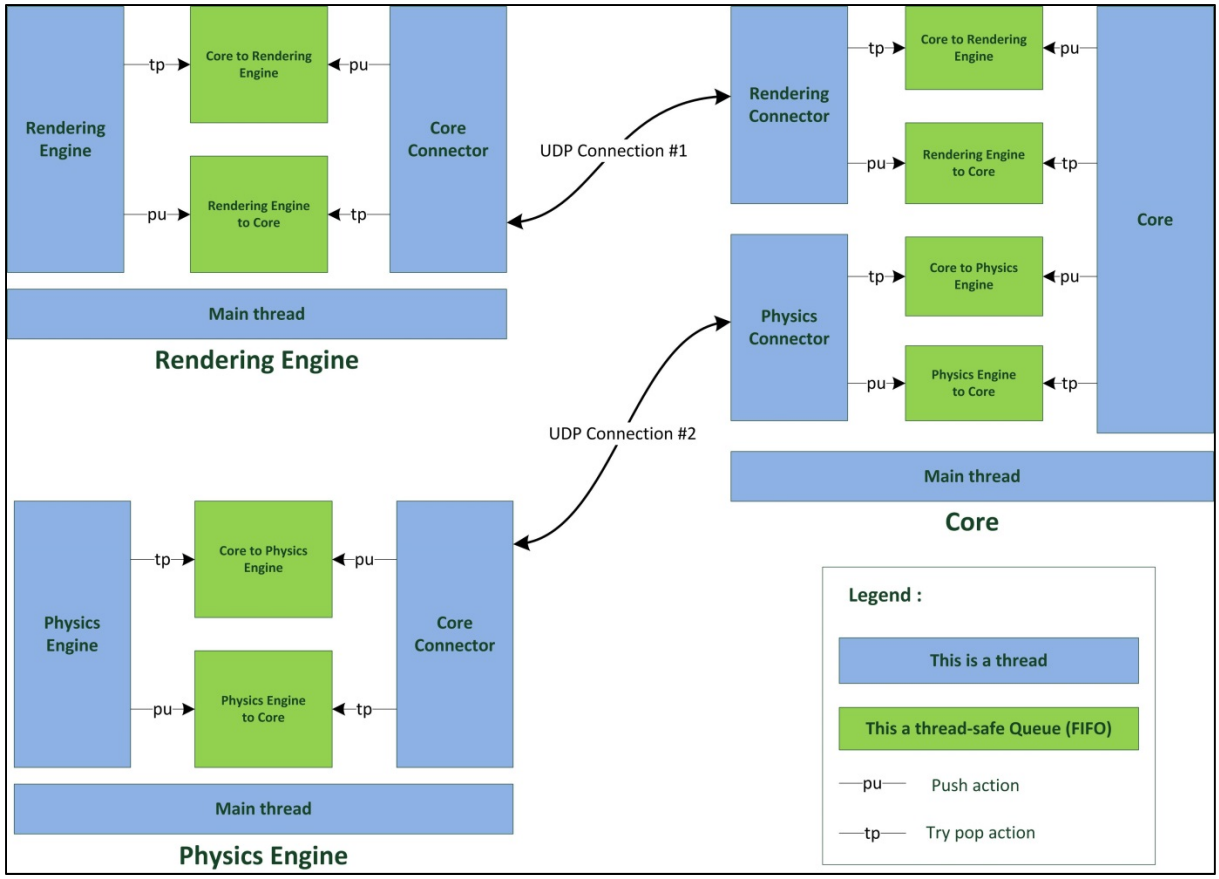


Figure 3.4 Architecture du prototype orienté *substrates*

3.4 Scénarios de tests de performances

Dans cette section, nous allons soumettre nos deux versions du prototype à divers conditions expérimentales afin d'observer leur comportement. Nous n'allons évidemment modifier qu'un seul paramètre à la fois pour étudier son impact sur les performances.

3.4.1 Tests sans virtualisation

3.4.1.1 Protocole expérimental

Dans notre scénario, nous n'utilisons qu'un seul ordinateur dont les caractéristiques sont indiquées dans le tableau 3.1.

Tableau 3.1 Configuration de test du prototype

CPU	RAM	GPU	OS
Intel Core i7 3820	16 GB	NVIDIA GeForce GTX660	Ubuntu 12.04 LTS

Dans nos résultats, nous définissons les métriques suivantes, mesurées du point de vue du *core* :

- **MLIT** ou *Main Loop Iteration Time*, qui est la durée nécessaire pour effectuer une itération du *workflow*. Son inverse fournit le nombre d'images par secondes ou FPS;
- **TSP** ou *Time Spent with Physics* défini par la différence $t_{PhysicsSimulationResult} - t_{SimulatePhysics}$. C'est le temps gaspillé par le *core* en attendant la réponse du moteur physique;
- **TSR** ou *Time Spent with Rendering* défini par la difference $t_{SceneDataUpdated} - t_{UpdateSceneData}$. C'est le temps gaspillé par le *core* en attendant la réponse du moteur de rendu.

La mesure du TSP est la somme de trois temps : le temps de traitement du moteur physique et les temps de transmission des messages *SimulatePhysics* (requête) et *PhysicsSimulationResults* (réponse). En revanche, le TSR n'est pas réellement la somme de

trois temps. En effet, le moteur de rendu peut fonctionner parallèlement au moteur physique. Nous aurions pu choisir d'envoyer la réponse *SceneDataUpdated* uniquement après la fin de la génération de l'image par le moteur de rendu (il aurait fallu la renommer plus correctement en *SceneRendered*), plutôt que de l'envoyer juste après la mise à jour des objets, mais cela aurait réduit les performances. En effet, notre objectif est d'essayer d'augmenter le parallélisme entre les modules pour réduire les temps de traitement. Enfin, nos mesures sont réalisées sur 5000 itérations du *workflow*, soit l'équivalent de 5000 générations d'images successives, ou encore 5000 échantillons de MLIT. Cette valeur a été choisie arbitrairement car nous avons constaté que le MLIT fluctuait très peu en fonction du temps, comme en témoigne l'écart type indiqué ci-après dans les résultats.

3.4.1.2 Résultats de MOBB et SOBB en conditions normales

D'après la figure 3.5, on constate que MOBB fournit des résultats très satisfaisants. Avec un MLIT moyen de 11.66 ms, 2.8 fois inférieur à la limite maximale de 33.33 ms, on atteint un nombre moyen d'images par secondes de 1/11.6e-3 = 85, soit 180% supérieur aux 30 FPS minimum pour garantir une fluidité de l'animation.

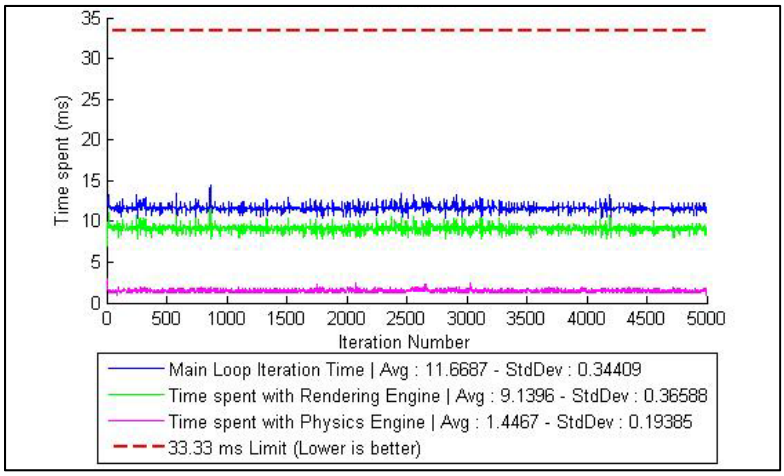


Figure 3.5 Résultats de performances de MOBB

Dans le cas de SOBB cette fois-ci, la figure 3.6 nous montre que ce dernier s'en tire moins bien que MOBB, bien que les performances demeurent très satisfaisantes. Le MLIT passe de 11.66 ms à 13.15 ms cette fois-ci, soit une augmentation de 12.7% du temps de traitement. Ainsi, le nombre moyen d'images par seconde est de 76, soit 153% supérieur à la limite minimum de 30 FPS.

Par ailleurs, le TSP passe de 1.44 ms à 5.91 ms, ce qui est normal étant donné l'*overhead* rajouté par les *connectors*. Pour rappel, SOBB dispose d'une architecture plus lourde en termes d'intermédiaires de communication. La bibliothèque enet fait de son mieux pour transmettre les messages en UDP, le plus tôt possible. Le fait qu'elle communique à travers une interface virtuelle, à savoir la boucle locale du système d'exploitation, dont la latence moyenne mesurée est de 0.025 ms, constitue déjà un avantage, par rapport à une interface physique, comme Ethernet.

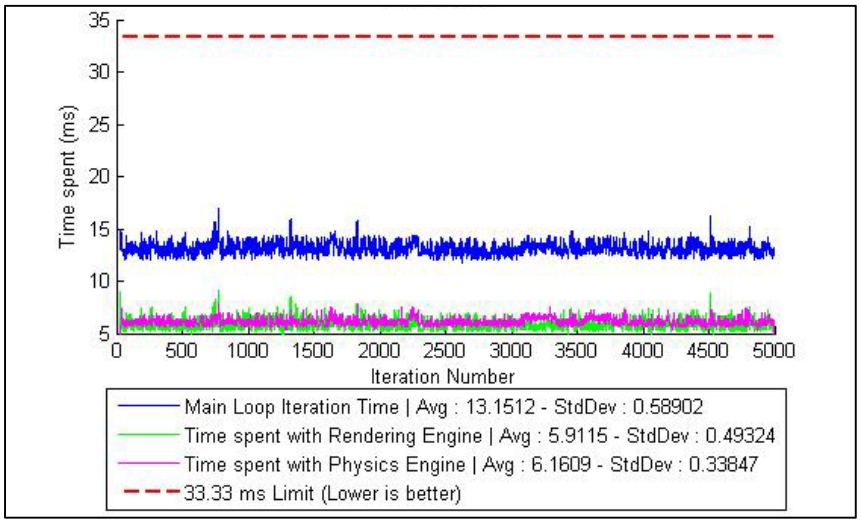


Figure 3.6 Résultats de performances de SOBB

Toutefois, on constate un résultat assez surprenant à première vue : comparé à MOBB, le TSR diminue de 9.13 à 5.91 ms. Attention, cela ne veut pas dire que le temps de traitement du moteur de rendu a diminué. Les temps de traitements du moteur de rendu et du moteur physique demeurent constants en moyenne avec de très faibles variations (nous l'avons

mesuré mais cela n'apparait pas sur les figures). Pour comprendre cette diminution, il faut revenir au *workflow,* qui pour rappel vise à augmenter le parallélisme entre les modules au lieu de garder des phases entièrement séquentielles. Lorsque le *core* en est à l'itération n, l'augmentation du TSP dans le cas de SOBB laisse plus de temps au moteur de rendu pour terminer de générer la frame n-1. A la réception de la requête *UpdateSceneData* pour la frame n, ce dernier a eu plus de temps pour progresser dans le traitement de la frame n-1, il répond donc plus rapidement avec *SceneDataUpdated*. Le TSR mesure la différence $t_{SceneDataUpdated} - t_{UpdateSceneData}$ du point de vue du core, il est donc normal qu'il rétrécisse en passant de MOBB à SOBB.

3.4.1.3 Influence de la latence du réseau

Comme nous l'avions vu dans la partie architecture de ce mémoire, la latence qui influence les temps de communications entre les modules peut avoir un effet négatif sur la qualité de service du prototype, si elle n'est pas maîtrisée. C'est ce que l'on se propose d'étudier dans cette partie.

L'influence de la latence du réseau n'a évidemment pu être testée que sur SOBB, puisqu'il s'agit de la seule version distribuée du prototype. Ses modules communiquent toujours à travers la boucle locale, mais nous avons utilisé cette fois-ci un outil, nommé tc (pour *traffic control*) permettant d'altérer la latence, qui est quasiment nulle à l'origine. Voici un exemple de la commande utilisée : tc qdisc add dev lo root netem delay 1ms.

Dans cette commande, qdisc signifie *queuing discipline*, dev signifie *device*, lo se refère à *loopback* et netem signifie *net emulator*. Cet exemple montre l'addition d'un délai de 1 ms sur la boucle locale du système d'exploitation. Le RTT (*Round Time Trip*) ou la valeur retournée par un ping est donc le double, soit 2 ms.

D'après la figure 3.7 qui présente les conséquences d'une augmentation de la latence, on constate qu'au-delà d'un délai de 4 ms, le MLIT commence à passer au-dessus de la limite

maximale de 33.33 ms. Dans ces conditions, SOBB ne peut plus fournir des performances de simulation décentes. A partir de 5 ms de délai, les balles ne se déplaçaient plus de manière fluide et nous avons pu observer une animation saccadée.

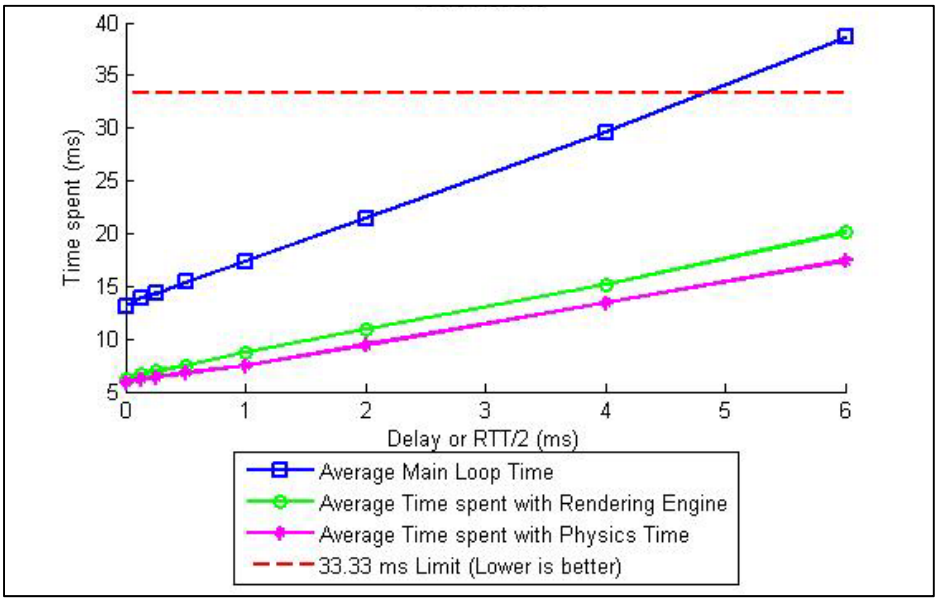


Figure 3.7 Influence de la latence du réseau
sur les performances de SOBB

3.4.1.4 Influence de la quantité de modules

Outre la latence du réseau qui est un paramètre critique à prendre en compte dans notre approche orientée *substrates*, nous devons aussi étudier l'influence de la quantité de modules sur les performances du prototype. En effet, dans notre cas nous n'avons considéré qu'une composition de trois modules, mais rien n'empêche de créer un jeu ayant plus de modules spécialisés

Développer des modules peut rapidement devenir chronophage. Nous aurions pu par exemple développer un module d'intelligence artificielle, permettant à un objet de traverser le champ de balles qui rebondissent en évitant toute collision. Toutefois, nous avons choisi une autre

approche très rapide à mettre en place : utiliser des requêtes factices à destination du moteur physique.

L'idée est simple : pour une même itération, on envoie une première requête dite efficace permettant de simuler la physique des balles. Puis nous envoyons n requêtes supplémentaires dites factices, avec un indicateur leur permettant d'être ignorées par le moteur physique. Le but est de simuler une augmentation des requêtes échangées entre les modules. Évidemment, dans le cas de requêtes efficaces, il y aurait un temps de traitement additionnel, ce qui n'est pas le cas avec les requêtes factices. Cela reste toutefois utile : on peut tester le prototype dans le cas le plus idéal où les temps de traitement sont négligeables. Ainsi, des requêtes supplémentaires efficaces ne pourraient qu'impacter plus négativement les performances et ne feront jamais mieux que les requêtes factices : nos résultats permettront ainsi de placer une limite d'efficacité maximale au meilleur des cas.

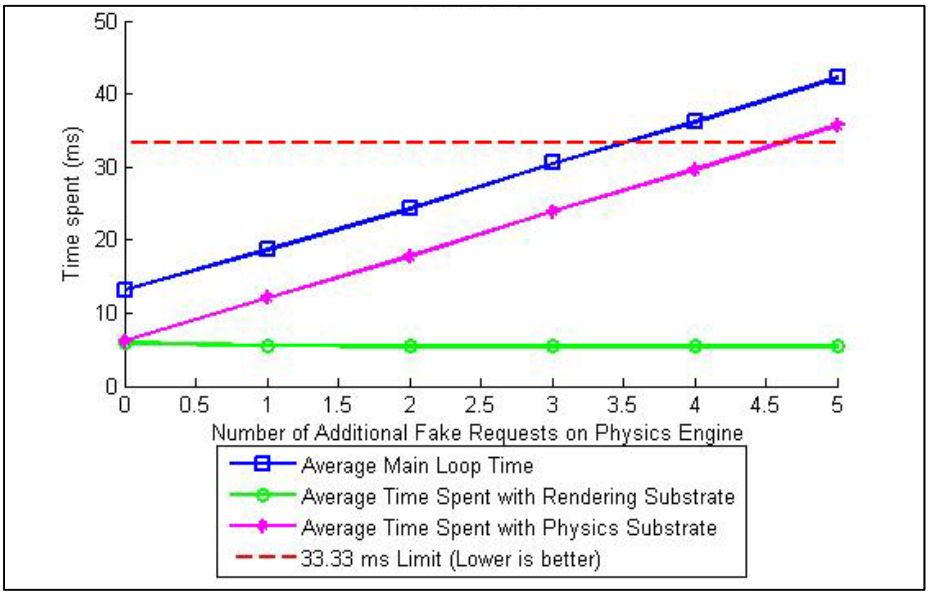


Figure 3.8 Influence de la quantité de *substrates*
sur les performances de SOBB

La figure 3.8 nous montre qu'à partir de trois requêtes factices supplémentaires (sans compter la requête efficace), les performances de SOBB se dégradent. Le TSR reste constant

tout au long de la simulation ce qui est normal. Toutefois, le TSP augmente étant donné que le *core* gaspille plus de temps en communiquant avec le moteur physique, à cause des requêtes factices supplémentaires.

3.4.2 Tests avec virtualisation

3.4.2.1 Protocole expérimental

Dans le scénario avec virtualisation, nous utilisons toujours le même ordinateur. Toutefois, nous avons choisi d'utiliser Windows 7 comme système d'exploitation hôte pour héberger Ubuntu 12.04 LTS comme système invité. Notre hyperviseur, VMWare Player 5.02, est une version allégée et gratuite de Workstation 9, logiciel à licence commerciale.

Les machines virtuelles peuvent utiliser jusqu'à 2 cœurs de processeurs en même temps, disposent de 2 Go de mémoire vive et sont connectées en réseau via un NAT virtuel. L'accélération graphique matérielle, très bien supportée depuis la version 5 de VMWare Player est évidemment activée, permettant l'accélération des traitements liés au moteur de rendu.

Tous nos tests sont effectués avec trois machines virtuelles lancées simultanément. Dans le cas de MOBB, bien qu'une seule VM ne soit nécessaire, nous conservons les deux autres allumées. Cela permet de mieux comparer les performances avec SOBB, qui dans un premier temps sera exécuté sur une même VM (et donc deux inutilisées), puis sur trois VMs, avec un *substrate* par VM.

3.4.2.2 Résultats de MOBB

La virtualisation apporte toujours un *overhead* plus ou moins important selon les cas, qui détériore les performances, comme l'indiquait Nae *et al.* (2009). Toutefois, les performances obtenues par MOBB en environnement virtualisé sont très satisfaisantes. On constate que le

MLIT moyen est de 13.08 ms, soit environ 76 FPS. Comparé au MOBB non virtualisé, dont le MLIT était de 11.66 ms, la baisse de performances n'est que de 10.9%.

Il faut noter que cette efficacité est atteinte grâce aux nombreux efforts entrepris par VMWare pour améliorer la gestion de l'accélération graphique matérielle (depuis août 2012 environ). Les tâches créées par le moteur de rendu nécessitant un processeur graphique peuvent désormais s'exécuter avec des conditions proches du natif, c'est-à-dire sans virtualisation. Précédemment, nous avions temporairement essayé l'hyperviseur VirtualBox d'Oracle (logiciel libre, contrairement à Player qui est uniquement gratuit), mais les performances étaient désastreuses (moins de 20 FPS en moyenne), à cause d'un support encore précaire de l'accélération graphique matérielle.

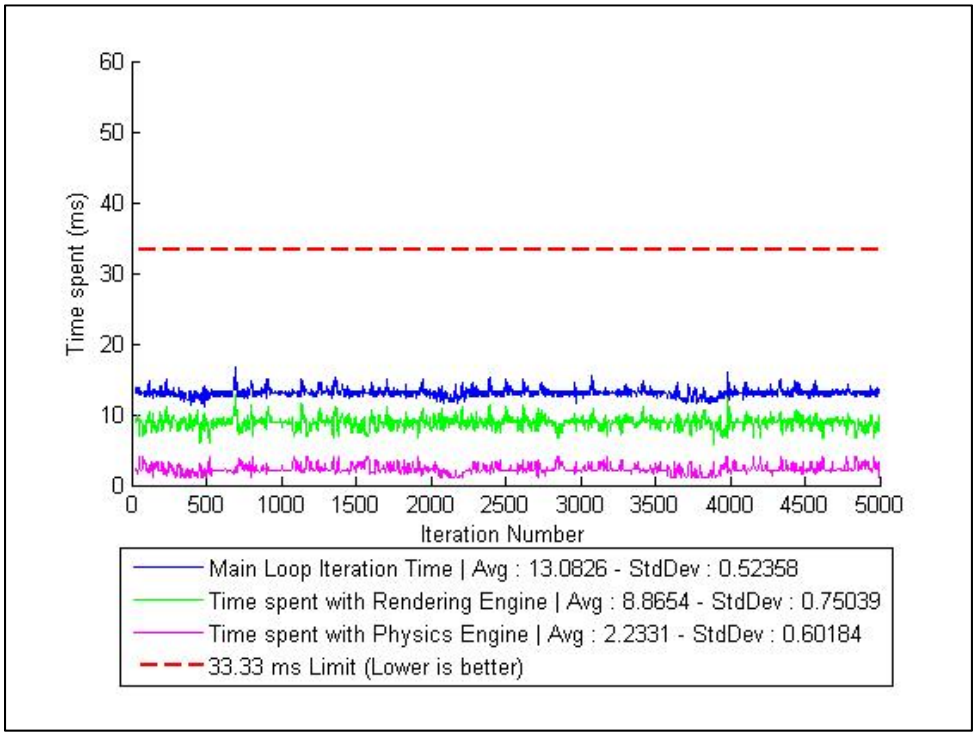


Figure 3.9 Résultats de performances de MOBB,
en environnement virtualisé

3.4.2.3 Résultats de SOBB

Lorsque SOBB est entièrement hébergé sur une seule VM, le MLIT moyen est de 22.55 ms, comme l'indique la figure 3.10. La latence moyenne, que nous avons mesuré avec un ping sur la boucle locale, était de 4e-3 ms, soit un délai de 2e-3 ms. Les performances de SOBB hébergé sur une VM sont toujours satisfaisantes, avec un MLIT inférieur à 33.33 ms, mais la dégradation comparée au MOBB virtualisé est de 42%.

On constate que cette augmentation est principalement due au TSP, qui passe de 2.23 ms pour le MOBB à 10.34 ms pour le SOBB hébergé sur une VM.

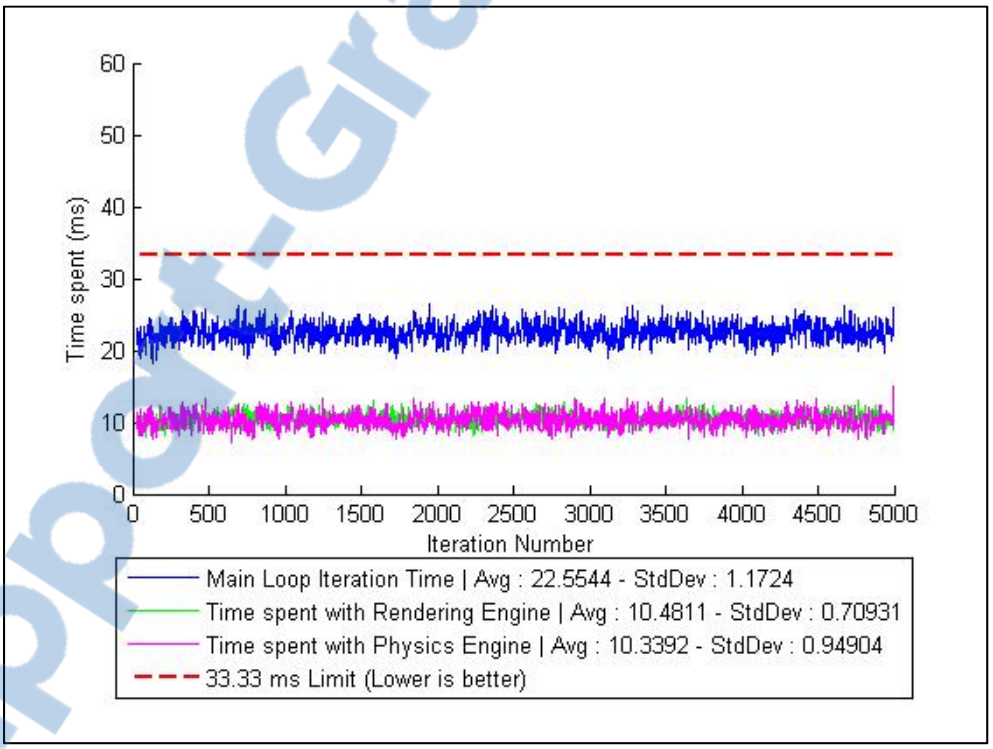


Figure 3.10 Résultats de performances de SOBB,
en environnement virtualisé,
avec les trois *substrates* sur la même VM

Enfin, comme le montre la figure 3.11, lorsque SOBB est exécuté avec un *substrate* par VM, les performances demeurent satisfaisantes mais se dégradent encore. Avec un MLIT moyen

de 27.11 ms, la baisse d'efficacité par rapport au SOBB entièrement hébergé par une VM est de 17%.

Cela est principalement dû à l'augmentation de la latence entre les modules. Ces derniers ne communiquent plus via la boucle locale, dont la latence moyenne mesurée était de 4e-3 ms mais à travers le réseau NAT virtualisé par Player, caractérisé par une latence moyenne de 2.5e-1 ms.

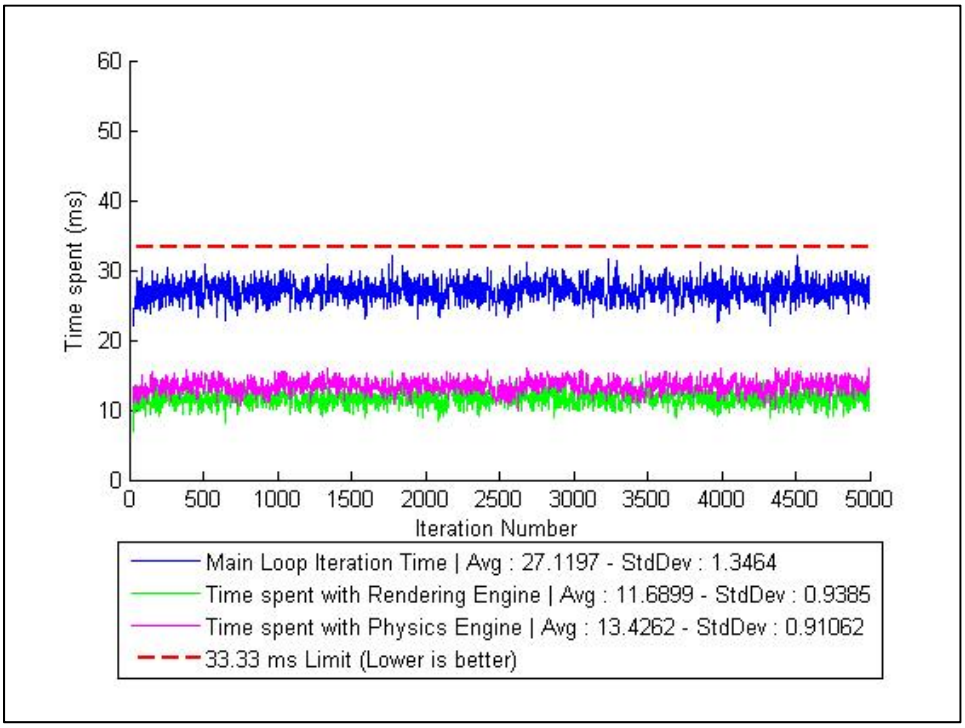


Figure 3.11 Résultats de performances de SOBB,
en environnement virtualisé, avec un *substrate* par VM

3.5 Conclusion du chapitre

Dans ce chapitre, nous avons comparé les performances d'un jeu vidéo développé selon l'approche classique qui est monolithique, avec notre approche orientée *substrates*. Nous avons expérimentalement démontré que l'approche monolithique surpasse la nôtre en termes de performances, car cette dernière rajoute une dépendance vis-à-vis de la latence du réseau. Enfin, nous avons constaté sans surprise que la virtualisation avait un impact négatif sur les performances, bien qu'elles demeurent acceptables. Dans le prochain chapitre, nous allons décrire notre algorithme de dimensionnement dynamique des ressources.

CHAPITRE 4

ALGORITHME DE DIMENSIONNEMENT DYNAMIQUE DE RESSOURCES PROPOSE

Dans ce chapitre, nous allons décrire notre algorithme de dimensionnement dynamique de ressources. Nous montrerons comment ce dernier permet de superviser une application orientée *substrates*, dans le but de maintenir une qualité de service constante, grâce à une allocation ou une libération de ressources. Nous décrirons également notre approche choisie pour garantir une consolidation au sein d'une infrastructure.

4.1 Objectifs et contributions

Les objectifs de notre algorithme de gestion de ressources sont de :

- tenter de maintenir un temps de réponse satisfaisant en s'adaptant aux besoins d'une application hébergée dans le *cloud*, afin qu'elle puisse s'exécuter dans les conditions spécifiées par son SLA;
- permettre d'ajuster de manière dynamique les ressources allouées aux machines virtuelles, en fonction des besoins de l'application;
- effectuer de la consolidation de serveurs en migrant les machines virtuelles et en éteignant les machines physiques inutilisées, dans le but de réduire la consommation énergétique.

Comparé à l'algorithme original proposé par Rui *et al.* (2012), notre contribution inclut :

- la prise en charge d'applications orientées *substrates* et non pas seulement multi-tiers;
- la possibilité de superviser plusieurs applications simultanément et non pas une seule;
- la migration de machines virtuelles pour deux scénarios : la consolidation mais aussi la possibilité d'allouer plus de ressources à une machine virtuelle dont la machine physique hôte est saturée;
- la définition d'un environnement orienté objet pour l'algorithme, plus explicite que les notations de l'article original;

4.2 Hypothèses de l'algorithme

Les hypothèses de notre algorithme permettent de définir certaines limitations. Tout d'abord, si dans notre architecture, nous avions vu que plusieurs *substrates* d'une même composition pouvaient être hébergés par différents *datacenters*, notre algorithme lui est conçu pour gérer des compositions où tous les *substrates* appartiennent au même *datacenter*.

De même, comme nous le verrons plus tard, si plusieurs applications sont en sous-approvisionnement, notre algorithme ne permet pas de les approvisionner de manière simultanée mais séquentielle.

Par ailleurs, tout comme Rui *et al.* (2012), notre algorithme est multidimensionnel et rajoute la supervision de la latence du réseau au sein d'une application. Toutefois, en raison des limitations de notre environnement de test décrit dans le dernier chapitre, notre algorithme ne sera testé qu'en fonction d'une seule dimension (le processeur), en raison des fonctionnalités limitées de notre simulateur.

Enfin, l'algorithme demeure relativement simple et n'utilise pas de fonction d'optimisation. Par exemple, il ne cherche pas à réduire le nombre de migrations, qui pourtant devrait être évité si possible. L'algorithme est ainsi agnostique de l'application et se base uniquement sur les temps de réponse des processus et les taux d'utilisation des ressources pour prendre des décisions.

4.3 Présentation de l'algorithme

La figure 4.1 illustre l'environnement orienté objet dans lequel notre algorithme est implémenté.

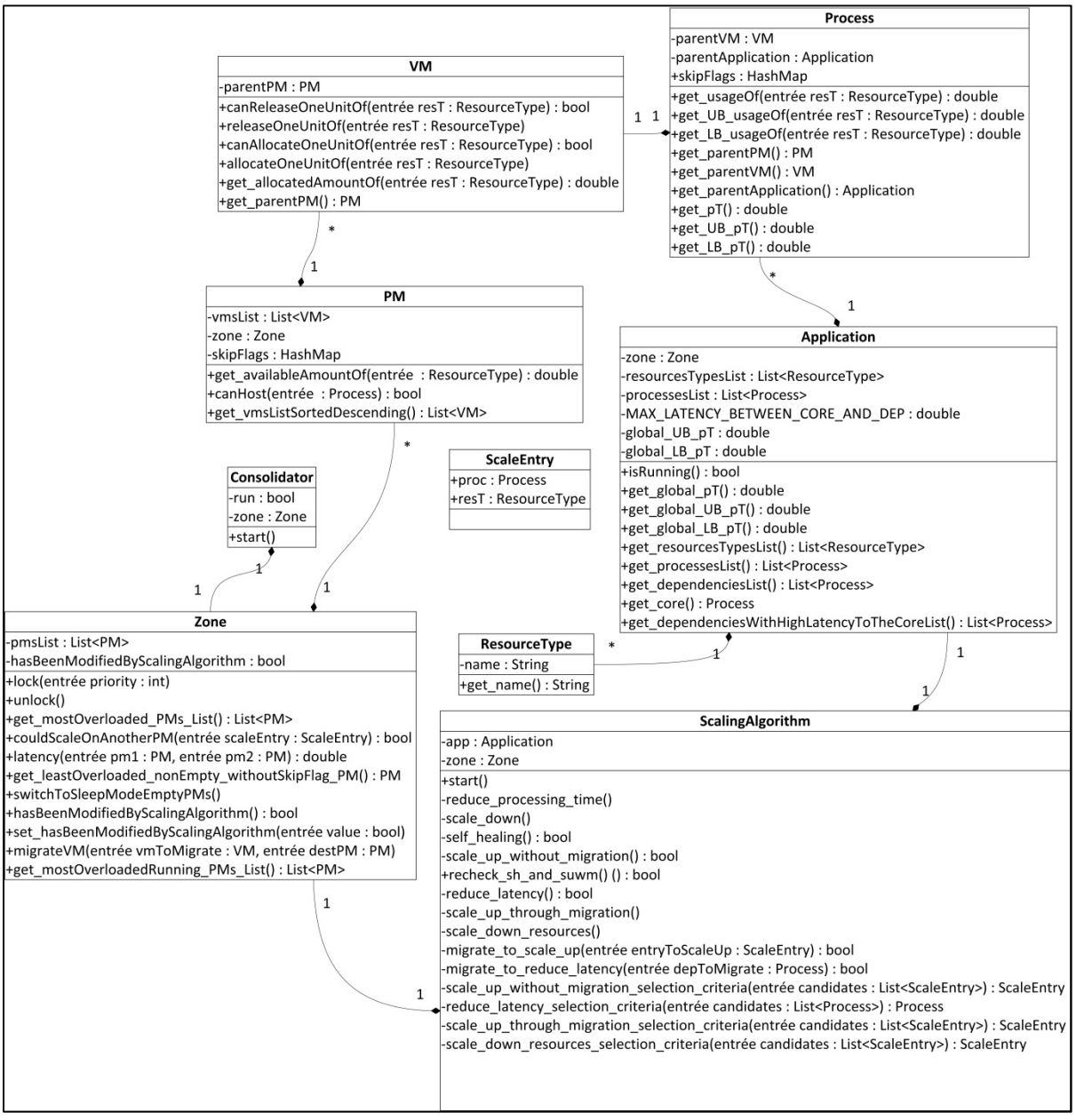


Figure 4.1 Modélisation UML de l'environnement de l'algorithme

Les classes PM, VM et Process permettent respectivement de modéliser une machine physique, une machine virtuelle et un processus.

La classe Application permet de modéliser une instance de composition (simplifiée avec l'appellation application), donc un ensemble de *substrates* regroupant un *core* et plusieurs *dependencies*. Dans notre cas, chaque *substrate* est une instance de la classe Process, hébergée sur une machine virtuelle différente.

La classe ScalingAlgorithm intègre l'ensemble des fonctionnalités de notre algorithme de dimensionnement dynamique. Pour chaque instance de la classe Application, une instance de la classe ScalingAlgorithm lui sera associée pour assurer la supervision.

La classe Zone définit une région dans un *datacenter,* qui consiste en un groupe de machines physiques. En effet, pour permettre à plusieurs machines virtuelles d'être modifiées et migrées, il est nécessaire de trouver une stratégie pour éviter des situations de concurrence. Par exemple, si une machine physique PM_1 dispose d'un seul cœur CPU de libre, et que trois machines virtuelles ($PM_1.VM_1$, $PM_1.VM_2$ et $PM_1.VM_3$) ont besoin d'un cœur supplémentaire au même instant, il faut sérialiser la demande pour éviter d'avoir une allocation impossible. Dans ce cas, la première instance de la classe ScalingAlgorithm associée à la première instance de la classe Application demandera le verrou de la zone, afin que l'algorithme soit le seul à la modifier et allouera un cœur CPU à la machine virtuelle qui en a besoin ($PM_1.VM_1$ par exemple). Ensuite, les autres instances de ScalingAlgorithm devront successivement obtenir le verrou de la zone pour tenter une migration vers une machine physique ayant au moins un cœur de CPU libre.

4.3.1 Pseudo-code sélectif

Dans cette partie, nous présentons les codes de notre algorithme ainsi que ceux d'autres méthodes que nous estimons pertinents. Pour avoir une description des algorithmes et des

méthodes qu'ils utilisent, il faut se référer à la documentation sélective fournie dans la partie 4.3.2. Celle-ci est indispensable pour comprendre le principe de fonctionnement.

4.3.1.1 Méthodes de la classe ScalingAlgorithm

Dans cette partie, nous nous concentrerons d'abord sur la définition de chaque méthode contenue dans la classe ScalingAlgorithm, qui pour rappel contient l'ensemble des mécanismes de notre algorithme de dimensionnement dynamique des ressources.

Le point d'entrée de notre algorithme est la méthode start, définie dans l'algorithme 4.1. En supervisant le temps de réponse d'une application donnée, la méthode start permet de déterminer s'il y a un sur ou un sous approvisionnement.

Algorithme 4.1 Classe ScalingAlgorithm - Méthode start

```
1.  while ( app.isRunning() ) :
2.          global_pT = app.get_global_pT()
3.          if      ( global_pT > app.get_global_UB_pT() ) :
4.                  zone.lock(0)
5.                  reduce_processing_time()
6.                  zone.unlock()
7.          else if ( global_pT < app.get_global_LB_pT() ) :
8.                  zone.lock(1)
9.                  scale_down()
10.                 zone.unlock()
```

Si la méthode start détecte un temps de réponse au-dessus du seuil supérieur de tolérance, la méthode reduce_processing_time définie dans l'algorithme 4.2 est appelée pour tenter de mettre fin au sous-approvisionnement. Celle-ci contient 4 méthodes (décrites plus tard) qui tentent de réduire le temps de traitement d'une application.

Algorithme 4.2 Classe ScalingAlgorithm - Méthode reduce_processing_time

```
1.  if ( self_healing() ) :
2.          return
3.  if ( scale_up_without_migration() ) :
4.          return
5.  if ( reduce_latency() ) :
6.          return
7.  scale_up_through_migration()
```

Si au contraire, la methode start détecte un temps de traitement en dessous du seuil inférieur, cela est probablement dû à un sur-approvisionnement. Pour réduire les coûts de consommation, elle appelle la méthode scale_down qui tente de libérer le surplus de ressources allouées.

Algorithme 4.3 Classe ScalingAlgorithm - Méthode scale_down

```
1.  scale_down_resources()
```

La méthode self_healing définie dans l'algorithme 4.4 est la première à être appelée par la méthode reduce_processing_time. Son but est de voir s'il est possible de mieux répartir les ressources entre les machines virtuelles. L'idée est que si un processus p1 a un temps de réponse et un taux d'utilisation pour une ressource r1 en dessous du seuil inférieur, et qu'un processus p2 a un temps de réponse et un taux d'utilisation pour une ressource r1 au-dessus du seuil supérieur de tolérance, on retire une unité de la ressource r1 de la machine virtuelle hébergeant p1, pour la donner à la machine virtuelle hébergeant p2. Evidemment, cela implique que les deux machines virtuelles résident sur la même machine physique.

Algorithme 4.4 Classe ScalingAlgorithm - Méthode self_healing

```
    return type    : bool
```

```
1.  problemSolved = false
2.  toBalance = List<VM>()
3.  resourcesTypesList = app.get_resourcesTypesList()
4.  processesList = app.get_processesList()
5.  breakAndCheck = false
6.  do :
7.          toBalance.clear()
8.          breakAndCheck = false
9.          foreach ( permutationOfPair(proc1,proc2) in processesList ) :
10.                 foreach ( resT in resourcesTypesList ) :
11.                         if      ( proc1.get_UsageOf(resT) <
    proc1.get_LB_UsageOf(resT)
12.                                 AND proc1.get_pT() < proc1.get_LB_pT()
13.                                 AND proc2.get_UsageOf(resT) >
    proc2.get_UB_UsageOf(resT)
14.                                 AND proc2.get_pT() > proc2.get_UB_pT()
15.                                 AND proc1.get_parentPM() == proc2.get_parentPM()
16.                                 AND proc1.get_parentVM().canReleaseOneUnitOf(resT)
17.                                 ) :
18.                                 toBalance[0] = proc1.get_parentVM()
19.                                 toBalance[1] = proc2.get_parentVM()
20.                                 toBalance[0].releaseOneUnitOf(resT)
21.                                 toBalance[1].allocateOneUnitOf(resT)
22.                                 zone.set_hasBeenModifiedByScalingAlgorithm(true)
23.                                 breakAndCheck = true
24.                                 break
25.                 if( breakAndCheck ) :
26.                         break
27.         problemSolved = app.get_global_pT() < app.get_global_UB_pT()
```

```
28. while (     app.isRunning()
29.         AND  !toBalance.isEmpty()
30.         AND  !problemSolved
31.         )
32. return problemSolved
```

La seconde méthode appelée par la méthode reduce_processing_time est scale_up_without_migration. Celle-ci est décrite dans l'algorithme 4.5. Son but est de tenter d'allouer des ressources à un processus dont le temps de traitement et le taux d'utilisation pour une ressource donnée sont au-dessus du seuil supérieur de tolérance. Cette méthode n'est effective que si la machine physique hôte dispose de ressources libres.

Algorithme 4.5 Classe ScalingAlgorithm - Méthode scale_up_without_migration

```
    return type    : bool

1.  problemSolved = false
2.  toScaleUp = List<ScaleEntry>()
3.  resourcesTypesList = app.get_resourcesTypesList()
4.  processesList = app.get_processesList()
5.  do :
6.          toScaleUp.clear()
7.          foreach ( proc in processesList ) :
8.                  if ( proc.get_pT() > proc.get_UB_pT() ) :
9.                          foreach ( resT in resourcesTypesList ) :
10.                                 if ( proc.get_UsageOf(resT) >
    proc.get_UB_UsageOf(resT)
11.                                 AND proc.get_parentVM().canAllocateOneUnitOf(resT)
12.                                 ) :
13.                                 toScaleUp.append(ScaleEntry(proc, resT))
```

```
14.         if ( !toScaleUp.isEmpty() ) :
15.                 entryToScaleUp = scale_up_without_migration_selection_criteria(
    toScaleUp )
16.                 entryToScaleUp.proc.get_parentVM().allocateOneUnitOf(resT)
17.                 zone.set_hasBeenModifiedByScalingAlgorithm(true)
18.         problemSolved = app.get_global_pT() < app.get_global_UB_pT()
19. while   (       app.isRunning()
20.         AND     !toScaleUp.isEmpty()
21.         AND     !problemSolved
22.         )
23. return problemSolved
```

Méthode antagoniste de scale_up_without_migration, scale_down_resources décrite dans l'algorithme 4.6 libère des ressources allouées aux processus ayant un temps de traitement en dessous du seuil inférieur.

Algorithme 4.6 Classe ScalingAlgorithm - Méthode scale_down_resources

```
1.  toScaleDown = List<ScaleEntry>()
2.  resourcesTypesList = app.get_resourcesTypesList()
3.  processesList = app.get_processesList()
4.  do :
5.          toScaleDown.clear()
6.          foreach ( proc in processesList ) :
7.                  if ( proc.get_pT() < proc.get_LB_pT() ) :
8.                          foreach ( resT in resourcesTypesList ) :
9.                                  if ( proc.get_UsageOf(resT) <
    proc.get_LB_UsageOf(resT)
10.                                 AND   proc.get_parentVM().canReleaseOneUnitOf(resT)
11.                                 ) :
```

```
12.                                   toScaleDown.append( ScaleEntry(p, resT) )
13.         if ( !toScaleDown.isEmpty() ) :
14.                 entryToScaleDown = scale_down_selection_criteria( toScaleDown )
15.                 entryToScaleDown.proc.get_parentVM().releaseOneUnitOf(resT)
16.                 zone.set_hasBeenModifiedByScalingAlgorithm(true)
17.         problemSolved = app.get_global_pT() > app.get_global_LB_pT()
18. while   (       app.isRunning()
19.         AND     !toScaleDown.isEmpty()
20.         AND     !problemSolved
21.         )
```

La méthode recheck_sh_and_suwm décrite dans l'algorithme 4.7 permet de réaliser un self_healing, suivi d'un scale_up_without_migration en cas d'échec. Celle-ci est utilisée au sein des méthodes reduce_latency ainsi que scale_up_through_migration.

Algorithme 4.7 Classe ScalingAlgorithm - Méthode recheck_sh_and_suwm

```
1.  if ( self_healing() ) :
2.          return true
3.  if ( scale_up_without_migration() ) :
4.          return true
5.  return false
```

La troisième méthode à être appelée par reduce_processing_time est reduce_latency. Celle-ci est en troisième position car elle peut occasionner des migrations de machines virtuelles, que l'on souhaite éviter à cause des inconvénients listés dans le chapitre 1. Son objectif est de superviser la latence qui sépare la machine virtuelle hébergeant le *core* d'une application, et les machines virtuelles qui hébergent les *dependencies*. Si cette latence est au-dessus d'un certain seuil fixé, reduce_latency sélectionne une des machines virtuelles concernées et hébergeant les *dependencies,* puis tente de la migrer vers une machine physique ayant une latence plus faible vis-à-vis de la machine physique hébergeant le *core*.

Algorithme 4.8 Classe ScalingAlgorithm - Méthode reduce_latency

```
     return type    : bool

1.   problemSolved = false
2.   candidateDeps = List<Process>()
3.   resourcesTypesList = app.get_resourcesTypesList()
4.   processesList = app.get_processesList()
5.   migrationDone = false
6.   do :
7.           if ( migrationDone ) :
8.                   if ( recheck_sh_and_suwm() ) :
9.                           break
                     migrationDone = false
10.          candidateDeps.clear()
11.          candidateDeps = app.get_DependenciesWithHighLatencyToTheCore ()
12.          if ( !candidateDeps.isEmpty() ) :
13.                  depToMigrate = reduce_latency_selection_criteria(candidateDeps)
14.                  if ( migrate_to_reduce_latency(depToMigrate) ) :
15.                          migrationDone = true
16.                  problemSolved = app.get_global_pT() < app.get_global_UB_pT()
17.  while  (       app.isRunning()
18.          AND   !candidateDeps.isEmpty()
19.          AND   !problemSolved
20.          )
21.  foreach ( proc in processesList ):
22.          proc.skipFlags["migrateToReduceLatency"] = false
23.  return problemSolved
```

La méthode migrate_to_reduce_latency est utilisée par la méthode reduce_latency pour sélectionner une machine physique de destination, puis y migrer la machine virtuelle concernée.

Algorithme 4.9 Classe ScalingAlgorithm - Méthode migrate_to_reduce_latency

```
     input          : Process depToMigrate
     return type    : bool

1.   mOPMsList = zone.get_mostOverloaded_PMs_list()
2.   core = app.get_core()
3.
4.   foreach ( mOPM in mOPMsList ) :
5.         if       ( mOPM.canHost(depToMigrate)
6.                  AND zone.latency( depToMigrate.get_parentPM(), core.get_parentPM()
     ) > zone.latency( mOPM, core.get_parentPM() )
7.                  ) :
8.                  zone.migrate( depToMigrate.get_parentVM(), mOPM )
9.                  zone.set_hasBeenModifiedByScalingAlgorithm(true)
10.                 return true
11.  depToMigrate.skipFlags["migrateToReduceLatency"] = true
12.  return false
```

Enfin, la dernière méthode appelée en cas de sous-approvisionnement est scale_up_through_migration. Sa faible priorité (quatrième position) vient du fait qu'elle peut occasionner des migrations et des allocations de ressources. En effet, si une machine virtuelle mv1 nécessite des ressources, mais que sa machine physique hôte est saturée, l'idée est de voir si l'on peut allouer des ressources supplémentaires à mv1 en la migrant vers une autre machine physique ayant des ressources disponibles.

Algorithme 4.10 Classe ScalingAlgorithm - Méthode scale_up_through_migration

```
    return type    : bool

1.  toScaleUp = List<ScaleEntry>()
2.  resourcesTypesList = app.get_resourcesTypesList()
3.  processesList = app.get_dependenciesList()
4.  migrationDone = false
5.  do :
6.          if ( migrationDone ):
7.                  if ( recheck_sh_and_suwm() ) :
8.                          break
9.                  migrationDone = false
10.         toScaleUp.clear()
11.         foreach ( proc in processesList ) :
12.                 if ( proc.get_pT() > proc.get_UB_pT() ) :
13.                         foreach ( resT in resourcesTypesList ) :
14.                                 if      (       proc.get_UsageOf(resT) >
                                                        proc.get_UB_UsageOf(resT)
15.                                         AND     zone.couldScaleUpOnAnotherPM(
                                                        ScaleEntry(proc, resT) )
16.                                 ) :
17.                                 toScaleUp.append( ScaleEntry(proc, resT) )
18.         if ( !toScaleUp.isEmpty() ) :
19.                 entryToScaleUp  =  scale_up_through_migration_selection_criteria(
20. toScaleUp )
21.                 if ( migrate_to_scale_up( entryToScaleUp ) ) :
22.                         migrationDone = true
23.                 problemSolved = app.get_global_pT() < app.get_global_UB_pT()
24. while   (       app.isRunning()
```

```
25.         AND   !toScaleUp.isEmpty()
26.         AND   !problemSolved
27.         )
28.   foreach ( proc in processesList ):
29.         foreach ( resT in resourcesTypesList ):
30.               proc.skipFlags[“migrate_to_scale_up”][resT] = false
```

La méthode migrate_to_scale_up décrite dans l’algorithme 4.11 est utilisée par scale_up_through_migration pour sélectionner une machine physique de destination, et y migrer la machine virtuelle concernée.

Algorithme 4.11 Classe ScalingAlgorithm - Méthode migrate_to_scale_up

```
      input          : ScaleEntry entryToScaleUp
      return type    : bool

1.    proc = entryToScaleUp.proc
2.    resT = entryToScaleUp.resT
3.    requiredAmountOfR = proc.get_parentVM().get_allocatedAmountOf(resT) +
      oneUnitOf(resT)
4.    mOPMsList = zone.get_most_overloaded_PMs_list()
5.    foreach ( mOPM in mOPMsList ) :
6.          if    ( mOPM.canHost(proc)
7.                AND mOPM.get_availableAmountOf(resT) >= requiredAmountOfR )) :
8.                zone.migrate(proc.get_parentVM(), mOPM)
9.                proc.get_parentVM().allocateOneUnitOf(resT)
10.               zone.set_hasBeenModifiedByScalingAlgorithm(true)
11.               return true
12.   depToMigrate.skipFlags[“migrateToScaleUp”][resT] = true
13.   return false
```

4.3.1.2 Méthodes de la classe Consolidator

La méthode start de la classe Consolidator est le point d'entrée de l'algorithme de consolidation. Cette méthode tente selon l'approche *best fit decreasing*, de migrer les machines virtuelles situées sur les machines physiques les moins utilisées, vers les machines physiques les plus utilisées. Dès qu'une machine physique est vidée, celle-ci est mise en veille.

Algorithme 4.12 Classe Consolidator - Méthode start

```
1.  while( run ) :
2.      zone.lock(1)
3.      if( zone.hasBeenModifiedByScalingAlgorithm() ):
4.          zone.set_consolidatorSkipFlagsOfEachPM_toFalse()
5.          zone.set_hasBeenModifiedByScalingAlgorithm(false)
6.      lOPM = zone.get_leastOverloaded_NonEmpty_WithoutSkipFlag_PM()
7.      vmsList = lOPM.get_VMsList_sortedDescending()
8.      mOPMs = zone.get_MostOverloadedRunningPMsList()
9.      atLeastOneVM_HasBeenMigrated = false
10.     foreach( vm in vmsList ):
11.         foreach( mOPM in mOPMs ) :
12.             if  ( mOPM.canHost(vm)
13.                 AND mOPM != vm.get_parentPM()
14.                 ) :
15.                 zone.migrate( vm, mOPM )
16.                 mOPMs = zone.get_MostOverloadedRunningPMsList()
17.                 atLeastOneVM_HasBeenMigrated = true
```

```
18.                                    break
19.          if( !atLeastOneVM_HasBeenMigrated ) :
20.                    mOPM.skipFlags["consolidator"] = true
21.          zone.switchToSleepModeEmptyPMs()
22.          zone.unlock()
```

4.3.1.3 Méthodes de la classe PM

Algorithme 4.13 Classe PM - Méthode canHost

```
         input          : Process proc
         return type    : bool

1.   resourcesTypesList = app.get_resourcesTypesList()
2.   foreach ( resT in resourcesTypesList ) :
3.            if ( selfPM.get_availableAmountOf(resT) <
     proc.get_parentVM().get_allocatedAmountOf(resT) ) :
4.                     return false
5.   if ( proc.isCore() ) :
6.            dependenciesList = proc.get_parentApplication().get_dependenciesList()
7.            foreach ( dep in dependenciesList ) :
8.                     if (          proc.get_parentApplication().MAX_LATENCY_BETWEEN_
     CORE_AND_DEP < latency( selfPM, dep.get_parentPM() ) ) :
9.                              return false
10.  else :
11.           core = proc.get_parentApplication().get_core()
12.                    if ( proc.get_parentApplication().MAX_LATENCY_BETWEEN_
     CORE_AND_DEP < latency( selfPM, core.get_parentPM() ) ) :
13.                             return false
14.  return true
```

4.3.1.4 Méthodes de la classe Application

Algorithme 4.14 Classe Application - Méthode get_DependenciesWithHighLatencyToTheCore

```
     return type    : List<Process>

1.   core = get_core()
2.   depsList = get_dependenciesList()
3.   candidateDeps = List<Process>()
4.   foreach ( dep in depsList ) :
5.           if ( latency( dep.get_parentPM(), core.get_parentPM() ) >
     MAX_LATENCY_BETWEEN_CORE_AND_DEP
6.                   AND dep.skipFlags["migrateToReduceLatency"] == false ) :
7.                   candidateDeps.append( dep )
8.   return candidateDeps
```

4.3.2 Documentation sélective

La documentation sélective explique le rôle des méthodes les plus importantes de notre algorithme de dimensionnement et de son environnement. Pour mieux distinguer nos contributions, nous avons inclus dans notre documentation une légende de couleurs permettant de faire une comparaison entre les fonctionnalités initialement présentées par Rui *et al.* (2012) et les nôtres:

- un fond vert indique que la méthode a été implémentée telle quelle;
- un fond bleu indique que la méthode a été implémentée avec une amélioration apportée;
- un fond blanc indique que la méthode n'existait pas dans l'article original.

4.3.2.1 Classe Application

Tableau 4.1 Documentation de la classe Application

Méthode ou attribut	Description
bool isRunning()	Retourne vrai si l'application est en cours d'exécution, faux le cas échéant.
double get_global_pT()	Retourne une valeur synthétisant le temps de traitement de l'application. On définit un échantillon de temps de traitement de l'application comme la durée nécessaire pour effectuer une itération de notre *workflow*. Ainsi, pour plus de stabilité, cette méthode retourne une moyenne pondérée glissante (*weighted moving average* ou WMA), calculée à partir de n échantillons de temps de traitement. Ceci est une méthode bloquante qui attend de collecter n échantillons. On appelle la quantité n taille de fenêtre de la WMA.
double get_global_UB_pT()	Retourne le seuil supérieur du temps de traitement de l'application (UB signifiant *Upper Bound*). Au-delà de cette limite, le SLA est violé. Cette limite permet de détecter un sous-approvisionnement.
double get_global_LB_pT()	Retourne le seuil inférieur du temps de traitement de l'application (LB signifiant *Lower Bound*). En dessous de cette limite, on peut envisager la possibilité de libérer des ressources quitte à avoir un temps de traitement supérieur, mais tout en restant en dessous de la limite maximale définie par le SLA. Cette méthode permet de détecter un sur-approvisionnement.
List<ResourceType> get_resourcesTypesList()	Retourne la liste des types de ressources utilisées par l'application. Cela peut être par exemple les cœurs de processeurs, la mémoire, la bande passante du réseau local, etc. Par choix de conception, la latence est exclue de cette liste car c'est une dimension qui n'est pas réellement une ressource.
List<Process> get_processesList()	Retourne la liste des tous les processus (instances de *substrates*) qui constituent l'application. Cela inclut donc le *core* et les *dependencies*.
List<Process> get_dependenciesList()	Retourne la liste des *dependencies* de l'application.

Tableau 4.1 Documentation de la classe Application (suite)

Méthode ou attribut	Description
Process get_core()	Retourne le *core* de l'application.
int MAX_LATENCY_BETWEEN_CORE_AND_DEP	Seuil supérieur de la latence entre une *dependency* et un *core*. C'est une constante qu'il est préférable de ne pas dépasser.
List<Process> get_dependenciesWithHighLatencyToTheCoreList()	Retourne la liste des *dependencies* de l'application qui ont une latence vis-à-vis du *core* supérieure à MAX_LATENCY_BETWEEN_CORE_AND_DEP.

4.3.2.2 Classe Process

Tableau 4.2 Documentation de la classe Process

Méthode ou attribut	Description
double get_usageOf(**ResourceType** resT)	Retourne le taux d'utilisation de la ressource resT par le processus.
double get_UB_usageOf(**ResourceType** resT)	Retourne le seuil supérieur du taux d'utilisation de la ressource resT. Cette limite permet d'aider à la détection d'une situation de sous approvisionnement.
double get_LB_usageOf(**ResourceType** resT)	Retourne le seuil inférieur du taux d'utilisation de la ressource resT. Cette limite permet d'aider à la détection d'une situation de sur approvisionnement.
PM get_parentPM()	Retourne une référence vers la machine physique qui héberge la machine virtuelle hébergeant le processus.
VM get_parentVM()	Retourne une référence vers la machine virtuelle hébergeant le processus.
Application get_parentApplication()	Retourne une référence vers l'application à laquelle le processus appartient.
double get_pT()	Retourne une valeur synthétisant le temps de traitement du processus. On définit un échantillon de temps de traitement d'un processus comme la durée nécessaire pour effectuer une tâche au sein d'une itération donnée. Ainsi, pour plus de stabilité, cette méthode retourne une moyenne pondérée glissante, calculée à partir de n échantillons de temps de traitement. Ceci est une méthode bloquante qui attend de collecter n échantillons.

Tableau 4.2 Documentation de la classe Process (suite)

Méthode ou attribut	Description
double get_UB_pT()	Retourne le seuil supérieur du temps de traitement du processus. Cette limite permet d'aider à la détection d'une situation de sous approvisionnement.
double get_LB_pT()	Retourne le seuil inférieur du temps de traitement du processus. Cette limite permet d'aider à la détection d'une situation de sur approvisionnement.

4.3.2.3 Classe ScaleEntry

Tableau 4.3 Documentation de la classe ScaleEntry

Méthode ou attribut	Description
Process proc	Processus.
ResourceType resT	Type de ressource.
ScaleEntry(Process proc**, ResourceType** resT**)**	Constructeur de la classe.

4.3.2.4 Classe Zone

Tableau 4.4 Documentation de la classe Zone

Méthode ou attribut	Description
void lock(**int** priority)	Cette méthode permet de verrouiller la zone. Si la zone est libre, le verrou est obtenu immédiatement. Si le verrou est déjà détenu par une autre instance de ScalingAlgorithm, le demandeur sera mis dans une des deux files d'attente, selon la priorité indiquée par la valeur de *priority*. La valeur 0 correspond à la priorité la plus élevée. Dans notre implémentation, il existe deux niveaux de priorité : 0 et 1. Les demandeurs présents dans la file d'attente 1 doivent attendre que la file d'attente 0 soit vide avant d'espérer obtenir le verrou.
void unlock()	Cette méthode permet de déverrouiller la zone.

Tableau 4.4 Documentation de la classe Zone (suite)

Méthode ou attribut	Description
List<PM> get_mostOverloaded_PMs_List()	Retourne la liste des machines physiques classées par ordre décroissant de taux d'utilisation global. Dans le cas multidimensionnel, on définit le taux d'utilisation global comme étant le produit des taux d'utilisation de chaque ressource. Les machines en mode veille sont également retournées.
bool couldScaleOnAnotherPM(**ScaleEntry** scaleEntry)	Vérifie si l'on pourrait augmenter une ressource allouée à un processus sur une autre machine physique (via migration). Pour cela le processus pour une ressource donnée doit : • Avoir un *skipFlag* inactif • Avoir une allocation inférieure à la quantité totale que possède la machine physique qui en a le plus dans la zone
Double getLatency(**PM** pm1**,PM** pm2)	Retourne la latence entre les machines physiques pm1 et pm2. Dans notre implémentation, on suppose que si une des machines physiques est éteinte, la latence est nulle.
PM get_leastOverloaded_nonEmpty_ withoutSkipFlag_PM()	Retourne la machine physique la moins utilisée contenant au moins une machine virtuelle et n'ayant pas de *skipFlag* avec la valeur true.
void switchToSleepModeEmptyPMs()	Cette méthode vérifie s'il existe des machines physiques n'hébergeant aucune machine virtuelle et les met en veille.
void migrateVM(**VM** vmToMigrate, **PM** destPM)	Migre la machine virtuelle vmToMigrate vers la machine physique destPM.
List<PM> get_mostOverloaded Running_PMs_List()	Retourne la liste des machines physiques allumées classées par ordre décroissant de taux d'utilisation global.

4.3.2.5 Classe PM

Tableau 4.5 Documentation de la classe PM

Méthode ou attribut	Description
double getAvailableAmountOf (**ResourceType** resT)	Retourne la quantité disponible de la ressource resT.

Tableau 4.5 Documentation de la classe PM (suite)

Méthode ou attribut	Description
bool canHost(**Process** proc)	Retourne vrai si la machine physique peut héberger le processus proc, faux le cas échéant.
List<VM> get_vmsListSortedDescending()	Retourne la liste des machines virtuelles hébergées par ordre décroissant de taux d'utilisation global.

4.3.2.6 Classe VM

Tableau 4.6 Documentation de la classe VM

Méthode ou attribut	Description
bool canReleaseOneUnitOf(**ResourceType** resT)	Retourne vrai si la machine virtuelle peut libérer une unité de la ressource resT, faux le cas échéant.
Void releaseOneUnitOf(**ResourceType** resT)	Libère une unité de la ressource resT.
bool canAllocateOneUnitOf(**ResourceType** resT)	Retourne vrai si la machine virtuelle peut allouer une unité de la ressource resT, faux le cas échéant.
void allocateOneUnitOf(**ResourceType** resT)	Alloue une unité de la ressource resT.
double getAllocatedAmountOf(**ResourceType** resT)	Retourne la quantité allouée de la ressource resT.
PM get_parentPM()	Retourne une référence vers la machine physique qui héberge la machine virtuelle hébergeant le processus.

4.3.2.7 Classe ScalingAlgorithm

Tableau 4.7 Documentation de la classe ScalingAlgorithm

Méthode ou attribut	Description
void start()	Point d'entrée de l'algorithme. Cette méthode lance la procédure de dimensionnement automatique tant que l'application est encore en marche. Selon le temps de traitement de l'application, elle détecte s'il y a un sur ou sous approvisionnement et appelle les méthodes nécessaires pour résoudre le problème. Lors du verrouillage de la zone, la priorité donnée au sous-approvisionnement est supérieure à celle du sur-approvisionnement, car il faut d'abord tenter de résoudre une violation du SLA (via la méthode reduce_processing_time) avant de tenter d'économiser de l'énergie en libérant des ressources allouées (via la méthode scale_down).
void reduce_processing_time()	Cette méthode regroupe l'ensemble des procédures permettant de mettre un terme à la violation du SLA, équivalent à une situation de sous-dimensionnement, en tentant de réduir le temps de traitement de l'application.
void scale_down()	Cette méthode tente de mettre un terme au surdimensionnement.
bool self_healing()	Le *self healing* tente d'optimiser la distribution des ressources au sein des processus d'une même application. Ainsi, il essaye de voir si une machine virtuelle dans la zone de sur-approvisionnement peut donner des ressources à un machine virtuelle dans la zone de sous-approvisionnement. Retourne vrai si le problème est corrigé, faux le cas échéant.
bool scale_up_without_migration()	Cette méthode tente d'augmenter les ressources allouées à chaque machine virtuelle. Retourne vrai si le problème est corrigé, faux le cas échéant.
bool recheck_sh_and_suwm()	Cette méthode exécute en série un self_healing et un scale_up_without_migration. Retourne vrai si le problème est corrigé, faux le cas échéant.
bool reduce_latency()	Cette méthode permet de réduire le temps de traitement de l'application en tentant de migrer les *dependencies* les plus éloignées en terme de latence, à proximité du *core*. Retourne vrai si le problème est corrigé, faux le cas échéant.

Tableau 4.7 Documentation de la classe ScalingAlgorithm (suite)

Méthode ou attribut	Description
void scale_up_through_migration( )	Cette méthode permet de réduire le temps de traitement de l'application en tentant de migrer les machines virtuelles dans la zone de saturation et dont les machines physiques hôtes n'ont plus de ressources disponibles, vers des machines physiques ayant des ressources disponibles.
void scale_down_resources()	Cette méthode tente de libérer des ressources allouées lorsqu'une machine virtuelle est dans la zone de sur-approvisionnement.
bool migrate_to_scale_up(**ScaleEntry** entryToScaleUp)	Appelée par la méthode scale_up_through_migration, cette méthode permet de trouver la meilleure machine physique pour y migrer une machine virtuelle.
bool migrate_to_reduce_latency(**Process** depToMigrate)	Appelée par la méthode reduce_latency, cette méthode permet de trouver la meilleure machine physique pour y migrer une machine virtuelle.
ScaleEntry scale_up_without_migration _ selection_criteria(**List<ScaleEntry>** candidates)	Appelée par la méthode scale_up_without_migration, cette méthode utilise un critère pour sélectionner une ScaleEntry parmi une liste de candidats. Dans notre implémentation, on sélectionne le processus proc dont le temps de traitement est le plus élevé par rapport à son seuil supérieur, donné par la différence : proc.get_pT() – proc.get_UB_pT().
Process reduce_latency_ selection_criteria(**List<Process>** candidates)	Appelée par la méthode reduce_latency, cette méthode utilise un critère pour sélectionner un processus parmi une liste de candidats. Dans notre implémentation, on sélectionne le processus dont la latence vis-à-vis du *core* est la plus élevée.
ScaleEntry scale_up_through_migration _ selection_criteria(**List<ScaleEntry>** candidates)	Appelée par la méthode scale_up_through_migration, cette méthode utilise un critère pour sélectionner une ScaleEntry parmi une liste de candidats. Dans notre implémentation, on sélectionne le processus proc dont le temps de traitement est le plus élevé par rapport à son seuil supérieur, donné par la différence : proc.get_pT() – proc.get_UB_pT().
ScaleEntry scale_down_resources_ selection_criteria(**List<ScaleEntry>** candidates)	Appelée par la méthode scale_down_resources, cette méthode utilise un critère pour sélectionner une ScaleEntry parmi une liste de candidats. Dans notre implémentation, on sélectionne le processus proc dont le temps de traitement est le plus faible par rapport à son seuil inférieur, donné par la différence : proc.get_LB_pT() – proc.get_pT().

4.4 Conclusion du chapitre

Dans ce chapitre, nous avons présenté le principe de fonctionnement de notre algorithme de dimensionnement dynamique des ressources, en détaillant l'environnement orienté objet qu'il requiert ainsi que les algorithmes qu'il implémente. Dans le chapitre suivant, nous évaluerons son efficacité à travers divers scénarios de tests de performances et discuterons de ses directions futures.

CHAPITRE 5

ANALYSE DES RESULTATS EN ENVIRONNEMENT SIMULE

Dans ce chapitre, nous allons tester notre algorithme de dimensionnement dynamique de ressources. Nous justifierons l'usage d'un simulateur et soumettrons notre algorithme à divers scénarios pour évaluer son efficacité et souligner ses paramètres à optimiser. Ces derniers seront essentiellement liés au nombre d'applications fonctionnant simultanément dans une même zone, au dimensionnement initial des applications et enfin le choix d'une taille pour la fenêtre de la moyenne glissante. Nous testerons également notre algorithme de consolidation pour justifier son utilité dans l'optimisation des ressources allouées et la réduction des coûts d'énergie.

5.1 Présentation de l'environnement de simulation

Pour tester l'efficacité de notre algorithme, nous avons choisi d'utiliser un simulateur. Les avantages par rapport à un environnement réel sont les suivants :

- il est possible de réaliser des simulations sur des *datacenters* virtuels de taille conséquente (plus de 100 machines physiques). Une telle infrastructure n'est pas toujours à notre disposition. Il en est de même pour le coût des licences logicielles, incluant dans notre cas les systèmes d'exploitation et hyperviseurs;
- un simulateur fournit des résultats très rapidement. Durant nos tests, nous avons constaté que le temps de simulation pouvait être 250 fois plus faible que le temps réel;
- les paramètres de simulations sont très rapides à modifier et il est plus facile d'automatiser l'exécution de plusieurs scénarios différents;
- modéliser un prototype tel que SOBB est très simple. Implémenter des fonctionnalités de dimensionnement dynamique telles que la modification du nombre de cœurs CPU ou la supervision des ressources utilisées l'est encore plus. Un simulateur est donc extrêmement pratique pour mettre en place un banc d'essai et tester un algorithme, le plus vite possible.

Le simulateur que nous avons utilisé est SimGrid. Il s'agit d'un logiciel libre écrit en C permettant de simuler des infrastructures informatiques. Comme nous avons choisi d'utiliser le langage C++ avec l'environnement Qt, les fonctionnalités de SimGrid utilisées ont été enveloppées dans des classes.

D'autre part, nous avons dû étendre les capacités du simulateur. Par exemple, SimGrid ne supportait pas la loi d'Amdahl. Celle-ci permet dans notre cas de calculer le gain en performance qu'une application peut atteindre en fonction de sa quantité de code parallèle.

5.2 Scénarios de tests de performances

5.2.1 Protocole expérimental

Pour tester notre algorithme, nous avons modélisé le *workflow* de SOBB (décrit dans la figure 3.2) au sein de SimGrid.

Pour mesurer l'efficacité de notre algorithme, nous nous sommes intéressés à deux métriques principales :

- temps de violation du SLA (en secondes) : plus il est faible, mieux c'est. 0 signifie que l'application n'a pas souffert de sous-approvisionnement. Le temps de violation du SLA est l'ensemble des temps durant lesquels l'application a eu un temps de réponse (ou MLIT) au-dessus de 33.33 ms;
- machines physiques actives en moyenne :

$$\text{machines physiques actives en moyenne } = \frac{1}{n}\sum_{i=1}^{n} PMsActivesDurantLaTranche(i)$$

où n représente le nombre total de tranches de temps. Plus la quantité de machines physiques actives en moyenne est faible, mieux c'est. Cet indicateur est intéressant pour évaluer la qualité de la consolidation. Dans notre implémentation, cet échantillonnage a été fait à une fréquence de 100Hz. Nous avons constaté que le choix de ce taux d'échantillonnage offre un

bon compris, car c'est une valeur suffisamment élevée pour assurer une précision des mesures, sans pour autant générer des fichiers de données trop volumineux à analyser.

D'autre part, comme indiqué dans les résultats du troisième chapitre avec le prototype réel, les temps de traitement relevés pour le moteur de rendu étaient 7 fois plus importants que ceux du moteur physique. De ce fait, dans les paramètres de Simgrid, nous avons réglé la longueur d'une tâche du moteur de rendu comme étant 7 fois plus importante que celle du moteur physique. Toutefois, nous pouvons toujours altérer la portion de code parallèle. Comme l'indique la loi d'Amdahl, plus la portion de code parallèle est faible, moins un programme bénéficie de l'addition de cœurs CPU.

Le tableau 5.1 fournit les paramètres par défaut :

Tableau 5.1 Paramètres par défaut du simulateur

Paramètre	Valeur
Etat de la consolidation (instance de la classe Consolidator)	active
Méthode d'allocation initiale des machines virtuelles	*next fit*
Machines physiques disponibles dans la zone	100
Nombres d'applications	8
Taille de la fenêtre de la moyenne glissante	8
Nombre de cœurs de processeurs alloués au départ à chaque machine virtuelle	1
Nombre de cœurs de processeurs par machine physique	6
Nombre de processeurs par machine physique	1
Nombre d'itérations du *workflow*	2000

Dans le cas de la méthode d'allocation initiale des machines virtuelles, nous avons utilisé le *next fit*, car dans nos conditions expérimentales, il aboutit à un agencement équivalent au *first fit decreasing*, tout en étant plus rapide. En effet, nos conteneurs (les machines physiques) sont tous de même taille et nos objets (les machines virtuelles) sont également de tailles identiques au départ. La disposition du *next fit* est donc identique à celle du *first fit decreasing*.

Tableau 5.2 Paramètres par défaut de l'application

Paramètre	Valeur
Limite supérieure du temps de traitement (en ms)	30
Limite inférieure du temps de traitement (en ms)	26

Tableau 5.3 Paramètres par défaut des *dependencies*

	Valeurs	
Paramètre	**Moteur de rendu**	**Moteur physique**
Taux d'utilisation des cœurs de processeurs (fixé, car ne peut être simulé)	90%	90%
Taux d'utilisation des cœurs de processeurs – seuil supérieur	70%	70%
Taux d'utilisation des cœurs de processeurs – seuil inférieur	30%	30%
Portion de code parallèle	80%	80%
Limite supérieure du temps de traitement (en ms)	20	1
Limite inférieure du temps de traitement (en ms)	14	0.3

Enfin, même en conservant des paramètres identiques, deux simulations n'offrent pas des résultats parfaitement similaires. En effet, chacune a un comportement pseudo aléatoire. Cela est essentiellement dû à la phase où plusieurs processus de dimensionnement tentent de verrouiller la zone de manière concurrente. Pour cela, nous allons effectuer des simulations de Monte Carlo. Toutefois, nous verrons qu'à travers l'analyse des moyennes et des écarts types, ces résultats sont très proches. Pour chaque scénario, nous ferons 100 simulations.

La figure 5.1 donne un aperçu de notre environnement de test réduit à deux applications et trois machines physiques. Dans ce schéma, on désigne par la notation $Z_x.PM_y.VM_z$ la machine virtuelle d'indice z, hébergée par la machine physique d'indice y et contenue dans la zone d'indice x. De même, on désigne par la notation $App_a.Sub_b$ le *substrate* d'indice b de l'application d'indice a.

Dans la figure 5.1, nous retrouvons les hypothèses de notre algorithme de dimensionnement dynamique de ressources, où une instance de ScalingAlgorithm par application (désignée par ScalingProcess) est nécessaire et où une unique instance de la classe Consolidator pour toute la zone (désignée par ConsolidatorProcess) suffit. Ces processus sont tous regroupés sur une machine physique de supervision désignée par MonitoringPM.

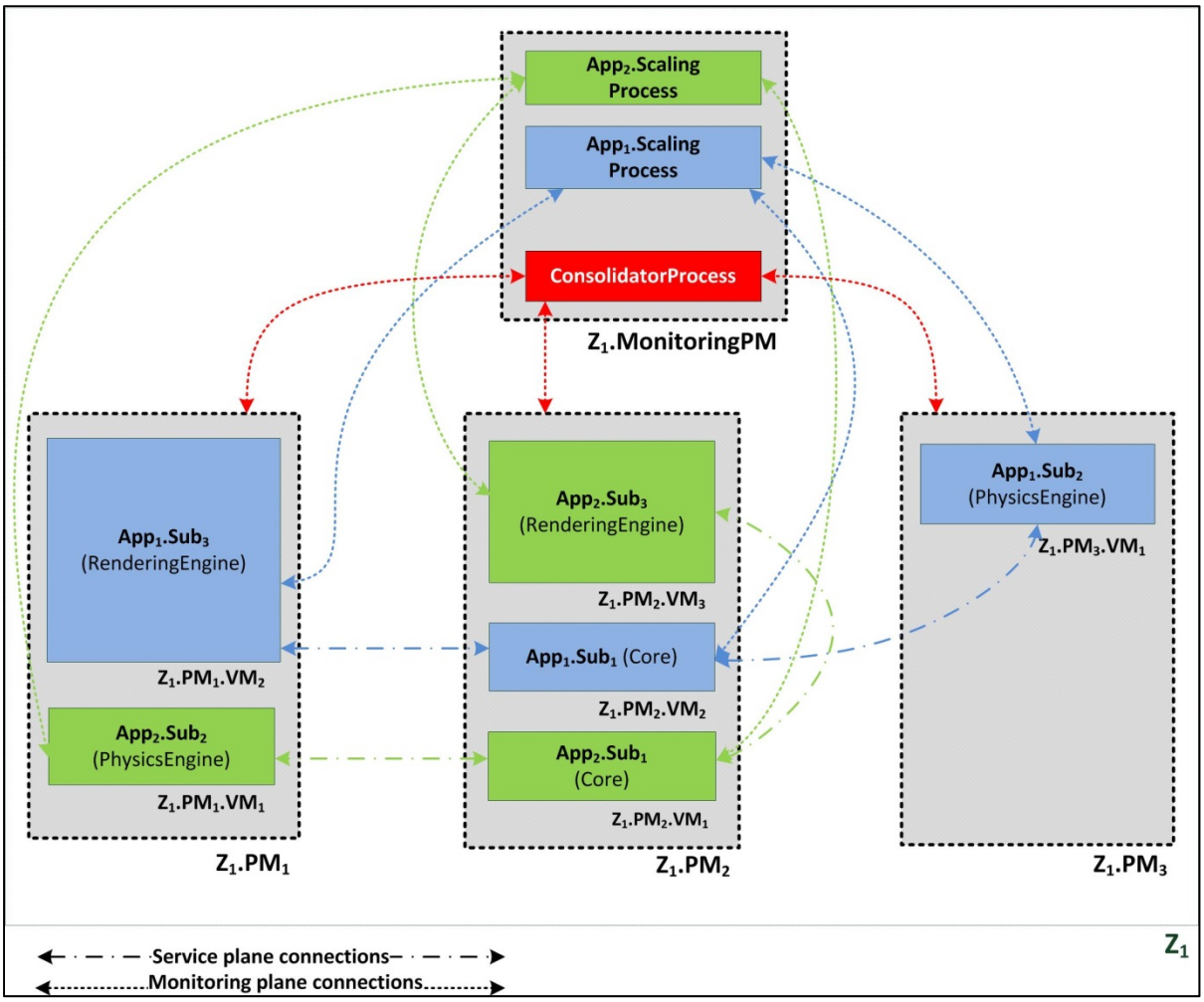


Figure 5.1 Environnement de test de l'algorithme de dimensionnement dynamique

Enfin, comme l'indique la légende de la figure 5.1, les flèches intitulées *Service plane connections* désignent les connections entre *substrates*, qui permettent d'échanger des requêtes pour progresser dans le *workflow*. Les flèches intitulées *Monitoring plane connections* indiquent les flux de supervision entre :

- machines virtuelles et instances de ScalingAlgorithm auxquelles elles sont rattachées;
- machines virtuelles et l'unique instance de Consolidator.

5.2.2 Scénario de base

Le scénario de base est un scénario de simulation qui reprend les paramètres par défaut définis précédemment.

5.2.2.1 Résultats détaillés

La figure 5.2 nous montre les résultats d'une simulation faite avec le scénario de base. Le graphe donne l'évolution du MLIT de chaque application en fonction du temps. On constate qu'au départ, le MLIT de chaque application est largement au-dessus du seuil supérieur (en pointillés rouge, intitulé UB). Les applications souffrent de sous-approvisionnement, car les machines virtuelles qui les hébergent sont initialement dimensionnées avec un unique cœur de processeur. Ce problème est résolu de manière successive grâce à l'intervention de chaque instance de ScalingAlgorithm : chaque application voit son temps de traitement passer en dessous du seuil supérieur.

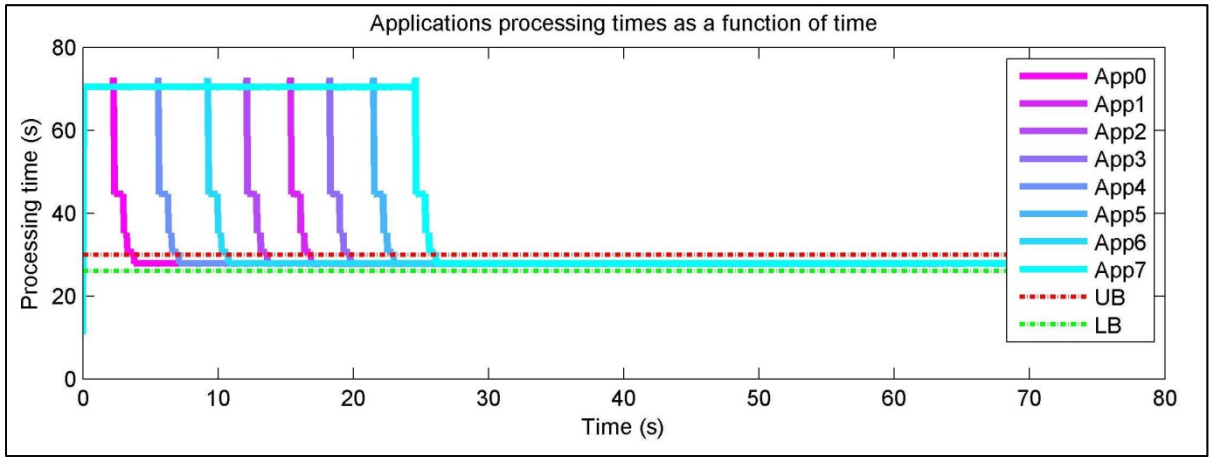


Figure 5.2 Évolution des MLITs de chaque application en fonction du temps

Les figures 5.3 et 5.4 montrent bien que les temps de traitement des moteurs de rendu et des moteurs physiques baissent successivement pour chaque application. Bien qu'ils ne passent pas en dessous de la zone supérieure (intitulée UB, en pointillés rouge), la baisse des temps de traitement est suffisante pour baisser le temps de traitement global de l'application, c'est-à-dire le MLIT. L'algorithme de dimensionnement abandonne donc les opérations d'approvisionnement dès que le problème est réglé au niveau du temps de réponse global.

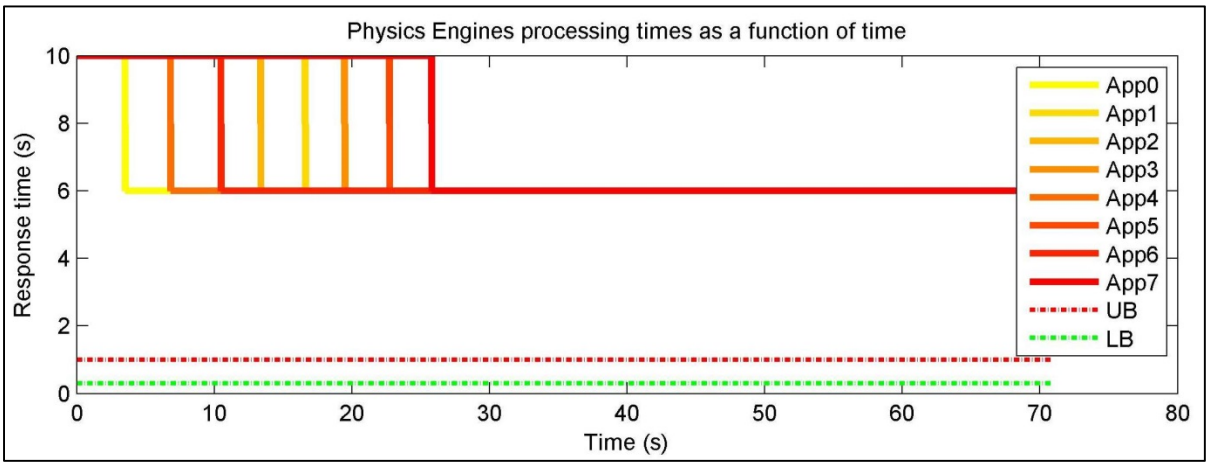


Figure 5.3 Évolution des temps de traitement du moteur physique de chaque application en fonction du temps

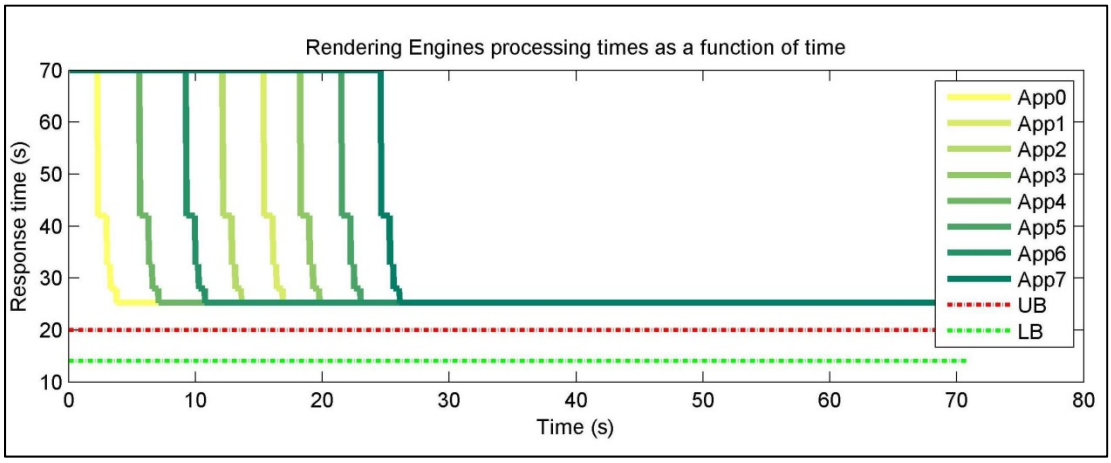


Figure 5.4 Évolution des temps de traitement du moteur de rendu de chaque application en fonction du temps

D'autre part, la figure 5.5 donnant l'évolution du nombre de machines physiques en fonction du temps montre bien qu'au départ, seulement 4 machines physiques sont allumées. Ceci est normal puisqu'au début de la simulation nous avons 8 applications nécessitant un cœur pour chacun de ses trois *substrates*, soit 24 cœurs au total. Comme chaque machine physique est dotée de 6 cœurs, il faut 24/6 = 4 machines physiques au départ pour héberger l'ensemble des applications. Entre le temps t=0 s et t=25 s, on observe une augmentation des machines physiques actives. Ceci est dû aux migrations pour effectuer des *scale up* des machines virtuelles vers de nouvelles machines physiques ayant des ressources disponibles, dont certaines sont éteintes. Juste après le temps t=28 s, on observe une diminution d'une machine physique : ceci est dû au travail du consolidateur.

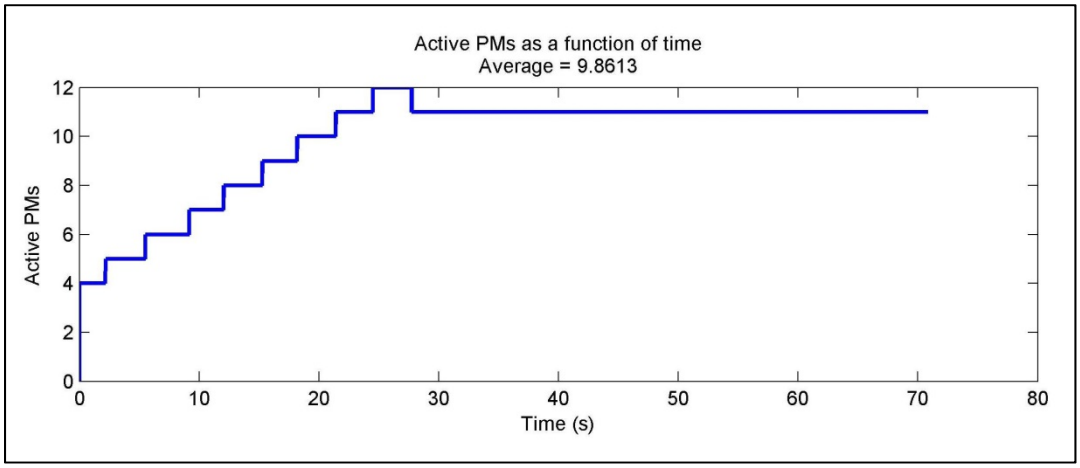


Figure 5.5 Évolution du nombre de machines physiques actives
en fonction du temps

5.2.2.2 Synthèse de 100 simulations

La figure 5.6 donne une synthèse de 100 simulations, afin d'étudier la stabilité de l'algorithme. On s'aperçoit que la moyenne des temps de violation moyens du SLA est de 14.84 s, avec un faible écart-type de 0.31. De même, on constate que les machines physiques actives en moyenne sont de 9.8 avec un faible écart type de 0.02. Ces deux résultats signifient que d'une simulation à une autre, les résultats varient peu. Pour des conditions

expérimentales identiques, notre algorithme fournit des résultats stables, malgré les variables pseudo-aléatoires sur lesquelles nous n'avons pas de contrôle.

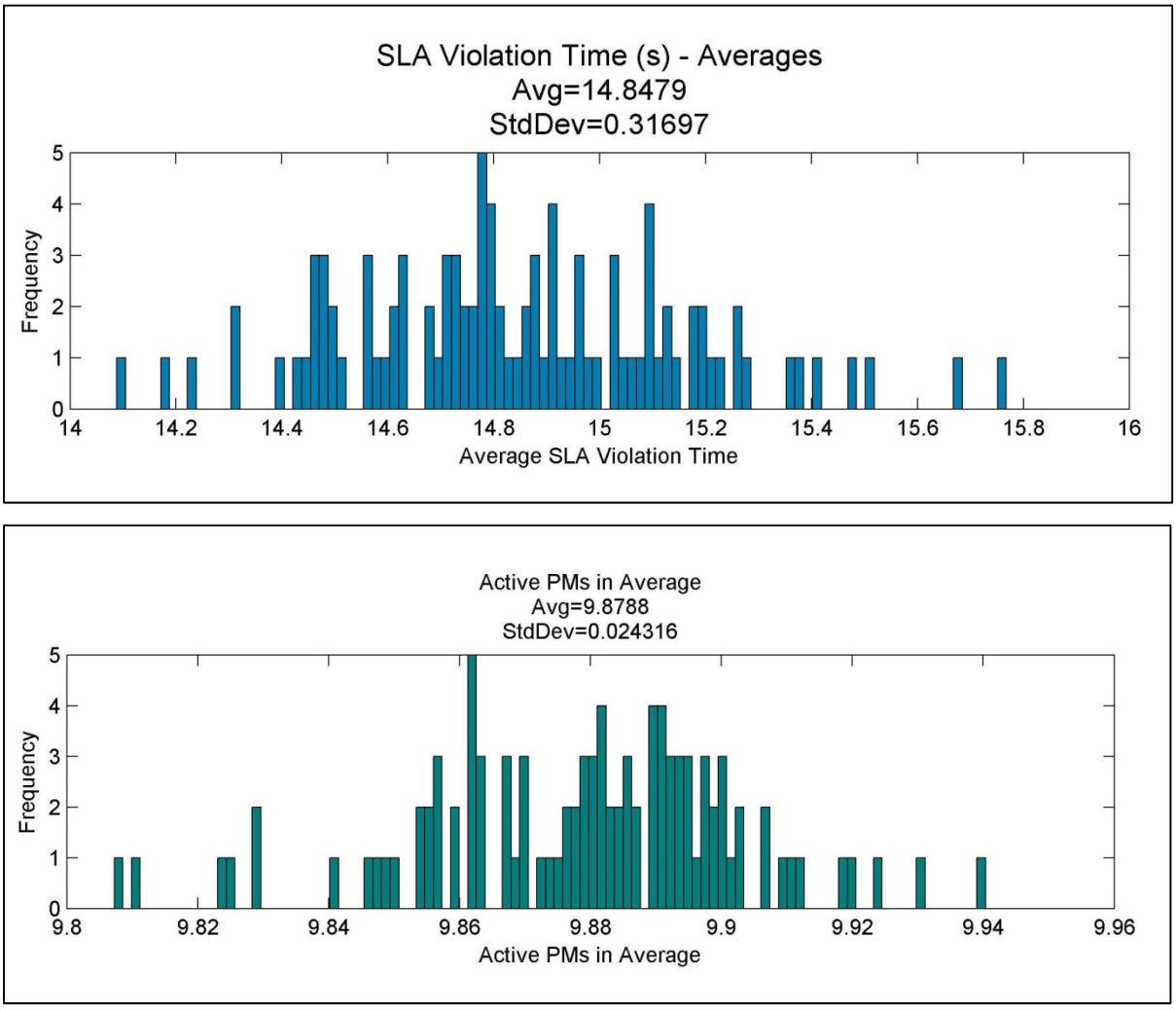


Figure 5.6 Synthèse de 100 simulations

5.2.3 Influence du nombre d'applications

Dans cette expérience, on fait varier le nombre d'applications du scénario de base de 2 à 64. Le but est de matérialiser l'inconvénient principal de la technique de verrouillage d'une zone : plusieurs processus de dimensionnement ne peuvent fonctionner en parallèle, mais de manière séquentielle. La figure 5.7 illustre bien les conséquences de cette approche : plus le

nombre d'applications mal dimensionnées durant la même période augmente (elles le sont toutes au départ dans le cas du scénario de base), plus les processus de dimensionnement doivent en moyenne patienter avant d'obtenir le verrou. Ainsi, le temps de violation moyen du SLA augmente. Cette expérience montre que l'évolution du temps moyen de violation du SLA en fonction du nombre d'applications est linéaire.

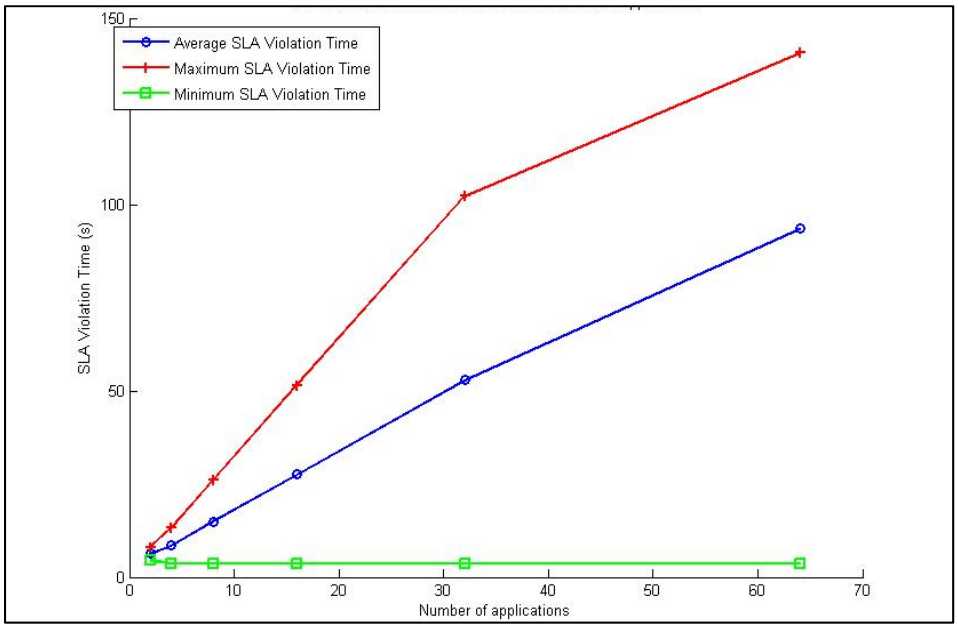


Figure 5.7 Influence du nombre d'applications

5.2.4 Influence de la quantité initiale de ressources allouées

La figure 5.8 illustre une expérience qui consiste à faire varier le nombre de cœurs initialement alloués aux machines virtuelles de 1 à 5. On constate que le temps de violation du SLA diminue lorsque l'allocation initiale augmente. Ce résultat est facile à interpréter puisqu'il montre que plus le dimensionnement d'une application est proche du cas idéal dès le départ, moins l'algorithme de dimensionnement gaspillera du temps pour tenter de résoudre une situation de sous-approvisionnement. Cette expérience montre bien l'intérêt de bien dimensionner initialement les machines virtuelles, comme le permet notre modèle d'information présenté dans le chapitre deux.

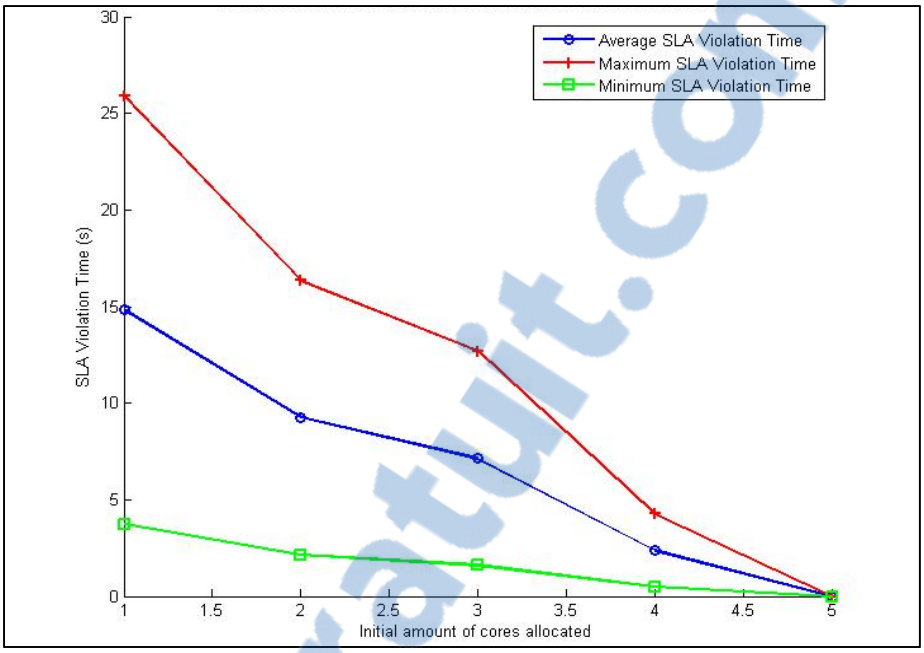


Figure 5.8 Influence de la quantité initiale de ressources allouées

5.2.5 Influence de la taille de la fenêtre de la moyenne glissante

Dans cette expérience, nous faisons varier la taille de la fenêtre de la moyenne glissante (WMA) de 2 à 16. Pour rappel, celle-ci permet de mieux synthétiser une série de temps de traitement consécutifs pour prendre des décisions plus pertinentes en cas de sous ou sur approvisionnement. L'objectif de la WMA est d'atténuer l'effet des pics de temps de traitement isolés dans le temps, pour éviter un comportement trop réactif de l'algorithme de dimensionnement. On constate d'après la figure 5.9 que l'évolution du temps moyen de violation du SLA en fonction de la taille de la fenêtre de la moyenne glissante est linéaire. La taille de la fenêtre doit donc être choisie minutieusement : il faut faire un compromis entre stabilité de l'algorithme (fenêtre WMA plus grande) et sa rapidité de traitement (fenêtre WMA plus petite).

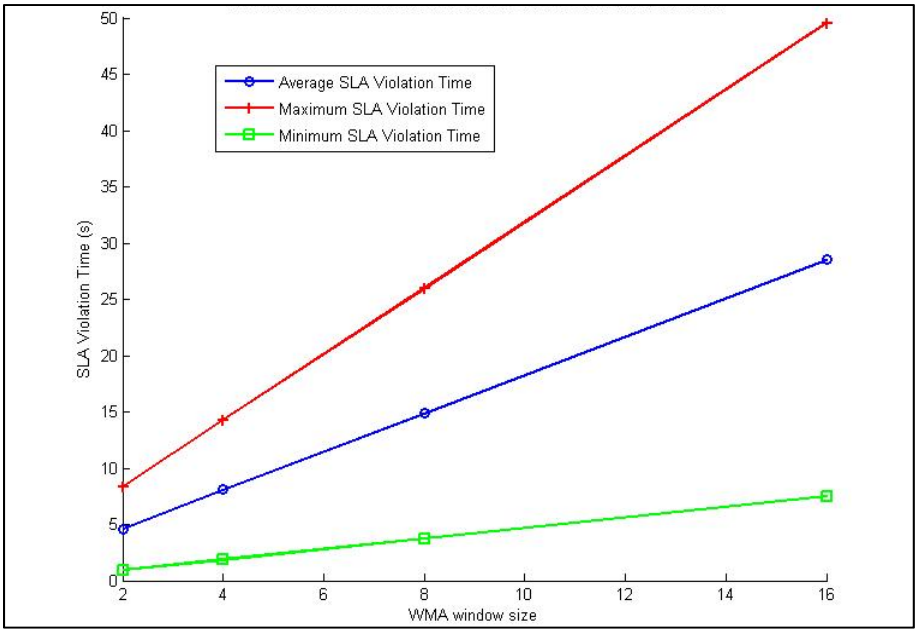


Figure 5.9 Influence de la taille de la fenêtre de la moyenne glissante

5.2.6 Efficacité de la consolidation

Dans cette partie, nous testons l'efficacité de notre algorithme de consolidation décrit dans le quatrième chapitre. Pour rappel, celui-ci est implémenté par la classe Consolidator.

5.2.6.1 Scénario de base sans consolidation

L'histogramme de la figure 5.10 donne la répartition du nombre moyen de machines physiques actives pour le scénario de base lorsque la consolidation est désactivée. Sur 100 simulations, on voit qu'il y a en moyenne 10.49 machines physiques actives, avec un faible écart type de 0.03.

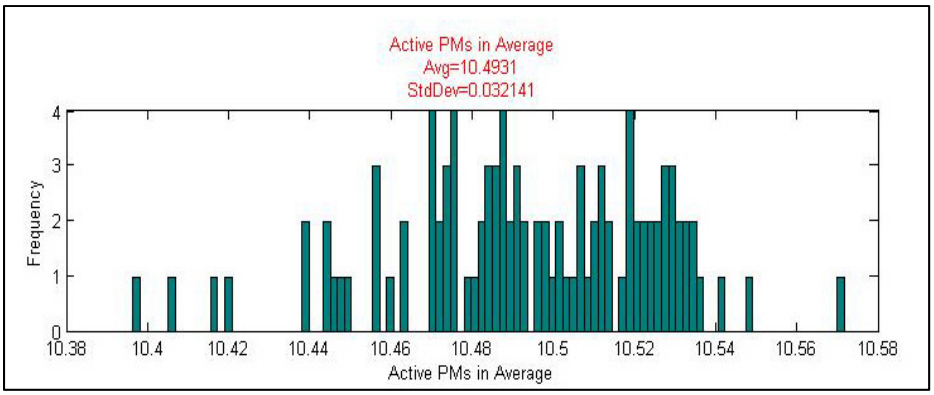


Figure 5.10 Machines physiques actives en moyenne,
scénario de base sans consolidation

5.2.6.2 Scénario de base avec consolidation

L'histogramme de la figure 5.11 donne cette fois-ci la répartition du nombre moyen de machines physiques actives durant le scénario de base non modifié, donc avec consolidation. On voit qu'en moyenne, il y a 9.87 machines physiques actives, avec un faible écart type de 0.029. Ce résultat est inférieur de seulement 6% par rapport au scénario de base sans consolidation. Cette faibe diminution est due à la stratégie d'allocation initiale des machines virtuelles efficace : le *next fit*. L'amélioration apportée par le consolidateur est donc relativement faible.

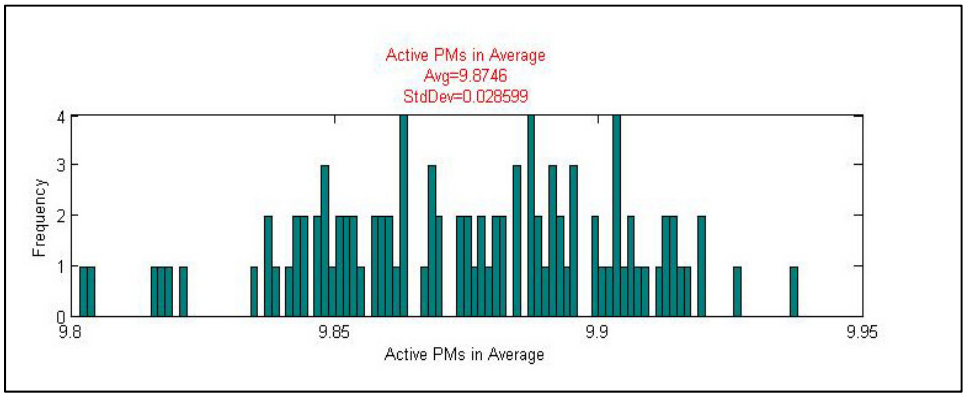


Figure 5.11 Machines physiques actives en moyenne,
scénario de base avec consolidation

5.2.6.3 Scénario de base sans consolidation, avec allocation initiale de type *round robin*

Cette fois-ci, nous modifions le scénario de base en désactivant la consolidation et en allouant les machines virtuelles avec une technique de *round robin* qui détériore fortement le degré de consolidation initial : chaque machine virtuelle nouvellement instanciée est allouée à une machine physique différente, puis on boucle lorsque l'on atteint le nombre maximal de machines physiques de la zone. On constate ici que le nombre moyen de machines physiques actives est de 24 soit 2.28 fois la quantité constatée dans l'expérience sans consolidation (10.49). On en conclut sans surprise que la stratégie d'allocation initiale des machines virtuelles est très importante.

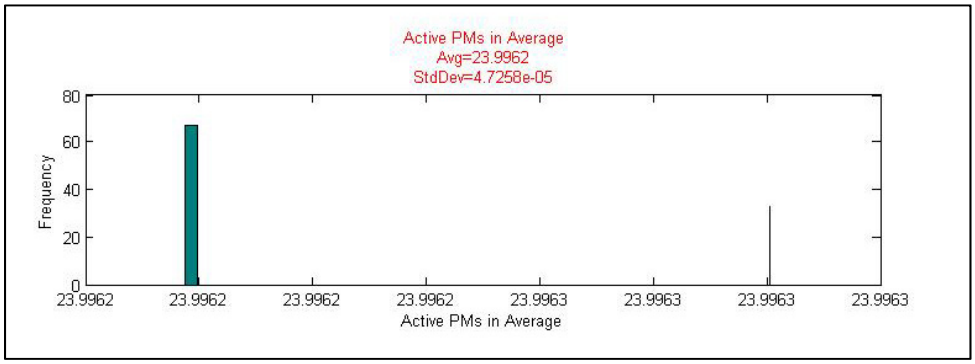


Figure 5.12 Machines physiques actives en moyenne,
scénario de base sans consolidation, avec allocation initiale de type *round robin*

5.2.6.4 Scénario de base avec consolidation, avec allocation initiale de type *round robin*

Cette fois-ci, nous conservons la consolidation du scénario de base mais nous continuons à allouer les machines virtuelles avec une technique de *round robin*. Nous constatons que le nombre de machines physiques actives est désormais de 14.03. Cela est 61% inférieur au cas sans consolidation avec allocation initiale de type *round robin* : le consolidateur rattrape le gaspillage effectué par cette allocation initiale que nous avions volontairement choisi comme étant inefficace. Certes, avec 14.03 machines physiques actives, par rapport à 9.8 si l'allocation initiale était de type *next fit*, nous restons 43% au-dessus. Etant donné que l'on

utilise des heuristiques qui n'offrent pas des solutions optimales, ces résultats montrent que la technique d'allocation initiale des machines virtuelles ne doit pas être négligée et offre une complémentarité très intéressante au consolidateur.

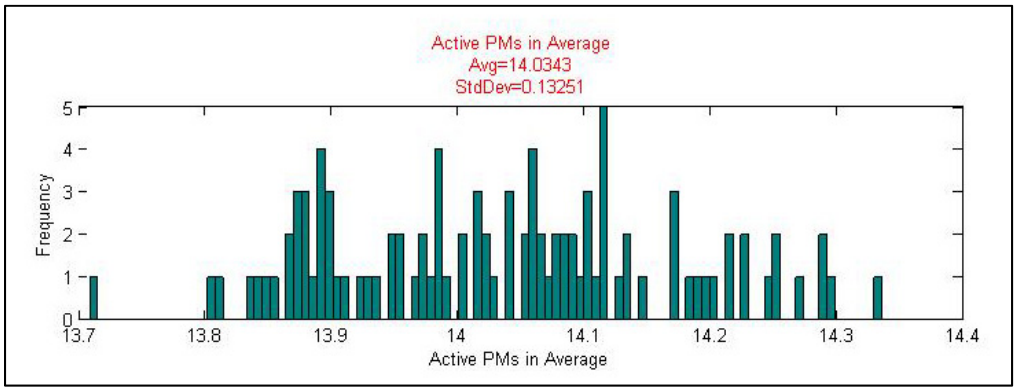


Figure 5.13 Machines physiques actives en moyenne,
scénario de base avec consolidation, avec allocation initiale de type *round robin*

5.3 Conclusion du chapitre

Dans ce chapitre, les scénarios de tests ont montré que notre algorithme de dimensionnement dynamique de ressources permet de mettre un terme à une situation de sous-approvisionnement en allouant des ressources aux applications mal dimensionnées. Nous avons étudiés plusieurs facteurs ayant un impact sur l'efficacité de notre algorithme. Tout d'abord, plus le nombre d'applications est élevé plus l'algorithme nécessite du temps pour mettre un terme à une situation de sous-approvisionnement. Deuxièmement, mieux une application est correctement dimensionnée à l'état initial, moins l'algorithme gaspille du temps pour trouver le dimensionnement idéal. Troisièmement, la taille de la fenêtre de la moyenne glissante joue un rôle central dans l'efficacité de l'algorithme et doit être choisie pour offrir un bon compromis entre vivacité de l'algorithme et stabilité des temps de traitement relevés. Enfin, nous avons constaté l'intérêt de l'algorithme de consolidation, qui a permis de réduire le gaspillage effectué par une méthode d'allocation initiale des machines virtuelles peu optimale.

CONCLUSION

Au terme de ce mémoire, nous avons présenté deux contributions majeures.

La première, dans le domaine du *cloud gaming*, s'est penchée sur la possibilité de distribuer des jeux vidéo à travers une architecture infonuagique orientée services, permettant à des nouveaux acteurs, les *substrates providers* de fournir des modules, les *substrates*, qui lorsque composés par des *service providers* constituent une application utilisable par des clients. Nous nous sommes intéressés aux limites de cette approche, liées notamment aux contraintes temps réel des jeux vidéo. Initialement, ces dernières sont essentiellement présentes à cause de la latence entre le joueur et les serveurs distants dans le cas du *cloud gaming* classique utilisant des jeux monolithiques, en plus des temps de traitement. Néanmoins, dans notre approche, des contraintes de latence s'étendent également au *workflow* de l'application, désormais distribuée, que l'on a cherché à caractériser par le MLIT.

La comparaison des deux *benchmarks* que nous avons développé a montré que l'approche traditionnelle pour construire des jeux vidéo, qui est monolithique, est robuste et offre de meilleures performances. Toutefois, notre conception orientée *substrates* est intéressante pour son approche distribuée. Avec plus de moyens, nous aurions pu comme Zhao *et al.* (2012) construire un prototype plus sophistiqué et surtout plus optimisé, en profitant de l'avantage apporté par l'aspect distribué de notre paradigme des *substrates*. Zhao *et al.* (2012) a d'ailleurs constaté une augmentation des performances.

La seconde contribution majeure de ce mémoire est la conception d'un algorithme de dimensionnement dynamique de ressources pour notre architecture, permettant de maintenir une qualité de service constante durant la phase de service. Son comportement dynamique permet de superviser en continu les temps de traitement des *substrates* ainsi que les taux d'utilisation des ressources consommées, pour tenter de mettre un terme aux situations de sous ou sur approvisionnement. D'autre part, il permet également d'assurer une consolidation des ressources en effectuant de la migration de machines virtuelles.

Directions futures de l'architecture orientée services

Dans le cas de notre architecture de virtualisation de jeux vidéo orientée services, nous nous sommes contentés de définir le rôle, les objectifs et les interactions de chaque module de l'architecture, au niveau de chaque phase. La première étape pour compléter notre architecture est la description du fonctionnement interne de ses modules. Si certains reposent sur des mécanismes simples, tel que le *Publication Engine*, d'autres en revanche nécessitent des algorithmes sophistiqués qui vont au-delà de ce mémoire. C'est essentiellement le cas du *Combination Finder Engine*, qui pour rappel a pour but de trouver la meilleure combinaison de *substrates providers* pour une demande de réservation d'une composition. Ce problème qui relève de l'optimisation combinatoire n'a pas été traité car il nécessite des algorithmes non triviaux.

La seconde direction future est de compléter notre architecture en se concentrant sur tous les scénarios pouvant entraîner des erreurs ou des exceptions. Par exemple, que se passe-t-il lorsqu'une composition nécessite un *substrate* qui n'est plus disponible ? Quelles sont les requêtes échangées par les modules de l'architecture pour résoudre ce problème ? Les réponses à ces questions peuvent être partiellement définies dans les règles métiers (*business rules*) de l'architecture.

Les deux directions précédentes sont indispensables à la robustesse de l'architecture. Néanmoins, si notre architecture a fait l'objet d'une vérification d'un point de vue théorique, pour assurer son intégrité, elle n'a pas été implémentée pour être validée. L'implémentation de l'architecture, vérifiée par l'exécution de tests unitaires, est indispensable à sa validation. Cela permettra de s'assurer qu'aucune fonctionnalité ou connexion entre modules n'a été omise durant la phase de conception. L'implémentation de l'architecture peut se faire en environnement simulé (plus rapide et plus simple), réel (plus exigent mais plus fiable), ou hybride (certaines fonctionnalités sont simulées, d'autres sont réellement implémentées).

Directions futures du prototype de jeu orienté substrates

Mis à part les tests de performances, le but de notre prototype était de démontrer la viabilité de notre paradigme des *substrates*. Comme nous l'avions mentionné dans le troisième chapitre, notre prototype s'est résumé à un *benchmark* qui ne prenait en compte que la seconde étape du *workflow* du *cloud gaming*, qui est le traitement à distance du jeu. A l'avenir, il serait très intéressant de développer un vrai jeu vidéo et d'implémenter les trois étapes du *cloud gaming* : transmission des commandes, traitement à distance du jeu et renvoi d'un flux vidéo.

D'autre part, développer un vrai jeu vidéo est un *challenge,* mais pourrait permettre de découvrir d'autres avantages et inconvénients du paradigme orienté *substrates*. Par exemple, en revenant à notre prototype, supposons que l'on souhaite ajouter un module de son. Ce dernier jouerait une musique de fond et devrait jouer un son dès qu'une balle entre en contact avec le sol. Si dans l'approche monolithique du *cloud gaming* cela ne semble pas un défi, notre approche orientée *substrates* pose un problème. Comment synchroniser l'image, générée par le moteur de rendu, avec le son, généré par le moteur de son ? Pour rappel, le moteur de rendu et le moteur de son peuvent être dans deux centres de données distincts avec des latences vis-à-vis du terminal du joueur et du *core* variables. Devons-nous créer un *substrate* qui fusionne et synchronise les deux flux avant de les transmettre vers le logiciel client du joueur ? Ou alors doit-on transmettre les deux flux séparément et les synchroniser au niveau du logiciel client du joueur ? Cette question ouverte montre que par rapport à l'architecture monolithique, l'approche orientée *substrates* peut compliquer l'architecture d'un jeu et soulever de nouvelles problématiques.

Directions futures de l'algorithme de dimensionnement dynamique de ressources

Notre algorithme de dimensionnement dynamique, relativement simple, pourrait être amélioré sur de nombreux points. Le premier étant le remplacement de la technique de blocage d'une zone, par l'approche du *three way handshake*. Au prix d'une implémentation

plus complexe qui nous aurait demandé du temps, elle aurait permis un verrouillage plus fin des machines physiques et aurait amélioré la vivacité de l'algorithme en augmentant le parallélisme lorsque plusieurs instances de ScalingAlgorithm auraient souhaité agir simultanément. L'annexe 3 illustre en partie son fonctionnement dans le cas d'une demande de migration. Pour le premier message, une machine physique source A envoie une requête indiquant une demande de migration vers une machine physique de destination B. Pour le second message, si B dispose des ressources nécessaires, elle accepte la demande de A en réservant temporairement les ressources. Pour le troisième message, lorsque A reçoit cette confirmation, elle reconfirme cet accord en notifiant B. Si au bout d'un temps donné, A ne reçoit pas de réponse après son premier message, elle envoie sa demande à une autre machine physique. De même, si au bout d'un temps donné, B ne reçoit pas une confirmation définitive de A à travers le troisième message, elle libère les ressources qu'elle avait réservées pour A en supposant que A s'est finalement désistée.

D'autre part, une autre amélioration possible de l'algorithme serait l'inclusion d'un module de prédiction de charge. Couplé à un module d'apprentissage, il pourrait prévoir le comportement du temps de réponse ou encore les taux d'utilisation des ressources en analysant un historique. En raison de l'aspect très nerveux et imprévisible des jeux vidéo (la consommation de ressources peut varier fortement d'une fraction de seconde à l'autre, et d'un niveau de jeu à un autre), l'intérêt d'un module de prédiction serait peut-être anecdotique pour une prédiction immédiate mais pourrait être important pour une prédiction à long terme.

Enfin, la fonctionnalité la plus importante que notre algorithme n'inclut pas est sans aucun doute une approche d'optimisation combinatoire. Par exemple, supposons qu'un processus (hébergé sur une machine virtuelle vm1) ait un temps de réponse trop élevé. De même, imaginons que les taux d'utilisation du processeur et de la mémoire vive de vm1 sont au-delà du seuil supérieur. Imaginons également qu'une unité supplémentaire de cœur CPU coûte plus cher pour le fournisseur d'IaaS qu'une unité de mémoire vive et qu'en plus, la machine physique hôte ne dispose plus de mémoire vive mais dispose de quelques cœurs de

processeur. Serait-il préférable d'augmenter immédiatement l'allocation de cœurs de processeur (ce qui reviendrait plus cher que la mémoire vive), ou faudrait-il migrer vm1 vers une autre machine physique avec de la mémoire ? Quitte à subir des coûts temporaires de migration, la mémoire coûterait moins cher au long terme. Toutefois, nous ne connaissons pas le comportement de l'application et rien n'indique que l'addition de mémoire résoudrait le problème. La résolution de ce genre de problème d'optimisation est complexe mais permettrait indéniablement d'obtenir des performances bien meilleures : l'algorithme de dimensionnement ferait des choix plus judicieux.

Concernant l'environnement de test, une implémentation réelle est certes très exigeante en temps de développement mais fournit les résultats les plus réalistes. Elle permet également de prendre en compte les phénomènes non modélisés ou les fonctionnalités non supportées par un simulateur. Dans notre cas par exemple, Simgrid ne peut être utilisé que pour tester une ressource : le processeur. La mémoire vive ou encore une carte graphique ne sont pas supportées. De même, Simgrid est incapable de simuler un niveau de charge pour une unité de calcul devant traiter une tâche (un processeur par exemple). Cela sera peut-être le cas des versions futures.

ANNEXE I

Vue schématique des modules d'un jeu vidéo

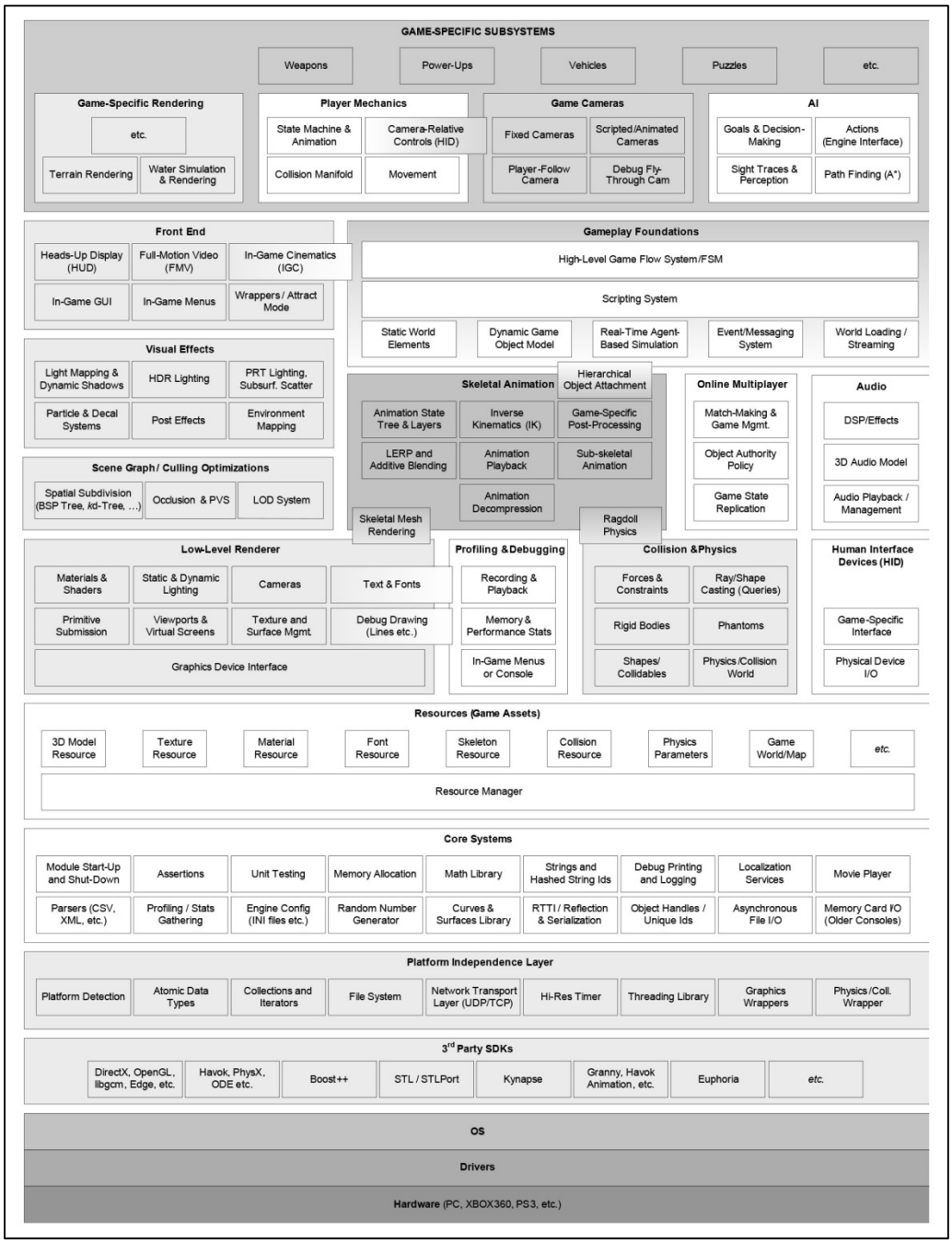


Figure-A I-1 Vue schématique des modules d'un jeu vidéo
Tirée de Gregory (2009)

ANNEXE II

Exemples de *workflows* de boucles principales

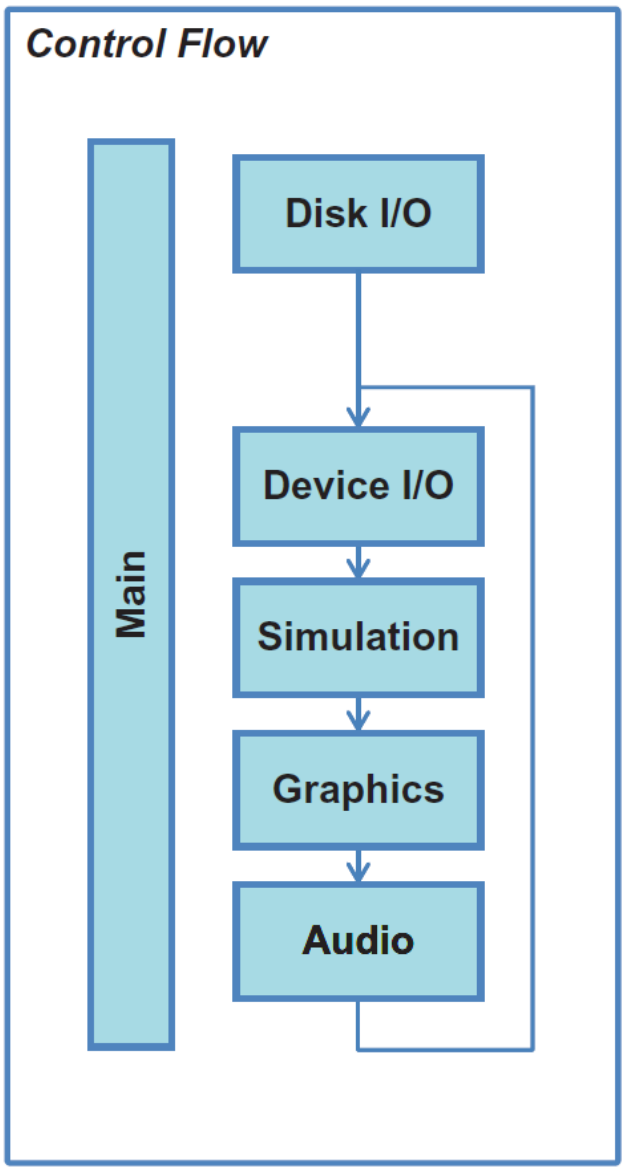


Figure-A II-1 *Workflow* d'une boucle principale simple
Tirée de Steed *et al.* (2009)

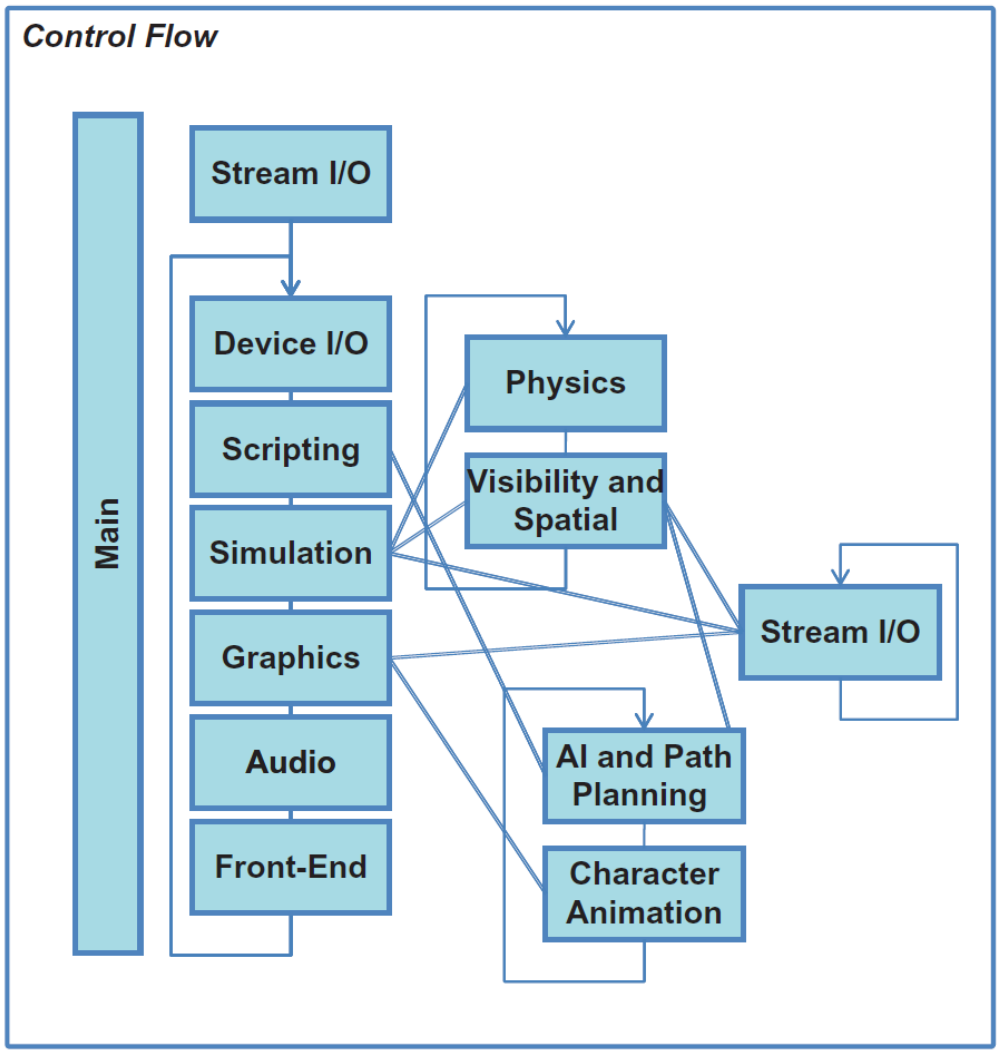


Figure-A II-2 *Workflow* d'une boucle principale sophistiquée, incluant plusieurs modules fonctionnant en parallèle
Tirée de Steed *et al.* (2009)

ANNEXE III

Principe du *three way handshake*

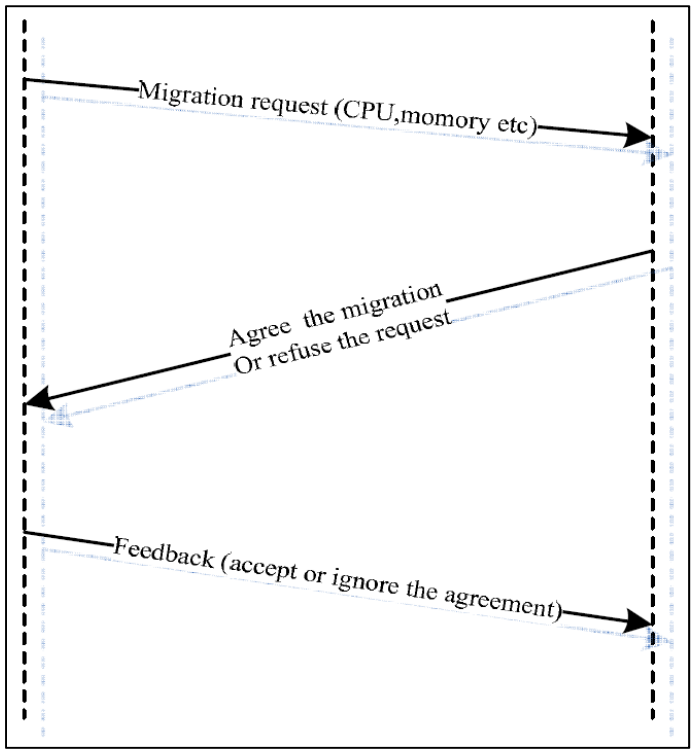


Figure-A III-1 Principe du *three way handshake*
Tirée de Yang *et al.* (2011)

BIBLIOGRAPHIE

Barboza, D.C., H. Lima, E.W.G. Clua et V.E.F Rebello. 2010. « A Simple Architecture for Digital Games on Demand Using Low Performance Resources under a Cloud Computing Paradigm ». In *2010 Brazilian Symposium on Games and Digital Entertainment (SBGAMES)*. (Florianopolis, Nov. 8-10 2010), p. 33-39.

Belqasmi, F., C. Azar, M Soualhia, N. Kara et R. Glitho. 2011. « A virtualized infrastructure for IVR applications as services ». In *Proceedings of ITU Kaleidoscope 2011: The fully networked human ? - Innovations for future networks and services*. (Cape Town, Dec. 12-14 2011), p. 1-7.

Beloglazov, A. et R. Buyya. 2013. « Managing Overloaded Hosts for Dynamic Consolidation of Virtual Machines in Cloud Data Centers under Quality of Service Constraints ». *IEEE Transactions on Parallel and Distributed Systems*. vol. 24, n°7, p. 1366-1379.

Beloglazov, A. et R. Buyya. 2012. « Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in Cloud data centers ». *Concurrency and Computation: Practice & Experience*, vol. 24, n°13, p. 1397-1420.

Beltran, M., Carlos Rey Juan et A. Guzman. 2012. « An automatic machine scaling solution for cloud systems ». In *19th International Conference on High Performance Computing (HiPC)*. (Pune, Dec. 18-22 2012), p. 1-10.

Bhatia, A.K., M. Hazra et S.K. Basu. 2009. « Better-Fit Heuristic for One-Dimensional Bin-Packing Problem ». In *Advance Computing Conference (IACC 2009)*. (Patiala, India, March 6-7 2009), p. 193-196.

Buyya, Rajkumar, James Broberg et Andrzej M. Goscinski. 2011. *Cloud Computing: Principles and Paradigms*. Hoboken, New Jersey : John Wiley & Sons, 664p.

Enet. 2013. « Enet ». In *enet: enet*. En ligne. <http://enet.bespin.org/>. Consulté le 8 janvier 2014.

Erl, Thomas, Ricardo Puttini et Zaigham Mahmood. 2013. *Cloud Computing: Concepts, Technology & Architecture*. Westford, MA : Prentice Hall, 528p.

Furht, Borko et Armando Escalante. 2010. *Handbook of Cloud Computing*. Boca Raton, FL : Springer, 634p.

Futuremark. 2013. « 3DMark ». In *Futuremark, best PC benchmarks and system performance tests*. En ligne. <http://www.3dmark.com/>. Consulté le 8 janvier 2014.

Gallagher, Michael D. 2011. « Entertainment Software Association : ESSENTIAL FACTS ABOUT THE COMPUTER AND VIDEO GAME INDUSTRY 2011, SALES, DEMOGRAPHIC AND USAGE DATA ». En ligne. 16p. http://www.theesa.com/facts/pdfs/ESA_EF_2011.pdf. Consulté le 7 septembre 2013.

Glinka, Frank, Allaithy Raed, Sergei Gorlatch et Alexander Ploss . 2009. « A service-oriented interface for highly interactive distributed applications ». In *Euro-Par'09 Proceedings of the 2009 international conference on Parallel processing*. (Delft, Aug. 25-28 2009), p. 266-277.

Gregoire, Marc, Nicholas A. Solter et Scott J. Kleper. 2011. *Professional C++*. Indianapolis, IN : Wiley Publishing, Inc., 1104p.

Gregory, Jason. 2009. *Game Engine Architecture*. Boca Raton, FL : A K Peters/CRC Press, an imprint of Taylor & Francis Group, an Informa business, 864 p.

Gueyoung, Jung, A. Matti Hiltunen , Kaustubh R. Joshi, Richard D. Schlichting et Calton Pu. 2010. « Mistral: Dynamically Managing Power, Performance, and Adaptation Cost in Cloud Infrastructures ». *In ICDCS '10 Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems*. (Genova, June 21-25 2010), p. 62-73. Washington, DC, USA : IEEE Computer Society.

Herlihy, Maurice et Nir Shavit. 2012. *The Art of Multiprocessor Programming*. Waltham, MA : Morgan Kaufmann, 536p.

Irrlicht Engine. 2013. In *Irrlicht Engine – A free open source 3D engine*. En ligne. <http://irrlicht.sourceforge.net/ >. Consulté le 8 janvier 2014.

Marinescu, Dan C. 2013. *Cloud Computing: Theory and Practice*. Waltham, MA : Morgan Kaufmann, 416p.

Maurina, Edward F. 2008. *Multiplayer Gaming and Engine Coding for the Torque Game Engine*. Wellesley, MA : GarageGames Inc., 450p.

McShaffry, Mike et David Graham. 2012. *Game Coding Complete*. 4e éd. Boston, MA : Cengage Learning, 960 p.

Mills, K., J. Filliben et C. Dabrowski. « Comparing VM-Placement Algorithms for On-Demand Clouds ». 2011. In *CLOUDCOM '11 Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*. (Athens, Nov. 29 - Dec. 1 2011), p. 91-98. Washington, DC, USA : IEEE Computer Society.

Nae,Vlad, Radu Prodan, Thomas Fahringer et Alexandru Iosup. 2009. « The impact of virtualization on the performance of Massively Multiplayer Online Games ». In *NetGames '09 Proceedings of the 8th Annual Workshop on Network and Systems*

Support for Games. (Paris, November 23-24, 2009), p. 1-6. NJ, USA : IEEE Press Piscataway.

Qt Project. 2013. « Qt 4 ». In *Qt Project*. En ligne. <http://qt-project.org/>. Consulté le 8 janvier 2014.

Rabin, Steve. 2009. *Introduction to Game Development*. 2nde éd., Coll. « Course Technology ». Boston, MA : Cengage Learning, 1008 p.

Rui, Han, Guo Li, M.M. Ghanem et Guo Yike. 2012. « Lightweight Resource Scaling for Cloud Applications ». In *CCGRID '12 Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. (Ottawa, May 13-16 2012), p. 644-651. Washington, DC, USA : IEEE Computer Society.

Sedgewick, Robert et Kevin Wayne. 2011. *Algorithms*. 4e éd. Boston, MA : Pearson Education, 976p.

Shea, Ryan et Jiangchuan Liu. 2013. « Cloud Gaming : Architecture and Performance ». *IEEE Network*, vol. 27, n°4, août 2013, p. 16-21.

Shekhar, Srikantaiah, Kansal Aman et Zhao Feng. 2008. « Energy aware consolidation for cloud computing ». In *HotPower'08 Proceedings of the 2008 conference on Power aware computing and systems*. (San Diego, CA, Dec. 2008), p. 1-5. Berkeley, CA, USA : USENIX Association.

Shrivastava, V., P. Zerfos, Kang-Won Lee, H. Jamjoom, Yew-Huey Liu et S. Banerjee. 2011. « Application-aware virtual machine migration in data centers ». In *INFOCOM, 2011 Proceedings IEEE*. (Shangai, April 10-15 2011), p. 66-70.

Skiena, Steven S. 2008. *The Algorithm Design Manual*. 2e éd. London : Springer-Verlag, 730p.

Sosinsky, Barry. 2009. *Cloud Computing Bible*. Indianapolis, IN : Wiley Publishing, Inc., 532p.

Steed, Anthony et Manuel Fradinho Oliveira. 2009. *Networked Graphics: Building Networked Games and Virtual Environments*. Burlington, MA : Morgan Kaufmann, 536p.

Stein, Johannes et Aung Sithu Kyaw. 2011. *Irrlicht 1.7.1 Realtime 3D Engine Beginner's Guide*. Birmingham, UK : Packt Publishing, 272p.

Stephens, Rod. 2008. *Beginning Database Design Solutions*. Indianapolis, IN : Wiley Publishing, Inc., 552p.

Verma, Akshat, Puneet Ahuja et Anindya Neogi. 2008. « pMapper: power and migration cost aware application placement in virtualized systems ». In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*. (Leuven, Belgium, 01-Dec. 05 2008), p. 243-264. New York, NY, USA : Springer-Verlag New York, Inc.

Von Hagen, William. 2008. *Professional Xen Virtualization*. Indianapolis, IN : Wiley Publishing, Inc., 405p.

Williams, Anthony. 2012. *C++ Concurrency in Action: Practical Multithreading*. Shelter Island, NY : Manning Publications, 325p.

Wood, Timothy, Prashant Shenoy, Arun Venkataramani et Mazin Yousif. 2007. « Black-box and gray-box strategies for virtual machine migration ». In *NSDI'07 Proceedings of the 4th USENIX conference on Networked systems design & implementation*. (Cambridge, MA, April 11-13 2007), p. 17.

Yang, Ke, Jianhua Gu, Tianhai Zha et Guofei Sun. 2011. « An Optimized Control Strategy for Load Balancing Based on Live Migration of Virtual Machine ». In *CHINAGRID'11 Proceedings of the 2011 Sixth Annual ChinaGrid Conference*. (Liaoning, Aug. 22-23 2011), p. 141-146.

Yazir, Yagiz Onat, Chris Matthews, Roozbeh Farahbod, Stephen Neville, Adel Guitouni, Sudhakar Ganti etYvonne Coady. 2010. « Dynamic Resource Allocation in Computing Clouds Using Distributed Multiple Criteria Decision Analysis ». In *CLOUD '10 Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing*. (Miami, FL, July 5-10 2010), p. 91-98. Washington, DC, USA : IEEE Computer Society.

Zhao, Zhou, Kai Hwang et Jose Villeta. 2012. « Game cloud design with virtualized CPU/GPU servers and initial performance results ». In *ScienceCloud '12 Proceedings of the 3rd workshop on Scientific Cloud Computing Date*. (Delft, June 18-22 2012), p. 23-30. New York, NY, USA : ACM.