MASTER EN SCIENCES INFORMATIQUES

"Mobile Cloud Offloading" : évaluation d'un compromis entre énergie et temps de réponse

Homble, Nicolas

Award date:
2016

Awarding institution:
Universite de Namur

Université de Namur
Faculté d'informatique
Année académique 2015–2016

"Mobile Cloud Offloading" : évaluation d'un compromis entre énergie et temps de réponse

Nicolas HOMBLÉ

Promoteur : _____ (Signature pour approbation du dépôt - REE art. 40)
Marie-Ange REMICHE

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Résumé

Dans ce mémoire, nous analysons le compromis énergie-temps de réponse inhérent aux systèmes mobiles avec migration d'opérations vers un cloud. Pour ce faire, nous nous appuyons sur une étude théorique de ce compromis réalisée dans l'article *Analysis of the Energy-Response Time Tradeoff for Mobile Cloud Offloading Using Combined Metrics*. Nous réalisons cette analyse sur base de simulations par évènements discrets du modèle mathématique défini dans cet article. Nous implémentons dans un premier temps un outil de simulation conforme aux spécifications du modèle mathématique. Ensuite, nous comparons leurs analyses de ce compromis aux analyses réalisées sur base de nos simulations. Cette comparaison fait suite à une analyse critique de leur approche.

Mots-clés — compromis énergie-temps de réponse, file d'attente, système mobile, migration vers un cloud, simulation par évènements discrets

Abstract

Our objectif is to analyse the tradeoff between energy and responses time appearing in mobile cloud offloading systems. Our work is based on the paper *Analysis of the Energy-Response Time Tradeoff for Mobile Cloud Offloading Using Combined Metrics*. They propose the analyses of this tradeoff using queueing theory. In our work, we use discret event simulation to simulate their proposed mathematical model of the system. We first build up a simulation tool according to their specified mathematical model. We then propose a discussion of their mathematical developments and accordingly compare our own analysis with theirs.

Index terms — energy-response time tradeoff, queueing system, offloading, mobile cloud computing, discret event simulation

Remerciements

J'adresse en premier lieu mes remerciements les plus vifs au professeur Marie-Ange Remiche. Non seulement elle m'a guidé tout au long de la rédaction de ce mémoire par ses multiples lectures (et relectures), mais elle n'a cessé durant cette année de prendre le temps de me rencontrer afin de répondre à mes questions, et me prodiguer aide et conseil. Sans elle, bien des points seraient demeurés confus et obscurs. Qu'elle trouve ici le témoignage de ma plus sincère reconnaissance.

J'adresse également des remerciements tout particulier à mon épouse, Claire Ducarme, pour son soutient sans faille durant toute cette année particulière à plus d'un titre puisqu'elle à vu naitre également notre fille Camille que je remercie de sa présence et de ses premiers sourires. En plus de son soutient elle fut une lectrice attentive me permettant d'améliorer sans cesse la qualité de mon travail.

Enfin, je remercie toutes les personnes qui ont participé de près ou de loin à l'aboutissement de ce travail, soit par leur oreille attentive, soit par leurs conseils, soit tout simplement par leur soutient.

Table des matières

Introduction

Nous voyons de nos jours un accroissement massif de l'utilisation de technologies mobiles. Smartphones et tablettes sont devenus incontournables (Cavazza, 2014). Le nombre d'applications développées sur ce support, tant pour les professionnels que pour les particuliers, suit la même tendance. Un simple passage sur l'un ou l'autre "app store" permet de se rendre compte de la quantité et de la diversité des applications disponibles.

Naturellement, ces applications tentent d'offrir à leurs utilisateurs la même expérience qu'ils ont sur des systèmes plus puissants (ordinateur portable ou ordinateur fixe). Les opérations qu'elles effectuent (calculs, lecture/écriture de fichiers, etc.) nécessitent par conséquent de plus en plus de ressources pour offrir aux utilisateurs une expérience confortable : processeurs plus puissants, mémoire vive et de stockage plus grande, interfaces réseaux plus rapides, écrans haute définition, etc. Hors, ces ressources sont fortement contraintes sur les appareils mobiles. Certaines opérations trop lourdes empêchent même le portage d'applications sur ce type de supports.

De plus, les batteries de ces appareils souffrent également. Leur temps de vie diminue drastiquement à mesure que l'utilisation du mobile s'intensifie. Il n'est pas rare aujourd'hui de voir la batterie d'un terminal mobile se décharger en un peu plus d'une heure d'utilisation ininterrompue. Hors, pour la majeure partie des utilisateurs de mobiles, l'augmentation de ce temps de vie est la préoccupation première (Kumar *et al.*, 2013).

Il est dès lors devenu indispensable, à moins d'assister à un bond technologique majeur, de trouver des solutions visant à soulager les mobiles d'opérations trop lourdes. Parmi ces solutions, celle qui est habituellement employée aujourd'hui est la migration de certaines de ces opérations du mobile vers des infrastructures de cloud. On appelle couramment cette technique le "mobile cloud offloading". Le terme *offload* dans ce contexte fait référence à la *migration* d'opérations dans le but de soulager le mobile. Cette technique permet d'allier la force du "cloud computing" à la flexibilité qu'offre l'utilisation de terminaux mobiles au quotidien.

Cependant, il est nécessaire de choisir les opérations à migrer avec discernement. Certaines, par exemple, ne peuvent s'exécuter en dehors du mobile. C'est le

cas d'opérations nécessitant l'accès aux périphériques de l'appareil (caméra, clavier, écran, micro, etc). En outre, les canaux de communication permettant la migration varient en vitesse, en disponibilité et en consommation d'énergie. Il est donc souhaitable d'optimiser cette migration en prenant en compte ces différents paramètres dès lors qu'on souhaite offrir à l'utilisateur les meilleures performances à moindre coût énergétique. Ces performances se mesurent, entre autres, par le temps de réponse (ou délai) qu'une application fournit aux différentes requêtes d'un utilisateur. Hors, les attentes d'un utilisateur varient selon l'application utilisée. Par exemple, l'envoi d'une vidéo sur Youtube afin qu'elle soit compressée et mise à disposition de tiers est une opération qui peut être effectuée plusieurs heures après la capture vidéo. Par contre, durant une partie de "Fast chess", le calcul du prochain coup de l'IA doit être rapide et prendre tout au plus quelques secondes. On peut considérer que toute application présente un certain degré de tolérance aux délais qui lui est propre.

En général, les applications avec un grand degré de tolérance sont des candidates de choix pour économiser l'énergie des mobiles. Effectivement, on peut retarder l'exécution de certaines de leurs opérations jusqu'à l'obtention des conditions optimales de consommation d'énergie. Par exemple, attendre l'accès à un réseau WiFi de qualité pour transmettre des données vers un cloud. A contrario, une application peu tolérante utilisera soit les ressources du mobile pour effectuer ses opérations, soit en migrera certaines par l'interface réseau disponible immédiatement sans se soucier de sa consommation. Il existe donc, pour une application, un compromis entre énergie consommée et temps de réponse attendu appelé *compromis énergie-temps de réponse*. Ce compromis peut être étudié selon différentes stratégies de migration afin d'être optimisé selon les besoins des applications concernées. Cette optimisation a déjà fait l'objet de nombreuses recherches ((Wu *et al.*, 2013), (Wu et Wolter, 2014) par exemple). L'objectif de ces recherches est l'étude du choix de la stratégie de migration d'une application et de l'interface la plus appropriée pour ce faire. De plus, il est possible aujourd'hui de morceler une application en plusieurs opérations pouvant être traitées séparément, la décision de les migrer peut donc se prendre pour chacune d'elles individuellement.

Le comportement d'un système permettant la migration d'opérations est complexe. Tout d'abord, la fréquence d'arrivée des opérations soumises au choix de migration dépend du comportement de l'application en cours d'utilisation. Ensuite, la vitesse de traitement d'une opération, tant localement par le mobile, qu'à distance par un cloud, dépend du nombre d'instructions à exécuter. Enfin, la vitesse de transmission d'une opération quelle que soit l'interface réseau utilisée dépend de la taille des données à transférer (opération complète ou uniquement les inputs) et de la qualité du réseau au moment du transfert. Pour étudier tous les cas possibles et en tirer des statistiques cohérentes, il faut observer le système en

fonction pendant de longues périodes de temps et s'assurer qu'il évolue dans des conditions très différentes afin de capturer tous les cas d'utilisation du système.

Nous utilisons les outils mathématiques de la théorie des files d'attente pour modéliser et analyser ces systèmes. L'objectif premier est de proposer un modèle mathématique du système étudié, d'implémenter un outil de simulation de ce modèle dans un langage de programmation adéquat et enfin, par sa simulation répétée, d'en analyser les performances. Nous utilisons l'approche dite *simulation par évènements discrets* pour réaliser cette implémentation. Elle permet, dans notre cas, de capturer le cheminement des opérations dans le système sous la forme d'une suite d'évènements discrets. Chaque évènement arrive à un moment bien précis et modifie l'état de notre système. Il est alors possible de collecter des données sur l'ensemble des opérations réalisées au sein du système et d'en extraire des statistiques, qui permettent par la suite d'analyser le comportement du modèle. Ce modèle étant une représentation du système réel, les conclusions que l'on en tire lui sont applicables dans la mesure où ce modèle reflète de manière valable cette réalité.

Notre mémoire comporte l'analyse du compromis énergie-temps de réponse évoqué précédemment, en utilisant la simulation par évènements discrets d'un modèle mathématique du système mobile avec migration d'opérations vers un cloud. Pour ce faire, nous modélisons ce système sous la forme d'un réseau de files d'attente très particulier. Nous proposons une simulation de ce dernier dans le cadre de deux stratégies de migration afin de comparer leurs performances respectives. Le système que nous modélisons est le suivant. Le terminal mobile (que nous appelons simplement *mobile*) est muni de deux interfaces physiques pour la transmission réseau : la première est une interface cellulaire, la seconde une interface WiFi. Le système d'exploitation du mobile prend en charge la décision de migrer une opération vers un cloud ou de la traiter localement. Le traitement local, la transmission par une interface réseau et le traitement par le cloud sont modélisés par une file d'attente aux spécifications dépendant des détails d'implémentation du système. Nous considérons deux stratégies de migration. La première, nommée *stratégie ininterrompue* dans la suite, utilise l'interface WiFi dès que possible et bascule sur l'interface cellulaire dès que la connexion WiFi est interrompue. La seconde stratégie, nommée *stratégie interrompue*, assigne les opérations lors de leur arrivée à l'une ou l'autre des deux interfaces et ce en parallèle. Dans ce cas-ci, lorsque le WiFi est interrompu, les opérations qui lui sont assignées s'accumulent en attendant d'être traitées une fois la connexion à nouveau disponible.

La comparaison de ces stratégies est réalisée, dans ce travail, par le biais d'une métrique appelée "Energy response-time weighted product". Cette métrique, proposée par l'auteur de notre article de référence (Wu *et al.*, 2015) (disponible dans sa version originale à l'annexe E), permet de mesurer le compromis énergie-temps de

réponse. Une pondération adaptée de l'une et l'autre de ces performances permet de favoriser leur importance relative.

Notre outil de simulation s'appuie principalement sur le modèle mathématique proposé dans l'article "Analysis of energy-response time Tradeoff for mobile cloud offloading using combined metrics" (Wu *et al.*, 2015). L'analyse qui y est proposée est limitée aux performances moyennes (vu la trop grande complexité analytique du modèle). Nous utiliserons les résultats théoriques proposés dans cet article pour valider notre outil de simulation. Nous comparerons ensuite l'analyse théorique proposée par l'auteur de l'article à l'analyse que nous réalisons dans les mêmes conditions mais cette fois en simulant le système à l'aide de notre outil de simulation.

Organisation du texte

Nous décrivons ici succinctement les différents chapitres de notre mémoire afin de guider le lecteur dans son parcours.

Dans le chapitre 1, nous décrivons le système mobile avec migration. Nous relatons le principe de migration et l'intérêt des stratégies de migration mises en place. Nous vérifions également que celles-ci, d'une part, rejoignent les besoins propres aux utilisateurs de système mobile, et, d'autre part, soient réalisables techniquement. Ce chapitre est très général et intéressera tout lecteur qui souhaite comprendre le cadre dans lequel se situe ce mémoire.

Dans le chapitre 2, nous détaillons le modèle mathématique proposé dans notre article de référence du système décrit au premier chapitre. Nous spécifions ensuite ce modèle.

Dans le chapitre 3, nous commençons, aux sections 3.1 et 3.2, par détailler l'implémentation de notre outil de simulation. Le lecteur souhaitant poursuivre notre travail d'implémentation ou utiliser cet outil pourra s'y référer par la suite, par exemple, s'il souhaite obtenir plus d'informations sur une classe particulière de cet outil. Connaitre les détails de l'architecture de l'outil de simulation n'est pas essentiel à la bonne compréhension de l'analyse du compromis énergie-temps de réponse que nous réalisons au chapitre 4. Nous procédons à la vérification de l'implémentation à la section 3.3. Cette section est sans conteste la plus importante de notre mémoire car elle met en évidence nos analyses sur l'article de référence. Celles-ci sont à la base, d'une part, du processus de développement de l'outil de simulation, et, d'autre part, des analyses du chapitre 4.

Dans le chapitre 4, nous comparons, pour le compromis énergie-temps de réponse, les analyses proposées par l'auteur de notre article de référence aux analyses que nous réalisons sur base de simulations obtenues à l'aide de l'outil détaillé au chapitre précédent.

Chapitre 1

Description du système mobile avec migration

Comme nous l'avons vu dans l'introduction, nous souhaitons analyser le compromis énergie-temps de réponse d'un système mobile avec migration d'opérations sur un cloud. Ce compromis existe du fait de la possibilité pour un tel système mobile de décider, pour chaque opération, soit de leur exécution locale au moyen de ses ressources propres, soit de leur exécution à distance par un cloud. Un tel système est composé de trois ensembles d'activités permettant la prise en charge et l'exécution des opérations :

1. L'activité liée à l'exécution locale des opérations
2. Les activités liées à l'exécution distante des opérations
3. L'activité liée à la décision de la migration

Le premier et le troisième ensemble d'activités sont pris en charge par le mobile, le deuxième, par contre, est conjointement pris en charge par le mobile et le cloud. Ces ensembles d'activités, que nous allons détailler sous peu, prennent en charge des *opérations*, c'est-à-dire des entités responsables de l'exécution de parties d'application. Par exemple, l'enregistrement d'une vidéo par une application ad hoc pourrait se diviser en trois opérations. La première traite les images brutes reçues de la caméra et les transmet à la deuxième opération qui se charge de les compresser pour limiter la taille de la vidéo sur le disque. La troisième opération se charge, quant à elle, d'enregistrer le résultat sur le disque et présente éventuellement la vidéo dans une galerie. Lorsqu'elle est décomposée en opérations, une application est beaucoup plus flexible quant à son éventuelle migration. En effet, si on ne décompose pas une application nécessitant l'accès aux périphériques du mobile, il n'est pas possible de la migrer car elle ne peut alors pas être complètement exécutée à distance, même si toutes les opérations qu'elle contient ne sont pas nécessairement soumises à cette contrainte. Une fois l'application partitionnée adé-

quatement, il est probable qu'elle puisse être partiellement migrée. Dans l'exemple envisagé précédemment, la première et la troisième opération ne pourraient être migrées car elles dépendent du mobile pour le bon déroulement de leur exécution. Par contre, la deuxième, pourrait être exécutée en dehors du mobile. Il suffit par exemple qu'un service dédié à l'exécution de cette opération soit disponible sur un système distant, que le mobile lui transmette le contenu à compresser et qu'il le reçoive en retour.

Dans le cadre des applications mobiles, une application partitionnée donne donc naissance, comme nous venons de le suggérer, à deux catégories d'opérations : les *migrables* et les *non-migrables*. Les opérations non-migrables sont inconditionnellement soumises à l'activité d'exécution locale qui est entièrement prise en charge par les ressources du mobile. Elles ne peuvent en aucun cas être migrées car elles accèdent soit à des périphériques du mobile durant leur exécution (la caméra, le micro, le port USB, etc.), soit elles réagissent à des stimuli de l'utilisateur (interaction avec le clavier, l'écran, le mouvement, etc.). Les opérations migrables, par contre, sont potentiellement exécutables à distance.

Dans ce chapitre, nous détaillons dans un premier temps, les trois ensembles d'activités de notre système de migration aux sections 1.1, 1.2, 1.3, respectivement. Par la suite, nous mettons en évidence les problèmes de performance d'un tel système à la section 1.4. Ces performances dépendent en grande partie des stratégies de migration mises en place. Par stratégies, nous entendons les politiques employées pour migrer effectivement les opérations. Par exemple, le choix de l'interface physique utilisée pour la migration des opérations en fonction du type de celles-ci. Nous détaillons ces stratégies à la section 1.5. Enfin, une architecture présentant les caractéristiques de notre système de migration sera détaillée dans la section 1.6.

1.1 L'activité liée à l'exécution locale des opérations

Cette activité est entièrement prise en charge par les ressources du mobile (CPU, mémoire, I/O device, etc). Les opérations exécutées durant cette activité peuvent être migrables ou non-migrables. Cependant, comme nous souhaitons analyser le comportement du système dans le cadre de la migration des opérations, seule nous importe l'observation des opérations migrables. Pour pouvoir se concentrer sur elles uniquement, nous considérons que leur exécution durant cette activité est prioritaire. Elles sont ainsi toujours exécutées avant les opérations non-migrables. Durant l'activité, le temps d'exécution d'une opération dépend de la puissance de traitement du mobile. Précisément, ce temps dépend du compor-

tement séquentiel d'exécution des opérations par le mobile. L'énergie nécessaire dépend, quant à elle, de la puissance dissipée par les différentes ressources mises à contribution durant l'exécution. Cette énergie est prélevée sur la batterie du mobile.

1.2 Les activités liées à l'exécution distante des opérations

Ces activités, conjointement prises en charge par le mobile et le cloud, sont au nombre de trois :

1. Choix de l'interface de transmission des opérations
2. Transmission des opérations
3. Exécution des opérations

L'activité liée au choix de l'interface de transmission des opérations est prise en charge par un processus exécuté au sein de l'OS du mobile. Celui-ci a pour but de choisir l'interface réseau à utiliser pour la transmission des opérations vers le cloud. Les décisions qu'il prend dépendent de la stratégie de migration qui est d'application. Nous avons évoqué dans l'introduction les deux stratégies de migration que nous considérons pour ce système : la stratégie ininterrompue et la stratégie interrompue. Les particularités de chacune d'elles ainsi que les raisons du choix de ces stratégies sont exposées à la section 1.5. Le délai encouru par la prise de décision et l'énergie que cela consomme au niveau du mobile est négligeable.

L'activité de transmission des opérations est prise en charge par les interfaces réseau du mobile. Nous en considérons deux dans ce système. La première est une interface cellulaire de type 2G/3G, la seconde est une interface Wlan de type WiFi. Le temps de transmission d'une opération dépend de l'interface réseau utilisée tout comme l'énergie consommée durant la transmission. On constate que l'interface cellulaire est plus lente que l'interface WiFi et qu'elle consomme plus (Cuervo *et al.*, 2010), (Shu *et al.*, 2013),(Balasubramanian *et al.*, 2009). On considère également l'interface cellulaire comme étant disponible en permanence (IBPT, 2016) contrairement à l'interface WiFi disponible par intermittence selon la disponibilité de *Hotspot*. Comme pour l'exécution locale d'une opération, l'énergie consommée pour la transmission est prélevée sur la batterie du mobile et correspond à la puissance dissipée par l'interface réseau durant la transmission.

Enfin, l'activité d'exécution des opérations est prise en charge par le cloud. On peut voir celui-ci comme un espace dans lequel les opérations migrées sont prises en charge par différents services afin d'être exécutées. Différents types d'opérations impliquent différents types de services. Ces services utilisent les ressources physiques du cloud pour assurer leur fonctionnement. Ces ressources sont issues d'un

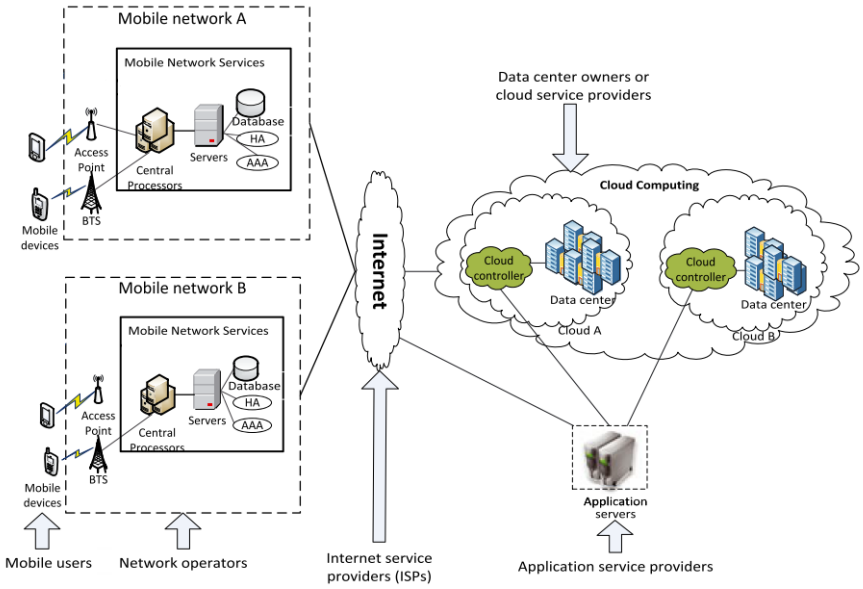


FIGURE 1.1 – Migration d'une opération (Kumar *et al.*, 2013)

ensemble de systèmes informatiques inter-connectés appelé *data-center*. Tous les services possibles ne sont pas tous installés nativement dans le cloud. Au besoin, si un service n'est pas disponible, le cloud peut le télécharger depuis un serveur d'applications (Kumar *et al.*, 2013), qui est responsable du maintien de ces services (disponibilité, mise à jour, etc). Le temps d'exécution d'une opération dans le cloud dépend de la puissance de traitement de celui-ci. En général, les data-center qui prennent en charge physiquement les opérations permettent l'exécution en parallèle d'un grand nombre d'entre-elles. L'énergie consommée pour exécuter une opération sur le cloud n'est pas prélevée sur la batterie du mobile.

Concernant l'ensemble des activités liées au traitement distant, il faut préciser que la connexion entre le mobile et le cloud n'est pas une connexion directe. Schématiquement on peut décrire le cheminement des opérations sur le réseau comme le propose la figure 1.1. Sur celle-ci, on constate en effet qu'une opération transmise par l'une ou l'autre interface réseau transite tout d'abord par l'antenne relais (*base station* du réseau en question, c'est-à-dire un *acces point (AP)* pour l'interface WiFi ou une *base transceiver station (BTS)* pour le réseau cellulaire). Ces antennes sont propres à chaque *opérateur de réseau*. Cela n'empêche pas un même mobile de pouvoir passer par un opérateur de réseau différent selon l'interface utilisée. L'opération est ensuite acheminée vers le *processeur central* de l'opérateur de réseau, lui-même relié à plusieurs serveurs fournissant des services tels que l'authentification de l'utilisateur et les autorisations qui lui sont associées. Toutes ces informations sont stockées dans les bases de données de l'opérateur de réseau. En-

suite, l'opération est acheminée par l'opérateur de réseau vers le cloud en passant par Internet. A son arrivée dans le cloud, l'opération est analysée par un contrôleur afin de l'orienter vers le service adéquat permettant son exécution. Comme nous l'avons vu, si le service n'est pas disponible au sein du cloud, le contrôleur se charge de le télécharger depuis un serveur d'applications. Par la suite, le service prend en charge l'opération et l'exécute en utilisant les ressources du data-center du cloud. Enfin, le résultat est renvoyé à la source en suivant le cheminement inverse.

1.3 L'activité liée à la décision de la migration

Cette troisième activité est prise en charge par un processus exécuté au sein de l'OS du mobile. Celui-ci décide, pour chaque opération migrable, s'il est préférable de la transmettre ou non à l'ensemble des activités liées à l'exécution distante. Une opération migrable n'est donc pas nécessairement migrée : elle peut très bien être traitée localement au sein du mobile. Le choix opéré a pour objectif l'optimisation du compromis énergie-temps de réponse que nous détaillons dans la section suivante.

1.4 Compromis énergie-temps de réponse

Le compromis entre énergie et temps de réponse intervient dès lors que le mobile a l'opportunité de se délester de l'exécution locale de certaines opérations en les migrant vers un cloud. Dans un tel système, l'énergie consommée par le mobile et le temps d'exécution d'une opération ne sont plus uniquement fonction des caractéristiques propres des ressources du mobile. En effet, il faut prendre en compte également le coût de la migration ainsi que le gain réalisé par l'utilisation du cloud.

Du point de vue de la rapidité d'exécution, l'exécution locale des opérations au sein du mobile est moins rapide que l'exécution sur le cloud. Cependant, lors de l'exécution distante, il faut prendre également en compte les délais de la migration des opérations. Ces délais dépendent de l'interface réseau utilisée.

Du point de vue de l'énergie consommée, pour l'activité d'exécution locale, il faut prendre en compte la puissance dissipée durant l'exécution des opérations. Par contre, pour les activités d'exécution distante, l'énergie consommée par le cloud n'entre pas en compte contrairement à l'énergie consommée par les interfaces réseau utilisées durant l'activité de transmission des opérations. Ici encore, cette énergie est fonction de la puissance dissipée par les interfaces réseau pendant ce temps.

Le compromis énergie-temps de réponse apparait lorsque l'on souhaite réduire

d'une part l'énergie consommée et d'autre part le temps de réponse attendu. Ce temps de réponse est intrinsèquement lié aux attentes d'un utilisateur selon l'application utilisée, alors que l'énergie consommée doit être minimisée autant que possible, pour autant que cela n'entrave pas l'expérience de l'utilisateur.

L'étude de ce compromis est donc inhérent au besoin de migrer certaines opérations pour soulager le mobile et les conditions pour ce faire. Comme nous l'avons vu, d'une part, les interfaces ont des caractéristiques différentes en termes de disponibilité, de consommation énergétique et de vitesse de transmission, et, d'autre part, les attentes de l'utilisateur varient en fonction de l'application utilisée. Il est important, pour analyser ce compromis, d'établir des stratégies de migration correspondant aux attentes de l'utilisateur et aux caractéristiques des interfaces réseau. L'outil de décision de migration doit connaitre les circonstances dans lesquelles l'utilisation de l'une ou l'autre stratégie sera favorisée, et cela selon le type d'opérations à exécuter. Dans le cadre de ce mémoire, nous nous concentrerons sur l'étude des stratégies évoquées plus haut (cf. section 1.2) et présentées dans (Wu *et al.*, 2015). Dans la section suivante, nous détaillons ces stratégies.

1.5 Stratégies de migration

Les stratégies de migrations ont pour but de définir la politique d'utilisation des interfaces réseau pour le transfert des opérations vers le cloud. Ces stratégies dépendent des applications et de leurs attentes en termes de temps de réponse aux actions de l'utilisateur. Prenons deux exemples, une partie d'échec et l'application d'enregistrement de vidéo que nous avons évoqué lors de la description du partitionnement d'une application (cf. introduction du chapitre 1). Pour la partie d'échec, les opérations à transmettre au cloud doivent utiliser l'interface disponible la plus rapide. La raison est simple, l'utilisateur ne peut se permettre d'attendre trop longtemps entre chaque coup. Le mieux étant alors de privilégier l'interface WiFi, et d'utiliser l'interface cellulaire, même si plus coûteuse, lorsque le réseau WiFi est indisponible. Par contre, l'enregistrement d'une vidéo peut attendre quelques heures avant d'être disponible à la lecture. Dans ce cas, on peut aisément attendre la disponibilité d'un réseau WiFi pour la transmission de la vidéo brute vers le cloud afin qu'elle y soit compressée et privilégier ainsi l'économie d'énergie.

En résumé, tout utilisateur souhaite économiser un maximum d'énergie pour pouvoir utiliser son mobile le plus longtemps possible sans devoir le recharger. Cependant, cela ne peut se faire au détriment d'une expérience d'utilisation inconfortable. Si on se concentre uniquement sur les applications migrables, celles qui nécessitent des temps de réponse courts doivent bénéficier d'une stratégie de migration permettant un accès rapide et fiable au Cloud, alors qu'une application

qui ne le nécessite pas pourra se permettre des interruptions de connexions avec pour objectif l'économie d'énergie.

Ces deux cas nous ont amené à considérer la *stratégie ininterrompue* pour des applications nécessitant des temps de réponse courts et la *stratégie interrompue* pour celles qui ne le nécessitent pas. Cette dernière est naturellement orientée sur l'économie d'énergie.

Lorsque la stratégie ininterrompue est d'application, le processus de décision privilégie l'interface WiFi si elle est disponible. Dans le cas contraire, il utilise l'interface cellulaire. Avec cette stratégie, les opérations migrées sont transmises au cloud par un flux ininterrompu.

Lorsque la stratégie interrompue est d'application, le processus de décision transmet les opérations en parallèle sur les deux interfaces réseau. Pour ce faire, il faut que le mobile soit muni d'un protocole de transport fiable, en mode connecté, permettant la gestion d'un flux de données passant en même temps par deux interfaces réseau distinctes. Il existe à notre connaissance deux candidats possibles : le protocole *Stream Control Transmission Protocol (SCTP)* (Stewart *et al.*, 2000) et le protocole *MultiPath TCP (MPTCP)* (Ford *et al.*, 2013). Tous deux travaillent au niveau de la couche transport du modèle OSI. Le protocole MPTCP semble le plus adapté car il gère des *flux TCP* (Postel, 1981) en parallèle. Cette caractéristique, d'une part, le rend compatible avec les applications actuelles utilisant le protocole TCP, et ce, sans devoir les modifier, et, d'autre part, lui permet de passer pratiquement inaperçu au sein des systèmes d'interconnexion réseau tel que proxies et firewalls. Les opérations transmises par l'interface cellulaire, quant à elles, ne subissent aucune interruption. Par contre, lorsque l'interface WiFi est indisponible, les opérations transmises par cette interface s'accumulent et attendent la disponibilité du réseau WiFi.

Avant de conclure ce chapitre, nous exposons dans la section suivante un exemple d'architecture illustrant le système de migration mobile.

1.6 Mobile Assistance Using Infrastructure (MAUI) : une architecture de migration

Pour terminer, décrivons succinctement l'architecture MAUI (Cuervo *et al.*, 2010) qui permet la prise en charge de la migration d'applications. Celle-ci est particulièrement intéressante, car elle s'intègre parfaitement au système que nous avons décrit.

L'objectif poursuivi par cette architecture est la migration de code pour des applications smartphone afin d'économiser l'énergie consommée par celui-ci. Pour ce faire, MAUI propose un environnement de développement dans lequel les dévelop-

peurs peuvent marquer les méthodes sujettes à migration (celles qui sont migrables dans notre système). Lorsqu'une telle méthode doit être exécutée, MAUI exécute un processus d'optimisation pour décider de son exécution locale ou distante. A l'issu de chaque exécution distante, MAUI récolte des informations sur les conditions de la migration afin de parfaire son processus de décision. Si la connexion est interrompue, MAUI est capable de poursuivre l'exécution de la méthode localement.

On retrouve dans cette architecture les trois ensembles d'activité constitutifs de notre système, à savoir le processus d'exécution local, le processus d'exécution distant et le processus de décision de la migration. Cependant, le cloud est ici remplacé par un serveur MAUI. Celui-ci contient l'environnement permettant l'exécution des méthodes migrées. Le principe de fonctionnement est similaire à ce que nous avons décrit pour le système que nous souhaitons simuler. Comme nous venons de le voir, les applications sont partitionnées en méthodes migrables et non-migrables par le développeur de l'application. Ces méthodes sont ensuite soumises à un processus de décision qui va les orienter soit vers le serveur MAUI pour qu'elles soient exécutées en dehors du smartphone, soit elles sont exécutées localement par le smartphone. Le processus de décision de la migration, appelé dans cette architecture le *framework d'optimisation*, prend en compte dans ces décisions d'une part, le coût de la migration d'une méthode et le gain encouru par la migration, et, d'autre part, la qualité de la connexion réseau entre le smartphone et le serveur MAUI. Concernant les méthodes, MAUI évalue la quantité de données nécessaire à transférer vers le serveur pour leur exécution. Il compare ce coût au gain de cycles CPU économisé par le smartphone. Pour la qualité de la connexion réseau, il analyse continuellement le débit disponible et la latence entre le smartphone et le serveur. A partir de ces données propres aux méthodes et à la qualité du réseau, le processus choisi d'exécuter les méthodes à distance ou localement.

1.7 Conclusion

Nous avons décrit, dans ce chapitre, l'ensemble du système que nous souhaitons modéliser et simuler, ceci afin d'évaluer le compromis énergie-temps de réponse inhérent à la migration possible d'opérations du mobile vers le cloud. Nous avons décrit ce compromis et ses origines. Nous avons exposé deux stratégies de migrations répondant chacune à des besoins différents en termes de performances attendues. Nous avons enfin proposé un exemple d'architecture de migration : MAUI illustrant ce système. Le prochain chapitre sera dévolu à la modélisation du système mobile avec migration.

Chapitre 2

Modélisation du système mobile avec migration

Dans ce chapitre, nous proposons l'utilisation d'un réseau de files d'attente afin de construire le modèle mathématique du système mobile avec migration décrit dans le chapitre précédent. Les propriétés des différents composants de ce système s'adaptent parfaitement à une modélisation par files d'attente. En effet, dans la description du système, nous avons vu que le temps de traitement d'une opération que ce soit lors de son exécution locale, lors de son exécution distante ou encore lors de sa transmission par une interface réseau, est composé du temps d'attente éventuel imposé par le traitement des opérations précédentes et de son temps de traitement propre. C'est la raison pour laquelle une file d'attente est le candidat idéal car d'une part, le temps de service du serveur de la file modélise le temps de traitement de l'opération et, d'autre part, le temps passé dans la file avant d'arriver au serveur modélise le temps de traitement des opérations précédentes.

Nous proposons à la section 2.1 de décrire le réseau de files d'attente modélisant le système et justifions les choix posés lors de sa construction. Ce modèle est proposé par l'auteur de notre article de référence (Wu *et al.*, 2015) qui, pour rappel, réalise la même analyse que la nôtre mais par une approche analytique. Dans la section 2.2, nous spécifions ce modèle afin qu'il reflète le comportement aléatoire du système mobile avec migration.

2.1 Description du modèle mathématique

Dans la suite, les termes *réseau*, *réseau de files* et *réseau de files d'attente* sont interchangeables. Si le contexte d'emploi du terme réseau rendait ambigu son utilisation, nous préciserions de quel type de réseau il s'agit. Le graphique proposé dans la figure 2.1 représente le réseau de files d'attente modélisant le système

mobile avec migration. Il est inspiré de celui présenté dans (Wu *et al.*, 2015), à la figure 1 de l'article. Ce réseau est composé de trois parties, chacune modélisant les trois ensembles d'activités du système. Nous retrouvons naturellement modélisé :

1. L'activité liée à l'exécution locale des opérations
2. Les activités liées à l'exécution distante des opérations
3. L'activité liée à la décision de la migration

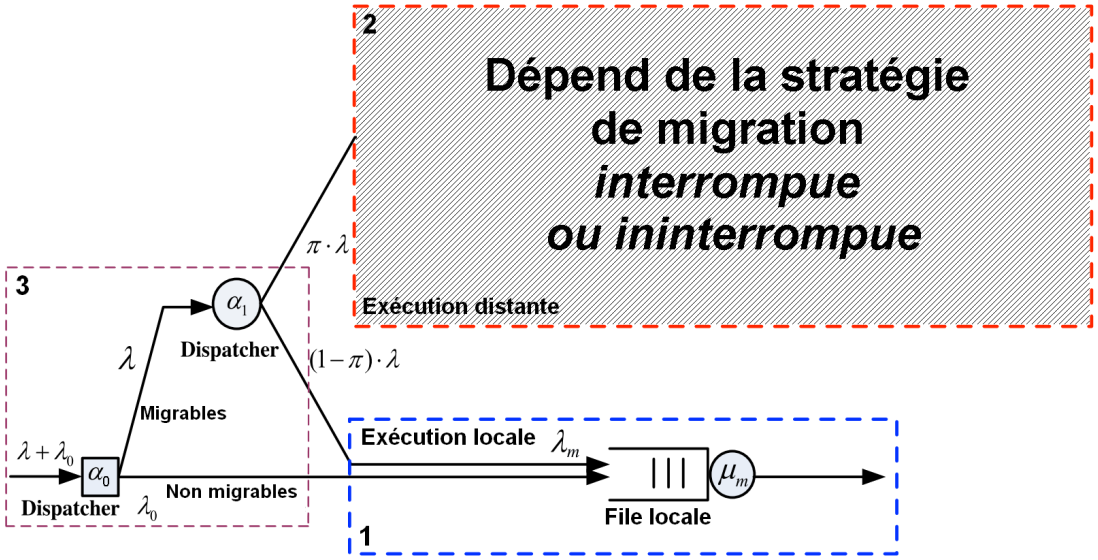


FIGURE 2.1 – Représentation du modèle mathématique inspiré de (Wu *et al.*, 2015), figure 1.

Nous proposons de décrire pas à pas le modèle en reprenant chaque ensemble d'activités du système. Nous commençons par la modélisation de l'activité d'exécution locale à la section 2.1.1, ensuite les activités d'exécution distante à la section 2.1.2 et enfin l'activité de décision de la migration à la section 2.1.3.

2.1.1 Modélisation de l'activité liée à l'exécution locale des opérations

L'activité liée à l'exécution locale des opérations du système est modélisée comme une simple file d'attente dans le cadre inférieur en pointillé dans la figure 2.1 (cadre 1). Nous l'appelons *file locale*.

L'exécution d'une opération est représentée ici comme un serveur réalisant une opération à la vitesse moyenne de μ_m opérations par unité de temps. Autrement

dit, une opération est exécutée en moyenne en $1/\mu_m$ unité de temps. La consommation moyenne d'énergie par opération est dès lors p_m/μ_m joules représentant la puissance dissipée par les ressources du mobile durant l'exécution de l'opération, avec p_m la puissance nominale du serveur. Nous avons vu que le mobile traite les opérations de manière séquentielle. Ce comportement est modélisé par la *discipline de service FIFO* (First-in First-out) de la file. L'utilisation de cette discipline de service signifie que lorsqu'une opération est en cours d'exécution, les suivantes s'accumulent dans la file. Lorsque l'exécution d'une opération est terminée, la suivante dans la file démarre la sienne, et ce dans l'ordre de leur arrivée.

De plus, comme nous l'avons vu, seule nous importe, pour nos analyses, l'observation des opérations migrables. Par conséquent, cette file gère en priorité les opérations migrables, ce qui veut dire que dès qu'une opération migrable arrive dans cette file, elle se place devant la première opération non-migrable s'y trouvant éventuellement. De plus, si le serveur de la file est occupé par une opération non-migrable, son traitement est immédiatement stoppé au profit de l'opération migrable. Par cette politique de priorité, du point de vue des opérations migrables, les non-migrables n'existent tout simplement pas.

La raison de ce choix d'ordonnancement avec priorité s'explique uniquement par la mesure de performance qui nous intéresse. En effet, celle-ci ne concerne que l'exécution des opérations migrables.

2.1.2 Modélisation des activités liées à l'exécution distante des opérations

Les activités liées à l'exécution distante sont modélisées au moyen d'un réseau de files d'attente dans le cadre supérieure en pointillé dans la figure 2.1 (cadre 2). Le réseau de file est différent selon la stratégie de migration qui est d'application (section 1.5). Chaque stratégie est dès lors modélisée par un réseau de files d'attente particulier. Le réseau modélisant la stratégie interrompue est visible à la figure 2.2 et le réseau de la stratégie ininterrompue à la figure 2.3. Nous présentons dans un premier temps les points communs aux deux stratégies, nous reviendrons ensuite sur leurs particularités.

Points communs

Nous appelons *file cellulaire* la file d'attente modélisant l'interface cellulaire et *file WiFi* celle modélisant l'interface WiFi. L'assignation des opérations à l'une ou l'autre file dépend de la stratégie de migration. Le temps de transfert de chaque interface fonction des conditions radio est modélisé par le temps de service d'un serveur. Les qualités de celui-ci sont fonction de la qualité de la transmission radio. La transmission d'une opération étant représentée ici comme un serveur

réalisant cette transmission à la vitesse moyenne de μ_g opérations par unité de temps, pour la file modélisant l'interface cellulaire. Autrement dit, une opération est transmise en moyenne en $1/\mu_g$ unité de temps. De la même manière, on a pour la file WiFi en moyenne μ_w opérations transmises par unité de temps avec, cette fois, une opération transmise en moyenne en $1/\mu_w$ unité de temps. Nous avons vu que le temps de transmission d'une opération dépend de la taille de l'opération à transmettre et du débit qu'offre l'interface réseau utilisée. Pour toute interface réseau, on peut alors considérer que

$$\mu = s/E[X] \tag{2.1}$$

où $E[X]$ est la taille moyenne d'une opération, et s la vitesse de transmission de l'interface réseau.

Avant de définir la discipline de service des deux files, nous fixons un point de vocabulaire afin d'éviter toute ambiguïté sur les termes employés pour décrire une file d'attente. Nous posons à cet effet que le terme *buffer*, dès à présent, fait référence à l'endroit où les opérations s'accumulent dans une file en attendant d'être traitées par son serveur.

La discipline des deux files est FIFO : les opérations sont donc transmises dans leur ordre d'arrivée. Le temps de transmission d'une opération est donc au final, composé de son temps effectif de transmission (temps passé dans le serveur de la file) et du temps qu'elle aura éventuellement attendu dans la file (temps passé dans le buffer de la file).

Tout comme pour la file locale, l'énergie moyenne consommée par opération est p_g/μ_g joules pour la file cellulaire et p_w/μ_w joules pour la file WiFi avec, cette fois, p_g et p_w les puissances nominales respectives des serveurs de la file cellulaire et de la file WiFi.

Une fois la transmission réalisée, c'est au cloud d'exécuter l'opération. Le cloud, comme pour l'exécution locale, est modélisé par une file d'attente simple de vitesse moyenne de traitement μ_c que nous appelons *file cloud*. Il exécute donc en moyenne une opération en $1/\mu_c$ unité de temps. Cependant, dans ce cas, le nombre de serveurs est de très grande taille. Tout se passe comme s'il était infini, c'est-à-dire que dès qu'une opération arrive, elle est immédiatement prise en charge. Le temps d'exécution total d'une opération est, dès lors, égal au temps passé dans le serveur qu'elle aura rejoint. Ce choix de modélisation se justifie par la capacité du cloud à traiter un grand nombre d'opérations en parallèle comme nous l'avons vu à la section 1.2. Les opérations ne s'accumulant jamais dans la file, celle-ci n'a pas de discipline de service particulière. Nous avons également vu à la section 1.2 que l'énergie du cloud n'est pas prélevée sur la batterie du mobile, elle ne nous intéresse donc pas pour l'évaluation du compromis énergie-temps de réponse. C'est la raison pour laquelle nous considérons dans notre modèle que le cloud ne consomme pas d'énergie. Voyons à présent les particularités de chaque stratégie.

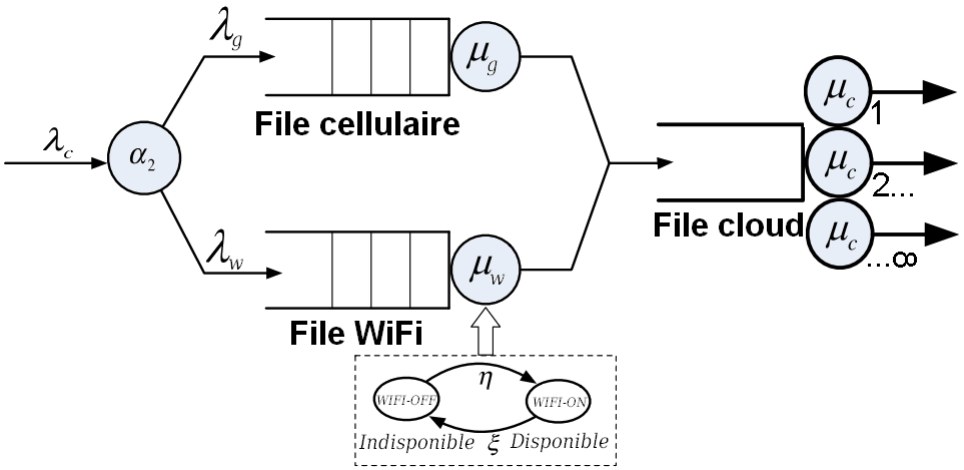


FIGURE 2.2 – Modèle de file d'attente caractérisant le comportement de la stratégie interrompue, inspiré de la figure 5 de (Wu *et al.*, 2015)

Stratégie interrompue

Le réseau de files d'attente propre à la stratégie interrompue est représenté à la figure 2.2. Comme nous l'avons vu précédemment, dans cette stratégie, le processus responsable du choix de l'interface peut transmettre les opérations par les deux interfaces en parallèle. Ce choix a pour objectif d'optimiser le compromis énergie-temps de réponse. Ce processus est modélisé par le *dispatcher* α_2 dont les décisions sont modélisées par une probabilité de transmettre une opération par l'une ou l'autre file.

L'interface cellulaire est modélisée comme une simple file ce qui correspond, pour cette stratégie, au comportement voulu de cette interface dans le système, c'est-à-dire être accessible en permanence. Par contre, l'interface WiFi est modélisé comme une file dont le serveur alterne entre des périodes d'activité et des périodes d'inactivité. Ce comportement du serveur représente le comportement du réseau WiFi qui alterne entre des périodes de disponibilité et des périodes d'indisponibilité.

Stratégie ininterrompue

Le réseau de files d'attente propre à la stratégie ininterrompue est proposé à la figure 2.3. Pour cette stratégie, le processus responsable du choix de l'interface n'a pas d'optimisation à réaliser. Il doit simplement transmettre les opérations à migrer par l'interface WiFi si elle est disponible et, dans le cas contraire, par l'interface cellulaire. De ce fait, nous pouvons modéliser ce comportement par

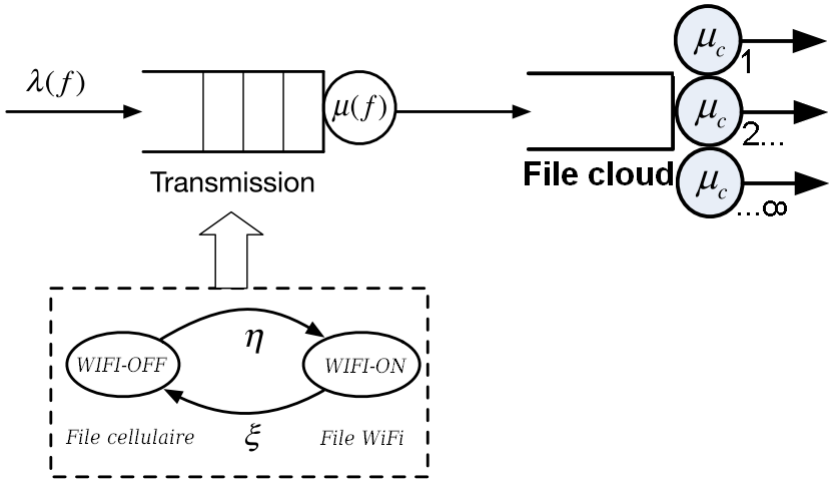


FIGURE 2.3 – Modèle de file d'attente caractérisant le comportement de la stratégie ininterrompue, inspiré de la figure 3 de (Wu *et al.*, 2015)

une file d'attente dont le serveur change d'état selon la disponibilité du réseau WiFi. Si le réseau WiFi est disponible, le serveur de la file est dans l'état *WIFI-ON* et transmet les opérations selon les caractéristiques du serveur modélisant la transmission par l'interface WiFi. Si le réseau WiFi est indisponible, le serveur de la file est dans l'état *WIFI-OFF* et transmet alors les opérations selon les caractéristiques du serveur modélisant la transmission par l'interface cellulaire.

2.1.3 Modélisation de l'activité liée à la décision de la migration

Terminons la description de la modélisation des activités telle que représentée dans la figure 2.1 en développant le modèle de l'activité liée à la décision de migration. Celle-ci est modélisée par les *dispatcher* α_0 et le *dispatcher* α_1 dans le cadre inférieur gauche de la figure 2.1 (cadre 3). Le dispatcher α_0 modélise la distinction entre opérations migrables et non-migrables. Les opérations non-migrables sont inconditionnellement transmises à la file locale. Comme nous l'avons vu, ces opérations sont traitées, dans tous les cas, après les opérations migrables. Les opérations migrables, quant à elles, sont acheminées vers le dispatcher α_1 qui modélise le processus de décision de la migration exécuté au sein du mobile. Nous faisons l'hypothèse, dans notre modèle, que le dispatcher α_1 réalise un choix aléatoire pour déterminer le chemin suivi par les différentes opérations migrables d'une même application. En effet, prendre en compte la complexité d'implémentation en

opérations d'une application, dans notre modèle, serait accorder trop d'importance à ce niveau de détail par rapport à la mesure de performance qui nous intéresse.

Dans le chapitre précédent, nous avons considéré que le temps mis par le processus de décision de la migration ainsi que celui mis par le processus de choix de l'interface de transmission étaient négligeables, tout comme l'énergie consommée pendant ce temps. En conséquence, dans le modèle mathématique, les dispatchers α_0, α_1 et α_2 font des choix instantanés qui ne consomment pas d'énergie.

2.2 Spécification du modèle mathématique

La spécification du modèle mathématique doit nous permettre de refléter le comportement aléatoire du système mobile avec migration. Par exemple, nous avons vu que l'arrivée des opérations dans le système dépend du comportement de l'application en cours d'utilisation. Pour un grand nombre d'applications différentes dont on observe le comportement durant une longue période de temps, on peut considérer que l'arrivée des opérations dans le système a un comportement aléatoire. Pour simuler l'arrivée aléatoire d'opérations dans le système, on peut utiliser un *processus de renouvellement* dont l'objectif est de produire des instants d'arrivée d'opérations à des intervalles de temps aléatoires. Autre exemple, le temps d'exécution d'une opération au sein de l'activité locale dépend du nombre d'instructions qu'elle comporte. Nous avons modélisé la prise en charge de cette exécution par le serveur de la file locale. Nous avons dit que le temps d'exécution d'une opération était en moyenne de $1/\mu_m$ unités de temps, ce qui correspond, pour rappel, en *moyenne* à μ_m opérations exécutées par unité de temps. Pour modéliser le temps d'exécution *propre* d'une opération, on peut cette fois associer à chaque exécution un temps donné par une variable aléatoire. Celle-ci est distribuée selon une fonction de répartition particulière dont l'espérance est $1/\mu_m$. Spécifier le modèle mathématique sert à caractériser son comportement aléatoire. Pour ce faire, nous proposons de reprendre le cheminement d'une opération dans le système et de spécifier pour chaque composant qu'elle traverse son comportement aléatoire. Nous commençons bien entendu par spécifier, en tout premier lieu, le processus d'arrivée des opérations.

2.2.1 Processus d'arrivée des opérations

Le premier comportement à spécifier est l'arrivée des opérations dans le système. Pour ce faire, nous l'avons évoqué plus haut, nous proposons d'utiliser un *processus de renouvellement*. Un tel processus est, par définition, un processus aléatoire $\{T_n; n \in \mathbb{N}_0\}$ décrivant les instants T_n aléatoires où se produisent des évènements particuliers. Dans le cas qui nous concerne, ces évènements sont les

arrivées d'opérations dans le système. On appelle ces évènements des *renouvellements*. Un processus de renouvellement est tel que la distance séparant deux renouvellements successifs est une collection $\{X_n; n \in \mathbb{N}_0\}$ de variables aléatoires supposées indépendantes et identiquement distribuées, selon une fonction de répartition F, tel que $F(0) = 0$ et $F(\infty) = 1$. On parle encore de processus de renouvellement de loi F (Bass, 2011).

Pour caractériser l'arrivée des opérations dans notre système, nous utilisons un processus de renouvellement particulier de *loi exponentielle* de paramètre λ appelé *processus de Poisson*. Intuitivement, cela signifie que les opérations arrivent dans le système à intervalle de temps donné par une collection de variables aléatoires X_n distribuées comme des exponentielles de paramètre λ. L'indice n fait alors référence à l'intervalle séparant l'arrivée de la $(n-1)^{\text{ième}}$ opération de la $n^{\text{ième}}$ opération pour $n > 1$. Pour $n = 1$, il s'agit de l'intervalle de temps séparant l'arrivée de la première opération du temps 0. Le paramètre λ de l'exponentielle, aussi appelé *facteur d'échelle*, influe sur la durée des intervalles. Autrement dit, sur la fréquence d'arrivée des opérations. Plus λ est grand, plus la fréquence d'arrivée est grande, plus λ est petit et plus la fréquence est petite. En moyenne, le nombre d'opérations arrivant par unité de temps dans le système est égale à λ, on parle encore ici du taux moyen d'arrivées ou fréquence moyenne d'arrivées. L'intervalle moyen de temps entre deux arrivées est égale à $1/\lambda$.

Une propriété remarquable des processus de Poisson, que nous allons utiliser par la suite, est que la superposition de deux processus de Poisson indépendants de paramètres respectifs λ_1 et λ_2 revient considérer un processus de Poisson de paramètre $(\lambda_1 + \lambda_2)$. La réciproque est vraie également : un processus de Poisson de paramètre λ peut être décomposé en deux processus de Poisson indépendants de paramètre respectif λ_1 et λ_2 avec $\lambda = \lambda_1 + \lambda_2$.

2.2.2 Arrivée des opérations dans le système et décision de leur migration

Le processus de Poisson que nous utilisons pour caractériser l'arrivée des opérations dans le système est de paramètre $(\lambda + \lambda_0)$ où λ correspond au taux moyen d'arrivée des opérations migrables et λ_0 au taux moyen d'arrivée des opérations non-migrables.

Toutes les opérations arrivent, en tout premier lieu, dans le dispatcher α_0 (cadre 3) qui redirige les opérations migrables vers le dispatcher α_1 et les opérations non-migrables vers la file locale. Par la propriété des processus de Poisson vue ci-dessus, le processus d'arrivée des opérations migrables dans le dispatcher α_1 est donc un processus de Poisson de paramètre λ. Les opérations non-migrables arrivent, quant à elles, dans la file locale selon un processus de Poisson de paramètre λ_0.

Nous avons vu que le dispatcher α_1 opère un choix aléatoire quant à la redirection d'une opération migrable, soit vers la file locale si elle n'est pas migrée, soit vers la file cloud si elle est migrée. Ce comportement aléatoire est modélisé par une probabilité π_{α_1} que l'opération soit migrée et une probabilité $(1 - \pi_{\alpha_1})$ qu'elle ne le soit pas, avec $0 \leq \pi_{\alpha_1} \leq 1$. On observe deux cas particuliers :

- si $\pi_{\alpha_1} = 0$, alors toutes les opérations sont exécutées par la file locale
- si $\pi_{\alpha_1} = 1$, alors toutes les opérations sont exécutées par la file cloud

Grâce à ce paramètre π_{α_1}, le dispatcher α_1 pourra être implémenté de telle manière à optimiser la consommation d'énergie et les délais de transmission. Après transmission, une opération migrée arrive soit à la file cloud selon un processus de Poisson de paramètre $\lambda_c = \pi_{\alpha_1} \cdot \lambda$, soit à la file locale selon un processus de Poisson de paramètre $\lambda_m = (1 - \pi_{\alpha_1}) \cdot \lambda$. Nous commençons par spécifier la file locale, nous reviendrons ensuite sur la file cloud et sur les files cellulaire et WiFi.

2.2.3 Exécution des opérations par la file locale

Nous avons vu que les opérations provenant du dispatcher α_1 arrivent dans la file locale selon un processus de Poisson de paramètre λ_m et que les opérations provenant du dispatcher α_0 arrivent dans cette file selon un processus de Poisson de paramètre λ_0. Si on regarde l'arrivée de toute opération confondue dans cette file, elle est caractérisée par un processus de Poisson de paramètre $(\lambda_0 + \lambda_m)$. Cependant, nous avons vu également que seules les opérations migrables nous intéressent dans l'analyse du compromis énergie-temps de réponse. C'est la raison pour laquelle la file locale a une politique de priorité ayant pour effet que, du point de vue des opérations migrables, tout se passe comme si les opérations non-migrables n'existent pas. Dès lors, on peut mesurer nos performances en ne regardant que les opérations migrables. L'arrivée des opérations y est donc caractérisée par un processus de Poisson de paramètre λ_m.

Nous avons vu également que le serveur de la file exécute une opération à la vitesse moyenne de μ_m opérations par unité de temps, autrement dit, en moyenne une opération en $1/\mu_m$ unité de temps. Dès lors, nous modélisons le temps d'exécution des opérations dans cette file par une collection de variables aléatoires respectant cette contrainte sur la moyenne. Notre choix se porte naturellement sur des distributions exponentielles de paramètre μ_m. Ces caractéristiques en font une *file d'attente M/M/1* (Leemis et Park, 2006).

2.2.4 Transmission des opérations vers la file cloud

Spécifions à présent la transmission des opérations vers la file cloud pour chaque stratégie en commençant par la stratégie interrompue.

Stratégie interrompue

Pour cette stratégie, comme nous l'avons vu, les décisions du dispatcher α_2 sont mises en œuvre par une probabilité de transmettre une opération par l'une ou l'autre file.

Avec une probabilité π_{α_2} sachant que $0 \leq \pi_{\alpha_2} \leq 1$, les opérations sont transmises à la file cellulaire et avec une probabilité $(1 - \pi_{\alpha_2})$, elles sont transmises à la file WiFi. Nous avons deux cas particulier :

- si $\pi_{\alpha_2} = 0$, alors toutes les opérations sont transmises à la file WiFi.
- si $\pi_{\alpha_2} = 1$, alors toutes les opérations sont transmises à la file cellulaire.

Les opérations arrivent dans la file cellulaire selon un processus de Poisson de paramètre $\lambda_g = \pi_{\alpha_2} \cdot \lambda_c$, pour un taux moyen d'arrivées de λ_g opérations par unité de temps, ce qui correspond à une opération en moyenne toutes les $1/\lambda_g$ unité de temps.

Nous avons vu que le serveur de la file cellulaire exécute une opération à la vitesse moyenne de μ_g opérations par unité de temps, c'est-à-dire une opération en moyenne en $1/\mu_g$ unité de temps. A nouveau, nous faisons le choix de modéliser le temps d'exécution des opérations dans cette file par une collection de variables aléatoires distribuées comme des exponentielles de paramètre μ_g. A l'instar de la file locale, les caractéristiques de la file cellulaire en font une file d'attente M/M/1.

Pour la file WiFi, les opérations arrivent selon un processus de Poisson de paramètre $\lambda_w = (1 - \pi_{\alpha_2}) \cdot \lambda_c$, pour un taux moyen d'arrivée de λ_w opérations par unité de temps, ce qui correspond à une opération en moyenne toutes les $1/\lambda_w$ unité de temps. Nous avons vu que le serveur de la file WiFi exécute une opération à la vitesse moyenne de μ_w opérations par unité de temps, c'est-à-dire une opération en moyenne en $1/\mu_w$ unité de temps. Nous modélisons le temps d'exécution des opérations dans cette file par une collection de variables aléatoires distribuées comme des exponentielles de paramètre μ_w.

Nous avons vu également que le serveur de cette file alterne des périodes d'activité et des périodes d'inactivité dont une représentation de la trajectoire est disponible à la figure 2.4. Nous modélisons ce comportement par un processus de Markov alternant entre deux états. Ceux-ci sont *WIFI-ON* (interface WiFi disponible) et *WIFI-OFF* (interface WiFi indisponible). Le temps de séjour dans chaque état est aléatoire et distribué comme une exponentielle de paramètre η pour l'état WIFI-OFF et ξ pour l'état WIFI-ON. Le temps moyen d'une période d'activité (WIFI-ON) est de $1/\xi$ unité de temps alors que le temps moyen d'une période d'inactivité (WIFI-OFF) est de $1/\eta$ unité de temps. Nous appelons cette file *une file d'attente* $M/M/1_{ON/OFF}$.

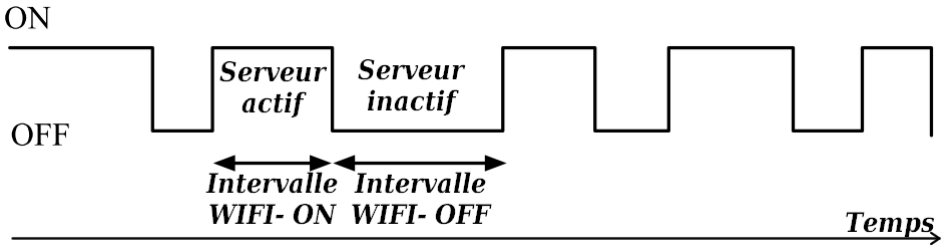


FIGURE 2.4 – Alternance de la disponibilité du WiFi pour la stratégie interrompue, inspiré de la figure 2 de (Wu *et al.*, 2015)

Stratégie ininterrompue

Pour la stratégie ininterrompue, comme nous l'avons vu, le processus de décision n'opère pas de choix liés à l'optimisation du compromis énergie-temps de réponse. Il transmet les opérations à l'interface disponible, en privilégiant le WiFi. Tout se passe comme si les opérations arrivaient directement dans une file, appelée *file transmission* dont l'état dépend de la disponibilité du WiFi. Si le WiFi est indisponible, le serveur de la file est dans l'état WIFI-OFF. Si le WiFi est disponible, il est dans l'état WIFI-ON. Rappelons que, pour cette stratégie, c'est le serveur aux caractéristiques du WiFi qui est utilisé dans l'état WIFI-ON alors que le serveur aux caractéristiques cellulaires est utilisé dans l'état WIFI-OFF. L'alternance entre ces états est également représentée à la figure 2.5. Le principe est le même que pour l'alternance entre les périodes d'activité et d'inactivité du serveur de la file WiFi dans la stratégie interrompue. Le processus de Markov caractérisant le passage d'un état à l'autre pour le serveur fonctionne de manière similaire. Dès lors, le processus global d'arrivée dans la file quel que soit l'état du serveur est alors un processus de Poisson de paramètre λ_c.

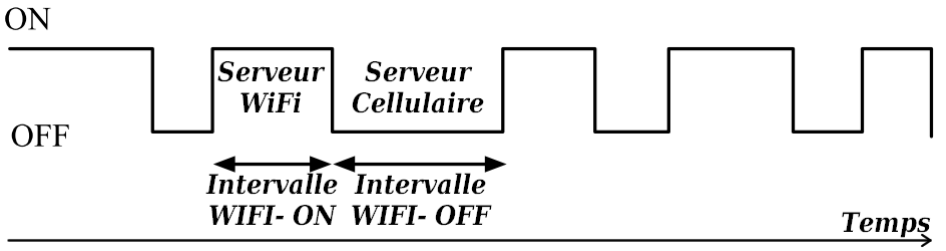


FIGURE 2.5 – Alternance de la disponibilité du WiFi pour la stratégie ininterrompue, inspiré de la figure 2 de (Wu *et al.*, 2015)

Le serveur WiFi utilisé dans l'état WIFI-ON exécute une opération à la vitesse moyenne de μ_2 [1] opérations par unité de temps, ce qui nous donne en moyenne une opération exécutée en $1/\mu_2$ unité de temps. Nous modélisons le temps d'exécution des opérations dans cet état par une collection de variables aléatoires distribuées comme des exponentielles de paramètre μ_2. Dès lors, cette file, dans l'état WIFI-ON, est une file M/M/1. Le serveur cellulaire utilisé dans l'état WIFI-OFF exécute, quant à lui, une opération à la vitesse moyenne de μ_1 opérations par unité de temps, ce qui nous donne en moyenne une opération exécutée en $1/\mu_1$ unité de temps. Nous modélisons le temps d'exécution des opérations dans cet état par une collection de variables aléatoires distribuées comme des exponentielles de paramètre μ_1. Dès lors, cette file, dans l'état WIFI-OFF est également une file M/M/1.

2.2.5 Exécution des opérations par la file cloud

Qu'importe la stratégie d'application, les opérations une fois transmises arrivent à la file cloud au taux λ_c. Les opérations sont toutes prises en charge par un serveur dès leur arrivée afin d'être exécutées et ce, à une vitesse moyenne de μ_c opérations par unité de temps autrement dit, en moyenne, une opération en $1/\mu_c$ unité de temps. Nous modélisons le temps d'exécution des opérations dans la file cloud par une collection de variables aléatoires distribuées comme des exponentielles de paramètre μ_c. Cette file ayant un nombre de serveurs considérés comme infini est appelée *file d'attente* $M/M/\infty$.

2.3 Conclusion

Dans ce chapitre, nous avons tout d'abord décrit le modèle mathématique du système mobile avec migration et justifié les choix posés lors de sa construction. Nous avons vu que ce modèle est composé d'un réseau de files d'attente et de dispatchers modélisant les différentes entités du système. La file locale et la file cloud modélisent respectivement l'exécution locale et distante des opérations. La file cellulaire, la file WiFi et la file transmission modélisent les interfaces de transmission réseau. Le dispatcher α_0 modélise la distinction entre opérations migrables et non-migrables, le dispatcher α_1 modélise le processus de décision de la migration et le dispatcher α_2 modélise le processus de choix de l'interface de transmission réseau.

1. Nous utilisons pour les vitesses de transmission des serveurs de cette file les indices 1 et 2 pour distinguer ceux-ci de la stratégie interrompue utilisant g et w. On considère que l'état WIFI-OFF est l'état indicé 1 utilisant le serveur aux caractéristiques de l'interface cellulaire et l'état WIFI-ON, l'état indicé 2 aux caractéristiques de l'interface WiFi.

Ensuite, nous avons spécifié chaque entité du modèle mathématique afin que celles-ci reflètent le comportement aléatoire du système. Pour ce faire, nous avons commencé par définir le processus d'arrivée des opérations. Pour les dispatchers, nous avons spécifié le comportement aléatoire de leur processus de décision et pour les différentes files d'attente nous avons spécifié le processus d'arrivée des opérations dans la file et les temps d'exécution de celles-ci au sein du serveur.

Pour terminer, nous proposons à la table 2.1, un tableau récapitulatif des paramètres attribués aux différentes entités du modèle mathématique. Nous nous référerons à ce tableau dans la suite lorsque nous affecterons des valeurs à ces paramètres afin d'analyser les performances des différentes entités dans différents cas de figure. Le choix des valeurs utilisées est réalisé de telle manière que le comportement des entités du modèle mathématique correspondent au comportement réel des composants du système de migration mobile qu'elle représente. Nous justifierons ces choix lorsqu'ils se présenteront.

TABLE 2.1 – Tableau récapitulatif des paramètres attribués aux différentes entités du modèle mathématique

Processus d'arrivée des opérations dans le système

Type	**Distribution**	**Paramètre**
Processus de Poisson	Exponentielle	λ

Dispatcher

Nom	**Paramètre**
α_1	π_{α_1}
α_2	π_{α_2}

Files d'attente

Nom	**Discipline**	**Puissance**	**Distribution**	**Paramètre**
File locale	FIFO	p_m	Exponentielle	μ_m
File cellulaire	FIFO	p_g	Exponentielle	μ_g
File WiFi	FIFO	p_w	Exponentielle	μ_w
File cloud	–	–	Exponentielle	μ_c

Chapitre 3

Implémentation de l'outil de simulation

Notre objectif dans ce mémoire est de concevoir un outil permettant la simulation du modèle mathématique dans le respect de sa spécification. Autrement dit, nous avons développé un programme qui implémente l'ensemble des entités du réseau de files d'attente du modèle mathématique et assure que leur comportement est conforme aux spécifications de celles-ci. Cet outil de simulation utilise, comme nous l'avons évoqué dans l'introduction, la technique de simulation par évènements discrets. Cette technique associe les changements d'états du système à des évènements discrets dans le temps. Ces changements d'états sont, entre autres, l'arrivée d'une opération au point d'entrée du réseau de files d'attente, la sortie d'une opération du serveur d'une de ces files, le passage de l'état actif à l'état inactif d'un serveur, etc. Pour réaliser la simulation du système, l'outil dispose de deux pièces maitresses. La première est une implémentation de la simulation de files d'attente qui modélise la prise en charge et l'acheminement des opérations. La seconde est un *simulateur* permettant la planification et le déclenchement, en temps voulu, des évènements générés par l'implémentation des entités qui simule le système. La notion de temps dans le simulateur est virtuelle. Le temps est certe divisé de manière discrète selon des unités spécifiées par celui qui analyse les performances du système (nous parlons en toute généralité d'*unité de temps* notées *ut*), mais elles ne correspondent en rien au temps d'exécution du programme.

Afin de pouvoir s'assurer de la correction de l'implémentation de l'outil de simulation, nous vérifions sa conformité aux spécifications du modèle mathématique dans la section 3.3. Pour ce faire, nous avons à notre disposition le fruit de l'étude analytique du modèle mathématique qui est proposé dans notre article de référence (Wu *et al.*, 2015). Vérifier l'implémentation revient, dans ce cas, à comparer différents résultats théoriques obtenus par l'analyse des entités des réseaux de files d'attente des deux stratégies que nous avons détaillées, aux résultats obtenus

par la simulation répétée de ces réseaux dans les mêmes conditions. A cette fin, nous paramétrons tout d'abord ces réseaux de files. Nous affectons aux différentes implémentations des entités des réseaux que nous simulons les valeurs employées dans (Wu *et al.*, 2015). Sous les mêmes conditions, résultats théoriques et simulés devraient être vraisemblablement les mêmes.

L'outil de simulation a été implémenté dans le langage *Java* en s'appuyant, pour le développement de certains modules [1], sur la librairie *Simulation Stochastique en Java (SSJ)* dans sa version 2.6.2 (L'Ecuyer *et al.*, 2002). Celle-ci propose notamment des générateurs de nombres aléatoires et des générateurs de variables aléatoires aux distributions diverses. Lorsque cela sera nécessaire, nous détaillerons les classes provenant de cette librairie au fur et à mesure de notre exposé.

Dans ce chapitre, nous commençons dans la section 3.1 par détailler l'implémentation du modèle mathématique et dès lors, spécifions le choix des composants qui intégreront les phénomènes aléatoires tels que spécifiés par les entités des réseaux de files d'attente. Dans la section 3.2, nous détaillons le simulateur exposant tout d'abord son principe de fonctionnement en toute généralité et ensuite son implémentation. Le simulateur, tel que nous le désignons dans ce travail, est une partie de notre outil de simulation, dont le rôle est de coordonner les différents évènements générés par les modules implémentés selon la spécification du modèle mathématique. Enfin, nous vérifions l'implémentation de notre outil de simulation à la section 3.3. Dans la suite, afin de décrire nos classes, nous nous aidons de diagramme de classes. Afin d'éviter de surcharger la lecture, nous ne proposons sur ces diagrammes que les classes, leurs attributs et méthodes nécessaires à la compréhension de l'implémentation de la partie du modèle que nous décrivons. Cependant, afin d'avoir une vue globale sur l'ensemble du diagramme représentant notre outil de simulation, nous proposons à l'annexe B le diagramme complet.

3.1 Implémentation du modèle mathématique

Afin de simuler le modèle mathématique décrit au chapitre précédent, il est nécessaire d'implémenter, d'une part, les différentes entités qui constituent ce modèle conformément à leurs spécifications, et, d'autre part, le processus de renouvellement d'arrivée des opérations. Pour rappel, ce processus est un processus de Poisson (section 2.2.1) et parmi les entités du modèle, nous avons vu différentes files d'attente et dispatchers (sections 2.2.2, 2.2.3, 2.2.4, 2.2.5). Ces différentes entités, comme nous l'avons vu, sont reliées entre-elles afin de former ce que nous avons appelé *un réseau de files d'attente*. Les opérations générées par le processus de Poisson arrivent dans ce réseau au niveau d'une entité que nous qualifions de

1. Un *module*, comme nous le verrons sous peu, est l'implémentation d'une entité du modèle mathématique. Un module peu donc se différencier en un disptacher ou une file d'attente.

point d'entrée. Dans le modèle mathématique, cette entité est le dispatcher α_0. Les opérations peuvent sortir du réseau en deux points, que nous qualifions de *points de sortie* et qui sont situés à la sortie de la file cloud et à la sortie de la la file locale. La figure 3.1 rappelle le schéma illustrant les réseaux de files d'attente de la stratégie interrompue et de la stratégie ininterrompue.

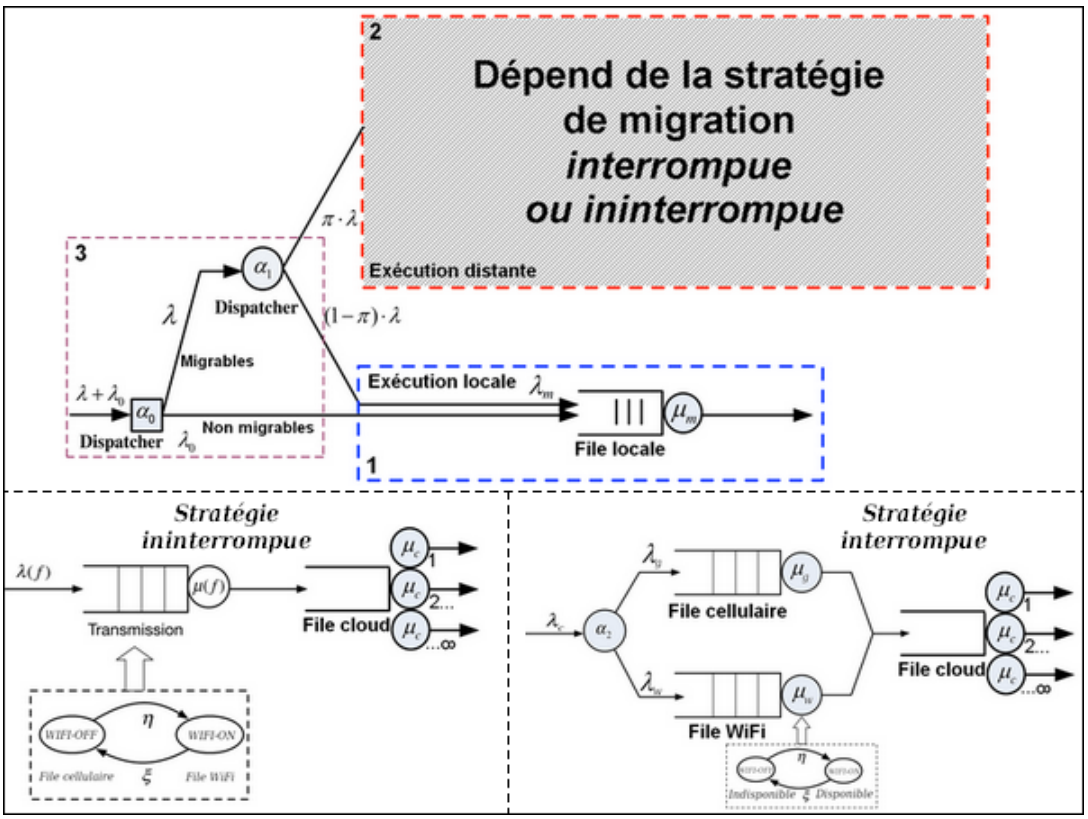


FIGURE 3.1 – Représentation du modèle mathématique inspirée de (Wu *et al.*, 2015)

Afin d'implémenter les différentes entités du modèle mathématique, nous avons développé une hiérarchie de classes que nous détaillons à la section 3.1.1. Nous abordons ensuite l'implémentation du processus de renouvellement à la section 3.1.2.

3.1.1 Hiérarchie de classes modélisant l'implémentation des entités du modèle

Dans notre implémentation du modèle mathématique, chacune des entités est représentée par une classe dont les responsabilités correspondent aux spécifications mathématiques de celle-ci. Le diagramme représentant la hiérarchie de ces classes est disponible à la figure 3.2.

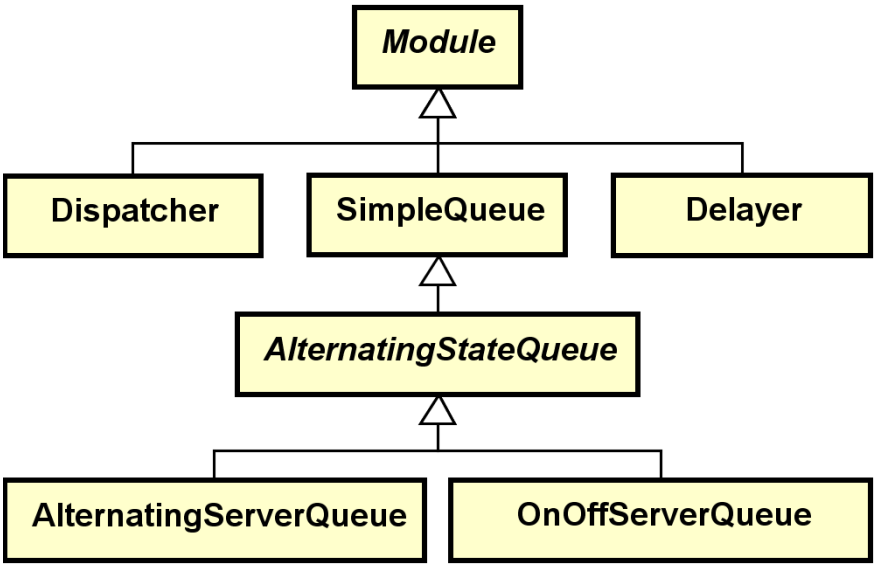


FIGURE 3.2 – Hiérarchie de classes représentant l'implémentation des entités du modèle mathématique

Dans un premier temps, nous décrivons cette hiérarchie de manière générale. Nous détaillerons ensuite les points importants pour chaque classe qui la compose. Dans la suite, lorsque nous parlons d'une classe, nous la notons en italique avec la première lettre en majuscule. Par contre, lorsque nous parlons, en tout généralité, d'une instance de cette classe, nous utilisons le nom de la classe en minuscule sans style particulier. Par exemple, dispatcher est une instance de la classe *Dispatcher*. Nous présentons, en tout premier lieu, la classe *Operation* qui n'est pas directement incluse dans cette hiérarchie mais dont les instances représentent les opérations qui cheminent à travers nos réseaux de files d'attente. Celles-ci sont donc naturellement manipulées par toutes les classes de notre hiérarchie. Le diagramme de cette classe est visible à la figure 3.3.

Une opération n'ayant d'autre utilité que la récolte d'informations temporelles et énergétiques, la classe qui la représente permet tout simplement de conserver cette information tout au long du cheminement de celle-ci à travers un réseau de

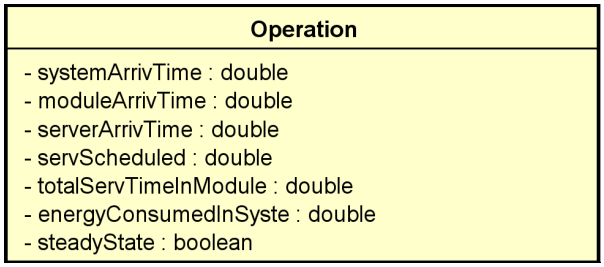


FIGURE 3.3 – Diagramme de la classe *Operation*

files. Elle contient les variables de classe suivante :

- *systemArrivTime*, qui est un double permettant de stocker le moment d'arrivée de l'opération dans le système. Celle-ci sert au calcul du temps passé par l'opération dans le réseau.
- *moduleArrivTime*, qui est un double permettant de stocker le moment d'arrivée de l'opération dans un module. Ce moment est mis à jour à chaque fois qu'une opération passe par un module. Cette variable sert principalement pour le calcul du temps passé par l'opération dans celui-ci.
- *servArrivTime*, qui est un double permettant de stocker le moment d'arrivée d'une opération dans le serveur d'une file d'attente. Elle sert au calcul du temps passé dans celui-ci.
- *totalServTimeInModule*, qui est un double permettant de stocker le temps total de service de l'opération dans un serveur. Elle est utilisée pour la file WiFi et la file transmission car, pour ces deux files, l'opération peut effectuer plusieurs passages par le ou les serveurs de la file. Cette variable sert donc d'accumulateur de temps pour chacun des passages.
- *energyConssumedInSystem*, qui est un double permettant de stocker l'énergie consommée par l'opération à travers le réseau. Cette variable est un accumulateur qui est mis à jour à chaque fois qu'une opération est traitée au sein d'un serveur consommant de l'énergie.
- *steadyState*, qui est un booléen permettant de savoir si l'opération est entrée dans le système alors que celui-ci est à l'état stationnaire.

Nous ne détaillerons pas, dans ce cas-ci, les méthodes de la classe car elles servent toutes essentiellement d'accesseur. Seules les méthodes commençant par *update* servent à mettre à jour les accumulateurs, et, celles commençant par *reset* à les remettre à zéro. Voyons à présent le détail des classes de notre hiérarchie.

Au plus haut niveau de celle-ci se trouve la classe *Module*, une classe abstraite qui contient les propriétés et comportements communs à l'ensemble des classes de la hiérarchie. Cette classe représente, dans le modèle mathématique, ce que nous appelons en toute généralité : *une entité*. Héritant directement de *Module*, nous

avons les classes suivantes :

- *Dispatcher*, qui est une implémentation de l'entité *dispatcher* du modèle mathématique. Les instances de cette classe assureront par exemple le rôle des dispatchers α_1 et α_2.
- *SimpleQueue*, qui est l'implémentation de la simple file d'attente du modèle mathématique. Les instances de cette classe assureront par exemple le rôle de la file locale ou, pour la stratégie interrompue, le rôle de la file cellulaire.
- *Delayer*, qui est une implémentation d'une file d'attente dont le nombre de serveurs est théoriquement infini. Une telle file prend en charge immédiatement toute opération qui lui est transmise. Les instances de cette classe assureront le rôle de la file cloud.

Au niveau suivant de la hiérarchie se trouve la classe *AlternatingStateQueue*, une classe abstraite qui hérite de la classe *SimpleQueue*. Par conséquent, c'est l'implémentation d'une simple file d'attente à laquelle est ajoutée la possibilité d'osciller entre deux états. Le comportement de cette simple file sera différent selon l'état dans lequel elle se trouve. Cette classe est abstraite car elle implémente le changement d'état mais délègue le comportement propre à adopter dans chacun d'eux aux classes qui en héritent. Ces classes héritières apparaissent au dernier niveau de la hiérarchie :

- *AlternatingServerQueue*, qui est une simple file d'attente dont le serveur a des propriétés différentes selon l'état dans lequel se trouve la file. Les instances de cette classe assureront le rôle de la file de transmission de la stratégie ininterrompue. En effet, pour rappel, les propriétés du serveur de cette file sont différentes selon l'état (*WIFI-ON* ou *WIFI-OFF*) de la file (voir section 2.1.2).
- *OnOffServeurQueue*, qui est une simple file d'attente dont le serveur oscille entre des périodes d'activité et d'inactivité. Les instances de cette classe assureront le rôle de la file WiFi de la stratégie interrompue.

Détaillons à présent ces classes en commençant par la classe *Module* dont le diagramme est visible à la figure 3.4.

La classe *Module*

Comme nous l'avons vu, nous avons choisi de faire de cette classe une classe abstraite car un module n'assume aucune responsabilité fonctionnelle au sein du réseau de files d'attente. Il est donc inutile de pouvoir instancier directement la classe *Module*. Par contre, cette classe contient les propriétés et comportements non-fonctionnels communs aux autres classes. Par exemple, pour construire un réseau de modules qui se spécialisera par la suite afin de produire un réseau de files d'attente, il est nécessaire de relier entre-eux chaque module, et ce, indépendamment des responsabilités propres à chacun en termes de prise en charge des

opérations.

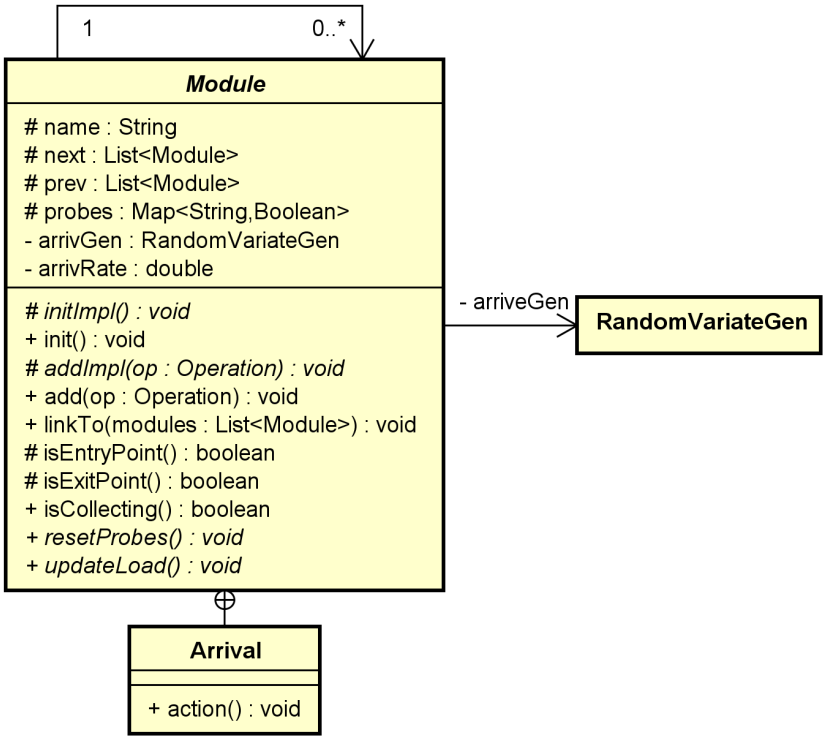


FIGURE 3.4 – Le diagramme de la classe *Module* implémentant les comportements communs des entités du modèle mathématique

La classe *Module* contient les variables suivantes :

- *name*, un nom servant à identifier le module au sein du réseau de files d'attente.
- *next* et *prev*, des références vers des listes de *Module*. La liste référencée par *next* contient les modules liés à la sortie d'un module et *prev* contient la liste des modules liés à l'entrée d'un module. Un module qui n'a pas de module à sa suite est un *point de sortie* du réseau de files d'attente. Un module qui n'a pas de module le précédant est un *point d'entrée* du réseau de files d'attente.
- *probes*, une référence vers un objet de type *Map* permettant d'associer à chaque *sonde* liée au module un booléen permettant de connaitre son état d'activité. Dans notre implémentation, une sonde est un objet particulier permettant, sur un module, la récolte de données à des fins statistiques. Nous décrirons en détail ces sondes et leurs particularités au fur et à mesure que nous les rencontrerons.
- *arrivGen* et *arrivRate* sont des variables qui ne sont utilisées que par les

modules qui sont des points d'entrée du réseau de files d'attente. La variable *arriveGen* référence un objet du type *RandomVariateGen* utilisé pour la génération des variables aléatoires lors de la simulation du processus de renouvellement. Nous verrons en détail à la section 3.1.2 l'implémentation de ce processus. La variable *arrivRate* est utilisée comme paramètre appliqué à la fonction de répartition utilisée pour la génération des variables aléatoires.

La classe *Module* contient les méthodes :

- *init()* qui permet d'initialiser le module. Durant cette initialisation, si le module en question est un point d'entrée, le tout premier évènement d'arrivée d'une opération est généré et transmis au simulateur afin d'enclencher le processus de renouvellement des opérations pour ce point d'entrée. Nous verrons en détail le mécanisme permettant la simulation du processus de renouvellement à la section 3.1.2. La méthode *init()* se termine par un appel à *initImpl()*, une méthode abstraite qui doit être implémentée par les classes héritant de *Module* afin d'y spécifier les instructions propres à leur besoins d'initialisation. L'appel de cette méthode sur chaque module du réseau de files d'attente, entre chaque simulation, est nécessaire afin de les réinitialiser.
- *add(op : Operation)*, qui permet l'ajout d'une opération à un module. Elle se termine par un appel à *addImpl(op : Operation)*, une méthode abstraite qui doit être implémentée par les classe héritant de *Module* afin de la spécifier leurs besoins propres. Cette méthode prend logiquement en paramètre l'opération qui doit être ajoutée au module. Nous reviendrons en détail sur la transmission des opérations d'un module à l'autre au moyen de cette méthode lorsque nous détaillerons les classes héritant de *Module*. C'est également lors du premier appel à cette méthode que sont initialisées les sondes du module. En effet, il n'est pas nécessaire de récolter des données sur un module par lequel ne serait jamais passée une opération.
- *linkTo(modules : List<Module>)*, qui permet de réaliser le lien entre un module et les modules suivants auxquels il est lié. Cette méthode permet la création du réseau de module en liant les modules les uns aux autres. Elle permet donc de garnir les listes référencées par les variables *prev* et *next* de chaque module.
- *isEntryPoint()*, qui permet de savoir si le module est un point d'entrée.
- *isExitPoint()*, qui permet de savoir su le module est un point de sortie.
- *isCollecting()*, qui permet de savoir si la collecte de données sur le module est active. Afin de s'assurer que ce soit bien le cas, cette méthode vérifie s'il existe sur la *Map* référencée par la variable *probes* des sondes toujours actives.
- *resetProbes()* et *updateLoad()* sont des méthodes abstraites. Elles doivent être implémentées par les classes héritant de *Module*. La méthode *resetProbes()*

sert à réinitialiser les sondes attachées au module. La méthode *updateLoad()*, quant à elle, est appelée lorsqu'il est nécessaire de collecter les données relatives au nombre d'opérations présentes dans le module.

La classe *Module* contient également la classe *Arrival*, une classe privée qui étend la classe *Event*. Cette classe représente, dans notre implémentation, l'évènement qui déclenche l'arrivée d'une opération dans le réseau de files d'attente. Autrement dit, l'arrivée d'une opération au point d'entrée du réseau. Elle n'est par conséquent utilisée que par un module faisant office de point d'entrée. L'ajout d'une opération au point d'entrée est réalisé en exécutant la méthode *action()* de cet évènement. Nous ne détaillons pas cette méthode dans l'immédiat car ce sera fait dans la section 3.1.2 puisque celle-ci fait partie de l'implémentation du processus de renouvellement. Cette classe est incluse dans la classe *Module* car il est nécessaire qu'elle puisse accéder aux variables *arrivGen* et *arrivRate*. De plus, *Arrival* ne fournit des services qu'à la classe *Module*. Il n'est donc pas nécessaire d'en faire une classe à part entière.

La classe *Disptacher*

Cette classe, héritée directement de *Module*, implémente le comportement des dispatchers du modèle mathématique décrit à la section 2.2.2. Les dispatchers, instances de la classe *Dispatcher*, reçoivent, en provenance des modules les précédents directement, des opérations qu'ils transmettent à l'un des deux modules auxquels ils sont liés. Le choix opéré quant à la transmission de l'opération est, comme nous l'avons vu, un choix aléatoire qui sera réalisé ici par un appel à la méthode *chooseModule()* dont le fonctionnement sera présenté ci-dessous. Le diagramme de cette classe est visible à la figure 3.5.

La classe *Dispatcher* contient les variables suivantes :

- *uniformRandGen*, qui est une référence vers un objet du type *MRG32K3*. Cet objet a pour fonction la génération uniforme de nombres aléatoires compris entre 0 et 1.
- *pi*, qui est un *double* représentant le paramètre utilisé dans l'algorithme réalisant le choix aléatoire du module vers lequel transmettre une opération.

La classe *Dispatcher* contient les méthodes suivantes :

- *initImpl()*, qui permet de réinitialiser le dispatcher. Ici, on se contente de réinitialiser le générateur de nombres aléatoires.
- *addImpl(Operation op)*, qui permet d'ajouter une opération à un dispatcher. Directement après son ajout, le choix du module auquel transmettre l'opération est réalisé en appelant *chooseModule()*. Enfin, l'opération est transmise au module choisi en appelant sur celui-ci sa méthode *add(Operation op)*.
- *chooseModule()*, qui permet, à l'arrivée d'une opération dans un dispatcher, de choisir vers quel module l'orienter. Dans le modèle mathématique, nous

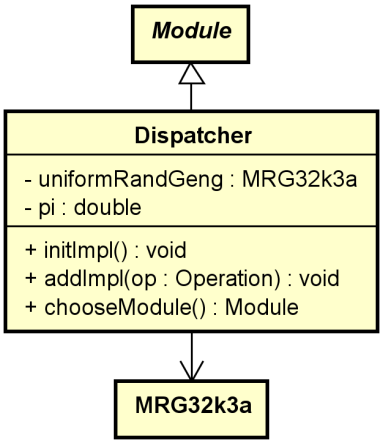


FIGURE 3.5 – Diagramme de la classe *Dispatcher* implémentant le comportement des dispatchers du modèle mathématique

avons spécifié ce choix comme étant une probabilité π que ce soit l'entité située au nord du dispatcher qui soit choisie et une probabilité $1 - \pi$ que ce soit l'entité située au sud qui soit choisie. Dans notre implémentation, nous réalisons ce choix en récupérant un nombre aléatoire compris entre 0 et 1 et en le comparant au paramètre *pi*. Si le nombre aléatoire généré se situe dans l'intervalle $]0, pi]$, on oriente l'opération vers le module situé à l'index 0 de *next*. Dans le cas contraire, on oriente l'opération vers le module situé à l'index 1 de *next*.

La classe *SimpleQueue*

Cette classe, héritée directement de *Module*, implémente le comportement des simples files d'attente du modèle mathématique. On retrouve, par exemple, parmi celles-ci, la file cellulaire de la stratégie interrompue. Cette classe contient la classe *SimpleQueue.Server* représentant le serveur de la file au sein duquel on simule l'exécution des opérations. Dans la suite, nous utilisons les termes *serveur* ou *serveur de la file* indifféremment lorsque nous parlons d'une instance de la classe *SimpleQueue.Server*.

Pour simuler l'exécution d'une opération, le serveur de la file récupère un nombre représentant le temps d'exécution d'une opération. Ce nombre est donné par une variable aléatoire générée par l'instance d'un *RandomVariateGen*. Cet objet est un générateur de variables aléatoires dont l'implémentation vient de la bibliothèque SSJ. La notion de temps dans nos simulations est représentée par une ligne du temps gérée par un simulateur dont nous détaillons le principe de fonc-

tionnement à la section 3.2. Pour simuler le temps passé par une opération dans un serveur, celui-ci transmet au simulateur un évènement qui sera déclenché au moment $T_D = T_A + T_E$, où T_A est le moment où l'opération entre dans le serveur et T_E le nombre aléatoire provenant de notre générateur. Dans ce cas, T_D représente le temps passé par l'opération dans le serveur. Lorsque ce temps est écoulé, l'évènement qui correspond à la sortie de l'opération du serveur est déclenché et l'opération est transmise au module suivant. Durant le temps que l'opération reste cantonnée dans le serveur, les opérations qui arrivent dans une *SimpleQueue* s'y s'accumulent.

Détaillons à présent le diagramme de la classe *SimpleQueue* et celui de la classe *SimpleQueue.Server* visible à la figure 3.6.

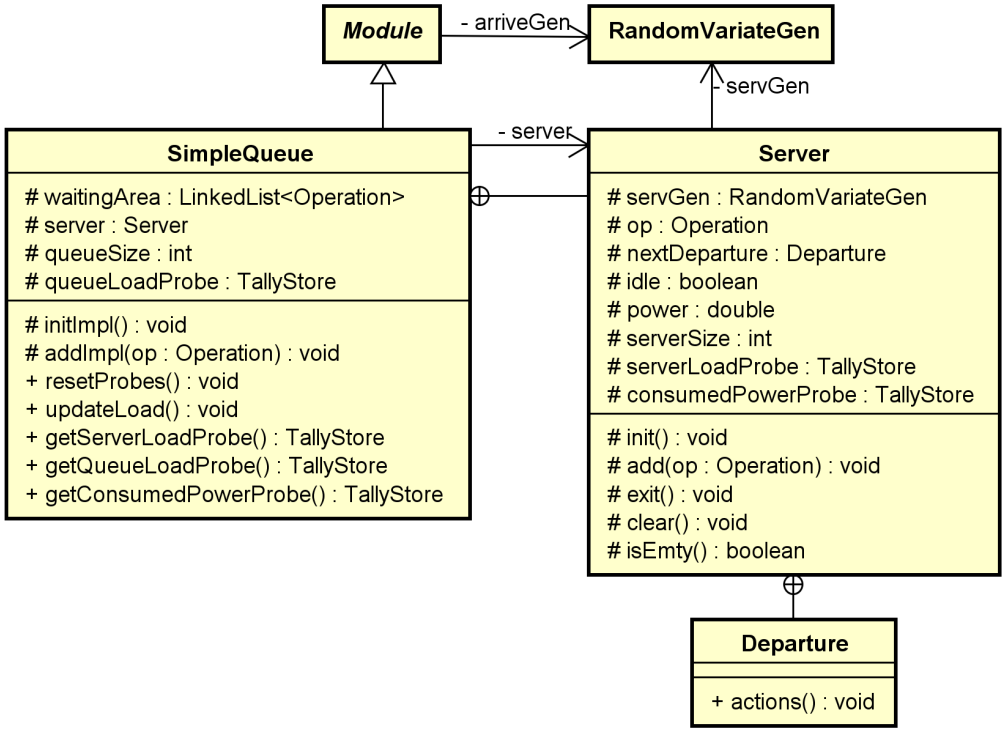


FIGURE 3.6 – Diagramme de la classe *SimpleQueue* implémentant le comportement des simple files d'attente du modèle mathématique

La classe *SimpleQueue* contient les variables suivantes :

– *waitingArea*, qui est une référence vers une liste chainée d'opérations. Elle représente le buffer de la file, c'est-à-dire l'endroit où les opérations s'accumulent, suivant leur ordre d'arrivée, lorsque le serveur de la file d'attente est occupé.

- *server*, qui est une référence vers un objet de type *SimpleQueue.Server* qui représente le serveur de la file d'attente.
- *queueSize*, qui est un entier représentant le nombre d'opérations dans la file d'attente, c'est-à-dire le nombre d'opérations accumulées dans la file en plus de l'opération dans le serveur.
- *queueLoadProbe*, qui est une référence vers un objet du type *TallyStore*. Cet objet permet le stockage de données récoltées sur cette file d'attente durant la simulation. En l'occurrence, les données stockées dans le *TallyStore* référencé par *queueLoadProbe* sont des entiers représentant le nombre d'opérations dans la file d'attente au moment de la récolte des données. Cette récolte a lieu à des intervalles de temps réguliers. Dès lors, à l'issue d'une simulation, il est possible d'effectuer différents calculs statistiques sur cette série de données. Par exemple, on peut extraire la moyenne du nombre d'opérations dans la file d'attente pour cette simulation et ensuite, calculer la variance ou encore l'écart-type de cette série de données par rapport à cette moyenne. Ces données seront utilisées pour mesurer les performances du système simulé. Notons que la classe *TallyStore* provient de la bibliothèque SSJ.

La classe *SimpleQueue* contient les méthodes suivantes :

- *initImpl()*, qui permet d'initialiser la file. Pour ce faire, la file d'attente est vidée d'éventuelles opérations subsistant d'une simulation précédente. Le serveur et les sondes liés à cette file d'attente sont ensuite initialisés.
- *addImpl(Operation op)*, qui permet d'ajouter une opération à la file d'attente. On commence par vérifier si le serveur est occupé. Dans ce cas, l'opération est ajoutée dans la liste chainée à la suite des autres. Dans le cas contraire, elle est placée directement dans le serveur en appelant sur la variable *server* la méthode *add(Operation op)*.
- *resetProbes()*, qui permet de réinitialiser les sondes de cette file d'attente.
- *updateProbes()*, qui permet d'enclencher la récolte ponctuelle de données sur la file d'attente, comme nous l'avons évoqué ci-dessus lors de la description de la variable *queueLoadProbe*. Lorsque cette méthode est appelée, toutes les sondes visées procèdent à une récolte.
- *getServerLoadProbe()*, *getQueueLoadProbe()* et *getConsumedPowerProbe()*, qui permettent d'obtenir les séries de données récoltées par les différentes sondes attachées à cette file d'attente. En l'occurrence, elles renvoient toutes le *TallyStore* qui leur est associé.

La classe *SimpleQueue.Server*

Cette classe privée est incluse dans la classe *SimpleQueue*. De ce fait, elle n'offre des services qu'à cette dernière et qu'aux classes qui en héritent. La classe *SimpleQueue.Server* contient également une classe de type *SimpleQueue.Server.Departure*

qui hérite de la classe *Event*. Cette classe *SimpleQueue.Server.Departure* représente les évènements de départ des opérations.

La classe *SimpleQueue.Server* contient les variables suivantes :

- *servGen*, qui est une référence vers l'objet de type *RandomVariateGen* servant à générer les variables aléatoires utilisées pour simuler le temps d'exécution des opérations dans le serveur.
- *op*, qui est une référence vers l'objet *Operation* se trouvant dans le serveur.
- *nextDeparture*, qui est une référence vers un objet *SimpleQueue.Server.Departure* qui représente l'évènement lié au départ de l'opération (*op*) du serveur.
- *idle*, qui est un booléen. Si sa valeur est *true*, le serveur est considéré comme étant inactif. Si sa valeur est *false*, le serveur est considéré comme étant actif. Pour les instances d'une *SimpleQueue*, la valeur de *idle* est toujours *false*. C'est uniquement pour les instances d'une *OnOffQueue* que la valeur d'*idle* pourra être passée à *true*.
- *power*, qui est un double représentant la puissance nominale du serveur de la file d'attente.
- *serverSize*, qui est un entier représentant la taille du serveur, c'est-a-dire, 0 si le serveur est vide, et 1, s'il est occupé.
- *serverLoadProbe* et *consumedPowerProbe*, qui sont des références vers des objets du type *TallyStore*. Ils représentent respectivement les sondes permettant la récolte de données sur la taille (charge) du serveur et sur la puissance dissipée par le serveur.

La classe *SimpleQueue.Server* contient les méthodes suivantes :

- *init()*, qui permet d'initialiser le serveur de la file d'attente. Pour ce faire, sa variable *idle* est mise à *false* car le serveur d'une file d'attente est toujours considéré comme actif au démarrage d'une simulation. La variable *serverSize* est mise à 0, *op* et *nextDeparture* à *null* et *servGen*, le générateur de variables aléatoires est initialisé.
- *add(Operation op)*, qui permet d'ajouter l'opération *op* au serveur et de lui affecter un temps d'exécution. La sortie de l'opération est ensuite planifiée en générant un évènement de départ transmis au simulateur. Le temps d'exécution de l'opération est récupéré auprès du générateur de variables aléatoires, référencé par *servGen* grâce à sa méthode *nextDouble()*. L'évènement de départ est transmis, quant à lui, au simulateur par un appel à *schedule(double delay)* sur *nextDeparture*. Ici, *delay* est la valeur récupérée depuis le générateur de variables aléatoires. Nous verrons plus précisément le principe de planification des évènements lorsque nous aborderons l'implémentation du simulateur.
- *exit()*, qui permet de sortir l'opération *op* se trouvant actuellement dans le serveur. Cette méthode est appelée au moment où l'évènement de départ de

cette opération est déclenché. Tout d'abord, à l'appel de cette méthode, la taille du serveur et de la file d'attente est ajustée. Ensuite, l'énergie consommée jusqu'ici par l'opération est mise à jour. Enfin, on ajoute l'opération sortante au module suivant et on récupère l'opération suivante en attente dans la file (la liste chainée référencée par *waitingArea*) que l'on ajoute au serveur.
- *clear()* et *isEmpty()*, qui permettent respectivement de vider le serveur et de vérifier s'il est vide.

La classe *SimpleQueue.Server.Departure*

Cette classe représente l'évènement de départ d'une opération du serveur de la file d'attente. Elle est incluse dans la classe *SimpleQueue.Server* étant donné qu'elle n'offre ses services qu'à celle-ci et qu'il est nécessaire qu'elle puisse accéder facilement à sa méthode *exit()*. Le classe *SimpleQueue.Server.Departure* hérite d'*Event* que nous détaillerons lorsque nous aborderons le principe de fonctionnement du simulateur. Elle n'implémente qu'une seule méthode, *action()*, qui est automatiquement exécutée par le simulateur lorsque l'évènement est déclenché. Cette méthode effectue un appel à la méthode *exit()* du serveur.

La classe *AlternatingStateQueue*

Cette classe, à l'instar de la classe *Module*, est une classe abstraite. Elle contient les propriétés et définit les comportements communs aux files pouvant osciller entre deux états. En l'occurrence, elle permet de définir la manière dont l'alternance entre ces deux états est réalisée. Par contre, elle délègue aux classes qui en héritent les comportements particuliers à adopter lors du changement d'état. Cette classe hérite directement de la classe *SimpleQueue* et contient la classe *AlternatingStateQueue.ChangeState*, qui hérite d'*Event* et représente l'évènement déclenchant le changement d'état de la file d'attente.

Détaillons à présent la classe *AlternatingStateQueue* dont le diagramme est visible à la figure 3.7.

La classe *AlternatingStateQueue* contient les variables suivantes :
- *state1*, qui est un booléen permettant de savoir dans quel état se trouve la file d'attente. Si *state1* est *true*, la file d'attente est dans l'état 1, sinon, elle est dans l'état 2.
- *state1ToState2Gen* et *state2ToState1Gen*, qui sont des références vers des générateurs de variables aléatoires de type *RandomVariateGen*. La période de temps que la file passe dans un état et dans l'autre est donné par les variables aléatoires générées par ces deux générateurs. Le générateur référencé par la variable *state1ToState2Gen* génère les variables aléatoires utilisées pour

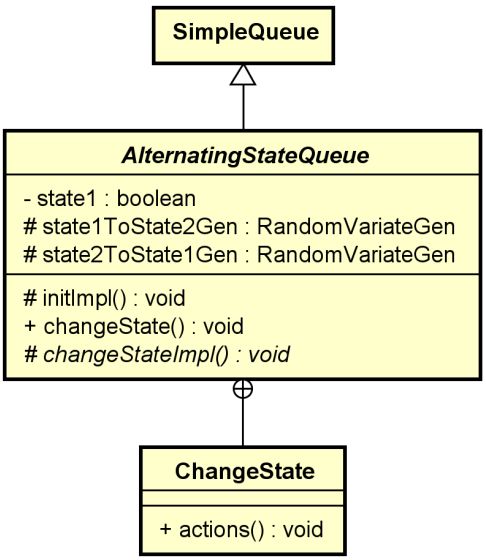


FIGURE 3.7 – Diagramme de la classe *AlternatingStateQueue* implémentant les comportements des files d'attente pouvant alterner entre deux états.

définir le temps passé dans l'état 1 et *state2ToState1Gen* génère les variables aléatoires utilisées pour définir le temps passé dans l'état 2. Bien sûr, de l'état 1, on ne peut passer qu'à l'état 2 et vice versa.

La classe *AlternatingStateQueue* contient les méthodes suivantes :

- *initImpl()*, qui permet d'initialiser la file d'attente. À cette fin, tout d'abord, *state1* est mise à *true* car nous avons choisi qu'une telle file d'attente démarre toujours dans son état 1. Ensuite, les deux générateurs de variables aléatoires sont initialisés. Enfin, le premier évènement de changement d'état de la file est transmis au simulateur. Pour ce faire, on appelle sur cet évènement la méthode *schedule(double delay)*. La variable *delay* représente ici le temps à l'issue duquel cet évènement doit se déclencher. En l'occurrence, il est obtenu en récupérant la valeur de la première variable aléatoire générée par la générateur de variables aléatoires référencé par *state1ToState2Gen*. Il représente donc bien la temps que la file d'attente passera dans l'état 1.
- *changeState()*, qui permet de réaliser le changement d'état de la file d'attente. Tout d'abord, cette méthode appelle *changeStateImpl* car, comme nous l'avons vu, les comportements particuliers à adopter dans un état sont délégués aux classes qui héritent de *AlternatingStateQueue*. Ensuite, une fois cette méthode exécutée, *changeState()* planifie le prochain changement d'état. Le processus est assez simple. Si *state1* est *true*, c'est-à-dire qu'avant l'appel à *changeState()* on était dans l'état 1, alors, au moment du change-

ment d'état, on passe à l'état 2. On peut donc en tout premier lieu affecter *false* à *state1* et ensuite générer le prochain évènement de changement d'état. Dans ce cas-ci, il s'agira donc du passage de l'état 2 à l'état 1. Si avant l'appel à cette méthode, on était dans l'état 2, c'est-à-dire avec *state1* à *false*, le principe aurait été similaire mais avec la planification du passage de l'état 1 à l'état 2.

- *changeStateImpl()*, qui est une méthode abstraite. Elle permet d'implémenter le comportement particulier à adopter par ces classes lors du changement d'état. Nous détaillerons cette méthodes lorsque nous aborderons ces classes.

La classe *AlternatingStateQueue.ChangeState*

Cette classe incluse, dans la classe *AlternatingStateQueue*, hérite d'*Event*. Les instances de cette classe sont les évènements transmis au simulateur afin de déclencher le changement d'état des files d'attentes du type *AlternatingStateQueue*. La méthode *action()* des instances de cette classe est appelée par le simulateur au moment où l'évènement se déclenche. L'action de cet évènement ne fait qu'appeler la méthode *changeState()* afin de procéder au changement d'état de la file d'attente.

La classe *OnOffServerQueue*

Cette classe hérite de la classe *AlternatingStateQueue*. Elle oscille donc entre deux états, ici représentés par l'oscillation entre l'état actif et l'état inactif du serveur de la file d'attente. Dans l'état 1, le serveur est actif et, dans l'état 2, le serveur est inactif. Hormis cette oscillation, les objets du type *OnOffServerQueue* se comportent comme les objets du type *SimpleQueue*, avec pour seule particularité que la variable *idle* de l'instance du *SimpleQueue.Server* peut être affectée à *true* et que, dans ce cas, les opérations arrivant dans la file s'accumulent tant que le serveur est inactif. Dans le modèle mathématique, cette classe représente la file WiFi de la stratégie interrompue. Le diagramme de cette classe est disponible à la figure 3.8.

Cette classe ne contient qu'une seule méthode additionnelle, la méthode *changeStateImpl*. Celle-ci est chargée, d'une part, de mettre à la première place de la file d'attente l'opération qui était dans le serveur si celui-ci devient inactif et, d'autre part, s'il devient actif, de lui ajouter l'opération en tête de file. Autrement dit, elle est chargée d'annuler l'exécution de l'opération en cours si le serveur devient inactif et de démarrer l'exécution de la première opération de la file si le serveur devient actif. Comme nous l'avons dit plus haut, le reste du comportement de la file est conforme à une simple file d'attente.

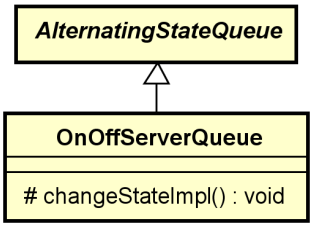


FIGURE 3.8 – Diagramme de la classe *OnOffServerQueue* implémentant le comportement d'une file d'attente dont le serveur oscille entre l'état actif et l'état inactif.

La classe *AlternatingServerQueue*

Cette classe hérite de la classe *AlternatingStateQueue*. Elle oscille donc entre deux états pour lesquels un serveur différent est utilisé, ceci afin de simuler le changement de propriété du serveur de la file transmission de la stratégie ininterrompue. En effet, dans notre implémentation, nous avons préféré utiliser deux instances de la classe *SimpleQueue.Server* plutôt qu'une seule pour laquelle il aurait fallut changer les propriétés à chaque changement d'état. Ici encore, à l'exception du changement d'état, le comportement de la file est exactement le même que celui d'une *SimpleQueue*. Seule la méthode *changeStateImpl()* nécessite donc d'être détaillée. Le diagramme de cette classe est proposé à la figure 3.9

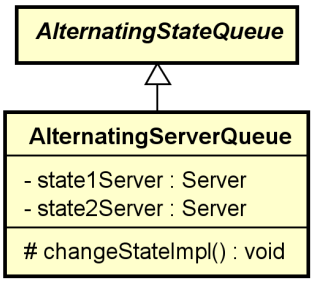


FIGURE 3.9 – Diagramme de la classe *AlternatingServerQueue* implémentant le comportement d'une file d'attente dont le serveur est différent selon l'état dans lequel se trouve la file.

Dans ce diagramme, on retrouve deux variables de classe référençant les serveurs propres à chaque état. Dans la pratique, la variable *server*, héritée de la classe *SimpleQueue*, référencera le serveur en cours d'utilisation.

La méthode *changeStateImpl()* vérifie, dans un premier temps, si le serveur est vide au moment du changement d'état. Si ce n'est pas le cas, elle annule l'exécution

de l'opération en cours et la remet en tête de la file d'attente. Ensuite, elle associe le serveur de l'état suivant à la variable *server* afin que celui-ci soit utilisé dans la suite. Enfin, elle ajoute l'opération qui se situe en tête de file au serveur utilisé.

La classe *Delayer*

Cette dernière classe de la hiérarchie, héritant de la classe *Module*, implémente le comportement d'une file d'attente dont le nombre de serveur est infini. Dans le modèle mathématique, c'est l'implémentation de la file cloud. Étant donné que le nombre de serveurs est infini et qu'il n'y a pas de consommation d'énergie pour ce type de file, cette classe ne contient pas de classe *Server*. A la place, toute opération qui entre dans cette file se voit attribuer immédiatement un évènement de sortie. A cette fin, la classe *Delayer* contient une classe *Delayer.Departure* chargée de la génération de ces évènements de sortie. Pour simuler le temps d'exécution d'une opération dans cette file, celle-ci récupère, comme pour les simples files d'attente, un nombre donné par une variable aléatoire générée par l'instance d'un *RandomVariateGen*.

Détaillons à présent le diagramme de cette classe visible à la figure 3.10. Nous commençons par la classe *Delayer* à proprement parler, nous décrirons ensuite la classe *Delayer.Departure* qu'elle contient.

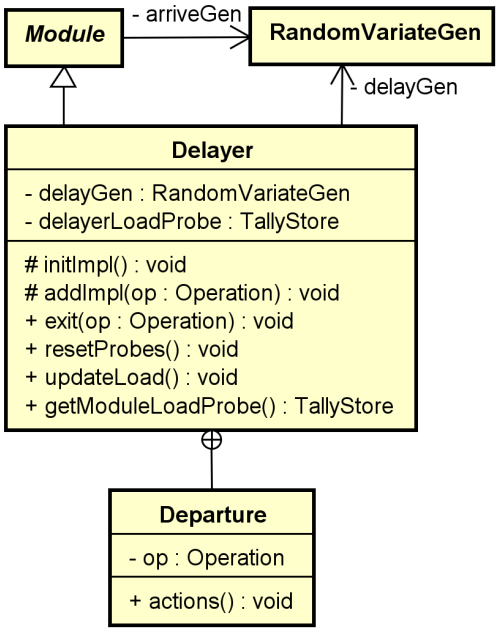


FIGURE 3.10 – Diagramme de la classe *Delayer* implémentant le comportement d'une file d'attente dont le nombre de serveur est infini

La classe *Delayer* contient les variables suivantes :

- *delayGen*, qui est une référence vers l'objet de type *RandomVariateGen* servant à générer les variables aléatoires utilisées pour simuler le temps d'exécution des opérations dans cette file.
- *delayLoadProbe*, qui est une référence vers un objet du type *TallyStore* permettant de stocker les données correspondant au nombre d'opérations dans la file. Ce nombre correspond en réalité au nombre d'opérations en cours d'exécution.

La classe *Delayer* contient les méthodes suivantes :

- *initImpl()*, qui permet d'initialiser la file. Pour ce faire, elle est tout d'abord vidée d'éventuelles opérations subsistant d'une simulation précédente. Ensuite, le générateur de variables aléatoires *delayGen* est initialisé. Enfin, la sonde *delayLoadProbe* est elle-même initialisée.
- *addImpl(Operation op)*, qui permet d'ajouter l'opération *op* à la file. On incrémente tout d'abord le nombre d'opérations dans la file puis on planifie la sortie de l'opération en générant un nouveau départ après un temps donné par une variable aléatoire générée par le générateur référencé par *delayGen*.
- *exit(Operation op)*, qui permet de sortir de la file l'opération *op*. A la sortie, on décrémente tout d'abord le nombre d'opérations dans la file puis on met à jour les éventuelles sondes récoltant des données sur les opérations passant par cette file.

La classe *Delayer.Departure*

Cette classe représente l'évènement de départ d'une opération d'un delayer. Elle est incluse dans la classe *Delayer* car elle n'offre des services qu'à celle-ci et qu'elle doit accéder facilement à sa méthode *exit(Operation op)*. Cette classe hérite d'*Event* et n'implémente que la méthode *action()* qui appelle la méthode *exit(Operation op)* du delayer afin de sortir l'opération qui a terminé son service dans cette file. Cette action est exécutée au moment où l'évènement de départ est déclenché.

Avant de passer au détail de l'implémentation du processus de renouvellement, nous décrivons une dernière classe. Celle-ci, bien que ne faisant pas partie directement de la hiérarchie représentant les entités du modèle mathématique, permet de gérer, pour l'ensemble d'un réseau de files d'attente, certains aspects propres à l'ensemble des modules du réseau et non à chacun d'eux individuellement.

La classe *QueueNetwork*

Cette classe représente en quelque sorte le conteneur à l'intérieur duquel un réseau de files est construit. Tout d'abord, cette classe est implémentée sous la

forme d'un singleton afin d'être accessible globalement. Elle permet notamment de conserver une référence vers chaque module du réseau afin, d'une part, de réinitialiser rapidement le réseau, et, d'autre part, d'exécuter des instructions qui nécessitent l'accès à tous les modules. Par exemple, c'est au sein de cette classe que nous déclenchons l'évènement qui réalise la mesure du nombre d'opérations sur chaque file. Cet évènement est implémenté sous la forme d'une classe incluse dans la classe *QueueNetwork*.

C'est également cette classe qui est responsable du démarrage de la simulation, de la détection du moment où le système atteint son état stationnaire et de la récolte des données d'énergie consommée et de temps passé par une opération dans l'ensemble du réseau.

Le diagramme de cette classe est visible sur la figure 3.11.

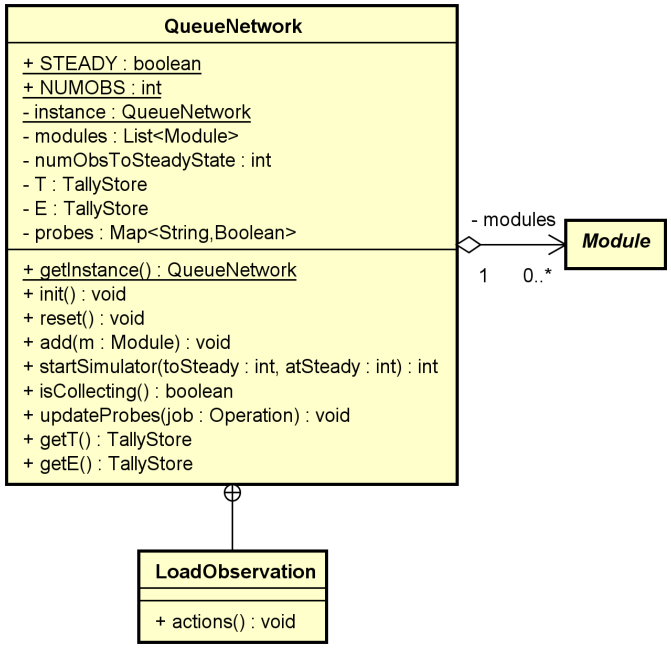


FIGURE 3.11 – Diagramme de la classe *QueueNetwork*, conteneur à l'intérieure duquel le réseau de file d'attente est créé et la simulation démarrée.

La classe *QueueNetwork* contient les variables suivantes :

- *STEADY*, qui est un booléen statique affecté à *true* si le système a atteint son état stationnaire et à *false* sinon.
- *NUMOBS*, qui est un entier statique représentant le nombre d'observations voulues sur les différents modules du réseau durant la simulation. Nous parlons ici des observations permettant d'obtenir les séries de données utilisées

pour calculer les mesures de performances de nos différentes files. Nous reviendrons sur cette notion à la section 3.3.5 lorsque nous verrons le protocole de simulation que nous avons mis en place.
- *instance*, qui est une référence vers une instance de *QueueNetwork* utilisée pour réaliser le singleton.
- *modules*, qui est une référence vers la liste des modules du réseau. Cette liste est garnie par chaque module au moment où il est créé. Pour ce faire, les modules s'enregistrent auprès du *QueueNetwork* au moment de leur instanciation.
- *numObsToSteadyState*, qui est un entier statique représentant le nombre d'observations nécessaires pour que le système atteigne son état stationnaire. Nous reviendrons également sur cette notion à la section 3.3.5.
- T et E, qui sont des références vers des objets du type *TallyStore*. Ils représentent respectivement les sondes permettant la récolte de données sur le temps passé par une opération dans le réseau et sur l'énergie consommée par une opération dans le réseau.
- *probes*, qui est une référence vers un objet de type *Map* permettant d'associer aux deux sondes T et E ci-dessus un booléen afin de connaitre leur état d'activité. Cette *Map* a la même fonction que celle vue au niveau de la classe *Module*.

La classe *QueueNetwork* contient les méthodes suivantes :

- *getInstance()*, qui permet l'accès à la variable *instance* et donc à l'instance unique de la classe *QueueNetwork*.
- *init()*, qui permet de mettre à *null* la variable *instance* afin de permettre la création d'une nouvelle instance de *QueueNetwork*.
- *reset()*, qui permet de réinitialiser le système, sans toute fois, dans ce cas, obliger la création d'une nouvelle instance de *QueueNetwork*.
- *add(Module m)*, qui permet aux modules du réseau de files à simuler de s'enregistrer auprès du *QueueNetwork*.
- *startSimulator(int toSteady, int atSteady)*, qui permet de démarrer le simulateur et d'affecter *toSteady* à la variable *numObsToSteadyState* afin de signifier au système le nombre d'observations avant d'arriver à l'état stationnaire, et, d'affecter *atSteady* à *NUMOBS* afin, cette fois, de lui signifier le nombre d'observations à réaliser à l'état stationnaire. Cette méthode initialise également le simulateur avant de le démarrer et appelle sur chaque module du réseau sa méthode *init()*.
- *isCollecting()*, qui permet de vérifier si l'ensemble des observations ont été réalisées sur les modules du réseau. Elle retourne *true* si c'est le cas et *false* sinon.
- *updateProbes(Operation op)*. Cette méthode est appelée par les modules qui

sont des points de sortie du système. Ceux-ci appellent cette méthode et passent en argument l'opération qui quitte celui-ci. Par cet appel, les sondes T et E sont mises à jour. De plus, à chaque sortie d'une opération, cette méthode appelle *isCollecting()* afin de vérifier si les différentes sondes des modules du réseau ont terminé leur collecte. Dans l'affirmative, la simulation est interrompue.
- *getT()* et *getE()*, qui permettent d'obtenir les séries de données récoltées par les sondes T et E. Elles renvoient le *TallyStore* qui leur est associé.

La classe *QueueNetwork.LoadObservation*

Cette classe représente l'évènement de déclenchement des mesures du nombre d'opérations sur les différentes files du réseau. Elle est incluse dans la classe *QueueNetwork* car elle doit accéder facilement à la liste des modules du réseau. Cette classe hérite d'*Event* et n'implémente que la méthode *action()* qui, d'une part, appelle sur chaque module du réseau la méthode *updateProbes()*, et, d'autre part, créer le prochain évènement de mesure sur les modules et le transmet pour planification au simulateur. Voyons à présent l'implémentation du processus de renouvellent.

3.1.2 Implémentation du processus de renouvellement

Comme nous l'avons vu, l'implémentation de ce processus est réalisé au niveau de la classe module dans la méthode *init* de celui-ci juste avant l'appel à sa méthode *initImpl()*. Seuls les modules étant des points d'entrée d'un réseau de files d'attente peuvent démarrer un tel processus. Concrètement, pour enclencher le processus, la variable *arrivGen* du module est instanciée afin de référencer un générateur de variables aléatoires. Ensuite, le premier renouvellement est produit en créant un nouvel évènement d'arrivée d'une opération transmis au simulateur. L'intervalle de temps auquel est planifié le déclenchement de cet évènement est lié au générateur référencé par la variable *arrivGen*. Les renouvellements successifs sont produits lors de l'appel à la méthode *action()* des évènements d'arrivée. De cette manière, il n'y a jamais qu'un et un seul évènement d'arrivée planifié sur la ligne du temps. Cela permet de limiter les ressources utilisées pour ce processus. Le diagramme de séquence de la figure 3.12 illustre ce principe.

Comme nous l'avons vu, lorsque le premier évènement d'arrivée est déclenché, sa méthode *action()* est exécutée. Une nouvelle opération est alors créée et ajoutée au point d'entrée du réseau qui est responsable du processus de renouvellement ayant généré cette arrivée. Ensuite, l'évènement d'arrivée de l'opération suivante est généré et planifié à un temps donné par une variable aléatoire fournie par le générateur référencé par *arrivGen*. Dans le cas du processus de Poisson que

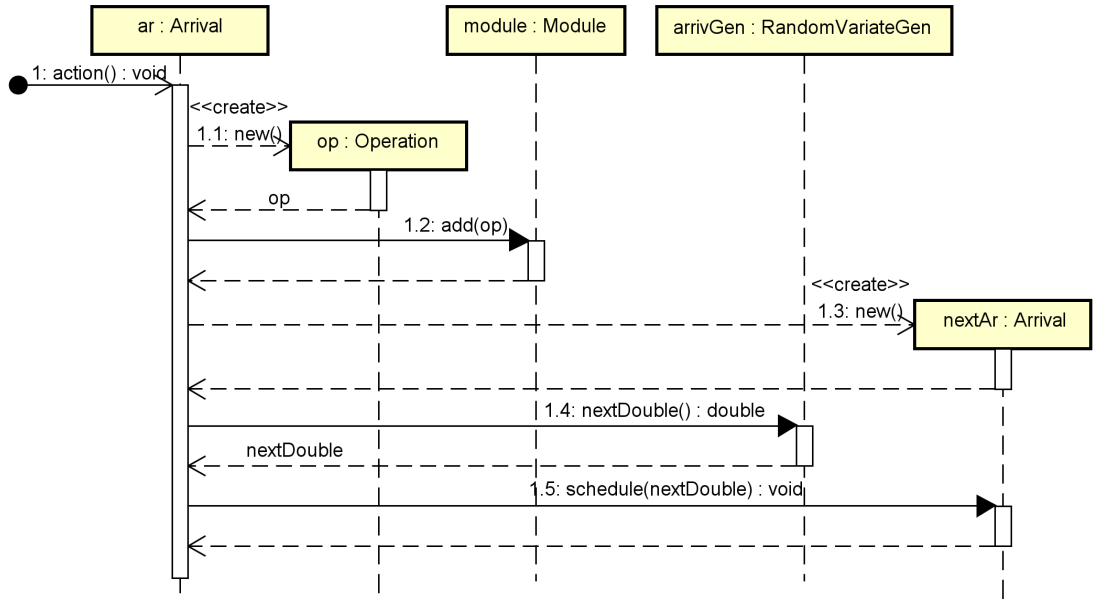


FIGURE 3.12 – Diagramme de séquence illustrant le principe du processus de renouvellement

nous utilisons, ce générateur est un *ExponentialGen*, c'est-à-dire un générateur de variables aléatoires distribuées comme des exponentielles. Par cette implémentation du processus de renouvellement, on génère des arrivées jusqu'à ce qu'une condition particulière vienne le stopper. Cette condition est paramétrée au sein du simulateur dont nous présentons à présent le principe de fonctionnement ainsi que l'implémentation.

3.2 Le simulateur

Afin de présenter le simulateur utilisé dans notre outil de simulation, nous commençons, dans la section 3.2.1, par décrire son principe de fonctionnement, puis dans la section 3.2.2, nous détaillons l'implémentation de celui-ci. Cette implémentation est issue de la bibliothèque SSJ.

3.2.1 Principe de fonctionnement

Le simulateur permet, comme nous l'avons évoqué ci-dessus, la gestion des évènements générés par les différents modules d'un réseau de files d'attente. Il a deux fonctions principales :

1. placer, en ordre utile sur la ligne du temps, les évènements qui lui sont transmis par les différents modules du réseau simulé,

2. déclencher ces évènements dans l'ordre chronologique.

Comme nous avons pu le voir précédemment, les phénomènes aléatoires d'un réseau sont simulés par ses différents modules. Par exemple, nous avons vu que la simulation du processus de Poisson est implémenté au niveau du point d'entrée du réseau. Le module qui prend en charge la simulation de ce processus génère des évènements d'arrivées d'opérations à des intervalles de temps donné, dans ce cas, par des variables aléatoires distribuées comme des exponentielles. Lorsque l'évènement est généré, il est transmis au simulateur qui a la charge de le placer au bon endroit sur la ligne du temps. Pour ce faire, le simulateur récupère l'intervalle de temps généré pour cet évènement et l'ajoute au moment présent afin d'obtenir le moment de déclenchement de l'évènement. Il place alors cet évènement sur la ligne du temps au moment qu'il a calculé.

Par exemple, au démarrage d'une simulation, le module faisant office de point d'entrée du réseau simulé génère, en tout premier lieu, l'évènement d'arrivée de la première opération. Admettons que l'intervalle de temps généré, pour l'arrivée de celle-ci, est 2ut. Comme le moment présent est $T_p = 0$, puisque la simulation vient de commencer, le simulateur place l'évènement de la première arrivée au moment $T_{A_1} = 0 + 2 = 2ut$. Le principe est le même à chaque évènement transmis au simulateur. Comme nous l'avons vu précédemment, dans l'implémentation de l'outil de simulation, nous avons quatre types d'évènements :

- *Arrival*, qui représente l'évènement d'arrivée d'une opération dans le réseau.
- *Departure*, qui représente l'évènement de sortie d'une opération du serveur.
- *ChangeState*, qui représente un changement d'état d'une file (pour la file transmission ou pour la file WiFi uniquement).
- *LoadObservation*, qui représente l'évènement déclenchant la prise de mesure du nombre d'opérations dans les différentes files du réseau.

Le simulateur est également responsable du déclenchement de ces évènements, ce qu'il réalise en exécutant la méthode *action()* de ceux-ci. Après avoir exécuté cette méthode, le simulateur supprime l'évènement de la ligne du temps et récupère l'évènement suivant afin d'exécuter à son tour l'action qu'il contient. Le simulateur procède ainsi tant qu'il y a des évènements à traiter sur la ligne du temps ou jusqu'à ce qu'une condition d'arrêt le stoppe. Afin d'illustrer le fonctionnement du simulateur, nous proposons à l'annexe A la simulation pas à pas d'une version simplifiée du réseau de files propre à l'exécution distante de la stratégie interrompue.

Nous détaillons à présent l'implémentation du simulateur de notre outil de simulation qui, pour rappel, provient de la bibliothèque SSJ.

3.2.2 Détail de l'implémentation du simulateur de la bibliothèque SSJ

Le diagramme de classe de la figure 3.13 représente l'implémentation du simulateur, on y retrouve les classes suivantes :

- La classe *Simulator*, qui représente le simulateur à proprement parler, et une interface *EventList*, qui représente la ligne du temps qui lui est liée. L'implémentation concrète de cette interface dépend de la structure de données souhaitée pour implémenter la ligne du temps. Nous y reviendrons sous peu.
- La classe *Event*, qui représente l'implémentation des évènements gérés par le simulateur. Cette classe est abstraite car il est nécessaire de l'étendre pour implémenter l'action exécutée au moment où l'évènement est déclenché. Ce choix d'implémentation est dû aux besoins propres des différents modules, ceux-ci ayant différents types d'action à exécuter en fonction de leurs spécificités. Comme nous l'avons vu, les quatre évènements de notre outil de simulation étendent tous cette classe.
- La classe *Sim*, qui permet de gérer une instance unique (singleton) de la classe *Simulator*.

Pour la description de ces classes, afin de ne pas encombrer la lecture de détails inutiles, nous avons choisi de ne représenter que les variables de classe et les méthodes nécessaires pour notre outil de simulation. Nous décrivons ces trois classes en commençant par la classe *Simulator*.

La classe *Simulator*

Comme nous venons de le voir, la pièce maitresse du simulateur est la classe *Simulator* dont les variables de classe sont les suivantes :

- *currentTime*, qui contient la valeur du temps présent que nous avons appelé T_p. Elle est mise à jour à chaque fois que le simulateur exécute un évènement.
- *eventList*, qui référence un objet du type *EventList*, qui représente la liste ordonnée chronologiquement selon l'ordre de déclenchement des évènements transmis au simulateur. Autrement dit, c'est l'objet qui représente la ligne du temps. *EventList* est une interface dont l'implémentation concrète dépend de la structure de données que l'on souhaite utiliser pour implémenter cette ligne. Selon le nombre d'évènements attendus et la probabilité que l'insertion d'un nouvel évènement perturbe plus ou moins l'ordre actuel, il est plus intéressant d'utiliser l'une ou l'autre structure. Chacune de celles-ci offre des avantages propres quant à la vitesse de ré-ordonnancement de la liste (Leemis et Park, 2006). Nous utilisons, pour nos simulations, comme structure de données une liste doublement chainée fournie par la classe *DoublyLinked*. Nous aurions pu tout aussi bien utiliser un arbre binaire fourni par

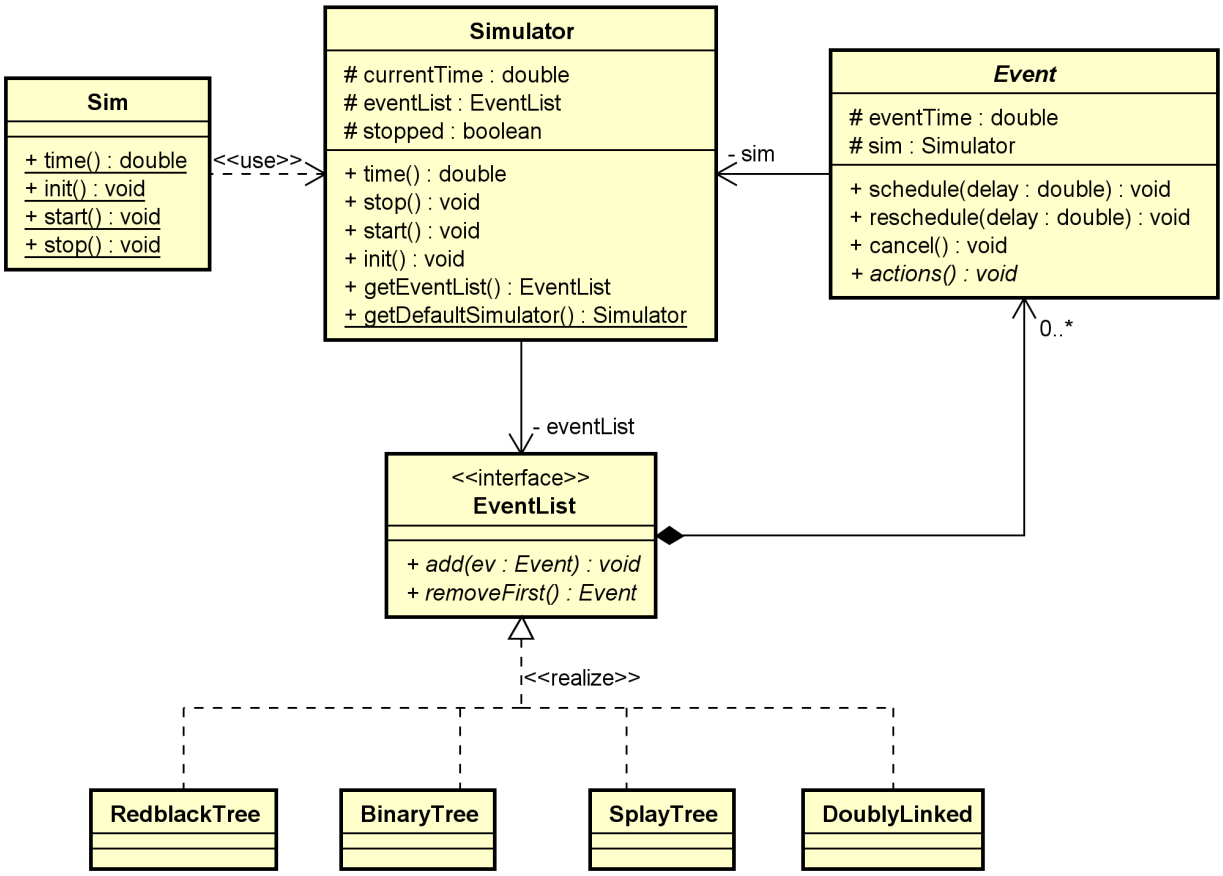


FIGURE 3.13 – Représentation du simulateur, de sa ligne du temps et des évènements qu'il gère

la classe *BinaryTree* car celui-ci offre les mêmes performances que la liste doublement chainée. Par contre, ces deux structures de données sont plus performantes que les deux autres disponibles, c'est-à-dire l'arbre de splay de la classe *SplayTree* et l'arbre rouge-noir de la classe *RedBlackTree*. Pour nous en assurer, nous avons réalisé plusieurs simulations avec chacune des structures disponibles [2].

– *stopped*, qui sert à représenter l'état actif ou inactif du simulateur. Cette variable est mise à *true* lorsqu'on souhaite interrompre la simulation.

Afin de représenter le comportement du simulateur, la classe *Simulator* est munie des méthodes suivantes :

2. Nous n'avons pas réalisé d'analyse statistique mais simplement observé les temps de simulation réels.

- *init()*, qui permet de réinitialiser le simulateur aux valeurs par défaut lorsqu'on souhaite démarrer une nouvelle simulation.
- *time()*, qui renvoie simplement la valeur du temps présent.
- *stop()*, qui permet d'affecter à la variable *stopped* la valeur requise pour que le simulateur s'arrête.
- *start()*, qui permet le bon déroulement d'une simulation en récupérant et en exécutant tour à tour les évènements comme nous l'avons vu ci-dessus. Celle-ci est le cœur du simulateur, c'est pourquoi nous la détaillons à présent en prenant appui sur le diagramme de séquence de la figure 3.14

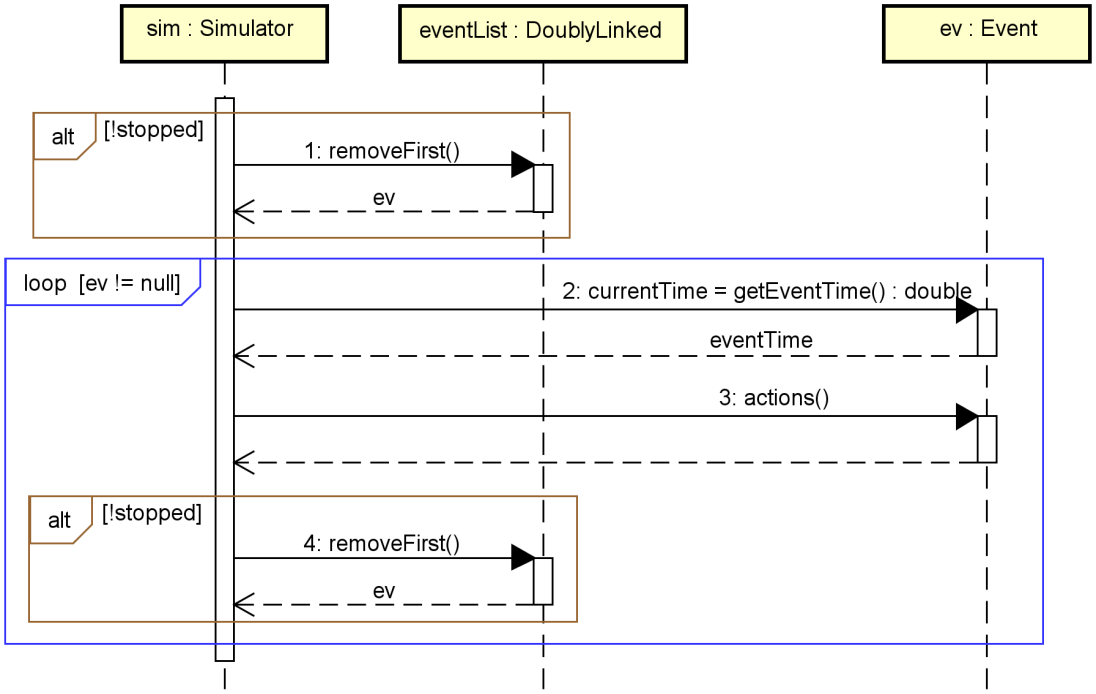


FIGURE 3.14 – Diagramme de séquence décrivant la récupération et l'exécution des évènements de la ligne du temps (rôle de la méthode *start()* de la classe *Simulator*)

Tout d'abord, nous constatons que le simulateur représenté ici par *sim*, une instance de la classe *Simulator*, vérifie si l'état de la simulation est actif (*stopped* ← faux). Si c'est le cas, il récupère le tout premier évènement (*ev*) sur la ligne du temps par un appel sur *eventList*, une instance de la classe *DoublyLinked*. Cet appel supprime *ev* de la ligne en question et le transmet à *sim*. Cette suppression entraine la mise à jour de la ligne du temps afin que le premier évènement sur celle-ci soit maintenant celui qui succède à *ev*. On entre ensuite dans une boucle qui itère tant qu'il existe un évènement à traiter. L'objet *sim* extrait d'*ev* son moment

d'exécution (*eventTime*) et met à jour le temps présent en affectant *eventTime* à *currentTime*. Ensuite, *sim* appelle sur *ev* l'action qui lui est liée. Enfin, *sim* réitère l'appel à *eventList* afin de récupérer l'évènement suivant jusqu'à ce qu'une condition d'arrêt stoppe la simulation (*stopped* ← vrai).

La classe *Event*

Cette classe, comme nous l'avons vu, est abstraite. Il faut donc l'étendre afin d'implémenter la méthode *actions()* selon les besoins du type d'évènement. Par exemple, un évènement d'arrivée et un évènement de départ auront des comportements différents et donc des actions différentes également.

La classe *Event* contient les variables suivantes :

- *eventTime*, qui représente le moment de déclenchement de l'évènement.
- *sim*, qui est une référence vers une instance de la classe *Simulator*.

La classe *Event* contient les méthodes suivantes :

- *schedule(double delay)*, qui permet de transmettre l'évènement au simulateur référencé par la variable *sim*. Le double *delay* représente l'intervalle de temps à utiliser pour la planification.
- *reschedule(double delay)*, qui permet les mêmes opérations que la méthode précédente mais cette fois pour un évènement qui était déjà planifié et pour lequel on souhaite modifier l'intervalle.
- *cancel()*, qui permet tout simplement de supprimer un évènement de la ligne du temps.
- *action()*, qui est une méthode abstraite dont l'implémentation est déléguée aux classes qui étendent *Event* afin de définir le type d'action à réaliser lors du déclenchement de l'évènement.

La classe *Sim*

La dernière classe à exposer est la classe *Sim* qui permet de travailler avec une instance unique de la classe *Simulator* en s'y référant par sa méthode *getDefaultSimulator()*. L'instance de la classe *Simulator*, dans ce cas, est un singleton créé au premier appel de cette méthode et réutilisé à chaque nouvel appel. Les fonctionnalités de cette classe sont plus limitées que celles de la classe *Simulator*. Par contre, l'utilisation d'un singleton de la classe *Simulator* facilite grandement la manipulation du simulateur. Étant donné que les méthodes disponibles dans la classe *Sim* sont suffisantes pour nos simulations, nous profitons des facilités qu'elle offre.

Nous avons vu en détail l'implémentation des entités du modèle mathématique ainsi que l'implémentation du simulateur qui forment les deux pièces maitresses de notre outil de simulation. Vérifions à présent cet outil afin de nous assurer que

son implémentation est conforme au spécifications du modèle mathématique qu'il représente.

3.3 Vérification de l'implémentation

La vérification de l'implémentation permet de s'assurer que cette dernière est conforme aux spécifications du modèle mathématique. Autrement dit, elle permet de s'assurer que les classes :

- *Dispatcher*
- *SimpleQueue*
- *Delayer*
- *AlternatingServerQueue*
- *OnOffServerQueue*

ainsi que le processus d'arrivée des opérations vus au chapitre précédent sont implémentés dans le respect des spécifications du modèle mathématique. Afin de réaliser cette vérification, il est tout d'abord nécessaire de définir différents scénarii de simulation. Chaque scénario permettant de vérifier une partie des spécifications. Concrètement, définir un scénario c'est choisir les modules à vérifier et paramétrer les générateurs de variables aléatoires de ceux-ci. Par exemple, comme nous l'avons vu à la section précédente, le module qui représente le dispatcher α_1 du modèle mathématique est une instance de la classe *Dispatcher*. Celui-ci contient une variable pi utilisée pour la simulation du choix aléatoire permettant d'orienter une opération vers l'un ou l'autre des deux modules qui lui sont liés. Pour définir le comportement du dispatcher, il faut affecter au paramètre pi une valeur comprise entre 0 et 1. Le choix posé influera sur le nombre moyen d'opérations transmises vers l'un ou l'autre module. On peut définir, pour ce faire, trois scénarii : deux correspondants à des valeurs extrêmes de pi et un avec une valeur intermédiaire. Pour les files d'attente simples et le processus de Poisson, il est suffisant de tester une seule affectation de paramètre pour les différents générateurs de variables aléatoires. En effet, si le comportement d'une file simple est correct pour une fréquence particulière de traitement des opérations, elle le sera aussi pour toute autre fréquence tant que la file reste stable. Pour les files d'attente pouvant alterner entre deux états, il est nécessaire de vérifier également que cette alternance soit correctement implémentée.

Une fois les scénarii définis, il faut définir les mesures de performance que l'on souhaite obtenir sur les différentes files. Nous pouvons ensuite procéder aux simulations de chaque scénario. Durant celles-ci, nous récoltons des séries de données sur chaque file impliquée dans ce scénario. Nous traitons ensuite statistiquement ces séries de données afin d'en extraire les mesures de performance définies préalablement. Dans ce mémoire, toutes ces mesures de performance sont des *moyennes*, ap-

pelées *moyennes empiriques* lorsqu'elles sont obtenues par simulation et *moyennes théoriques* ou encore *espérances* lorsqu'elles sont calculées théoriquement. Sur les moyennes empiriques, nous calculons un intervalle de confiance, c'est-à-dire un intervalle à l'intérieur duquel nous avons un certain degré de certitude que se situe la moyenne théorique. Nous travaillons avec un intervalle de confiance de 95%. Lorsque nous avons récolté toutes les moyennes empiriques souhaitées ainsi que leur intervalle de confiance, et ce pour chaque module du réseau, nous calculons les moyennes théoriques correspondantes. Pour calculer ces moyennes, nous avons à notre disposition l'étude théorique du modèle mathématique réalisée dans notre article de référence (Wu *et al.*, 2015). Il nous « suffit » d'utiliser les formules proposées par l'auteur de l'article pour calculer ces moyennes. Nous vérifions ensuite, pour chaque mesure de performance, si les espérances calculées se situent dans l'intervalle de confiance de la moyenne empirique. Dans l'affirmative, pour tous les scénarii définis, nous considérons que l'implémentation de l'outil de simulation est conforme aux spécifications du modèle mathématique.

Afin de détailler le processus de vérification, nous commençons, à la section 3.3.1, par définir les différents scénarii de simulation. Ensuite, nous définissons, à la section 3.3.2, les mesures de performance que nous souhaitons obtenir. Dans la section 3.3.3, nous détaillons le procédé mis en place lors de nos simulations afin de récolter les données nécessaires au calcul des moyennes empiriques et à la section 3.3.4, nous détaillons les formules utilisées afin de calculer les moyennes théoriques correspondantes. À la section 3.3.5, nous détaillons notre protocole de simulation. Nous comparons enfin, à la section 3.3.6, les résultats théoriques aux résultats empiriques afin de vérifier, à proprement parler, l'implémentation des modules de l'outil de simulation.

3.3.1 Définition des scénarii de simulation

Comme nous venons de le voir, définir un scénario de simulation revient à paramétrer les modules impliqués dans celui-ci, c'est-à-dire à affecter des valeurs aux différents générateurs de variables aléatoires de ces modules. Le choix des valeurs affectées est posé pour que les modules se comportent de manière similaire aux composants physiques qu'ils représentent. Afin de vérifier l'ensemble des modules implémentés, nous procédons en deux temps. Dans un premier temps, nous construisons le réseau de files de la stratégie interrompue et nous définissons quatre scénarii de simulation pour celui-ci. Dans un second temps, nous vérifions le module représentant la file transmission indépendamment du réseau de files de la stratégie ininterrompue.

Nous commençons par le détail des quatre scénarii de simulation de la stratégie

interrompue. Dans le tableau 3.1 ci-dessous, se trouvent les différents paramètres [3] et les valeurs qui leurs sont affectées.

TABLE 3.1 – Paramètres de simulation des scénarii de la stratégie interrompue

	PP	Dispatcher		File locale		File cellulaire		File WiFi				File cloud
	λ	π_{α_1}	π_{α_2}	μ_m	p_m	μ_g	p_g	μ_w	p_w	η	ξ	μ_c
Scénario 1	0.6	0	0.5	2	2	0.8	2.5	2	0.7	1	1	5
Scénario 2	0.6	1	0.5	2	2	0.8	2.5	2	0.7	1	1	5
Scénario 3	0.6	0.5	0.5	2	2	0.8	2.5	2	0.7	1	1	5
Scénario 4	0.6	0.5	0.5	2	2	0.8	2.5	2	0.7	2	5	5

Dans les trois premiers scénarii, on fait uniquement varier π_{α_1}. Le premier scénario, où $\pi_{\alpha_1} = 0$, permet de vérifier le comportement des modules dans le cas où toutes les opérations sont traitées par la file locale. Le deuxième scénario, où $\pi_{\alpha_1} = 1$, permet de vérifier le comportement des modules dans le cas où toutes les opérations sont transmises à la file cloud. Le troisième scénario, où $\pi_{\alpha_1} = 0.5$, permet de vérifier le comportement des modules dans le cas où, en moyenne, la moitié des opérations est traitée par la file locale et l'autre moitié transmise à la file cloud. Pour choisir les valeurs affectées aux générateurs de variables aléatoires des files du réseau, nous nous référons aux spécificités des composants physiques représentés par ces files. Par exemple, la file WiFi sera paramétrée de telle manière qu'en moyenne le temps passé par une opération dans le serveur de cette file corresponde au temps moyen mis par une opération pour être transmise d'un mobile à un cloud en passant l'interface physique WiFi du mobile. C'est pour cette raison également que l'appellation *unités de temps*, que nous avons utilisée en toute généralité lors de la spécification du modèle mathématique, est convertie en seconde dans nos scénarii. En effet, l'unité de temps de référence pour définir les spécificités des composants d'un mobile est la seconde. Cependant, il faut dès à présent préciser que la notion de temps durant une simulation est un temps simulé, il ne correspond en rien au temps qui s'écoule dans notre monde physique. L'ordinateur que nous utilisons simule environ 50000 secondes en une seconde réelle. Voyons à présent la justification des valeurs choisies en commençant par la file WiFi et par la file cellulaire.

Nous avons vu que le serveur de la file WiFi doit transmettre en moyenne une opération en μ_w unités de temps (section 2.2.4). Pour déterminer μ_w, nous nous basons sur les caractéristiques d'une interface physique WiFi. Dans ce cas, comme

3. Pour la définition complète des paramètres, se reporter à la section 2.2 et pour un résumé de ceux-ci, se reporter au tableau 2.1 de la section 2.3.

nous l'avons vu au chapitre précédent,

$$\mu_w = \frac{s_w}{E[X]} \tag{3.1}$$

où $E[X]$ est la taille moyenne d'une opération en Kilobyte (KB) et s_w est la vitesse nominale de transmission de l'interface WiFi en Kilobits par seconde (Kbps). Nous posons comme hypothèse qu'une opération fait $125KB$ en moyenne (Wu *et al.*, 2015) et que la vitesse de transmission nominale d'une interface physique WiFi est de $2000Kbps$ (Lee *et al.*, 2013). Nous convertissons les $2000Kbps$ en $250KBps$ (Kilobytes par seconde). Nous obtenons dès lors $\mu_w = 250KBps/125KB = 2op/s$ (opérations par seconde), ce qui signifie qu'une interface physique WiFi transfère en moyenne deux opérations de *125KB* par seconde ou encore une opération toutes les 0.5 secondes. Pour que la file WiFi simule un tel comportement, il faut que son serveur traite, en moyenne, 2 opérations par seconde ou encore que chaque opération reste 0.5 seconde dans le serveur avant d'en sortir. Dans ce cas, les temps de service des opérations dans ce serveur sont donnés par une collection de variables aléatoires distribuées comme des exponentielles de paramètre μ. Hors, en théorie, la valeur moyenne d'une collection de variables aléatoires distribuées comme des exponentielles de paramètre μ est égale à $1/\mu = 0.5$ alors $\mu = 2$. Ainsi, $\mu = \mu_w$ si la distribution est exponentielle. C'est la raison pour laquelle dans la spécification du modèle mathématique, nous utilisons directement comme paramètre de nos exponentielles la fréquence moyenne attendue pour le traitement des opérations au sein des serveurs de nos files d'attente [4]. Le même raisonnement est applicable à la file cellulaire avec cette fois-ci comme caractéristique de l'interface physique qu'elle représente $s_w = 800Kbps$ (toujours selon (Lee *et al.*, 2013)). Nous obtenons dès lors $\mu_w = 100KBps/125KB = 0.8$.

Pour la file WiFi, dans les trois premiers scénarii, on considère que les périodes de WIFI-ON et WIFI-OFF sont équivalentes. Le temps passé dans chacun de ces deux états est le même, raison pour laquelle nous affectons aux paramètres ξ et η la valeur 1. Dans le scénario 4, on cherche à s'assurer que l'alternance des états est réalisée dans les bonnes proportions pour cette file en particulier. On affecte alors à ξ la valeur 5 et à η la valeur 2. On garde pour les autres paramètres les valeurs utilisées pour le scénario 3. De ce fait, seules les mesures de performance de la file WiFi doivent varier entre le scénario 3 et le 4.

Ensuite, on pose comme hypothèse que le mobile exécute deux opérations en moyenne par seconde et que le cloud en exécute cinq en moyenne par seconde. Ceci semble raisonnable puisque le cloud exécute plus rapidement les opérations que le

4. Bien sûr, le même raisonnement n'est pas possible pour d'autres distributions. Comme par exemple des distributions d'Erlang de paramètres (k, μ). Dans ce cas, $\mu_w = k\mu$. Le paramètre μ à appliquer à la distribution est alors fonction de k et de la vitesse moyenne de traitement attendue du serveur de la file WiFi

mobile (un facteur 2.5 étant une base cohérente selon (Wu *et al.*, 2015)). Ainsi, on travaille également avec des distributions exponentielles, $\mu_m = 2$ et $\mu_c = 5$.

Concernant les puissances nominales pour la file locale, la file cellulaire et la file WiFi, nous utilisons respectivement les valeurs $p_m = 2w$, $p_g = 2.5w$ et $p_w = 0.7w$ (Balasubramanian *et al.*, 2009). Nous constatons, comme nous l'avons vu au chapitre précédent, que $p_g > p_w$ et $\mu_g < \mu_w$.

Pour le processus de Poisson décrivant les arrivées des opérations, nous posons comme hypothèse que $\lambda = 0.6$, ce qui signifie que les opérations susceptibles d'être migrées arrivent dans le dispatcher α_1 à raison de 0.6 opération par seconde. Nous ne faisons pas intervenir le dispatcher α_0 dans nos simulations car nous ne considérons que les opérations migrables. Le point d'entrée réel de notre réseau est donc le dispatcher α_1. Pour rappel, nous pouvons nous le permettre du fait de la discipline de service avec priorité de la file locale (section 2.2.3).

Pour les dispatchers, nous avons déjà expliqué ci-dessus les conséquences des choix des trois valeurs affectées à π_{α_1}. La raison de ces choix est la suivante : nous souhaitons vérifier que l'implémentation se comporte correctement pour les deux valeurs extrêmes de π_{α_1} et pour une valeur intermédiaire. Pour le dispatcher α_2, nous utilisons uniquement l'affectation $\pi_{\alpha_2} = 0.5$ puisque les valeurs extrêmes sont testées par le dispatcher α_1 et que ces deux dispatchers se comportent de la même manière. En procédant ainsi, nous sommes certain que la file WiFi et la file cellulaire seront mises à contribution durant les simulations où $0 < \pi_{\alpha_1} \leq 1$.

Pour la vérification de la file transmission, nous définissons deux scénarii. Les paramètres utilisés pour chacun d'eux sont proposés dans le tableau 3.2. Pour

TABLE 3.2 – Paramètres de simulation de la file transmission

	File transmission							
	λ_1	μ_1	p_1	λ_2	μ_2	p_2	η	ξ
Scénario 1	0,085714	0.8	2.5	0,214286	2	0.7	1	1
Scénario 2	0,085714	0.8	2.5	0,214286	2	0.7	2	5

rappel, celle-ci est vérifiée indépendamment du réseau de files de la stratégie ininterrompue. La raison sera expliquée dans la section 3.3.4, lorsque nous l'analyserons théoriquement. Pour cette file, nous avons dit que le serveur utilisé dans l'état WIFI-OFF, qui correspond à l'état 1, est celui aux propriétés de l'interface physique cellulaire, et, dans l'état WIFI-ON, qui correspond à l'état 2, le serveur aux propriétés de l'interface physique WiFi. Nous reprenons donc les mêmes valeurs que précédemment pour les paramètres de ces serveurs, la justification est la même. Dans ce cas, les opérations arrivent directement dans la file transmission selon un processus de Poisson de paramètre $\lambda_1 = 0,085714$, autrement dit, à la vitesse moyenne de $0,085714$ opérations par seconde dans l'état 1 et $\lambda_2 = 0,214286$

pour l'état 2. Ceci afin de réaliser la contrainte $\lambda_1/\mu_1 = \lambda_2/\mu_2$. Les raisons de cette contrainte seront également détaillées dans la section 3.3.4. C'est à cause de celle-ci que nous sommes forcé de vérifier l'implémentation de cette file en dehors du réseau de la stratégie ininterrompue. Enfin, dans le premier scénario, la file passe en moyenne le même temps dans l'état 1 que dans l'état 2. Par contre, dans le deuxième scénario, elle passe en moyenne 2/5 du temps dans l'état 2 et le reste dans l'état 1. Comme pour la file WiFi, ceci permet de s'assurer que l'alternance des états est implémentée dans les proportions voulues.

3.3.2 Choix des mesures de performance

Cette section et les deux suivantes sont certainement les plus importantes de notre mémoire car elles mettent en évidence l'analyse réalisée sur notre article de référence (Wu *et al.*, 2015). En effet, pour spécifier le modèle mathématique et, par la suite, pour implémenter ce modèle, il était nécessaire de comprendre l'analyse mathématique réalisée sur celui-ci par l'auteur de l'article. Hors, dans les sections 3.3.2, 3.3.3, 3.3.4, nous abordons ces analyses et la manière dont nous les avons interprétées afin d'obtenir les mêmes mesures de performance par simulation que l'auteur qui les a calculées théoriquement. De plus, nous avons détecté dans ses analyses une erreur de modélisation que nous sommes en mesure de corriger.

Il existe quatre types de mesures de performance utilisés dans l'article afin de comparer le compromis énergie-temps de réponse. Nous utiliserons également ceux-ci pour la vérification de notre implémentation. Ces mesures sont :

- le nombre moyen d'opérations dans une file, noté $\mathcal{N}$. Cette quantité est égale à la somme du nombre moyen d'opérations dans le buffer de la file ($\mathcal{L}$) et du nombre moyen d'opérations dans son serveur ($\mathcal{Q}$).
- la puissance moyenne dissipée par le serveur d'une file, notée $\mathcal{P}$. Cette quantité est égale au produit du nombre moyen d'opérations dans le serveur de la file ($\mathcal{Q}$) et de la puissance nominale de celui-ci.
- le temps de séjour moyen d'une opération, noté $\mathcal{T}$. Celui-ci s'observe soit sur une file, soit sur une branche d'un réseau de files, soit sur l'ensemble de ce réseau. Il correspond au temps moyen passé par une opération dans une file, sur une branche d'un réseau ou dans l'ensemble de celui-ci.
- l'énergie moyenne consommée par une opération dans une file, sur la branche d'un réseau ou dans l'ensemble de celui-ci, notée $\mathcal{E}$. Au niveau d'une file, celle-ci est directement liée au temps moyen passé par une opération dans le serveur de la file ($\mathcal{S}$). Au niveau d'une branche du réseau ou de son ensemble, cette mesure de performance dépend directement des files traversées.

Bien sûr, les mesures de performance réalisées sur le réseau de files sont intrinsèquement liées aux mesures de performance réalisées sur chacune des files qui le constitue. Par exemple, si on prend le temps moyen mis par une opération

pour traverser une branche d'un réseau, celui-ci correspond à la somme des temps moyens passés par une opération dans les files par lesquelles elle a cheminé sur cette branche. De plus, pour connaitre le temps moyen passé dans le réseau toute branche confondue, il faut prendre en compte la proportion d'opérations passées par chacune d'elles. Il est donc primordial de choisir des mesures de performance, d'une part, sur chaque file pour vérifier leur comportement individuel, et, d'autre part, sur le réseau pour vérifier que la manière d'acheminer les opérations est également correcte. Il est important de préciser que nous ne réalisons pas de mesures de performance sur les dispatchers. En effet, nous avons vu que le travail qu'ils réalisent est négligeable du fait des choix instantanés qu'ils posent. De plus, le nombre moyen d'opérations qui sont orientées vers l'un ou l'autre des deux modules liés à un dispatcher se répercute sur le taux moyen d'arrivées des opérations dans les files d'attente se situant à la suite de ce dispatcher. Ce taux moyen d'arrivée influera directement sur les mesures que nous réalisons sur ces files, ce qui implique que si toutes les files se situant à la suite d'un dispatcher se comportent correctement, on peut en déduire que le dispatcher se comporte correctement. Par conséquent, si toutes les files du réseau se comportent correctement, tous les dispatchers du réseau se comportent correctement également.

Voyons à présent comment sont réparties les mesures de performance choisies. Pour la file locale, nous avons :

- le nombre moyen d'opérations, noté $\mathcal{N}_m$
- la puissance moyenne dissipée, notée $\mathcal{P}_m$
- le temps de séjour moyen, noté $\mathcal{T}_m$
- l'énergie moyenne consommée, notée $\mathcal{E}_m$

Pour la file cellulaire, nous avons :

- le nombre moyen d'opérations, noté $\mathcal{N}_g$
- la puissance moyenne dissipée, notée $\mathcal{P}_g$
- le temps de séjour moyenne, noté $\mathcal{T}_g$
- l'énergie moyenne consommée, notée $\mathcal{E}_g$

Pour la file WiFi, nous avons :

- le nombre moyen d'opérations, noté $\mathcal{N}_w$
- la puissance moyenne dissipée, notée $\mathcal{P}_w$
- le temps de séjour moyen, noté $\mathcal{T}_w$
- l'énergie moyenne consommée, notée $\mathcal{E}_w$

Pour la file cloud, nous avons :

- le nombre moyen d'opérations, noté $\mathcal{N}_c$
- le temps de séjour moyen, noté $\mathcal{T}_c$

Pour la file transmission, nous avons :

- le nombre moyen d'opérations, notée $\mathcal{N}_t$
- la puissance moyenne dissipée dans l'état *WIFI-OFF*, notée $\mathcal{P}_1$

– la puissance moyenne dissipée dans l'état *WIFI-ON*, notée $\mathcal{P}_2$
– le temps de séjour moyen, noté $\mathcal{T}_t$
– l'énergie moyenne consommée, notée $\mathcal{E}_t$

Comme mesures de performance sur l'ensemble du réseau, nous avons :

– le temps de séjour moyen, noté $\mathcal{T}_r$
– l'énergie moyenne consommée, notée $\mathcal{E}_r$

On constate que pour la file locale, la file cellulaire et la file WiFi, nous travaillons avec les quatre mesures de performance. Pour la file cloud, nous ne considérons pas la puissance moyenne dissipée ni l'énergie moyenne consommée puisque nous avons considéré (section 2.1.2) que cette file ne consommait pas d'énergie. Pour la file transmission, nous travaillons avec les quatre mesures de performance mais nous dissocions la puissance moyenne dissipée pour l'état WIFI-ON et l'état WIFI-OFF. Voyons à présent comment obtenir ces mesures de performance, d'une part, par simulation, et, d'autre part, théoriquement en se référant à l'analyse réalisée dans notre article de référence.

3.3.3 Calcul des moyennes empiriques des mesures de performances

Comme nous l'avons évoqué, les mesures de performance obtenues par simulation sont calculées à partir de séries de données (appelées également séries d'observations). Nous verrons en détail, dans la section 3.3.5, le protocole de simulation que nous avons défini et dans lequel nous déterminons la taille et le nombre de séries de données à récolter et expliquons les raisons de ce choix. Dans cette section, nous nous concentrons uniquement sur la manière dont nous récoltons les données de ces séries. Nous commençons par le procédé de récolte du nombre d'opérations dans une file.

Comme nous l'avons vu dans l'implémentation de l'outil de simulation, nous réalisons ces mesures ponctuellement durant la simulation, c'est-à-dire à certains instants particuliers qui peuvent être situés à des intervalles de temps dont la taille est constante ou aléatoire. Par exemple, toutes les 10 secondes de simulation ou encore à des intervalles de temps aléatoires dont les valeurs sont données selon une collection de variables aléatoires quelconques. Au moment où l'observation est déclenchée, toutes les mesures de nombre d'opérations sur l'ensemble des files d'un réseau sont réalisées et les valeurs obtenues sont enregistrées. Le schéma du haut de la figure 3.15 illustre ce principe pour une file $M/M/1$ et $M/M/1_{ON/OFF}$. Sur celui-ci, on constate que, lors de la première observation, le nombre d'opérations dans la file est de 3, lors de la seconde, de 1, etc. Pour cette simulation, une série de huit mesures est réalisée. Cette série de données est cautionnée à la première ligne

du tableau 3.3. Dans la dernière colonne du tableau, on retrouve $\overline{\mathcal{N}}$[5], la moyenne du nombre d'opérations dans la file.

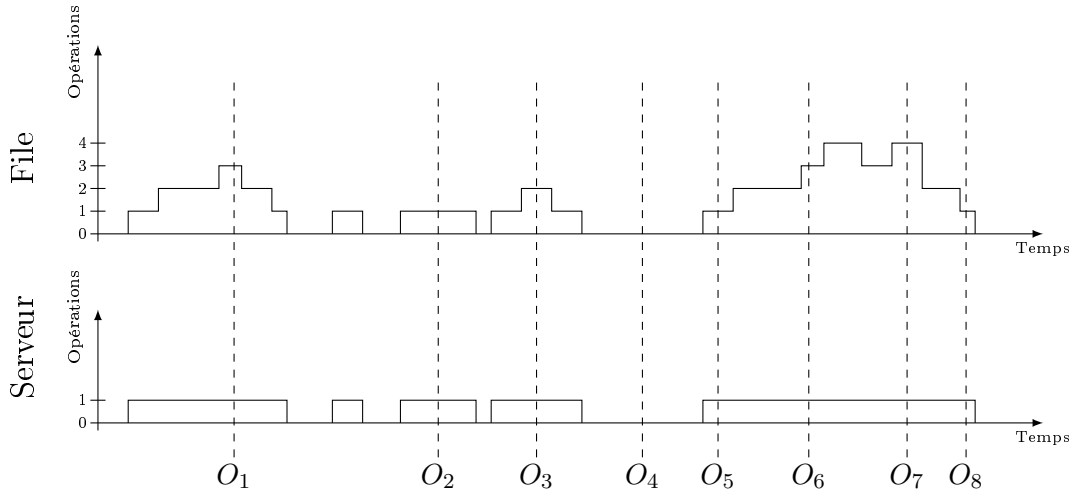


FIGURE 3.15 – Mesure du nombre d'opérations dans la file et son serveur

TABLE 3.3 – Tableau des observations du nombre d'opérations de la figure 3.15

Observation	1	2	3	4	5	6	7	8	Moy. emp
File	3	1	2	0	1	3	4	1	1.875
Server	1	1	1	0	1	1	1	1	0.875

Pour la récolte des données permettant le calcul de la puissance moyenne dissipée dans une file, il faut partir de sa définition :

$$\overline{\mathcal{P}} = p \cdot \overline{\mathcal{Q}} \tag{3.2}$$

avec p la puissance nominale du serveur de la file et $\overline{\mathcal{Q}}$ le nombre moyen d'opérations dans le serveur (sa charge). Dans ce cas, on observe ponctuellement si le serveur traite une opération. Bien entendu, à chaque observation, la valeur ne peut être que 1 ou 0 selon que le serveur traite ou non une opération à ce moment-là. Dès lors, on obtient une série de données composée de 0 et de 1 à partir de laquelle on peut calculer $\overline{\mathcal{Q}}$ et le multiplier par p afin d'obtenir $\overline{\mathcal{P}}$. Une autre possibilité, plus directe, est de multiplier à chaque observation le résultat obtenu par p. Dans ce cas, les valeurs de la série de données résultante sont constituées soit de p soit

5. Nous utilisons la notation avec une barre horizontale au dessus de la mesure de performance pour désigner une moyenne empirique, qui est ainsi différenciée de la moyenne théorique.

de 0. La moyenne calculée sur la série de données ainsi obtenue donne directement $\overline{\mathcal{P}}$. Sur le schéma du bas de la figure 3.15, se trouve la représentation graphique de l'évolution de la charge du serveur en fonction du temps. On constate, comme prévu, qu'à chaque observation la valeur mesurée est 1 ou 0. La série de données obtenue à partir des huit observations se trouve à la troisième ligne du tableau 3.3. La moyenne empirique de cette série ($\overline{\mathcal{Q}}$) est disponible dans la dernière colonne du tableau.

Pour les deux graphiques proposés, les observations sont réalisées à des instants choisis arbitrairement. En général, on choisira des intervalles de temps réguliers entre les différentes observations à réaliser, et ceci pour une question de facilité. Dans notre cas, cela n'a pas d'impact sur la qualité de l'échantillon récolté pour autant qu'il soit suffisamment grand. En effet, il n'y a pas, dans nos simulations, d'effet de répétition à intervalles de temps réguliers.

Dans l'implémentation, nous avons vu que le déclenchement d'une mesure de nombre d'opérations pour l'ensemble des files du réseau est réalisé par un évènement particulier (*LoadObservation*). Lorsque celui-ci est déclenché, toutes les sondes chargées de réaliser une mesure particulière enregistrent la valeur de celle-ci dans un *TallyStore* qui maintient un tableau représentant la série de données correspondant à la mesure de performance définie pour cette sonde. À l'issue de la simulation, il suffit de récupérer chaque *TallyStore* afin de procéder au traitement statistique de la série de données qu'il contient.

Concernant les temps de séjour moyen, les données récoltées ne sont plus issues d'observations ponctuelles déclenchées à des instants particuliers. Ces observations sont réalisées, cette fois, par opération. Autrement dit, on ne va plus observer une file ou le réseau à un moment donné, mais on va observer une opération lorsqu'elle passe par un point particulier. Par exemple, pour obtenir $\overline{\mathcal{T}_m}$, le temps de séjour moyen d'une opération dans la file locale, on va observer toutes les opérations qui sortent de cette file et récolter le temps qu'elles y sont restées. Si on souhaite connaitre le temps de séjour moyen dans le réseau, on va procéder de la même manière mais en observant cette fois les opérations à la sortie du réseau (ou aux sorties, s'il y en a plusieurs). Le temps de séjour d'une opération dans une file f est donnée par

$$\mathcal{T}_f = I_{D_f} - I_{A_f} \tag{3.3}$$

où I_{A_f} est le moment d'arrivée de l'opération dans la file et I_{D_f} son moment de sortie. Le temps de séjour d'une opération dans le réseau est, quant à lui, donné par

$$\mathcal{T}_r = I_{D_r} - I_{A_r} \tag{3.4}$$

avec I_{A_r}, le moment d'arrivée de l'opération dans le réseau et I_{D_r}, le moment de sortie de l'opération du réseau.

Le principe est le même pour récolter les données permettant de calculer la moyenne empirique de l'énergie consommée par une opération dans une file. Les observations sont réalisées à la sortie de la file. On a donc naturellement :

$$\mathcal{E}_f = \mathcal{S}_f \cdot p_f \tag{3.5}$$

où p_f est la puissance nominale de la file f, et $\mathcal{S}_f$ est le temps de service de l'opération de cette même file. Le temps de service d'une opération dans un serveur étant donné par :

$$\mathcal{S}_f = I_{D_{\mathcal{S}_f}} - I_{A_{\mathcal{S}_f}} \tag{3.6}$$

où $I_{A_{\mathcal{S}_f}}$ est le moment d'arrivée de l'opération dans le serveur de la file et $I_{D_{\mathcal{S}_f}}$ son moment de sortie.

L'énergie totale consommée par une opération à la sortie réseau est, quant à elle, donné par :

$$\mathcal{E} = \sum_f \mathcal{E}_f \tag{3.7}$$

comme la somme de $\mathcal{E}_f$, soit l'énergie consommée au sein de la file f.

3.3.4 Calcul de l'espérance des mesures de performance

Dans cette section, nous détaillons à présent les formules permettant d'obtenir les moyennes théoriques (espérances) des mesures de performance choisies. Pour nos calculs, nous commençons par la file locale et la file cellulaire de la stratégie interrompue. Toutes deux sont des files $M/M/1$. De ce fait, toutes les formules proposées sont valables pour ces deux files.

Analyse théorique de la file locale et de la file cellulaire

Pour la file locale, notre objectif est d'obtenir l'espérance du nombre d'opérations dans la file ($E[\mathcal{N}_m]$) [6], de la puissance dissipée par le serveur de la file ($E[\mathcal{P}_m]$), de l'énergie consommée par la file ($E[\mathcal{E}_m]$) et du temps de séjour dans la file ($E[\mathcal{T}_m]$).

Tout d'abord, nous commençons par le calcul de l'espérance du nombre d'opérations dans la file. Celle-ci étant une $M/M/1$, nous obtenons :

$$E[\mathcal{N}_m] = \frac{\rho_m}{1 - \rho_m} \tag{3.8}$$

6. On note, dans ce cas, la mesure de performance entre crochets précédée de E afin de signifier que l'on travaille avec une moyenne théorique.Ceci permet de bien distinguer la notation de la moyenne théorique de celle de la moyenne empirique.

avec

$$\rho_m = \frac{\lambda_m}{\mu_m} \tag{3.9}$$

où λ_m est le paramètre du processus de Poisson décrivant l'arrivée des opérations dans la file locale et μ_m, le paramètre de l'exponentielle décrivant les temps de service des opérations dans le serveur de cette file. Notons que par la présence du dispatcher α_1, nous avons :

$$\lambda_m = (1 - \pi_{\alpha_1}) \cdot \lambda \tag{3.10}$$

Il faut pouvoir ensuite calculer, pour cette file, $E[\mathcal{P}_m]$, qui est donné, dans notre article de référence, comme :

$$E[\mathcal{P}_m] = p_m \cdot Pr\{\mathcal{N}_m > 0\} \tag{3.11}$$

où p_m est la puissance nominale du serveur et $Pr\{\mathcal{N}_m > 0\}$ la probabilité qu'il y ait au moins une opération dans la file. Cette probabilité est égale à la charge moyenne du serveur qui est ρ_m.

Ensuite, pour $E[\mathcal{E}_m]$, l'espérance de l'énergie consommée par une opération dans cette file, on a :

$$E[\mathcal{E}_m] = \frac{1}{\lambda_m} \cdot E[\mathcal{P}_m] \tag{3.12}$$

$$= \frac{1}{\lambda_m} \cdot p_m \rho_m. \tag{3.13}$$

Après simplification, on obtient :

$$E[\mathcal{E}_m] = \frac{p_m}{\mu_m} \tag{3.14}$$

Ceci met en évidence la propriété suivante : l'énergie consommée est indépendante du taux d'arrivée des opérations dans une file $M/M/1$ (ce sera le cas également pour une file $M/M/1_{ON/OFF}$).

Enfin, l'espérance du temps de séjour dans cette file ($E[\mathcal{T}_m]$) s'obtient par la formule de Little à partir de l'espérance du nombre d'opérations dans la file. Nous avons donc :

$$E[\mathcal{T}_m] = \frac{1}{\lambda_m} \cdot E[\mathcal{N}_m] \tag{3.15}$$

Toutes ces expressions sont valables également pour la file cellulaire de la stratégie interrompue. Il suffit de remplacer l'indice m par g et de remarquer que λ_g est donné par :

$$\lambda_g = \pi_{\alpha_2} \cdot \lambda_c \tag{3.16}$$

avec

$$\lambda_c = \pi_{\alpha_1} \cdot \lambda \tag{3.17}$$

Analyse théorique de la file cloud

Cette file est une $M/M/\infty$. Seule l'espérance du nombre d'opérations ($E[\mathcal{N}_c]$) et l'espérance du temps de séjour ($E[\mathcal{T}_c]$) nous intéressent. En effet, pour rappel, on considère que la file cloud ne consomme pas d'énergie. Dans notre article de référence, l'espérance du nombre d'opérations dans cette file est donnée, par :

$$E[\mathcal{N}_c] = \frac{\lambda_c}{\mu_c} \tag{3.18}$$

avec

$$\lambda_c = \pi_{\alpha_1} \cdot \lambda \tag{3.19}$$

On constate ici que $E[\mathcal{N}_c]$ n'est autre que ρ_c, la charge moyenne des serveurs de cette file, ce qui s'explique par le fait qu'elle n'ait pas de buffer et que toute opération arrivant est prise en charge immédiatement.

La formule de Little s'applique également ici pour calculer l'espérance du temps de séjour :

$$E[\mathcal{T}_c] = \frac{1}{\lambda_c} \cdot E[\mathcal{N}_c] \tag{3.20}$$

Après avoir substitué $E[\mathcal{N}_c]$ par λ_c/μ_c et simplifié les λ_c, on obtient :

$$E[\mathcal{T}_c] = \frac{1}{\mu_c} \tag{3.21}$$

qui correspond tout simplement au temps de service moyen des serveurs. Encore une fois, ceci s'explique par la prise en charge immédiate des opérations, c'est-à-dire qu'elles ne peuvent rester dans cette file en moyenne que le temps que leur service soit terminé.

Analyse théorique de la file WiFi

Les files que nous avons vues jusqu'ici sont bien connues de la théorie des files d'attente et les formules qui découlent de leur analyse le sont également. Par contre, pour la file WiFi, qui est une file d'attente dont le serveur oscille entre des périodes d'activité et d'inactivité, l'analyse des performances est moins directe. Pour trouver l'espérance du nombre d'opérations dans cette file ($E[\mathcal{N}_w]$), l'auteur de l'article a dû procéder à l'analyse de la chaine de Markov décrivant cette file particulière. Il en a déduit des *équations de balance* à partir desquelles il a déterminé la formule permettant de calculer cette mesure de performance. Ce travail est loin d'être trivial et la description détaillée des étapes permettant d'y

arriver ne l'est pas non plus. Dans ce mémoire, nous ne détaillons pas le processus de transformation des équations de balance menant à la formule finale, car, d'une part, l'auteur ne propose pas l'ensemble des calculs qu'il réalise, et, d'autre part, la complexité mathématique dépasse le cadre de notre mémoire. Nous admettons donc, sans vérification mathématique, les formules proposées, d'autant plus que celles-ci concordent avec notre simulation (voir section 3.3.6).

Nous calculons pour cette file l'espérance du temps de séjour ($E[\mathcal{T}_w]$), l'espérance de la puissance dissipée ($E[\mathcal{P}_w]$) et l'espérance de l'énergie consommée ($E[\mathcal{E}_w]$). Nous commençons par l'espérance du nombre d'opérations dans cette file qui est donnée par l'expression :

$$E[\mathcal{N}_w] = \frac{\lambda_w}{\pi_{w_2}\mu_w - \lambda_w} + \frac{\pi_{w_1}\lambda_w\mu_w}{(\pi_{w_2}\mu_w - \lambda_w)(\xi + \eta)} \tag{3.22}$$

avec

$$\lambda_w = \pi_{\alpha_2} \cdot \lambda_c \tag{3.23}$$

et

$$\pi_{w_1} = \frac{\xi}{\eta + \xi} \tag{3.24}$$

$$\pi_{w_2} = \frac{\eta}{\eta + \xi} \tag{3.25}$$

où π_{w_1} est la probabilité que le serveur de la file soit dans l'état WIFI-OFF et π_{w_2} qu'il soit dans l'état WIFI-ON et ce, en régime stationnaire.

Pour le calcul de l'espérance de la puissance dissipée par le serveur de la file, l'auteur introduit le terme e qui peut prendre les valeurs 0 ou 1. Ce paramètre décrit l'état du serveur. Si $e = 1$, le serveur est actif (état WIFI-ON), si $e = 0$, le serveur est inactif (état WIFI-OFF). Ensuite, il donne la puissance dissipée par le serveur de la file comme étant :

$$E[\mathcal{P}_w] = p_w \cdot Pr[\mathcal{N}_w > 0, e_w = 1] \tag{3.26}$$

où $Pr[\mathcal{N}_w > 0, e_w = 1]$ respecte l'égalité suivante :

$$Pr[\mathcal{N}_w > 0, e_w = 1] = Pr[\mathcal{N}_w > 0 | e_w = 1] \cdot Pr[e_w = 1] \tag{3.27}$$

avec $Pr[e_w = 1]$ la probabilité que le serveur soit dans l'état actif (WIFI-ON), et $Pr[\mathcal{N}_w > 0 | e_w = 1]$ la probabilité d'avoir une opération dans la file sachant que le serveur est actif. Nous constatons que :

$$Pr[e_w = 0] = \pi_{w_1} \tag{3.28}$$

$$Pr[e_w = 1] = \pi_{w_2} \tag{3.29}$$

L'auteur nous dit ensuite que :

$$Pr[\mathcal{N}_w > 0 | e_w = 1] = \rho_w \tag{3.30}$$

et donc au final que :

$$E[\mathcal{P}_w] = p_w \cdot \rho_w \cdot Pr[e_w = 1] \tag{3.31}$$
$$= p_w \cdot \rho_w \cdot \pi_{w_2} \tag{3.32}$$

On pouvait considérer que :

$$\rho_w = \frac{\lambda_w}{\mu_w} \tag{3.33}$$

Cette hypothèse étant légitime car l'auteur définit ρ d'une manière générale comme étant λ/μ à différents endroits dans l'article et surtout il ne le définit jamais sous une autre forme [7]. De plus, les courbes qu'il obtient correspondent à cette définition de ρ. Cependant, nos simulations ne donnent pas les mêmes courbes [8]. Nous pouvons affirmer que l'auteur a considéré un mauvais ρ_w. Nous pensons que ρ_w, s'il est défini comme étant $Pr[\mathcal{N}_w > 0 | e_w = 1]$, est en fait donné par :

$$\rho_w = \frac{\lambda_w}{\pi_{w_2}\mu_w} \tag{3.34}$$

En effet, nous travaillons avec $Pr[\mathcal{N}_w > 0 | e_w = 1]$ soit la probabilité que le nombre d'opérations dans la file WiFi soit strictement positif lorsque le serveur est occupé. Dans le cas d'une $M/M/1$, il est connu que :

$$Pr[\mathcal{N}_w > 0, e_w = 1] = \rho_w \cdot \pi_{w_2} \tag{3.35}$$

avec

$$\pi_{w_2} = 1 \tag{3.36}$$
$$\rho_w = \frac{\lambda_w}{\mu_w} \tag{3.37}$$

7. Voir (Wu *et al.*, 2015) page 136 équation (1), page 138 équation (14) et (15), page 140 équation (33)

8. Ces courbes seront vues au chapitre suivant. En effet, l'auteur de l'article ne propose pas les calculs individuels des mesures de performance qu'il utilise pour analyser le compromis énergie-temps de réponse. Les courbes dont nous parlons ici seront donc vues à la section 4.2 de notre mémoire. Elles sont également disponibles dans notre article de référence (Wu *et al.*, 2015) à la page 141.

Dans le cas d'une $M/M/1_{ON/OFF}$, nous avons,

$$Pr[\mathcal{N}_w > 0, e_w = 1] = Pr[\mathcal{N}_w > 0 | e_w = 1] \cdot Pr[e_w = 1] \tag{3.38}$$
$$= Pr[\mathcal{N}_w > 0 | e_w = 1] \cdot \pi_{w_2} \tag{3.39}$$

Cette quantité vaut toujours λ_w/μ_w. Dès lors :

$$Pr[\mathcal{N}_w > 0 | e_w = 1] = \frac{\lambda_w}{\pi_{w_2}\mu_w} \tag{3.40}$$

soit

$$\rho_w \stackrel{\text{def}}{=} \frac{\lambda_w}{\pi_{w_2}\mu_w} \tag{3.41}$$

dans la suite et non comme λ_w/μ_w.

Cela a un sens puisque ρ_w est la proportion de temps pendant laquelle le serveur fournit un travail. Il ne peut cependant le fournir à la vitesse μ_w que dans $\pi_{w_2} \cdot 100$ % du temps. Cette remarque trouve également sont intuition dans le schéma de la figure 3.16. En effet, notons dans un premier temps que :

$$Pr[\mathcal{N}_w > 0, e_w = 1] = \frac{\sum_{i=1}^{n} A_i}{T_{tot}} \tag{3.42}$$

grâce à la théorie de l'ergodicité. On peut lire que la probabilité que le serveur soit en travail et actif (état WIFI-ON) est le rapport entre le temps total d'activité et le temps total de simulation. Sur la figure du haut est représentée l'activité d'un serveur $M/M/1$ et sur celle du bas, l'activité d'un serveur $M/M/1_{ON/OFF}$. On le voit sur la figure du bas, une portion de chaque unité de temps (à savoir $1 - \pi_{w_2}$) ne présente aucune activité pour le serveur ON/OFF. La proportion intervenant dans (3.42) doit en tenir compte. D'où l'intervention du facteur π_{w_2} dans (3.40).

Nous obtenons alors comme formule pour l'espérance de la puissance (en substituant (3.34) dans (3.32) et en simplifiant) :

$$E[\mathcal{P}_w] = p_w \cdot \frac{\lambda_w}{\mu_w} \tag{3.43}$$

La formule de Little s'applique également ici pour calculer l'espérance du temps de séjour :

$$E[\mathcal{T}_w] = \frac{1}{\lambda_w} \cdot E[\mathcal{N}_w] \tag{3.44}$$

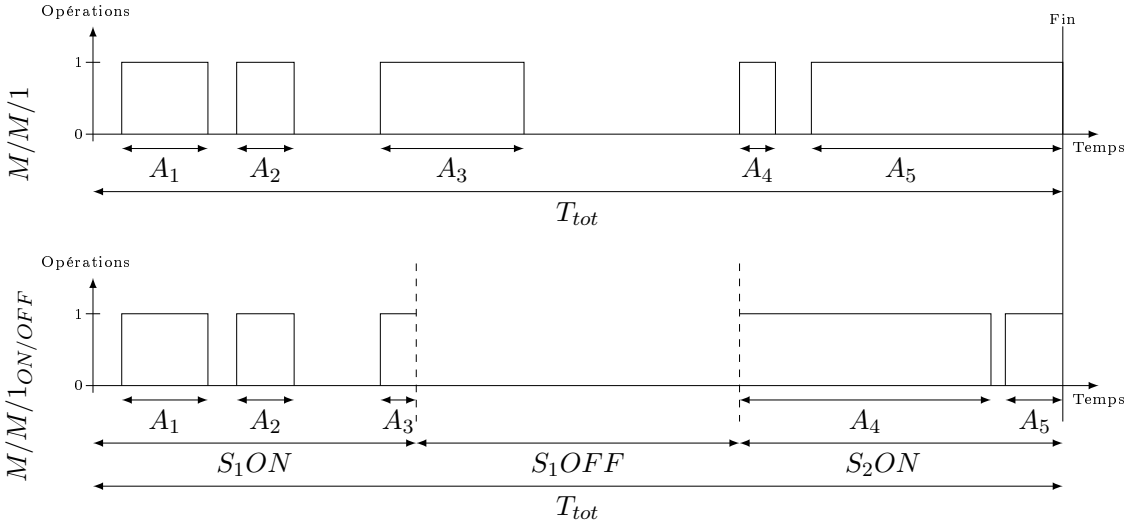


FIGURE 3.16 – Représentation graphique de $Pr[\mathcal{N} > 0, e = 1]$ pour une file $M/M/1$ et pour une file $M/M/1_{ON/OFF}$.

et l'espérance de l'énergie consommée est donnée par :

$$E[\mathcal{E}_w] = \frac{1}{\lambda_w} \cdot E[\mathcal{P}_w] \tag{3.45}$$

$$= \frac{1}{\lambda_w} \cdot p_w \rho_w \tag{3.46}$$

$$= \frac{p_w}{\mu_w} \tag{3.47}$$

Analyse théorique de la file de transmission

Pour cette file, afin de simplifier l'écriture, nous considérons que l'état WIFI-OFF de la file est l'état 1 et l'état WIFI-ON l'état 2. Pour rappel, dans l'état 1, la file se comporte comme une $M/M/1$ aux caractéristiques de l'interface physique cellulaire et, dans l'état 2, elle se comporte comme une $M/M/1$ aux caractéristiques de l'interface physique WiFi.

L'espérance du nombre d'opérations pour cette file est donnée par :

$$E[\mathcal{N}_t] = E[\mathcal{N}_1] + E[\mathcal{N}_2] \tag{3.48}$$

où $E[\mathcal{N}_t]$ est l'espérance du nombre d'opérations pour l'ensemble de la file et $E[\mathcal{N}_1]$, $E[\mathcal{N}_2]$, l'espérance du nombre d'opérations propres à chaque état. Afin de trouver la formule permettant de calculer $E[\mathcal{N}_t]$, l'auteur de l'article analyse la chaîne de

Markov décrivant cette file tout comme pour la file $M/M/1_{ON/OFF}$. Cependant, cette fois-ci, pour simplifier ses calculs, il pose la contrainte suivante :

$$\frac{\lambda_1}{\mu_1} = \frac{\lambda_2}{\mu_2} \tag{3.49}$$

Cette contrainte implique que $\rho_1 = \rho_2$ et donc que la charge des serveurs de la file soit à régime stationnaire équivalente, autrement dit que cette file se comporte en réalité de la même manière quel que soit l'état dans lequel on se trouve. Cette contrainte simplifie les calculs théoriques. Cependant, il n'est pas possible de la simuler sans avoir la maitrise du processus d'arrivée des opérations et de pouvoir modifier à chaque changement d'état le taux moyen de ces arrivées afin de réaliser la contrainte. Or, le processus d'arrivée des opérations dans cette file est un processus de Poisson de paramètre $\lambda_c = \pi_{\alpha_1} \cdot \lambda$. Nous ne pouvons pas modifier ce processus à moins de modifier soit le paramètre λ, soit le paramètre π_{α_1}. Cependant, dans ce cas, nous changeons également, à chaque changement d'état de la file transmission, le comportement du reste des modules du réseau. On constate donc que cette contrainte ne peut pas être simulée si cette file n'est pas un point d'entrée du réseau sur lequel on peut modifier à la volée les paramètres du processus d'arrivée des opérations. C'est la raison pour laquelle nous sommes obligés de vérifier le module représentant cette file en dehors du réseau de files de la stratégie ininterrompue. L'autre conséquence, liée au fait que cette contrainte ne soit pas applicable, est que nous n'aurons pas les mêmes résultats que l'auteur pour le compromis énergie-temps de réponse de cette stratégie. Cependant, nous pouvons quand même vérifier l'implémentation de ce module et donc être sûr que nos résultats, bien que différents, soient corrects. Prenant en compte cette contrainte, l'auteur nous donne les formules suivantes :

$$E[\mathcal{N}_1] = \pi_1 \cdot \frac{\lambda^*}{\mu^* - \lambda^*} \tag{3.50}$$

$$E[\mathcal{N}_2] = \pi_2 \cdot \frac{\lambda^*}{\mu^* - \lambda^*} \tag{3.51}$$

avec $\pi_1 = \frac{\xi}{\eta+\xi}$ et $\pi_2 = \frac{\eta}{\eta+\xi}$, les probabilités respectives d'être dans l'état 1 ou l'état 2 à régime stationnaire et λ^*, μ^*, deux quantités données par :

$$\lambda^* = \pi_1 \cdot \lambda_1 + \pi_2 \cdot \lambda_2 \tag{3.52}$$

$$\mu^* = \pi_1 \cdot \mu_1 + \pi_2 \cdot \mu_2 \tag{3.53}$$

Au final, puisque $\pi_1 + \pi_2 = 1$ on a :

$$E[\mathcal{N}_t] = \frac{\lambda^*}{\mu^* - \lambda^*} \tag{3.54}$$

$$\tag{3.55}$$

Pour la puissance dissipée, on a les formules suivantes :

$$E[\mathcal{P}_1] = p_1 \cdot \rho_1 \cdot \pi_1 \tag{3.56}$$
$$E[\mathcal{P}_2] = p_2 \cdot \rho_2 \cdot \pi_2 \tag{3.57}$$

avec p_1, ρ_1, p_2, ρ_2 les puissances nominales et les charges respectives du serveur de l'état 1 et de l'état 2.

L'espérance du temps de séjour dans la file est quant à elle donnée par :

$$E[\mathcal{T}_t] = \frac{1}{\lambda^*} \cdot E[\mathcal{N}_t] \tag{3.58}$$

et l'espérance de l'énergie consommée dans la file par :

$$E[\mathcal{E}_t] = \frac{1}{\lambda^*} \cdot (E[\mathcal{P}_1] + E[\mathcal{P}_2]) \tag{3.59}$$

Analyse théorique du temps de séjour et de l'énergie pour l'ensemble du réseau

Pour l'espérance de temps de séjour dans le réseau ($E[\mathcal{T}_r]$) et l'espérance de l'énergie consommée dans le réseau ($E[\mathcal{E}_r]$), on a :

$$E[\mathcal{T}_r] = \pi_{\alpha_1} E[\mathcal{T}_o] + (1 - \pi_{\alpha_1}) E[\mathcal{T}_m] \tag{3.60}$$
$$E[\mathcal{E}_r] = \pi_{\alpha_1} E[\mathcal{E}_o] + (1 - \pi_{\alpha_1}) E[\mathcal{E}_m] \tag{3.61}$$

avec $E[\mathcal{T}_o]$ et $E[\mathcal{T}_m]$, respectivement l'espérance du temps de séjour des opérations transmises vers la file cloud et l'espérance du temps de séjour dans la file locale. On applique le même raisonnement pour l'espérance de l'énergie consommée. On constate ici que les espérances pour le temps de séjour et pour l'énergie consommée sont directement liées à la probabilité qu'une opération soit migrée ou non. Après de multiples transformations, dont le détail est disponible à l'annexe D, on obtient :

$$E[\mathcal{T}_r] = \frac{1}{\lambda}(E[\mathcal{N}_g] + E[\mathcal{N}_w] + E[\mathcal{N}_c] + E[\mathcal{N}_m]) \tag{3.62}$$
$$E[\mathcal{E}_r] = \frac{1}{\lambda}(E[\mathcal{P}_g] + E[\mathcal{P}_w] + E[\mathcal{P}_m]) \tag{3.63}$$

Pour être précis, il faut substituer les indices g et w par 1 et 2 pour la stratégie ininterrompue.

3.3.5 Protocole de simulation

Le protocole de simulation définit trois paramètres appliqués à chaque simulation :

– le moment où le système atteint son état stationnaire
– la taille des séries de données
– l'intervalle de temps entre chaque mesure ponctuelle

Pour l'ensemble des scénarii simulés pour la vérification, nous considérons que le système atteint son état stationnaire lorsque les 1000 premières opérations en sont sorties. Pour justifier ce choix, nous proposons ci-dessous deux graphiques représentant, pour les scénarii 1 (figures 3.17) et 2 (figures 3.18) de la stratégie interrompue, l'évolution du temps de séjour moyen des opérations dans le réseau. Nous constatons qu'après 1000 observations au plus, cette moyenne converge, ce qui signifie que le système a atteint son état stationnaire. Pour les autres scénarii, nous procédons de la même manière pour nous assurer de la stabilité du système.

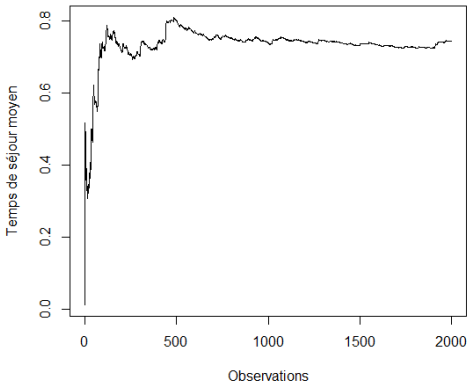


FIGURE 3.17 – Évolution du temps de séjour moyen d'une opération dans le réseau de la stratégie interrompue pour le scénario 1

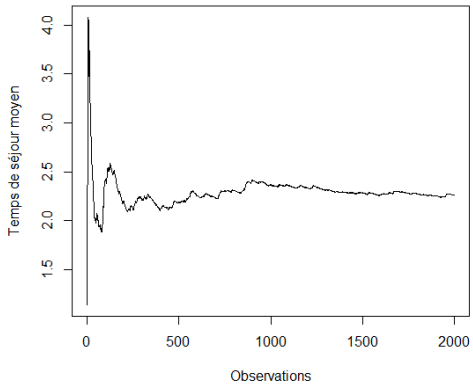


FIGURE 3.18 – Évolution du temps de séjour moyen d'une opération dans le réseau de la stratégie interrompue pour le scénario 2

Nous calculons les moyennes empiriques relatives à chaque mesure de performance sur base de 100000 mesures réparties en 100 séries de 1000 mesures. Cette technique de récolte des données est appelée *Batch mean*. Elle est inspirée de (Leemis et Park, 2006). Nous commençons la récolte lorsque le système est à l'état stationnaire.

Pour les intervalles de temps entre chaque mesure ponctuelle, nous avons choisi 15 secondes. Une simulation durera donc au minimum 15000 secondes et l'ensemble des simulations au moins 150000 secondes. Le temps réel de simulation d'un scénario est donc d'environ trois secondes, avec, pour rappel, environ 50000 secondes simulées en une seconde réelle.

3.3.6 Comparaison des résultats théoriques aux résultats empiriques

Afin de comparer les résultats empiriques aux résultats théoriques, nous proposons, pour chaque scénario, le tableau comparatif des résultats obtenus. Afin de ne pas encombrer la lecture, les calculs d'espérance sont détaillés à l'annexe C. Pour comparer les résultats, nous proposons de gauche à droite les colonnes suivantes :

- la moyenne empirique (*Moy. emp.*)
- la borne inférieure de l'intervalle de confiance (*Min(ic)*) et sa borne supérieure *Max(ic)*. Les bornes de cet intervalle sont définies comme :

$$[\overline{X} - 1.96 \cdot \frac{\sigma}{\sqrt{n}}, \overline{X} + 1.96 \cdot \frac{\sigma}{\sqrt{n}}]$$

  avec $\overline{X}$ la moyenne calculée sur la série de données, σ l'écart-type obtenu pour cette série, et n le nombre d'observations de cette série. La constante $1,96$ est utilisée pour calculer un intervalle de confiance de 95%.
- la moyenne théorique (*Espérance)*

Nous plaçons l'espérance entre les bornes de l'intervalle de confiance afin qu'il soit plus aisé de se rendre compte que la valeur de celle-ci est bien incluse dans l'intervalle.

Pour le premier scénario (tableau 3.4), nous constatons tout d'abord que le dispatcher a correctement orienté toutes les opérations vers la file locale puisqu'aucune opération n'est passée par la file WiFi, ni la file cellulaire ni la file cloud. Ensuite, nous constatons que les espérances calculées sur la file locale se situent dans l'intervalle de confiance. Pour ce scénario, l'ensemble du réseau s'est comporté comme attendu.

Pour le deuxième scénario (tableau 3.5), on constate cette fois, comme attendu, que toutes les opérations ont été transmises vers la file cloud. Toutes les espérances se situent dans l'intervalle de confiance. Le réseau s'est comporté comme prévu.

Pour le troisième scénario (tableau 3.6), nous constatons que les opérations ont été orientées vers la file locale et vers la file cloud comme prévu. Toutes les espérances se situent dans l'intervalle de confiance. On peut donc dire, d'une part, que le réseau s'est comporté correctement, et, d'autre part, que les périodes WIFI-ON et WIFI-OFF se sont enchainées comme prévu. Pour le dernier scénario (tableau 3.7), le constat est le même. En conclusion, l'ensemble des modules impliqués dans la stratégie interrompue sont implémentés correctement.

Voyons à présent le cas de la file transmission. L'ensemble des calculs d'espérance sont également proposés dans l'annexe C. Pour cette file, que ce soit pour le scénario 1 (tableau 3.8) ou pour le scénario 2 (tableau 3.9), toutes les espérances se situent également dans l'intervalle de confiance de la moyenne empirique de

Table 3.4 – Stratégie interrompue : scénario 1

		Moy. emp.	*Min(ic)*	*Espérance*	*Max(ic)*
F. locale	$\mathcal{N}_m$	0,432750	0,428302	0,428571	0,437198
	$\mathcal{P}_m$	0,604160	0,598964	0,600000	0,609356
	$\mathcal{T}_m$	0,714213	0,706333	0,714286	0,722092
	$\mathcal{E}_m$	0,999295	0,993340	1,000000	1,005249
F. cellulaire	$\mathcal{N}_g$	0,000000	0,000000	0,000000	0,000000
	$\mathcal{P}_g$	0,000000	0,000000	0,000000	0,000000
	$\mathcal{T}_g$	0,000000	0,000000	0,000000	0,000000
	$\mathcal{E}_g$	0,000000	0,000000	0,000000	0,000000
F. WiFi	$\mathcal{N}_w$	0,000000	0,000000	0,000000	0,000000
	$\mathcal{P}_w$	0,000000	0,000000	0,000000	0,000000
	$\mathcal{T}_w$	0,000000	0,000000	0,000000	0,000000
	$\mathcal{E}_w$	0,000000	0,000000	0,000000	0,000000
F. cloud	$\mathcal{N}_c$	0,000000	0,000000	0,000000	0,000000
	$\mathcal{T}_c$	0,000000	0,000000	0,000000	0,000000
Global	$\mathcal{T}_r$	0,714213	0,706333	0,714286	0,722092
	$\mathcal{E}_r$	0,999295	0,993340	1,000000	1,005249

Table 3.5 – Stratégie interrompue : scénario 2

		Moy. emp.	*Min(ic)*	*Espérance*	*Max(ic)*
F. locale	$\mathcal{N}_m$	0,000000	0,000000	0,000000	0,000000
	$\mathcal{P}_m$	0,000000	0,000000	0,000000	0,000000
	$\mathcal{T}_m$	0,000000	0,000000	0,000000	0,000000
	$\mathcal{E}_m$	0,000000	0,000000	0,000000	0,000000
F. cellulaire	$\mathcal{N}_g$	0,594750	0,587627	0,600000	0,601873
	$\mathcal{P}_g$	0,931350	0,922538	0,937500	0,940162
	$\mathcal{T}_g$	1,986988	1,958462	2,000000	2,015515
	$\mathcal{E}_g$	3,120775	3,099766	3,125000	3,141785
F. WiFi	$\mathcal{N}_w$	0,647870	0,640984	0,642857	0,654756
	$\mathcal{P}_w$	0,104776	0,103282	0,105000	0,106270
	$\mathcal{T}_w$	2,163969	2,134434	2,142857	2,193503
	$\mathcal{E}_w$	0,350964	0,348829	0,350000	0,353098
F. cloud	$\mathcal{N}_c$	0,118610	0,116380	0,120000	0,120840
	$\mathcal{T}_c$	0,200119	0,198863	0,200000	0,201376
Global	$\mathcal{T}_r$	2,270932	2,242802	2,271429	2,299062
	$\mathcal{E}_r$	1,728110	1,711301	1,737500	1,744919

Table 3.6 – Stratégie interrompue : scénario 3

		Moy. emp.	*Min(ic)*	*Espérance*	*Max(ic)*
F. locale	$\mathcal{N}_m$	0,176760	0,173615	0,176471	0,179905
	$\mathcal{P}_m$	0,300380	0,295343	0,300000	0,305417
	$\mathcal{T}_m$	0,586788	0,581561	0,588235	0,592016
	$\mathcal{E}_m$	0,999264	0,992976	1,000000	1,005552
F. cellulaire	$\mathcal{N}_g$	0,230140	0,226805	0,230769	0,233475
	$\mathcal{P}_g$	0,469275	0,462975	0,468750	0,475575
	$\mathcal{T}_g$	1,532905	1,518298	1,538462	1,547513
	$\mathcal{E}_g$	3,118798	3,098219	3,125000	3,139377
F. WiFi	$\mathcal{N}_w$	0,263020	0,259240	0,264706	0,266800
	$\mathcal{P}_w$	0,052311	0,051036	0,052500	0,053586
	$\mathcal{T}_w$	1,775557	1,755912	1,764706	1,795203
	$\mathcal{E}_w$	0,350509	0,348304	0,350000	0,352714
F. cloud	$\mathcal{N}_c$	0,060310	0,059005	0,060000	0,061615
	$\mathcal{T}_c$	0,200960	0,199787	0,200000	0,202132
Global	$\mathcal{T}_r$	1,217725	1,205717	1,219910	1,229733
	$\mathcal{E}_r$	1,363946	1,352679	1,368750	1,375212

Table 3.7 – Stratégie interrompue : scénario 4

		Moy. emp.	*Min(ic)*	*Espérance*	*Max(ic)*
F. locale	$\mathcal{N}_m$	0,176050	0,173002	0,176471	0,179098
	$\mathcal{P}_m$	0,299080	0,294504	0,300000	0,303656
	$\mathcal{T}_m$	0,586762	0,581518	0,588235	0,592007
	$\mathcal{E}_m$	0,999187	0,992872	1,000000	1,005503
F. cellulaire	$\mathcal{N}_g$	0,228460	0,225089	0,230769	0,231831
	$\mathcal{P}_g$	0,466000	0,459679	0,468750	0,472321
	$\mathcal{T}_g$	1,532909	1,518296	1,538462	1,547522
	$\mathcal{E}_g$	3,118768	3,098174	3,125000	3,139363
F. WiFi	$\mathcal{N}_w$	0,428650	0,423035	0,428572	0,434265
	$\mathcal{P}_w$	0,051716	0,050627	0,052500	0,052805
	$\mathcal{T}_w$	2,881732	2,844592	2.857147	2,918871
	$\mathcal{E}_w$	0,350787	0,348516	0,350000	0,353059
F. cloud	$\mathcal{N}_c$	0,059220	0,057655	0,060000	0,060785
	$\mathcal{T}_c$	0,200956	0,199781	0,200000	0,202132
Global	$\mathcal{T}_r$	1,502508	1,481394	1,493019	1,523622
	$\mathcal{E}_r$	1,364688	1,353238	1,368750	1,376138

chaque mesure de performance. Le module représentant cette file est donc implémenté correctement.

Table 3.8 – File transmission : scénario 1

	Moy. emp.	*Min(ic)*	*Espérance*	*Min(ic)*
$\mathcal{P}_1$	0,135900	0,132298	0,133929	0,139502
$\mathcal{P}_2$	0,037625	0,036510	0,037500	0,038740
$\mathcal{N}_t$	0,119990	0,117671	0,120000	0,122309
$\mathcal{T}_t$	0,794069	0,787459	0,800000	0,800680
$\mathcal{E}_t$	1,133708	1,122741	1,142857	1,144676

Table 3.9 – File transmission : scénario 2

	Moy. emp.	*Min(ic)*	*Espérance*	*Min(ic)*
$\mathcal{P}_1$	0,192275	0,187455	0,191327	0,197095
$\mathcal{P}_2$	0,021140	0,020418	0,021429	0,021862
$\mathcal{N}_t$	0,120580	0,117937	0,120000	0,123223
$\mathcal{T}_t$	0,979717	0,972348	0,980000	0,987085
$\mathcal{E}_t$	1,737063	1,725069	1,737500	1,749058

Nous pouvons dès lors considérer que l'implémentation de l'outil de simulation est correcte.

3.4 Conclusion

Dans ce chapitre, nous avons implémenté l'outil de simulation sur base du modèle mathématique vu au chapitre précédent. Nous avons réalisé cette implémentation dans le langage *Java* en prenant appui sur la bibliothèque SSJ. Nous avons décrit, d'une part, la hiérarchie de classe représentant l'implémentation des entités du modèle mathématique, et, d'autre part, l'implémentation du simulateur proposé par cette bibliothèque.

Nous avons ensuite procédé à la vérification de l'implémentation de l'outil en comparant, pour de même scénarii, différentes mesures de performance obtenues théoriquement et empiriquement. Dans le premier cas, nous avons calculé ces mesures de performance en utilisant les formules proposées dans notre article de référence. Dans le second cas, nous avons simulé, avec l'outil développé, ces scénarii et nous avons récolté des séries de données qui, après traitement statistique, nous on permis d'obtenir les moyennes empiriques correspondant aux mêmes mesures de performance.

Dans ce chapitre, nous avons également mis en évidence deux erreurs de modélisation commises par l'auteur de l'article. La première, que nous avons corrigée, concernait la définition de ρ de la file WiFi de la stratégie interrompue. La seconde concernait l'application d'une contrainte sur la file transmission. Elle n'est en réalité pas applicable lors d'une simulation et a fortiori pas non plus dans un système physique réel. Nous verrons dans le prochain chapitre l'impact de cette contrainte lorsque nous comparerons les analyses du compromis énergie-temps de réponse.

Chapitre 4

Analyse du compromis énergie-temps de réponse

Dans ce quatrième et dernier chapitre, nous allons revoir les analyses du compromis énergie-temps de réponse proposées par l'auteur de notre article de référence à la lumière, d'une part, de la définition correcte de ρ pour la file WiFi vue au chapitre précédent, et, d'autre part, en simulant la stratégie ininterrompue sans la contrainte imposée par l'auteur pour celle-ci. Pour mettre en évidence le compromis énergie-temps de réponse, nous utilisons une métrique appelée *Energy Response Time Weighted Product (ERWP)* que nous détaillons à la section 4.1. Pour nos analyses comparatives, nous utilisons les mêmes scénarii que ceux choisis par l'auteur. D'une part, nous les appliquons à l'étude théorique des modèles mathématiques pour lesquels nous avons corrigé la définition de ρ de la file WiFi. D'autre part, nous utilisons notre outil de simulation pour simuler ces scénarii et comparer les résultats obtenus empiriquement aux résultats théoriques originaux et corrigés de l'article. Normalement, les résultats empiriques obtenus pour la stratégie interrompue doivent être les mêmes que les résultats théoriques corrigés. Par contre, la stratégie ininterrompue offrira des résultats différents par l'impossibilité de simuler la contrainte imposée par l'auteur pour la file transmission de cette stratégie. La section 4.2 est dédiée à ces comparatifs.

4.1 Détail de la métrique : Energy Response Time Weighted Product (ERWP)

La métrique ERWP permettant de capturer le compromis énergie-temps de réponse s'exprime comme une fonction objective définie comme le produit pondéré de l'énergie moyenne consommée par une opération dans le système et de son temps de réponse moyen. Celui-ci est le temps moyen mis par une opération pour traverser

le système. La métrique s'exprime ainsi :

$$ERWP = E[\mathcal{E}_r]^w \cdot E[\mathcal{T}_r]^{1-w} \tag{4.1}$$

où $E[\mathcal{E}_r]$ et $E[\mathcal{T}_r]$ sont définis tels qu'au chapitre précédent et où w est un paramètre de pondération compris entre 0 et 1 permettant de donner plus de poids, soit à la consommation énergétique, soit au temps de réponse selon la valeur qui lui est affectée. Plus w tend vers 0 et plus on se soucie du temps de réponse. A l'inverse, plus w tend vers 1 et plus on se soucie de la consommation énergétique. Notons que si $w = 0.5$, on considère alors le temps de réponse aussi important que l'énergie consommée et on a comme expression de la métrique :

$$ERWP = \sqrt{E[\mathcal{E}_r] \cdot E[\mathcal{T}_r]} \tag{4.2}$$

Nous avons vu également au chapitre précédent que :

$$E[\mathcal{T}_r] = \frac{1}{\lambda}(E[\mathcal{N}_g] + E[\mathcal{N}_w] + E[\mathcal{N}_c] + E[\mathcal{N}_m]) \tag{4.3}$$

$$E[\mathcal{E}_r] = \frac{1}{\lambda}(E[\mathcal{P}_g] + E[\mathcal{P}_w] + E[\mathcal{P}_m]) \tag{4.4}$$

Par conséquent, nous pouvons exprimer la métrique sous la forme :

$$ERWP = \frac{1}{\lambda} \cdot (E[\mathcal{P}_g] + E[\mathcal{P}_w] + E[P_m])^w \cdot (E[\mathcal{N}_g] + E[\mathcal{N}_w] + E[N_c] + E[N_m])^{1-w} \tag{4.5}$$

Les transformations permettant d'arriver à cette expression dérivent de celles proposées à l'annexe D. Ici encore, pour être précis, il faut substituer les indices g et w par 1 et 2 pour la stratégie ininterrompue.

Étant donné que nous cherchons à minimiser le temps de réponse ainsi que l'énergie consommée, lorsque nous comparons nos stratégies par cette métrique, nous considérons que la plus intéressante est celle qui propose la valeur la plus faible pour une pondération particulière.

4.2 Comparatif des analyses du compromis énergie-temps de réponse

Dans cette section, nous allons comparer les analyses originales du compromis énergie-temps de réponse de notre article de référence aux analyses théoriques corrigées et à celles issues de nos simulations. Nous produirons donc pour chaque cas analysé trois graphiques que nous comparerons entre-eux.

4.2.1 Analyse de l'évolution de l'ERWP en fonction de w pour différents taux d'arrivées

Ce premier cas permet d'analyser l'impact du taux d'arrivée des opérations dans le système. Pour ce faire, on définit trois scénarii identiques pour les deux stratégies où seul le paramètre λ varie d'un scénario à l'autre. Les tableaux 4.1 et 4.2 proposent le récapitulatif des paramètres utilisés.

TABLE 4.1 – Paramètres de simulation des scénarii de la stratégie interrompue

	PP	Dispatcher		File locale		File cellulaire		File WiFi				File cloud
	λ	π_{α_1}	π_{α_2}	μ_m	p_m	μ_g	p_g	μ_w	p_w	η	ξ	μ_c
Scénario 1	0.2	0.5	$f(opt)$	2	2	0.8	2.5	2	0.7	1	1	5
Scénario 2	0.6	0.5	$f(opt)$	2	2	0.8	2.5	2	0.7	1	1	5
Scénario 3	1.0	0.5	$f(opt)$	2	2	0.8	2.5	2	0.7	1	1	5

TABLE 4.2 – Paramètres de simulation des scénarii de la stratégie ininterrompue

	PP	Dispatcher	File locale		File transmission						File cloud
	λ	π_{α_1}	μ_m	p_m	μ_1	p_1	μ_2	p_2	η	ξ	μ_c
Scénario 1	0.2	0.5	2	2	0.8	2.5	2	0.7	1	1	5
Scénario 2	0.6	0.5	2	2	0.8	2.5	2	0.7	1	1	5
Scénario 3	1.0	0.5	2	2	0.8	2.5	2	0.7	1	1	5

Pour ce cas, la proportion d'opérations migrées est fixée à 50% ($\pi_{\alpha_1} = 0.5$) et les autres paramètres correspondent aux valeurs utilisées dans le scénario 1 de la vérification de l'implémentation du chapitre précédent. Les justifications sont donc les mêmes. Il est nécessaire de s'arrêter cependant quelques instants sur le paramètre π_{α_2}. En effet, l'affectation $f(opt)$ employée signifie que la valeur de π_{α_2} sera choisie à chaque calcul de la métrique afin d'optimiser celle-ci, autrement dit de la minimiser. C'est-à-dire que, pour chaque calcul de la métrique, on va tester différentes valeurs de π_{α_2} et ne conserver que celle qui la minimise. De cette manière, on observe l'évolution de la métrique en fonction du paramètre w pour cette stratégie dans les conditions optimales. Dans ce mémoire, le calcul de π_{α_2} est réalisé avec la fonction *optimize(f, interval, tol)* implémentée dans le langage *R*. Les trois arguments que prend cette fonction sont respectivement :

- *f*, la fonction à optimiser. Dans notre cas c'est la métrique ERWP sous sa forme telle qu'à l'équation 4.5. Nous paramétrons cette fonction en utilisant les formules vues au chapitre précédent. En réalité, au lieu de chercher la

valeur de π_{α_2} directement, nous recherchons en fait la valeur théorique de λ_g qui minimise la métrique. En effet, une fois cette valeur obtenue il nous suffit d'utiliser la formule 3.16 afin de calculer π_{α_2} [1].

- *interval*, l'intervalle sur lequel l'optimisation est réalisée. Celui-ci sera toujours compris entre 0 et λ_c puisque π_{α_2} est compris entre 0 et 1. En effet, si $\lambda_g = \lambda_c$, $\pi_{\alpha_2} = 1$ et si $\lambda_g = 0$, $\pi_{\alpha_2} = 0$.
- *tol*, la précisions souhaitée. Nous le paramétrons à 0.000001. Autrement dit, on considère la valeur de λ_g comme étant celle qui minimise l'ERWP si cette valeur est la plus petite valeur calculée à 0.000001 près.

Pour obtenir l'évolution de l'ERWP pour chaque scénario en fonction de w, nous calculons sa valeur sur base des moyennes empiriques $\overline{T_r}$ et $\overline{E_r}$ récoltées par simulation en utilisant la formule 4.1. Pour le calcul théorique, on utilise la formule 4.5 à laquelle on affecte les paramètres des tableaux ci-dessus. Afin d'observer l'évolution de la métrique, nous allons calculer celle-ci pour cent valeurs de w. Ces valeurs vont de 0 à 1 par pas de 0.01. Sur les figures 4.1, 4.2 et 4.3 se trouvent les graphiques représentant l'évolution de la métrique en fonction de w pour les trois scénarii que nous avons détaillés ci-dessus.

Comme nous l'avons dit plus haut, le premier graphique propose les courbes calculées sur base des formules originales de notre article de référence. Nous nommons ce graphique dans la suite *graphique original*. Le deuxième graphique propose les courbes issues des formules pour lesquelles nous avons corrigé la définition de ρ de la file WiFi. Nous nommons ce graphique *graphique corrigé* dans la suite. Enfin, le troisième graphique propose les courbes obtenues par simulation. Nous le nommons *graphique des simulations*.

Comme on pouvait s'y attendre, pour la stratégie interrompue, les courbes du graphique des simulations correspondent à celles du graphique corrigé. On constate également que les courbes de la stratégie ininterrompue du graphique des simulations diffèrent comme prévu des courbes théoriques. Cela s'explique, comme nous l'avons vu au chapitre précédent, par l'impossibilité de simuler la contrainte mathématique qu'impose l'auteur à la file transmission dans les conditions de simulation de cette stratégie. Voyons à présent dans quelle mesure ces différences influent sur l'analyse de ces graphiques.

Prenons tout d'abord le graphique original. On constate que la stratégie ininterrompue est meilleure que la stratégie interrompue pour des valeurs faibles de w. Par contre, lorsque w s'approche de 1, la stratégie interrompue devient de plus en plus intéressante et finit même par surpasser la stratégie ininterrompue pour des valeurs de w supérieures à 0.7 (selon le paramètre λ utilisé). Cela signifie que,

1. Pour les résultats théoriques, on peut se limiter au calcul de λ_g. Par contre, pour nos simulations, nous devons impérativement travailler avec π_{α_2} que nous paramétrons au niveau du dispatcher α_2 car, dans ce cas, λ_g dépend de ce paramètre et non l'inverse.

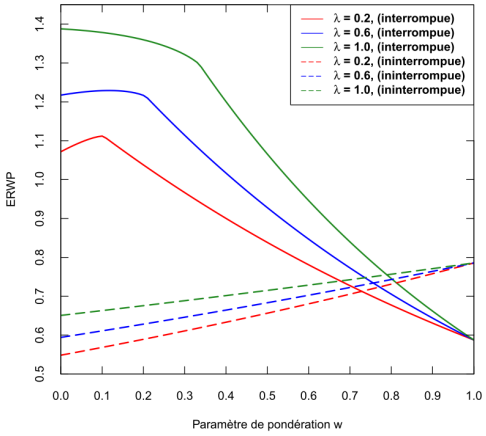


FIGURE 4.1 – Graphique original

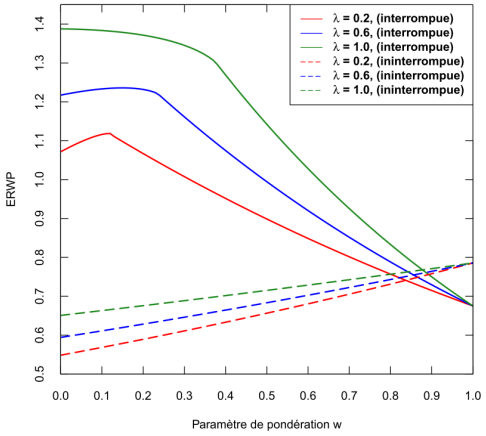


FIGURE 4.2 – Graphique corrigé

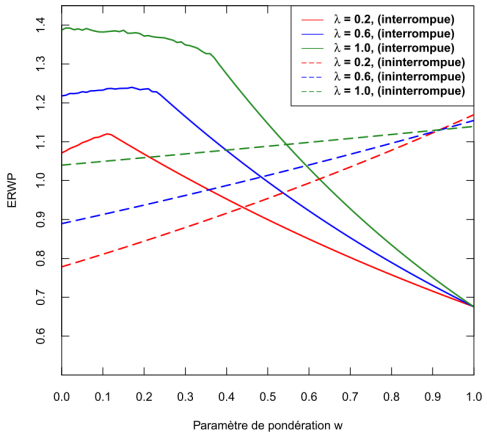


FIGURE 4.3 – Graphique des simulations

si l'on considère l'économie d'énergie plus importante que le temps de réponse, la stratégie interrompue est le meilleur choix pour ce cas. On constate également que pour les deux stratégies, le taux d'arrivée des opérations n'influe pas sur la consommation d'énergie car, comme nous l'avons vu, celle-ci est mesurée par opération et n'est fonction que de la puissance du serveur de la file et de sa vitesse d'exécution. Enfin, on constate que plus le taux d'arrivée augmente et plus la métrique augmente également. Cependant, la différence est moins marquée pour la stratégie ininterrompue que pour la stratégie interrompue.

Pour le graphique corrigé, on constate que les analyses sont les mêmes à ceci

près que le seuil à partir duquel la stratégie interrompue devient meilleure que la stratégie ininterrompue est $w > 0.8$.

Par contre, pour le graphique des simulations, les différences sont plus marquées. Dans un premier temps, on observe que bien que la stratégie ininterrompue continue d'être meilleure pour de faibles valeurs de w, la différence est bien moins marquée, et surtout, lorsqu'on regarde uniquement le temps de réponse, la marge entre les courbes de la stratégie ininterrompue est la même que celle entre les courbes de la stratégie interrompue. Les temps de réponse sont donc maintenant aussi sensibles chez l'un que chez l'autre aux taux d'arrivée des opérations. De plus, le seuil à partir duquel la stratégie interrompue est meilleure que la stratégie ininterrompue est bien plus bas et démarre à partir de $w > 0.4$. On constate enfin que pour la stratégie ininterrompue, contrairement aux deux premiers graphiques, l'énergie consommée cette fois dépend très légèrement du taux d'arrivée. Pour s'en convaincre, il faut observer les valeurs prises par l'ERWP pour cette stratégie lorsque $w = 1$, c'est-à-dire quand celle-ci ne prend en compte que l'énergie consommée. Cette dépendance est certes très faible mais tout de même observable.

4.2.2 Analyse de l'évolution de l'ERWP en fonction de w pour des couvertures WiFi différentes

Pour ce cas, le principe est fondamentalement le même que pour le précédent à ceci près que nous allons fixer le taux d'arrivée des opérations et faire varier cette fois la durée de la couverture WiFi. Les tableaux 4.3 et 4.4 résument les paramètres utilisés.

TABLE 4.3 – Paramètres de simulation des scénarii de la stratégie interrompue

	PP	Dispatcher		File locale		File cellulaire		File WiFi				File cloud
	λ	π_{α_1}	π_{α_2}	μ_m	p_m	μ_g	p_g	μ_w	p_w	η	ξ	μ_c
Scénario 1	0.6	0.5	$f(opt)$	2	2	0.8	2.5	2	0.7	1	1	5
Scénario 2	0.6	0.5	$f(opt)$	2	2	0.8	2.5	2	0.7	5	1	5
Scénario 3	0.6	0.5	$f(opt)$	2	2	0.8	2.5	2	0.7	10	1	5

Sur les figures 4.4, 4.5 et 4.6 se trouvent les graphiques représentant l'évolution de l'ERWP en fonction de w pour les trois scénarii que nous avons détaillés ci-dessus.

On constate, sur le graphique original, que la stratégie interrompue est moins performante que la stratégie ininterrompue lorsque les périodes sans couverture WiFi sont plus longues. On constate également que lorsque $w < 0.85$ la stratégie interrompue est d'autant plus performante que la couverture WiFi est longue. Ceci s'explique par le fait que plus w diminue plus on donne d'importance au temps de

TABLE 4.4 – Paramètres de simulation des scénarii de la stratégie ininterrompue

	PP	Dispatcher	File locale		File transmission						File cloud
	λ	π_{α_1}	μ_m	p_m	μ_1	p_1	μ_2	p_2	η	ξ	μ_c
Scénario 1	0.6	0.5	2	2	0.8	2.5	2	0.7	1	1	5
Scénario 2	0.6	0.5	2	2	0.8	2.5	2	0.7	5	1	5
Scénario 3	0.6	0.5	2	2	0.8	2.5	2	0.7	10	1	5

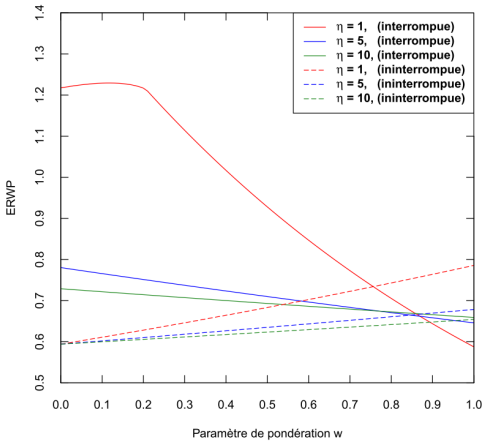


FIGURE 4.4 – Graphique original

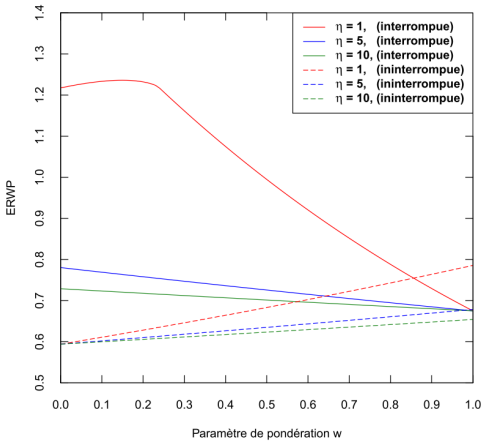


FIGURE 4.5 – Graphique corrigé

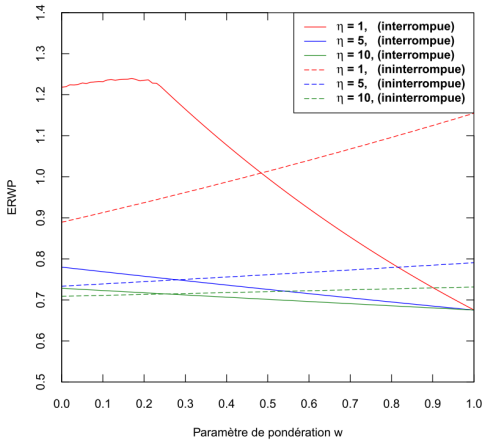


FIGURE 4.6 – Graphique des simulations

réponse. Hors celui-ci est d'autant plus petit que le serveur de la file WiFi est actif et donc traite les opérations qui arrivent dans cette file au lieu de les voir s'accumuler. Enfin, lorsqu'on ne regarde que l'énergie consommée, la stratégie interrompue est moins performante à mesure que la couverture est grande. Cette dernière analyse parait contre-intuitive puisque nous avons vu que l'énergie consommée était dépendante, pour cette stratégie, de la puissance et de la vitesse de traitement du serveur de la file uniquement. Normalement les périodes de disponibilité du WiFi ne devraient pas entrer en ligne de compte.

Si on se concentre maintenant sur le graphique corrigé, on constate qu'effectivement, avec la définition correcte de ρ pour la file WiFi, cette fois-ci, on obtient la même énergie consommée quelle que soit la couverture WiFi. Nous pouvons nous en convaincre en observant la convergence en un point des courbes pour $w = 1$ (on ne regarde alors que l'énergie consommée). Ce qui s'explique par le fait que dans une file $M/M/1_{ON/OFF}$ comme dans une file $M/M/1$, l'énergie consommée au niveau du serveur ne dépend que de la puissance nominale du serveur et de son taux de service (voir section 3.3.4 équations 3.14 et 3.45). Ce comportement dans le graphique original aurait pu mettre la puce à l'oreille de l'auteur de l'article quant à l'incohérence des courbes pour $w = 1$. On constate donc ici que ce n'est pas en dessous de $w = 0.85$ que la stratégie interrompue est plus performante à mesure que la couverture WiFi est plus longue, mais qu'en fait c'est vrai pour toute valeur de w en dessous de 1.

Enfin, le graphique des simulations montre que les courbes de la stratégie interrompue sont bien similaires à celles obtenues théoriquement avec ρ modifié. Pour cette stratégie, les analyses sont donc les mêmes. Par contre, on constate ici, pour la stratégie ininterrompue, que d'une part, si on se concentre sur l'économie d'énergie uniquement, celle-ci est dans tous les cas moins performante que l'autre stratégie, ce qui n'était pas le cas précédemment. D'autre part, on constate également, si on ne s'intéresse cette fois qu'au temps de réponse, que bien qu'elle soit plus performante que la stratégie interrompue, la différence est moindre. De plus, cette fois-ci, contrairement au graphique précédent, les temps de réponse pour des couvertures WiFi différentes varient au lieu de se situer au même point.

4.2.3 Analyse de l'évolution de l'ERWP en fonction de w considérant des valeurs de μ_c différentes

Pour ce cas, le principe est fondamentalement le même que pour les deux analyses précédentes, à ceci près que nous allons faire varier cette fois la vitesse de traitement des opérations par la file cloud. Les tableaux 4.5 et 4.6 résument les paramètres utilisés.

Sur les figures 4.7, 4.8 et 4.9 se trouvent les graphiques représentant l'évolution

TABLE 4.5 – Paramètres de simulation des scénarii de la stratégie interrompue

	PP	Dispatcher		File locale		File cellulaire		File WiFi				File cloud
	λ	π_{α_1}	π_{α_2}	μ_m	p_m	μ_g	p_g	μ_w	p_w	η	ξ	μ_c
Scénario 1	0.6	0.5	$f(opt)$	2	2	0.8	2.5	2	0.7	1	1	2
Scénario 2	0.6	0.5	$f(opt)$	2	2	0.8	2.5	2	0.7	1	1	6
Scénario 3	0.6	0.5	$f(opt)$	2	2	0.8	2.5	2	0.7	1	1	10

TABLE 4.6 – Paramètres de simulation des scénarii de la stratégie ininterrompue

	PP	Dispatcher	File locale		File transmission						File cloud
	λ	π_{α_1}	μ_m	p_m	μ_1	p_1	μ_2	p_2	η	ξ	μ_c
Scénario 1	0.6	0.5	2	2	0.8	2.5	2	0.7	1	1	2
Scénario 2	0.6	0.5	2	2	0.8	2.5	2	0.7	1	1	6
Scénario 3	0.6	0.5	2	2	0.8	2.5	2	0.7	1	1	10

de l'ERWP en fonction de w pour les trois scénarii que nous avons détaillés ci-dessus.

On constate sur le graphique original que plus la file cloud traite les opérations rapidement et plus le temps de réponse est faible. Ceci pour les deux stratégies dans les mêmes proportions. Si on se concentre sur l'économie d'énergie, la vitesse de traitement de la file cloud n'a logiquement aucun impact puisqu'on ne considère pas l'énergie consommée par celle-ci. Les conclusions sont les mêmes pour la version corrigée du graphique ainsi que pour la version obtenue par simulation.

4.2.4 Analyse de l'évolution de l'ERWP en fonction de π_{α_1} pour des valeurs fixes de w

Pour les deux prochains cas, nous allons cette fois observer le comportement de nos stratégies lorsque nous fixons w et que nous faisons varier la valeurs de π_{α_1}. Celle-ci, comme pour w précédemment, variera de 0 à 1 par pas de 0.01. Ces cas permettront d'analyser le comportement de nos stratégies pour des politiques de migration différentes. Nous commençons dans cette section par fixer trois valeurs de w : $w = 0.2$ (priorité au temps de réponse), $w = 0.5$ (même poids pour le temps de réponse que pour la consommation d'énergie), $w = 0.8$ (priorité à la consommation énergétique). Les tableaux 4.7 et 4.8 résument les paramètres utilisés.

Sur les figures 4.10, 4.11 et 4.12 se trouvent les graphiques représentant l'évolution de l'ERWP en fonction de π_{α_1} pour les trois scénarii que nous avons détaillés ci-dessus.

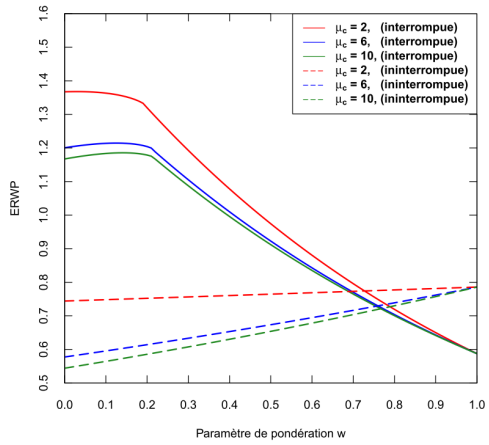


FIGURE 4.7 – Graphique original

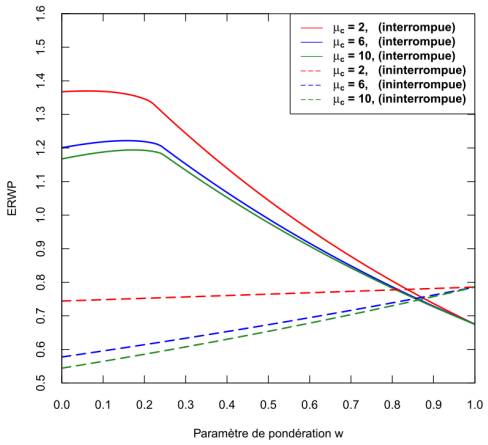


FIGURE 4.8 – Graphique corrigé

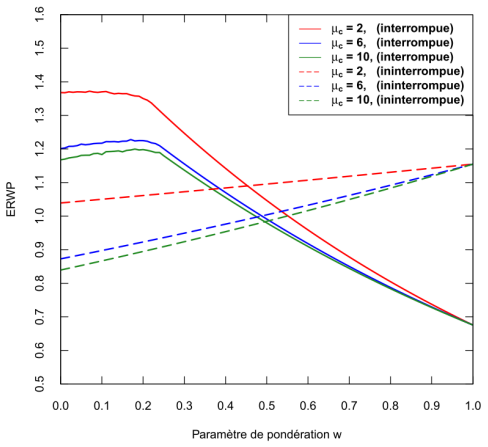


FIGURE 4.9 – Graphique des simulations

TABLE 4.7 – Paramètres de simulation des scénarii de la stratégie interrompue

	PP	Weight	Dispatcher	File locale		File cellulaire		File WiFi				File cloud
	λ	w	π_{α_2}	μ_m	p_m	μ_g	p_g	μ_w	p_w	η	ξ	μ_c
Scénario 1	0.6	0.2	$f(opt)$	2	2	0.8	2.5	2	0.7	1	1	5
Scénario 2	0.6	0.5	$f(opt)$	2	2	0.8	2.5	2	0.7	1	1	5
Scénario 3	0.6	0.8	$f(opt)$	2	2	0.8	2.5	2	0.7	1	1	5

TABLE 4.8 – Paramètres de simulation des scénarii de la stratégie ininterrompue

	PP	Weight	File locale		File transmission						File cloud
	λ	w	μ_m	p_m	μ_1	p_1	μ_2	p_2	η	ξ	μ_c
Scénario 1	0.6	0.2	2	2	0.8	2.5	2	0.7	1	1	5
Scénario 2	0.6	0.5	2	2	0.8	2.5	2	0.7	1	1	5
Scénario 3	0.6	0.8	2	2	0.8	2.5	2	0.7	1	1	5

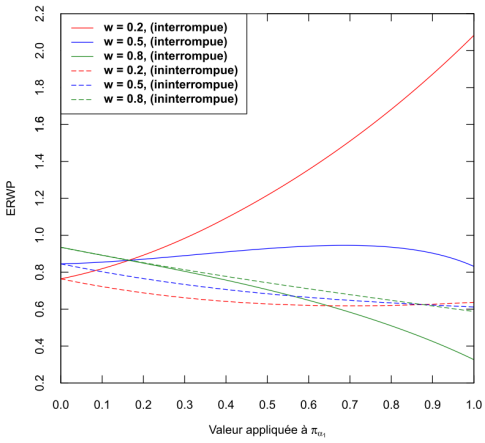


FIGURE 4.10 – Graphique original

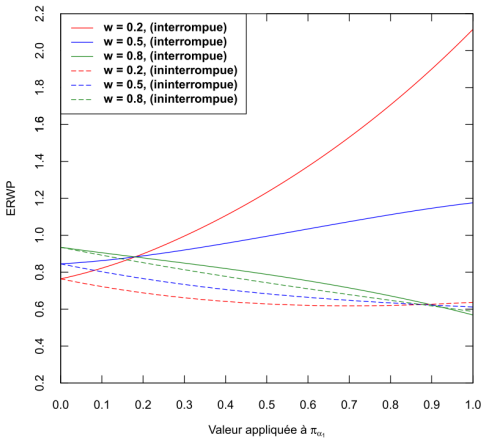


FIGURE 4.11 – Graphique corrigé

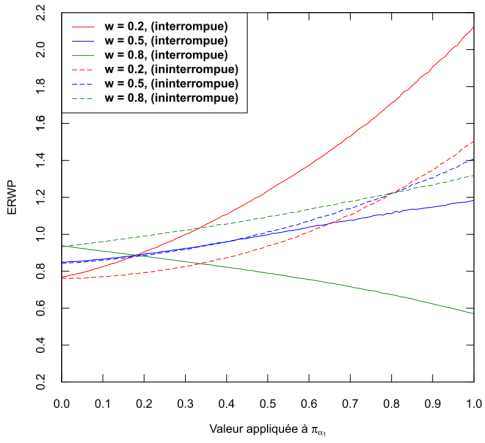


FIGURE 4.12 – Graphique simulations

Pour le graphique original, si $w = 0.2$, il est préférable de ne jamais migrer les opérations pour la stratégie interrompue, alors que pour la stratégie ininterrompue, il vaut mieux migrer la moitié des opérations. Cependant, à mesure que w se rapproche de 1, il est préférable pour les deux stratégies de migrer un maximum d'opérations sachant que la stratégie interrompue fournit de meilleurs résultats que l'ininterrompue.

Si on regarde le graphique corrigé, les conclusions sont les mêmes à ceci près que pour des valeurs de w proches de 1, on constate ici que les deux stratégies ont des performances similaires.

Les choses sont bien différentes sur le graphique des simulations. On constate dans ce cas que, pour la stratégie ininterrompue, il est toujours préférable de ne pas migrer d'opérations. En effet, quelle que soit la valeur du paramètre w, plus on migre d'opérations et plus les valeurs de la métrique augmentent. On constate également que si toutes les opérations sont migrées, on obtient des valeurs plus faibles pour la métrique à mesure que w augmente, autrement dit, à mesure que la consommation énergétique prend de l'importance. Les résultats obtenus pour la stratégie interrompue correspondent comme prévu aux résultats théoriques corrigés.

4.2.5 Analyse de l'évolution de l'ERWP en fonction de π_{α_1} pour différentes valeurs de μ_m

Pour ce dernier cas, nous observons l'évolution de l'ERWP en fonction de π_{α_1} pour différentes vitesses de traitement de la file locale. Nous fixons cette fois-ci la valeur de w à 0.5. Nous considérons dans ce cas la consommation énergétique aussi importante que le temps de réponse. Les tableaux 4.9 et 4.10 résument les paramètres utilisés.

TABLE 4.9 – Paramètres de simulation des scénarii de la stratégie interrompue

	PP	Weight	Dispatcher	File locale		File cellulaire		File WiFi				File cloud
	λ	w	π_{α_2}	μ_m	p_m	μ_g	p_g	μ_w	p_w	η	ξ	μ_c
Scénario 1	0.6	0.5	$f(opt)$	1	2	0.8	2.5	2	0.7	1	1	5
Scénario 2	0.6	0.5	$f(opt)$	2	2	0.8	2.5	2	0.7	1	1	5
Scénario 3	0.6	0.5	$f(opt)$	3	2	0.8	2.5	2	0.7	1	1	5

Sur les figures 4.13, 4.14 et 4.15 se trouvent les graphiques représentant l'évolution de l'ERWP en fonction de π_{α_1} pour les trois scénarii que nous avons détaillés ci-dessus.

Sur le graphique original, on observe que la stratégie ininterrompue est toujours préférée à la stratégie interrompue, quel que soit le taux de migration des

TABLE 4.10 – Paramètres de simulation des scénarii de la stratégie ininterrompue

	PP	Weight	File locale		File transmission						File cloud
	λ	w	μ_m	p_m	μ_1	p_1	μ_2	p_2	η	ξ	μ_c
Scénario 1	0.6	0.5	1	2	0.8	2.5	2	0.7	1	1	5
Scénario 2	0.6	0.5	2	2	0.8	2.5	2	0.7	1	1	5
Scénario 3	0.6	0.5	3	2	0.8	2.5	2	0.7	1	1	5

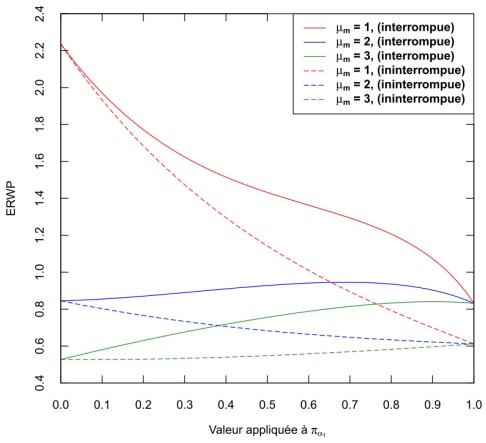


FIGURE 4.13 – Graphique original

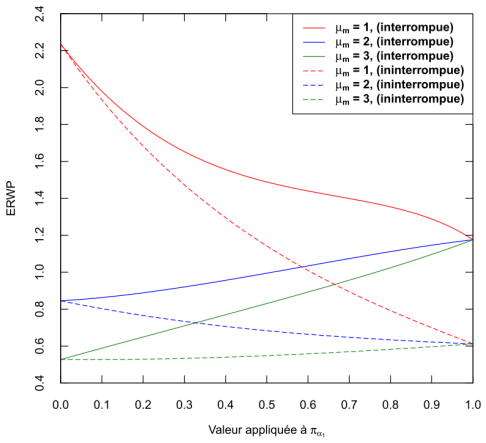


FIGURE 4.14 – Graphique corrigé

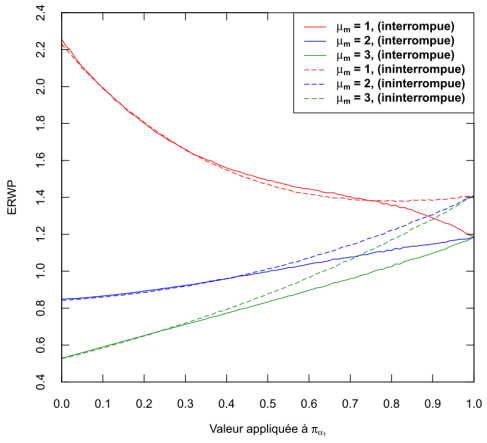


FIGURE 4.15 – Graphique des simulations

opérations. On constate également que plus la vitesse de traitement des opérations par la file locale est faible et plus il est intéressant de migrer les opérations. Cet intérêt diminue à mesure que la vitesse de traitement de cette file augmente. A tel point que pour $\mu_m = 3$, quelle que soit la stratégie, il est toujours plus intéressant de traiter les opérations localement.

Sur le graphique corrigé, on constate cette fois qu'il est toujours plus intéressant de traiter les opérations localement dès que $\mu_m \geq 2$ pour la stratégie interrompue. En effet, plus on migre d'opérations (plus π_{α_1} augmente) et plus la métrique augmente également.

Sur le graphique obtenu par simulation, on constate cette fois que, pour les deux stratégies, dès que $\mu_m \geq 2$, il n'est plus intéressant de migrer d'opérations. On constate également, contrairement au deux précédents graphiques, qu'au fur et à mesure que le taux d'opérations migrées augmente les deux stratégies ont des performances sensiblement équivalentes jusqu'à $\pi_{\alpha_1} = 0.4$ pour $\mu_m = 3$, $\pi_{\alpha_1} = 0.5$ pour $\mu_m = 2$ et enfin $\pi_{\alpha_1} = 0.7$ pour $\mu_m = 1$. Si on compare ce graphique au même graphique de la section précédente, on constate cette fois que, pour $w = 0.5$ si $\mu_m = 1$, il est maintenant plus intéressant en toute circonstance de migrer les opérations vers la file cloud. On peut donc considérer, dans ce cas, que la migration d'opérations fait sens lorsque $1 \geq \mu_m < 2$.

4.3 Conclusion

Dans ce chapitre, nous avons révisé les analyses du compromis énergie-temps de réponse proposées dans notre article de référence, d'une part, en corrigeant la définition de ρ pour la file WiFi de la stratégie interrompue, et, d'autre part, en simulant la stratégie ininterrompue sans la contrainte imposée dans cet article.

Tout d'abord, comme nous l'avions suggéré, pour la stratégie interrompue, les résultats obtenus théoriquement avec la définition correcte de ρ correspondent aux résultats obtenus par simulation.

Ensuite, d'une manière générale, lorsqu'on s'intéresse plus au temps de réponse qu'à l'économie d'énergie, dans tous les cas, la stratégie ininterrompue est plus intéressante que la stratégie interrompue. Cependant, le seuil à partir duquel cela n'est plus vrai est bien plus bas lorsque nous analysons par simulation que lors de l'analyse théorique.

Enfin, nous avons constaté que les différences apparaissant entre l'analyse théorique de la stratégie ininterrompue et l'analyse par simulation sont en général conséquentes, notamment lorsqu'il s'agit de déterminer le seuil à partir duquel la migration d'opérations devient plus intéressante que le traitement local. Dans ce cas, nous avons révélé qu'il fallait que la vitesse de traitement de la file locale soit deux fois moins grande que ce qui était requis par l'analyse théorique. Une telle

différence pose la question de l'intérêt réel d'imposer pour les analyses théoriques de la file transmission une contrainte qui, bien que facilitant les calculs des mesures de performance de celle-ci, ne correspond pas à une réalité physique. La qualité des analyses réalisées par la suite s'en trouve dégradée.

Conclusion générale

Dans ce mémoire, nous avons réalisé la simulation d'un système mobile avec possibilité de migration d'opérations vers un cloud afin d'analyser le compromis énergie-temps de réponse inhérent au système. Ce compromis vise à minimiser le temps de réponse du système et l'énergie qu'il consomme. Pour simuler le système, nous avons développé, en *Java*, un outil de simulation dont les spécifications proviennent d'un modèle mathématique de ce système prenant appui sur la théorie des files d'attente et réalisé par les auteurs de l'article *Analysis of the Energy-Response Time Tradeoff for Mobile Cloud Offloading Using Combined Metrics* (Wu *et al.*, 2015).

Partant de cet article, nous avons, dans un premier temps, décrit complètement le système mobile avec migration et mis en évidence les raisons de l'existence du compromis énergie-temps de réponse. Nous avons également décrit deux stratégies de migration pour lesquelles nous avons analysé ce compromis afin de déterminer les conditions dans lesquelles privilégier l'une à l'autre.

Nous avons, dans un second temps, analysé le modèle mathématique proposé dans l'article afin de l'implémenter sous la forme de l'outil de simulation dont nous avons parlé. Durant cette phase d'analyse, nous avons constaté deux problèmes de modélisation dans cet article.

Le premier problème de modélisation rencontré est une erreur dans la définition de ρ pour la file WiFi de la stratégie interrompue. Nous avons été capable non seulement de détecter cette erreur mais aussi de la corriger. Nous avons entrepris ensuite de comparer les analyses du compromis énergie-temps de réponse de la stratégie interrompue, d'une part, sans la correction de l'erreur, et, d'autre part, après sa correction, ceci afin d'ajuster les analyses originales proposées dans l'article. Nous avons, à cette occasion, mis en lumière qu'une partie des analyses réalisées par l'auteur auraient pu lui faire entrevoir la possibilité qu'une erreur se soit glissée dans sa modélisation.

Le second problème de modélisation est l'application, par l'auteur, d'une contrainte sur la vitesse d'arrivée dans la file transmission, ceci afin de faciliter l'analyse de cette file. Nous avons constaté que cette contrainte n'était pas simulable. En effet, lorsque nous simulons la stratégie ininterrompue à laquelle appartient cette

file, nous n'avons pas la possibilité de modifier les conditions de simulation pour appliquer cette contrainte, ces conditions étant dépendantes des entités précédent la file. Il est d'ailleurs remarquable que dans un système mobile avec migration, il ne soit pas non plus possible de maitriser cette contrainte pour les mêmes raisons.

Nous avons, ensuite, dans un troisième temps, procédé au développement de l'outil de simulation conformément aux spécifications du modèle mathématique pour lequel nous avons corrigé les erreurs de modélisation que nous venons d'exposer. Nous avons implémenté les entités du modèle sous la forme d'une hiérarchie de classe décrite au chapitre 3. Nous avons pris appuis pour certains aspects de l'outil sur la bibliothèque SSJ spécialisée dans la simulation par évènements discrets (principalement pour l'implémentation du simulateur de l'outil et de ses générateurs de variables aléatoires). Nous avons ensuite vérifié l'outil de simulation. Pour ce faire, nous avons, tout d'abord, construit deux réseaux de files à l'aide de l'outil, ceci afin de vérifier l'ensemble des modules implémentés. Nous avons, ensuite, défini plusieurs scénarii de simulation permettant de vérifier l'ensemble des spécifications de l'outil. Enfin, nous avons simulé ces scénarii et extrait de ces simulations différentes mesures de performance sur les différents modules. Nous avons comparé les mesures de performance obtenues empiriquement à l'espérance de celle-ci dans les mêmes conditions.

Une fois l'outil vérifié, nous avons, dans un quatrième temps, procédé à la simulation des scénarii utilisés dans notre article de référence pour l'analyse du compromis énergie-temps de réponse afin de comparer les résultats obtenus théoriquement par l'auteur aux résultats obtenus par simulation. Ces comparaisons avaient pour but principalement d'ajuster les analyses théoriques de l'auteur pour la stratégie ininterrompue sans application de la contrainte sur la file transmission. La simulation de la stratégie interrompue donne comme prévu les mêmes résultats que ceux attendus théoriquement une fois la définition de ρ corrigée.

Dans ce mémoire, tous les phénomènes aléatoires sont issus de variables aléatoires distribuées comme des exponentielles. Du point de vue théorique, ceci fait sens au vu de la complexité analytique qu'impliquerait l'emploi d'autres distributions. Par contre, par simulation, il est très aisé de changer ces distributions afin d'observer l'impact que celle-ci pourraient avoir sur les analyses réalisées. Nous ne l'avons pas fait nous même, mais dans une perspective d'avenir, il serait intéressant de se pencher sur cette possibilité. L'outil que nous avons développé est, en l'état, conçu de telle manière que les distributions des variables aléatoires des différents modules implémentés sont paramétrables. Toutes les distributions disponibles dans la bibliothèque SSJ sont utilisables.

Avant de terminer, nous souhaitons oser une critique personnelle sur notre article de référence et plus précisément sur son processus d'écriture et de révision. Durant cette année, nous avons lu et relu cet article et il nous a fallu plusieurs

mois de recherche avant d'être assuré qu'une erreur de modélisation avait échappée aux auteurs de celui-ci, le doute étant toujours tourné, en premier lieu, sur notre analyse et non sur les analyses proposées dans l'article. Dans ces recherches, nous avons constaté que la majeure partie de cet article était déjà traité dans d'autres article du même auteur, à tel point qu'on peut dire que plus de 50% du présent article est de la répétition parfois mot pour mot. Après avoir participé à de multiple discussions avec des académiques sensibilisés à cette question, il apparait que cette pratique de la multiplication des publications sur des sujets très proches, ce qui semble être le cas ici, est monnaie courante (sans mauvais jeu de mots) car le système de financement tend à récompenser la quantité. La qualité de la recherche, dans ce cas, ne risque-t-elle pas de se dégrader ? Nous savons que ces articles sont relus par des personnes compétentes dans le domaine ; chaque lecteur devant approuvé l'article avant que celui-ci fasse l'objet d'une publication dans une revue scientifique. Il nous semble que la qualité de la lecture d'un article n'est pas nécessairement à mettre à la charge du lecteur mais peut-être à la charge de la quantité sans doute croissante d'articles à relire. Il va s'en dire dans ce cas que l'attention et la vigilance vont décroissant avec l'augmentation du nombre de lecture à réaliser, surtout si d'un article à l'autre une partie non négligeable est de la répétition. N'est-il pas dès lors plus efficace de privilégier et de récompenser un travail plus conséquent faisant l'objet d'une longue période de réflexion et de remise en questions ?

Bibliographie

Balasubramanian, N., Balasubramanian, A. et Venkataramani, A. (2009). Energy consumption in mobile phones : a measurement study and implications for network applications. *In Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, pages 280–293.

Bass, R. F. (2011). *Stochastic Processes*. Cambridge university press, New York, United States of America.

Cavazza, F. (2014). Nous vivons maintenant dans un monde mobile. `http://www.fredcavazza.net/2014/04/30/vivons-maintenant-monde-mobile/`. [Dernière consultation le 8 mars 2016].

Cuervo, E., Balasubramanian, A., Cho, D.-k., Wolman, A., Saroiu, S., Chandra, R. et Bahl, P. (2010). Maui : making smartphones last longer with code offload. *In Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 49–62.

Ford, A., Raiciu, C., Handley, M. et Bonaventure, O. (2013). TCP Extensions for Multipath Operation with Multiple Addresses. RFC 6824 (Experimental).

IBPT (2016). Cartes de couverture : réseaux mobiles. `http://bipt.be/fr/consommateurs/internet/qualite-de-service/cartes-de-couverture-reseaux-mobiles`. [Dernière consultation le 4 mai 2016].

Kumar, K., Liu, J., Lu, Y.-H. et Bhargava, B. (2013). A survey of computation offloading for mobile systems. *Mobile Networks and Applications*, 18(1):129–140.

L'Ecuyer, P., Meliani, L. et Vaucher, J. (2002). SSJ : A framework for stochastic simulation in Java. *In Proceedings of the 2002 Winter Simulation Conference*, pages 234–242. Available at `http://simul.iro.umontreal.ca/ssj/indexe.html`.

Lee, K., Lee, J., Yi, Y., Rhee, I. et Chong, S. (2013). Mobile data offloading : How much can wifi deliver ? *IEEE/ACM Transactions on Networking (TON)*, 21(2):536–550.

Leemis, L. M. et Park, S. K. (2006). *Discrete-event simulation : A first course.* Pearson Prentice Hall Upper Saddle River, Upper Saddle River, New Jersey.

Postel, J. (1981). Transmission Control Protocol. RFC 793 (INTERNET STANDARD). Updated by RFCs 1122, 3168, 6093, 6528.

Shu, P., Liu, F., Jin, H., Chen, M., Wen, F., Qu, Y. et Li, B. (2013). etime : energy-efficient transmission between cloud and mobile devices. *In INFOCOM, 2013 Proceedings IEEE*, pages 195–199.

Stewart, R., Xie, Q., Morneault, K., Sharp, C., Schwarzbauer, H., Taylor, T., Rytina, I., Kalla, M., Zhang, L. et Paxson, V. (2000). Stream Control Transmission Protocol. RFC 2960 (Proposed Standard). Obsoleted by RFC 4960, updated by RFC 3309.

Wu, H., Knottenbelt, W. et Wolter, K. (2015). Analysis of the energy-response time tradeoff for mobile cloud offloading using combined metrics. *In Teletraffic Congress (ITC 27), 2015 27th International*, pages 134–142.

Wu, H., Wang, Q. et Wolter, K. (2013). Tradeoff between performance improvement and energy saving in mobile cloud offloading systems. *In 2013 IEEE International Conference on Communications Workshops (ICC)*, pages 728–732.

Wu, H. et Wolter, K. (2014). Tradeoff analysis for mobile cloud offloading based on an additive energy-performance metric. *In Proceedings of the 8th International Conference on Performance Evaluation Methodologies and Tools*, pages 90–97.

Annexe A

Illustration du fonctionnement du simulateur

Dans cette annexe, nous illustrons le principe de fonctionnement du simulateur en simulant un réseau de files d'attente dont les caractéristiques sont proches de la stratégie interrompue. Cependant, nous simplifions les spécifications des différents modules en travaillant notamment de manière déterministe et non aléatoire. Nous considérons que le dispatcher α_1 transmette une opération sur deux à la file locale et l'autre à la file cellulaire, et ce, en commençant par la file locale et nous n'utilisons qu'une seule file, la file cellulaire, pour la transmission des opérations vers la file cloud. Ce réseau de files d'attente est visible à la figure A.1. Nous stoppons la simulation une fois les quatre états suivants observés :

1. l'arrivée d'une opération dans une file vide,
2. l'arrivée d'une opération dans une file dont le serveur est en cours d'utilisation,
3. le transfert d'une opération de la file cellulaire à la file cloud,
4. la sortie d'une opération du système.

A l'origine, toutes les files sont vides et la ligne du temps est vierge. Le temps présent $T_p = 0$. Le premier évènement (A_1), correspondant à l'arrivée dans le système de la première opération (O_1), est créé par le système et transmis au simulateur afin d'enclencher le processus de renouvellement. Cet évènement est calculé à un intervalle $I_{A_1} = 1ut$ à partir de T_p. Le simulateur panifie cette arrivée au moment $T_p + I_{A_1} = 1$. L'état du système à ce moment est proposé à la figure A.1.

On démarre alors la simulation, ce qui a pour effet de passer la main au simulateur qui va récupérer le premier évènement disponible. Comme nous l'avons vu, le simulateur va, d'une part, ajuster le temps présent au moment d'exécution de cet évènement et, d'autre part, exécuter l'action qui lui est liée. Dans notre cas,

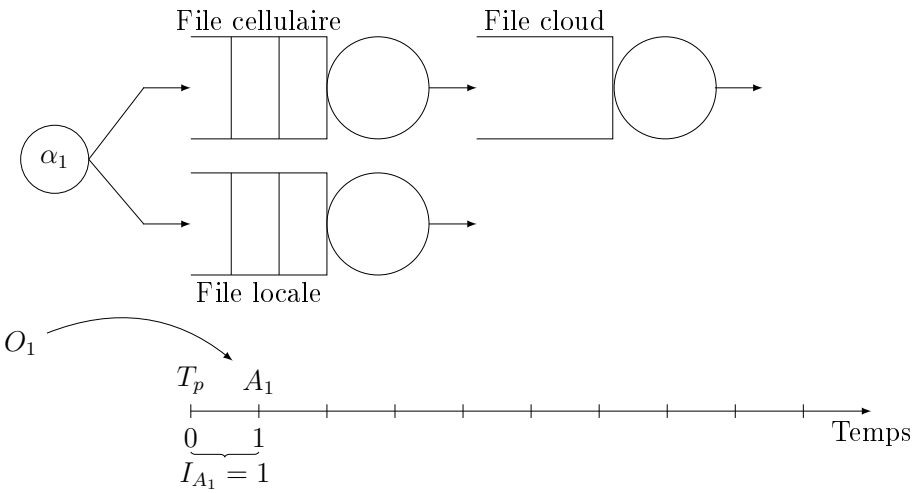


FIGURE A.1 – État du système à l'initialisation, la première arrivée A_1 est planifiée

le simulateur ajuste le temps présent à 1 et exécute ensuite l'action associée à A_1. Cette action, commune à tout évènement d'arrivée, procède en deux étapes :

1. générer l'arrivée suivante,
2. transmettre l'opération au point d'entrée du système.

La génération de la prochaine arrivée permet le renouvellement de l'arrivée des opérations dans le système. En l'occurrence, l'évènement d'arrivée (A_2) de l'opération (O_2) est calculée à un intervalle $I_{A_2} = 1ut$. Il est transmis au simulateur qui le planifie au moment $T_p + I_{A_2} = 2$. La seconde étape est la transmission au point d'entrée du système (le dispatcher α_1) de l'opération O_1. Le dispatcher α_1 choisit instantanément la file locale pour l'exécution de cette opération. Pour rappel, afin de simplifier le système, nous avons décidé que le dispatcher transmet les opérations à la file cellulaire et à la file locale alternativement en commençant par la file locale. La première opération (O_1) arrive dans la file locale au moment 1. Le serveur de la file étant vide, elle le rejoint immédiatement afin de démarrer son exécution. Celle-ci est calculée comme un intervalle $I_{D_1} = 2.5ut$ à partir de T_p. Le serveur transmet un évènement de départ D_1 au simulateur pour qu'il le planifie selon cet intervalle. Pour ce faire, celui-ci effectue $T_p + I_{D_1} = 3.5$ et planifie le départ de O_1 de la file locale au moment 3.5. L'état du système après l'exécution de l'évènement A_1 est visible à la figureA.2.

L'action liée à A_1 étant terminée, la main repasse au simulateur qui récupère l'évènement suivant (A_2) prévu au moment 2. Le simulateur ajuste T_p à 2 et exécute l'action liée à cet évènement. Premièrement, cette action génère A_3, l'arrivée de l'opération O_3 à un intervalle $I_{A_3} = 0.5ut$. A_3 est transmis au simulateur qui le planifie au moment $T_p + I_{A_3} = 2.5$. Deuxièmement, O_2, l'opération qui vient

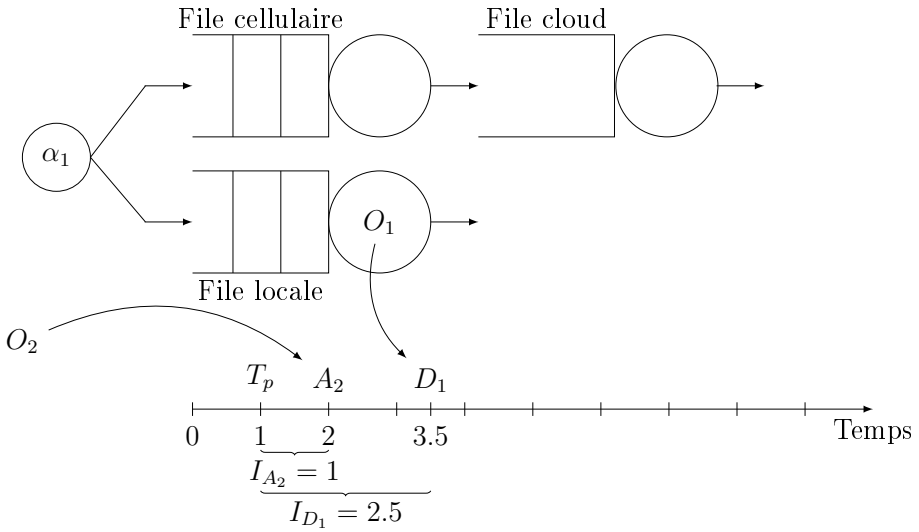


FIGURE A.2 – État du système après l'exécution de A_1, le premier évènement d'arrivée.

d'arriver dans le système est transmis au dispatcher α_1 qui l'oriente cette fois-ci vers la file cellulaire. La file étant vide, l'opération entre immédiatement dans le serveur et se voit attribuer un temps d'exécution de $I_{D_2} = 1ut$. L'évènement de départ (D_2), qui est lié à cette exécution, est transmis au simulateur qui le planifie au moment $T_p + I_{D_2} = 3$. L'état du système à ce moment-là est proposé à la figure A.3.

Après l'exécution de A_2, la main repasse au simulateur qui récupère A_3, le prochain évènement. Le simulateur ajuste tout d'abord T_p à 2.5. Comme pour les deux évènements traités précédemment, l'action associée génère tout d'abord l'évènement d'arrivée suivant A_4 de l'opération O_4 à un intervalle de $I_{A_4} = 2.5ut$. Cet évènement est transmis au simulateur qui le planifie au moment $T_p + I_{A_4} = 5$. Ensuite, l'opération O_3 est transmise au dispatcher qui l'oriente cette fois vers la file locale. Le serveur étant en activité, O_3 se met en attente dans la file. L'état du système à ce moment est exposé à la figure A.4.

Après l'exécution de A_3, la main repasse au simulateur qui récupère cette fois un évènement de départ D_2 et ajuste T_p à 3. L'action liée à un évènement de départ procède en deux étapes :

- transmettre l'opération au composant suivant ou sortir celle-ci du réseau de files d'attente,
- démarrer l'exécution de l'opération suivante s'il en existe une.

La première étape sort l'opération de la file et vérifie si elle doit être transmise ou non à un autre composant. Dans le cas présent, l'opération provient de la file

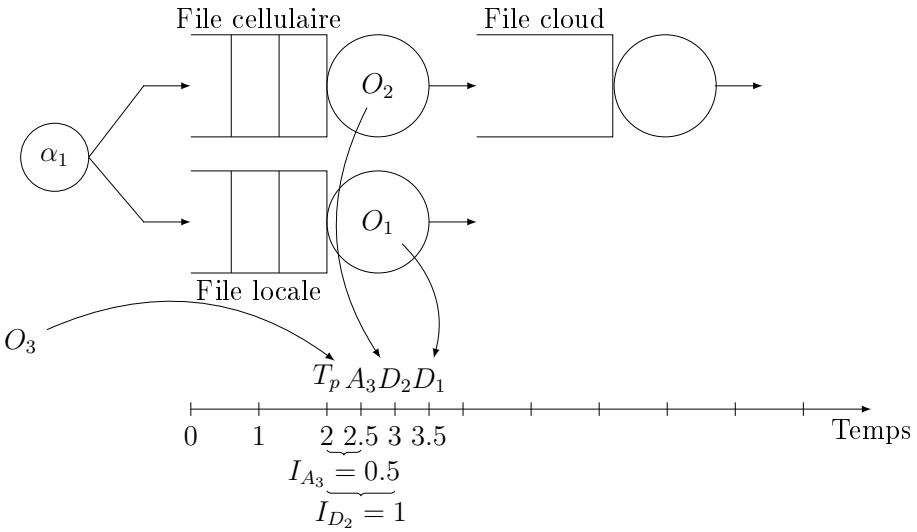


FIGURE A.3 – État du système après l'exécution de A_2, le deuxième évènement d'arrivée.

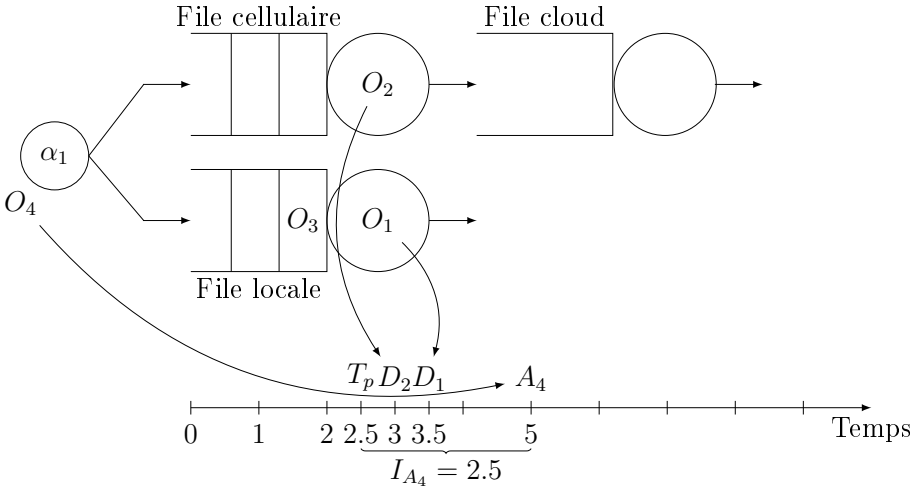


FIGURE A.4 – État du système après l'exécution de A_3, le troisième évènement d'arrivée.

cellulaire, elle est donc transmise à la file cloud. Le serveur de celle-ci étant vide, O_2 démarre immédiatement son exécution qui est calculée comme $I_{D_3} = 3ut$. Un évènement de départ D_3 du serveur est généré et transmis au simulateur qui le planifie au moment $T_p + I_{D_3} = 6$. Ensuite, on vérifie s'il n'y a pas une opération en attente dans la file cellulaire afin de démarrer son service. Ce n'est pas le cas ici. Dès lors, l'action se termine et la main est rendue au simulateur pour l'exécution

de l'évènement suivant. L'état du système à ce moment est proposé à la figure A.5.

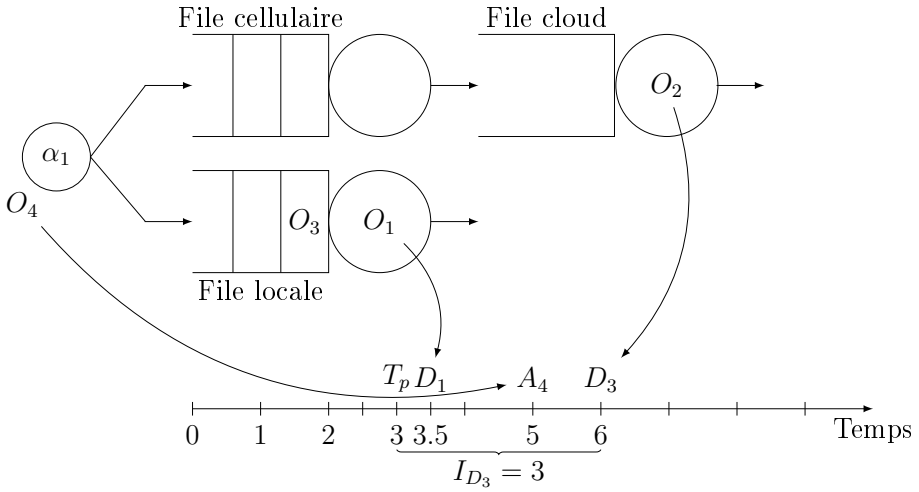


FIGURE A.5 – État du système après l'exécution de D_2, le premier évènement de départ.

L'évènement suivant est le départ de l'opération O_1 du serveur de la file locale. Le simulateur ajuste T_p à 3.5 et exécute l'action associée à un départ. Dans un premier temps, cette action sort l'opération du système car il n'y a pas d'autres composants liés à cette file. Le temps que l'opération est restée dans le système est alors calculé et conservé pour un éventuel traitement statistique ultérieur. Pour calculer le temps passé par cette opération dans le système, on soustrait du moment de sa sortie son moment d'arrivée. O_1 est arrivé au moment 1 et est sorti au moment $3ut$. Le temps (T_{O_1}) que l'opération a mis pour traverser le système est donc de $2ut$. Dans un second temps, on vérifie s'il n'y a pas une opération en attente dans la file afin de démarrer son service. Ici, O_3 est en attente, elle rejoint donc le serveur qui lui calcule un temps d'exécution de $I_{D_4} = 1ut$ et génère un évènement de départ transmis au simulateur qui le planifie à $D_4 = T_p + I_{D_4} = 4$. L'état du système à ce moment est visible à la figure A.6.

A ce stade, nous avons pu observer les quatre états requis. Nous stoppons la simulation.

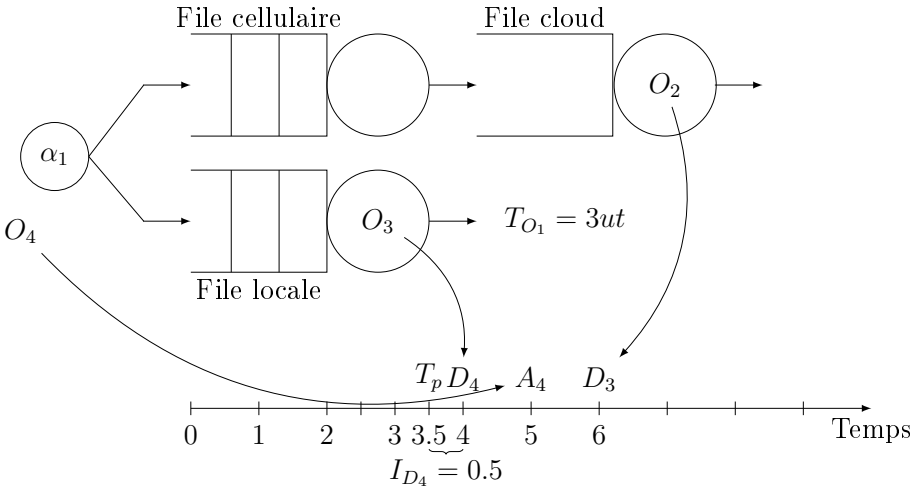


FIGURE A.6 – État du système après l'exécution de D_1, le deuxième évènement de départ.

Annexe B

Diagramme de classe de l'outil de simulation

Dans cette annexe, nous proposons le diagramme de classe de l'outil de simulation. Durant l'écriture du mémoire, nous avons préféré proposer les parties significatives de celui-ci lorsque cela était nécessaire, ceci afin d'éviter de surcharger la lecture de détails accessoires. Cette version complète contient toutes les classes utilisées pour implémenter l'outil. De ce fait, sur celui-ci, sont visibles également les classes propres à la *bibliothèque SSJ* qui sont directement utilisées par les classes de notre outil. On retrouve notamment sur celui-ci l'ensemble des classes permettant l'utilisation des générateurs de variables aléatoires.

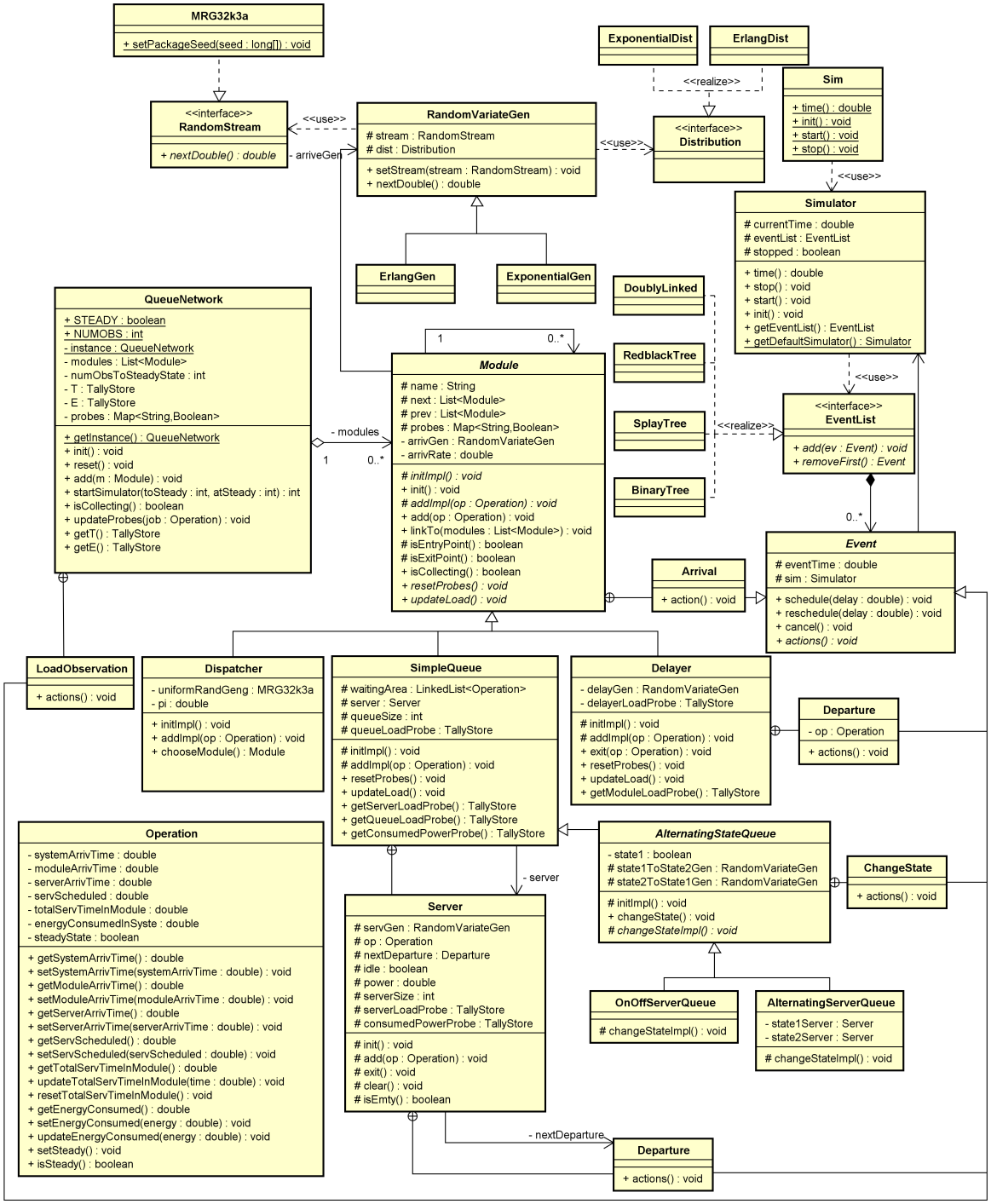


FIGURE B.1 – Diagramme de classe de l'outil de simulation

Annexe C

Détail des calculs d'espérance pour la vérification de l'implémentation

Dans cette annexe, nous proposons le détail des calculs d'espérance réalisés pour la vérification de l'implémentation de la section 3.3.6. Nous rappelons dans un premier temps les paramètres utilisés (tableau C.1 et C.2).

TABLE C.1 – Paramètres de simulation des scénarii de la stratégie interrompue

	PP	Dispatcher		File locale		File cellulaire		File WiFi				File cloud
	λ	π_{α_1}	π_{α_2}	μ_m	p_m	μ_g	p_g	μ_w	p_w	η	ξ	μ_c
Scénario 1	0.6	0	0.5	2	2	0.8	2.5	2	0.7	1	1	5
Scénario 2	0.6	1	0.5	2	2	0.8	2.5	2	0.7	1	1	5
Scénario 3	0.6	0.5	0.5	2	2	0.8	2.5	2	0.7	1	1	5
Scénario 4	0.6	0.5	0.5	2	2	0.8	2.5	2	0.7	2	5	5

TABLE C.2 – Paramètres de simulation de la file transmission

	File transmission							
	λ_1	μ_1	p_1	λ_2	μ_2	p_2	η	ξ
Scénario 1	0,085714	0.8	2.5	0,214286	2	0.7	1	1
Scénario 2	0,085714	0.8	2.5	0,214286	2	0.7	2	5

C.1 File locale

C.1.1 Scénario 1

$$\begin{aligned}\lambda_m &= \lambda \cdot (1 - \pi_{\alpha_1}) \\ &= 0.6 \cdot (1 - 0) \\ &\boxed{= 0.6}\end{aligned} \qquad \begin{aligned}\rho_m &= \frac{\lambda_m}{\mu_m} \\ &= \frac{0.6}{2} \\ &\boxed{= 0.3}\end{aligned} \qquad \begin{aligned}E[\mathcal{P}_m] &= p_m \cdot \rho_m \\ &= 2 \cdot 0.3 \\ &\boxed{= 0.6}\end{aligned}$$

$$\begin{aligned}E[\mathcal{N}_m] &= \frac{\rho_m}{1 - \rho_m} \\ &= \frac{0.3}{1 - 0.3} \\ &= \frac{0.3}{0.7} \\ &\boxed{= 0.428571}\end{aligned} \qquad \begin{aligned}E[\mathcal{T}_m] &= \frac{1}{\lambda_m} \cdot E[\mathcal{N}_m] \\ &= \frac{1}{0.6} \cdot 0.428571 \\ &\boxed{= 0.714286}\end{aligned} \qquad \begin{aligned}E[\mathcal{E}_m] &= \frac{p_m}{\mu_m} \\ &= \frac{2}{2} \\ &= \boxed{1}\end{aligned}$$

C.1.2 Scénario 2

$$\begin{aligned}\lambda_m &= \lambda \cdot (1 - \pi_{\alpha_1}) \\ &= 0.6 \cdot (1 - 1) \\ &\boxed{= 0}\end{aligned}$$

Si $\lambda_m = 0$ alors aucune opération ne passe par cette file. Toutes les espérances sont mises à 0 par défaut.

C.1.3 Scénario 3

$$\begin{aligned}\lambda_m &= \lambda \cdot (1 - \pi_{\alpha_1}) \\ &= 0.6 \cdot (1 - 0.5) \\ &\boxed{= 0.3}\end{aligned} \qquad \begin{aligned}\rho_m &= \frac{\lambda_m}{\mu_m} \\ &= \frac{0.3}{2} \\ &\boxed{= 0.15}\end{aligned} \qquad \begin{aligned}E[\mathcal{P}_m] &= p_m \cdot \rho_m \\ &= 2 \cdot 0.15 \\ &\boxed{= 0.3}\end{aligned}$$

$$E[\mathcal{N}_m] = \frac{\rho_m}{1-\rho_m} \qquad E[\mathcal{T}_m] = \frac{1}{\lambda_m} \cdot E[\mathcal{N}_m] \qquad E[\mathcal{E}_m] = \frac{p_m}{\mu_m}$$
$$= \frac{0.15}{1-0.15} \qquad = \frac{1}{0.3} \cdot 0.176470 \qquad = \frac{2}{2}$$
$$= \frac{0.15}{0.85} \qquad \boxed{= 0.588235} \qquad = \boxed{1}$$
$$\boxed{= 0.176471}$$

C.1.4 Scénario 4

Mêmes résultats qu'au scénario 3.

C.2 File cellulaire

C.2.1 Scénario 1

$$\lambda_c = \lambda \cdot \pi_{\alpha_1}$$
$$= 0.6 \cdot 0$$
$$\boxed{= 0}$$

Si $\lambda_c = 0$ alors aucune opération ne passe par cette file. Toutes les espérances sont mises à 0 par défaut.

C.2.2 Scénario 2

$$\lambda_c = \lambda \cdot \pi_{\alpha_1} \qquad \lambda_g = \lambda_c \cdot \pi_{\alpha_2} \qquad \rho_g = \frac{\lambda_g}{\mu_g}$$
$$= 0.6 \cdot 1 \qquad = 0.6 \cdot 0.5 \qquad = \frac{0.3}{0.8}$$
$$\boxed{= 0.6} \qquad \boxed{= 0.3} \qquad \boxed{= 0.375}$$

$$E[\mathcal{P}_g] = p_g \cdot \rho_g \qquad E[\mathcal{N}_g] = \frac{\rho_g}{1-\rho_g} \qquad E[\mathcal{T}_g] = \frac{1}{\lambda_g} \cdot E[\mathcal{N}_g] \qquad E[\mathcal{E}_g] = \frac{p_g}{\mu_g}$$
$$= 2.5 \cdot 0.375 \qquad = \frac{0.375}{1-0.375} \qquad = \frac{1}{0.3} \cdot 0.6 \qquad = \frac{2.5}{0.8}$$
$$\boxed{= 0.9375} \qquad \boxed{= 0.6} \qquad = \boxed{2} \qquad = \boxed{3.125}$$

C.2.3 Scénario 3

$$\begin{aligned} \lambda_c &= \lambda \cdot \pi_{\alpha_1} \\ &= 0.6 \cdot 0.5 \\ &\boxed{= 0.3} \end{aligned} \qquad \begin{aligned} \lambda_g &= \lambda_c \cdot \pi_{\alpha_2} \\ &= 0.3 \cdot 0.5 \\ &\boxed{= 0.15} \end{aligned} \qquad \begin{aligned} \rho_g &= \frac{\lambda_g}{\mu_g} \\ &= \frac{0.15}{0.8} \\ &\boxed{= 0.1875} \end{aligned}$$

$$\begin{aligned} E[\mathcal{P}_g] &= p_g \cdot \rho_g \\ &= 2.5 \cdot 0.1875 \\ &\boxed{= 0.46875} \end{aligned} \qquad \begin{aligned} E[\mathcal{N}_g] &= \frac{\rho_g}{1-\rho_g} \\ &= \frac{0.1875}{1-0.1875} \\ &= \frac{0.1875}{0.8125} \\ &\boxed{= 0.230769} \end{aligned} \qquad \begin{aligned} E[\mathcal{T}_g] &= \frac{1}{\lambda_g} \cdot E[\mathcal{N}_g] \\ &= \frac{1}{0.1875} \cdot 0.230769 \\ &\boxed{= 1.538462} \end{aligned} \qquad \begin{aligned} E[\mathcal{E}_g] &= \frac{p_g}{\mu_g} \\ &= \frac{2.5}{0.8} \\ &= \boxed{3.125} \end{aligned}$$

C.2.4 Scénario 4

Mêmes résultats qu'au scénario 3.

C.3 File WiFi

C.3.1 Scénario 1

$$\begin{aligned} \lambda_c &= \lambda \cdot \pi_{\alpha_1} \\ &= 0.6 \cdot 0 \\ &\boxed{= 0} \end{aligned}$$

Si $\lambda_c = 0$ alors aucune opération ne passe par cette file. Toutes les espérances sont mises à 0 par défaut.

C.3.2 Scénario 2

$$\lambda_c = \lambda \cdot \pi_{\alpha_1} \qquad \lambda_w = \lambda_c \cdot (1 - \pi_{\alpha_2}) \qquad \pi_{w_1} = \frac{\eta}{\eta + \xi} \qquad \rho_w = \frac{\lambda_w}{\pi_{w_2} \cdot \mu_w}$$

$$= 0.6 \cdot 1 \qquad = 0.6 \cdot 0.5 \qquad = \frac{1}{1+1} \qquad = \frac{0.3}{0.5 \cdot 2}$$

$$\boxed{= 0.6} \qquad \boxed{= 0.3} \qquad \boxed{= 0.5} \qquad \boxed{= 0.3}$$

$$\begin{aligned} E[\mathcal{N}_w] &= \frac{\lambda_w}{\pi_{w_2}\mu_w - \lambda_w} + \frac{\pi_{w_1}\lambda_w\mu_w}{(\pi_{w_2}\mu_w - \lambda_w)(\xi + \eta)} \\ &= \frac{0.3}{(0.5 \cdot 2) - 0.3} + \frac{0.5 \cdot 0.3 \cdot 2}{((0.5 \cdot 2) - 0.3)(1+1)} \\ &= \frac{0.3}{0.7} + \frac{0.3}{1.4} \\ &\boxed{= 0.642857} \end{aligned}$$

$$E[\mathcal{P}_w] = p_g \cdot \rho_g \cdot \pi_{w_2} \qquad E[\mathcal{T}_w] = \frac{1}{\lambda_w} \cdot E[\mathcal{N}_w] \qquad E[\mathcal{E}_w] = \frac{p_w}{\mu_w}$$

$$= 0.7 \cdot 0.3 \cdot 0.5 \qquad = \frac{1}{0.3} \cdot 0.642857 \qquad = \frac{0.7}{2}$$

$$\boxed{= 0.105} \qquad \boxed{= 2.142857} \qquad = \boxed{0.35}$$

C.3.3 Scénario 3

$$\lambda_c = \lambda \cdot \pi_{\alpha_1} \qquad \lambda_w = \lambda_c \cdot (1 - \pi_{\alpha_2}) \qquad \pi_{w_2} = \frac{\eta}{\eta + \xi} \qquad \rho_w = \frac{\lambda_w}{\pi_{w_2} \cdot \mu_w}$$

$$= 0.6 \cdot 0.5 \qquad = 0.3 \cdot 0.5 \qquad = \frac{1}{1+1} \qquad = \frac{0.15}{0.5 \cdot 2}$$

$$\boxed{= 0.3} \qquad \boxed{= 0.15} \qquad \boxed{= 0.5} \qquad \boxed{= 0.15}$$

$$\begin{aligned} E[\mathcal{N}_w] &= \frac{\lambda_w}{\pi_{w_2}\mu_w - \lambda_w} + \frac{\pi_{w_2}\lambda_w\mu_w}{(\pi_{w_2}\mu_w - \lambda_w)(\xi + \eta)} \\ &= \frac{0.15}{(0.5 \cdot 2) - 0.15} + \frac{0.5 \cdot 0.15 \cdot 2}{((0.5 \cdot 2) - 0.15)(1+1)} \\ &= \frac{0.15}{0.85} + \frac{0.15}{1.7} \\ &\boxed{= 0.264706} \end{aligned}$$

$$E[\mathcal{P}_w] = p_w \cdot \rho_w \cdot \pi_{w_2} \qquad E[\mathcal{T}_w] = \frac{1}{\lambda_w} \cdot E[\mathcal{N}_w] \qquad E[\mathcal{E}_w] = \frac{p_w}{\mu_w}$$
$$= 0.7 \cdot 0.15 \cdot 0.5 \qquad = \frac{1}{0.15} \cdot 0.264706 \qquad = \frac{0.7}{2}$$
$$\boxed{= 0.0525} \qquad \boxed{= 1.764706} \qquad = \boxed{0.35}$$

C.3.4 Scénario 4

$$\lambda_c = \lambda \cdot \pi_{\alpha_1} \qquad \lambda_w = \lambda_c \cdot (1 - \pi_{\alpha_2}) \qquad \pi_{w_2} = \frac{\eta}{\eta + \xi} \qquad \rho_w = \frac{\lambda_w}{\pi_{w_2} \cdot \mu_w}$$
$$= 0.6 \cdot 0.5 \qquad = 0.3 \cdot 0.5 \qquad = \frac{2}{7} \qquad = \frac{0.15}{0.285714 \cdot 2}$$
$$\boxed{= 0.3} \qquad \boxed{= 0.15} \qquad \boxed{= 0.285714} \qquad \boxed{= 0.262500}$$

$$\begin{aligned} E[\mathcal{N}_w] &= \frac{\lambda_w}{\pi_{w_2}\mu_w - \lambda_w} + \frac{\pi_{w_1}\lambda_w\mu_w}{(\pi_{w_2}\mu_w - \lambda_w)(\xi + \eta)} \\ &= \frac{0.15}{(0.285714 \cdot 2) - 0.15} + \frac{0.714286 \cdot 0.15 \cdot 2}{((0.285714 \cdot 2) - 0.15)(5 + 2)} \\ &= \frac{0.15}{0.421428} + \frac{0.214286}{2.949996} \\ &\boxed{= 0.428572} \end{aligned}$$

$$E[\mathcal{P}_w] = p_w \cdot \rho_w \cdot \pi_{w_2} \qquad E[\mathcal{T}_w] = \frac{1}{\lambda_w} \cdot E[\mathcal{N}_w] \qquad E[\mathcal{E}_w] = \frac{p_w}{\mu_w}$$
$$= 0.7 \cdot 0.262500 \cdot 0.285714 \qquad = \frac{1}{0.15} \cdot 0.428572 \qquad = \frac{0.7}{2}$$
$$\boxed{= 0.05250} \qquad \boxed{= 2.857147} \qquad = \boxed{0.35}$$

C.4 File cloud

C.4.1 Scénario 1

$$\lambda_c = \lambda \cdot \pi_{\alpha_1}$$
$$= 0.6 \cdot 0$$
$$\boxed{= 0}$$

Si $\lambda_c = 0$ alors aucune opération ne passe par cette file. Toutes les espérances sont mises à 0 par défaut.

C.4.2 Scénario 2

$$\lambda_c = \lambda \cdot \pi_{\alpha_1} \qquad E[\mathcal{N}_c] = \frac{\lambda_c}{\mu_c} \qquad E[\mathcal{T}_c] = \frac{1}{\mu_c}$$
$$= 0.6 \cdot 1 \qquad = \frac{0.6}{5} \qquad = \frac{1}{5}$$
$$\boxed{= 0.6} \qquad \boxed{= 0.12} \qquad \boxed{= 0.2}$$

C.4.3 Scénario 3

$$\lambda_c = \lambda \cdot \pi_{\alpha_1} \qquad E[\mathcal{N}_c] = \frac{\lambda_c}{\mu_c} \qquad E[\mathcal{T}_c] = \frac{1}{\mu_c}$$
$$= 0.6 \cdot 0.5 \qquad = \frac{0.3}{5} \qquad = \frac{1}{5}$$
$$\boxed{= 0.3} \qquad \boxed{= 0.06} \qquad \boxed{= 0.2}$$

C.4.4 Scénario 4

Mêmes résultats qu'au scénario 3.

C.5 File transmission

C.5.1 Scénario 1

$$\pi_1 = \frac{\xi}{\eta + \xi} \qquad \pi_2 = \frac{\eta}{\eta + \xi}$$
$$= \frac{1}{1+1} \qquad = \frac{1}{1+1}$$
$$\boxed{= 0.5} \qquad \boxed{= 0.5}$$

$$\lambda^* = \pi_1 \cdot \lambda_1 + \pi_2 \cdot \lambda_2 \qquad \mu^* = \pi_1 \cdot \mu_1 + \pi_2 \cdot \mu_2$$
$$= 0.5 \cdot 0.214286 + 0.5 \cdot 0.085714 \qquad = 0.5 \cdot 2 + 0.5 \cdot 0.8$$
$$\boxed{= 0.15} \qquad \boxed{= 1.4}$$

$$E[\mathcal{P}_1] = p_1 \cdot \rho_1 \cdot \pi_1 \qquad E[\mathcal{P}_2] = p_2 \cdot \rho_2 \cdot \pi_2 \qquad E[\mathcal{N}_t] = \frac{\lambda^*}{\mu^* - \lambda^*}$$
$$= 2.5 \cdot 0.107143 \cdot 0.5 \qquad = 0.7 \cdot 0.107143 \cdot 0.5 \qquad = \frac{0.15}{1.4 - 0.15}$$
$$\boxed{= 0.133929} \qquad \boxed{= 0.0375} \qquad \boxed{= 0.12}$$

$$E[\mathcal{T}_t] = \frac{1}{\lambda^*} \cdot E[\mathcal{N}_t] \qquad E[\mathcal{E}_t] = \frac{1}{\lambda^*} \cdot (E[\mathcal{P}_1] + E[\mathcal{P}_2])$$
$$= \frac{1}{0.15} \cdot 0.12 \qquad = \frac{1}{0.15} \cdot (0.133929 + 0.0375)$$
$$\boxed{= 0.8} \qquad \boxed{= 1.142857}$$

C.5.2 Scénario 2

$$\pi_1 = \frac{\xi}{\eta + \xi} \qquad \pi_2 = \frac{\eta}{\eta + \xi}$$
$$= \frac{5}{2 + 5} \qquad = \frac{2}{2 + 5}$$
$$\boxed{= 0.714286} \qquad \boxed{= 0.285714}$$

$$\lambda^* = \pi_1 \cdot \lambda_1 + \pi_2 \cdot \lambda_2 \qquad \mu^* = \pi_1 \cdot \mu_1 + \pi_2 \cdot \mu_2$$
$$= 0.714286 \cdot 0.085714 + 0.285714 \cdot 0.214286 \qquad = 0.714286 \cdot 0.8 + 0.285714 \cdot 2$$
$$\boxed{= 0.122449} \qquad \boxed{= 1.142857}$$

$$E[\mathcal{P}_1] = p_1 \cdot \rho_1 \cdot \pi_1 \qquad E[\mathcal{P}_2] = p_2 \cdot \rho_2 \cdot \pi_2$$
$$= 2.5 \cdot 0.107143 \cdot 0.714286 \qquad = 0.7 \cdot 0.107143 \cdot 0.285714$$
$$\boxed{= 0.191327} \qquad \boxed{= 0.021429}$$

$$E[\mathcal{N}_t] = \frac{\lambda^*}{\mu^* - \lambda^*} \qquad E[\mathcal{T}_t] = \frac{1}{\lambda^*} \cdot E[\mathcal{N}_t] \qquad E[\mathcal{E}_t] = \frac{1}{\lambda^*} \cdot (E[\mathcal{P}_1] + E[\mathcal{P}_2])$$
$$= \frac{0.122449}{1.142857 - 0.122449} \qquad = \frac{1}{0.122449} \cdot 0.12 \qquad = \frac{0.191327 + 0.021429}{0.122449}$$
$$\boxed{= 0.12} \qquad \boxed{= 0.98} \qquad \boxed{= 1.7375}$$

Annexe D

Transformation des équations pour le calcul de $E[\mathcal{T}_r]$ et $E[\mathcal{E}_r]$

Dans cette annexe, nous proposons le détail des étapes permettant de passer des équations :

$$E[\mathcal{T}_r] = \pi_{\alpha_1} E[\mathcal{T}_o] + (1 - \pi_{\alpha_1}) E[\mathcal{T}_m] \tag{D.1}$$
$$E[\mathcal{E}_r] = \pi_{\alpha_1} E[\mathcal{E}_o] + (1 - \pi_{\alpha_1}) E[\mathcal{E}_m] \tag{D.2}$$

aux équations :

$$E[\mathcal{T}_r] = \frac{1}{\lambda}(E[\mathcal{N}_g] + E[\mathcal{N}_w] + E[\mathcal{N}_c] + E[\mathcal{N}_m]) \tag{D.3}$$
$$E[\mathcal{E}_r] = \frac{1}{\lambda}(E[\mathcal{P}_g] + E[\mathcal{P}_w] + E[\mathcal{P}_m]) \tag{D.4}$$

Tout d'abord, rappelons que :

$$\lambda_c = \pi_{\alpha_1} \cdot \lambda$$
$$\Rightarrow \pi_{\alpha_1} = \frac{\lambda_c}{\lambda} \tag{D.5}$$
$$\lambda_m = (1 - \pi_{\alpha_1}) \cdot \lambda$$
$$\Rightarrow (1 - \pi_{\alpha_1}) = \frac{\lambda_m}{\lambda} \tag{D.6}$$

En substituant (D.5) et (D.6) dans (D.1) et (D.2) nous obtenons :

$$\begin{aligned} E[\mathcal{T}_r] &= \frac{\lambda_c}{\lambda} E[\mathcal{T}_o] + \frac{\lambda_m}{\lambda} E[\mathcal{T}_m] \\ &= \frac{1}{\lambda}(\lambda_c E[\mathcal{T}_o] + \lambda_m E[\mathcal{T}_m]) \end{aligned} \tag{D.7}$$

$$\begin{aligned} E[\mathcal{E}_r] &= \frac{\lambda_c}{\lambda} E[\mathcal{E}_o] + \frac{\lambda_m}{\lambda} E[\mathcal{E}_m] \\ &= \frac{1}{\lambda}(\lambda_c E[\mathcal{E}_o] + \lambda_m E[\mathcal{E}_m]) \end{aligned} \tag{D.8}$$

Dans (D.7), nous nous concentrons dans un premier temps sur $E[\mathcal{T}_o]$, qui est donné par :

$$E[\mathcal{T}_o] = \pi_{\alpha_2} E[\mathcal{T}_g] + (1 - \pi_{\alpha_2}) E[\mathcal{T}_w] + E[\mathcal{T}_c] \tag{D.9}$$

avec cette fois-ci :

$$\lambda_g = \pi_{\alpha_2} \cdot \lambda_c$$

$$\Rightarrow \pi_{\alpha_2} = \frac{\lambda_g}{\lambda_c} \tag{D.10}$$

$$\lambda_w = (1 - \pi_{\alpha_2}) \cdot \lambda_c$$

$$\Rightarrow (1 - \pi_{\alpha_2}) = \frac{\lambda_w}{\lambda_c} \tag{D.11}$$

En substituant (D.10) et (D.11) dans (D.9) et en appliquant la formule de Little, nous obtenons :

$$\begin{aligned} E[\mathcal{T}_o] &= \frac{\lambda_g}{\lambda_c} E[\mathcal{T}_g] + \frac{\lambda_w}{\lambda_c} E[\mathcal{T}_w] + E[\mathcal{T}_c] \\ &= \frac{1}{\lambda_c}(\lambda_g E[\mathcal{T}_g] + \lambda_w E[\mathcal{T}_w]) + E[\mathcal{T}_c] \\ &= \frac{1}{\lambda_c}(\lambda_g \cdot \frac{E[\mathcal{N}_g]}{\lambda_g} + \lambda_w \frac{E[\mathcal{N}_w]}{\lambda_w} + \frac{E[\mathcal{N}_c]}{\lambda_c}) \\ &= \frac{1}{\lambda_c}(E[\mathcal{N}_g] + E[\mathcal{N}_w] + E[\mathcal{N}_c]) \end{aligned} \tag{D.12}$$

sachant que :

$$E[\mathcal{T}_m] = \frac{1}{\lambda_m} E[\mathcal{N}_m] \tag{D.13}$$

Nous pouvons enfin substituer (D.12) et (D.13) dans (D.7) afin d'obtenir (D.3) :

$$\begin{aligned} E[\mathcal{T}_r] &= \frac{1}{\lambda}(\lambda_c E[\mathcal{T}_o] + \lambda_m E[\mathcal{T}_m]) \\ &= \frac{1}{\lambda}\{\lambda_c \frac{1}{\lambda_c}(E[\mathcal{N}_g] + E[\mathcal{N}_w] + E[\mathcal{N}_c]) + \lambda_m \frac{1}{\lambda_m} E[\mathcal{N}_m]\} \\ &= \frac{1}{\lambda}(E[\mathcal{N}_g] + E[\mathcal{N}_w] + E[\mathcal{N}_c] + E[\mathcal{N}_m]) \end{aligned} \tag{D.14}$$

Passons maintenant à la transformation de (D.8). La méthode de transformation est très similaire à ce que nous avons vu pour (D.7). Nous commençons par exprimer $E[\mathcal{E}_o]$ comme suit :

$$E[\mathcal{E}_o] = \pi_{\alpha_2} E[\mathcal{E}_g] + (1 - \pi_{\alpha_2}) E[\mathcal{E}_w] \tag{D.15}$$

En substituant (D.10) et (D.11) dans (D.15), nous obtenons :

$$\begin{aligned} E[\mathcal{E}_o] &= \frac{\lambda_g}{\lambda_c} E[\mathcal{E}_g] + \frac{\lambda_w}{\lambda_c} E[\mathcal{E}_w] \\ &= \frac{1}{\lambda_c}(\lambda_g E[\mathcal{E}_g] + \lambda_w E[\mathcal{E}_w]) \\ &= \frac{1}{\lambda_c}(\lambda_g \cdot \frac{E[\mathcal{P}_g]}{\lambda_g} + \lambda_w \frac{E[\mathcal{P}_w]}{\lambda_w} \\ &= \frac{1}{\lambda_c}(E[\mathcal{P}_g] + E[\mathcal{P}_w]) \end{aligned} \tag{D.16}$$

sachant que :

$$E[\mathcal{E}_m] = \frac{1}{\lambda_m} E[\mathcal{P}_m] \tag{D.17}$$

Nous pouvons enfin substituer (D.16) et (D.17) dans (D.8) afin d'obtenir (D.4) :

$$\begin{aligned} E[\mathcal{E}_r] &= \frac{1}{\lambda}(\lambda_c E[\mathcal{E}_o] + \lambda_m E[\mathcal{E}_m]) \\ &= \frac{1}{\lambda}\{\lambda_c \frac{1}{\lambda_c}(E[\mathcal{P}_g] + E[\mathcal{P}_w]) + \lambda_m \frac{1}{\lambda_m} E[\mathcal{P}_m]\} \\ &= \frac{1}{\lambda}(E[\mathcal{P}_g] + E[\mathcal{P}_w] + E[\mathcal{P}_m]) \end{aligned} \tag{D.18}$$

Ces transformations sont valables également pour la file transmission. Il suffit de remplacer les indices g et w par 1 et 2 respectivement.

Annexe E

Analysis of the Energy-Response Time Tradeoff for Mobile Cloud Offloading Using Combined Metrics

Dans cette dernière annexe nous proposons la version originale de notre article de référence.

2015 27th International Teletraffic Congress

Analysis of the Energy-Response Time Tradeoff for Mobile Cloud Offloading Using Combined Metrics

Huaming Wu
Free University of Berlin, Germany
Email: huaming.wu@fu-berlin.de

William Knottenbelt
Imperial College London, UK
Email: wjk@doc.ic.ac.uk

Katinka Wolter
Free University of Berlin, Germany
Email: katinka.wolter@fu-berlin.de

***Abstract*—Mobile offloading migrates heavy computation from mobile devices to cloud servers using one or more communication network channels. Communication interfaces vary in speed, energy consumption and degree of availability. We assume two interfaces: WiFi, which is fast with low energy demand but not always present and cellular, which is slightly slower has higher energy consumption but is present at all times. We study two different communication strategies: one that selects the best available interface for each transmitted packet and the other multiplexes data across available communication channels. Since the latter may experience interrupts in the WiFi connection packets can be delayed. We call it interrupted strategy as opposed to the uninterrupted strategy that transmits packets only over currently available networks.**

Two key concerns of mobile offloading are the energy use of the mobile terminal and the response time experienced by the user of the mobile device. In this context, we investigate three different metrics that express the energy-performance tradeoff, the known Energy-Response time Weighted Sum (EWRS), the Energy-Response time Product (ERP) and the Energy-Response time Weighted Product (ERWP) metric.

We apply the metrics to the two different offloading strategies and find that the conclusions drawn from the analysis depend on the considered metric. In particular, while an additive metric is not normalised, which implies that the term using smaller scale is always favoured, the ERWP metric, which is new in this paper, allows to assign importance to both aspects without being misled by different scales. It combines the advantages of an additive metric and a product.

The interrupted strategy can save energy especially if the focus in the tradeoff metric lies on the energy aspect. In general one can say that the uninterrupted strategy is faster, while the interrupted strategy uses less energy. A fast connection improves the response time much more than the fast repair of a failed connection. In conclusion, a short down-time of the transmission channel can mostly be tolerated.

***Index Terms*—Energy-Performance Tradeoff, Queuing Model, Offloading, Heterogeneous Networks, Mobile Cloud Computing**

I. Introduction

Mobile cloud computing aims at combining the strength of cloud computing with the convenience of mobile terminals. However, limited radio resources or limitations of other communication channels as well as lack of sufficient battery power may significantly impede the improvement of service quality [1] anticipated by using cloud services. Nonetheless computation offloading, which migrates computation-intensive tasks from mobile devices to a remote cloud infrastructure via a network, is a popular approach to alleviate the shortcomings of resource-constrained mobile devices. Since offloading an application to the cloud is not always possible or effective, the decision as to whether to execute a program locally or to offload it requires careful consideration of the nature of the computation and the communication channels available.

Mobile devices are often equipped with multiple wireless interfaces (e.g. cellular and WiFi) for data transfer, with different availabilities, delays and energy costs. Response time and energy consumption are two primary concerns for mobile systems that must be considered when making offloading decisions. However, different applications usually give different relative importance to both factors. For delay-tolerant applications (e.g. iCloud, Dropbox, RSS feeds and participatory sensing), response time is less critical and optimising energy usage is more relevant. Some information is not time-critical and its submission to the server may be delayed until the device enters an energy-efficient network. For delay-sensitive applications (e.g. speed chess game, face recognition, video conferencing and vehicular communications), fast response time is of primary concern while energy consumption is less important. The offloading scheme in which cloud services are accessible with short network latency (e.g. WiFi network) can serve in a better way. Therefore, there exists a fundamental tradeoff between energy consumption and response time in the expected usability of applications [2].

Recently, several researchers have worked on optimising the tradeoff between energy consumption and response time. In [3] and [4], the energy-delay tradeoff has been studied by deciding whether or not and by means of which communication interface to offload a whole application. Instead, an application can consist of several components, or jobs, that are treated separately, and thus offloading decisions should be made for every component. Accordingly, a queueing model has been proposed in [5] to capture the tradeoff between the energy consumption and the response time for mobile cloud offloading based on an additive energy-performance metric, where static and dynamic offloading polices were analysed.

Seamless offloading operation by switching between several transmission technologies has been proposed in [6]. In addition, this work examined the tradeoff between energy consumption for WiFi search and transmission rate when the WiFi network was intermittently available. A stochastic model for that scenario has been developed in [7] using various performance metrics and also intermittently available access links. Energy-efficient delayed network selection has been suggested in [2] and [8] to optimise the tradeoff between energy usage and delay in data transmission by intentionally deferring data transmission until the device meets an energy-efficient network.

The main contributions of this paper are as follows:

- We propose two strategies for mobile cloud offloading and analyse them by means of analytic queueing models: the uninterrupted offloading strategy and the interrupted offloading strategy. The uninterrupted strategy uses WiFi

978-1-4673-8422-3/15 $31.00 © 2015 IEEE
DOI 10.1109/ITC.2015.23

(which we assume to have the best transmission characteristics) whenever possible, but switches to a cellular interface if no WiFi connection exists [9]. The interrupted strategy assigns jobs upon arrival to one of two parallel interfaces with different characteristics which are modelled as two parallel queues[1] (e.g. cellular or WiFi transmission). Data transmission of the WiFi queue can be interrupted for short periods when the connection is lost.

- The models are compared using several metrics. We apply the previously studied Energy-Response time Weighted Sum (ERWS) and compare it with the Energy-Response time Product (ERP). After discussing advantages and disadvantages of both metrics we introduce the Energy-Response time Weighted Product (ERWP), which combines the advantages of both previously studied metrics.

The remainder of this paper is organised as follows. In Section II, we introduce the offloading system and the queueing model for offloading as well as the three considered metrics. In Section III, we analyse the uninterrupted offloading strategy based on the ERWP metric. The interrupted offloading strategy is proposed and analysed in Section IV. Section V evaluates metrics and models using numerical examples. Section VI concludes.

II. System Overview

In this section, we first introduce the overall offloading system, and then focus on its submodules: local execution and remote execution, respectively. Finally, we combine them by using metrics to capture the tradeoff between the mean energy consumption and mean response time.

A. The Offloading System

We model mobile offloading as a queueing system as depicted in Fig. 1. The mobile device, the cloud and the wireless networks are represented as queueing nodes to capture the resource contention and delay on the system. The mobile device executes an application with different types of jobs that can be classified into the following two classes. Each time a job is executed, a decision must be taken into which class it belongs:

- **Unoffloadable**: some jobs should be unconditionally processed locally on the mobile device, either because transferring the relevant information would take more time and energy or because these tasks must access local devices (camera, sensors, user interface, etc.) [10]. Local processing consumes battery power of the mobile device but there are no communication costs or delays.
- **Offloadable**: some jobs are flexible tasks that can be processed either locally on the mobile device, or remotely in a cloud infrastructure through computation offloading. Many tasks fall into this category, and the offloading decision depends on whether the communication costs outweigh the difference between local and remote costs [11].

We do not need to take offloading decisions for unoffloadable jobs. However, as for offloadable ones, the mobile device should judiciously make decisions that optimise the response time energy tradeoff expressed in one of the metrics defined at the end of this section.

[1]For simplicity we will call the two considered networks WiFi and cellular, but this could be any other technology with equivalent characteristics.

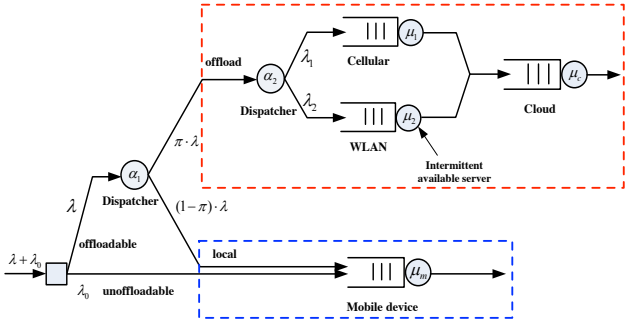


Figure 1. A queueing system for mobile cloud offloading

As indicated in Fig. 1, job arrivals at the mobile device are assumed to follow a Poisson process with an average arrival rate of $\lambda + \lambda_0$, where λ and λ_0 are the rates of offloadable and unoffloadable jobs, respectively. The arrival rate is based on the behavior of the application. The unoffloadable jobs with an arrival rate λ_0 are unconditionally executed locally. As for the offloadable ones, these arrive at rate λ, and the mobile device chooses to offload each job with a probability $0 \leq \pi \leq 1$. As in [12], jobs are offloaded to the cloud following a Poisson process with an average rate of $\lambda_c = \pi \cdot \lambda$, the offloading rate. Similarly, jobs that are proceed locally instead of being offloaded follow a Poisson process with rate $\lambda_m = (1 - \pi) \cdot \lambda$.

There are several ways to offload computation to the cloud, e.g. via a cellular connection (e.g. 2G, 3G and 4G), which is assumed to be the costly connection, or via intermittently available WiFi. We assume that the cellular interface can provide ubiquitous coverage for mobile devices in a wide area, but has lower data transmission rate and consumes more transmission energy than the WiFi interface. In many cases these assumptions are realistic. The mobile device, the cellular and WiFi connections are modelled as $M/M/1$-FCFS queues. The remote cloud is modelled as an $M/M/\infty$ queue, i.e. as a delay center [13]. We denote $1/\mu_m$ and $1/\mu_c$ the expected execution time of jobs on the mobile device and the cloud, respectively. The expected rates to transfer data to the cloud over the cellular network and WiFi are μ_1 and μ_2, respectively.

Two dispatchers are needed: α_1 is used to allocate the offloadable jobs either to the cloud or to the mobile device, while α_2 is used to offload the jobs either via a cellular connection or a WiFi network to the cloud. It should be noted that when $\pi = 0$, all offloadable jobs are processed locally, when $\pi = 1$, they are all offloaded to the cloud. The total cost, in terms of energy or response time for processing all offloadable jobs, is composed of the remote cost (sending some jobs to the cloud and waiting for the cloud to complete them), and the local cost (processing the rest jobs locally on the mobile device).

Our objective is to minimise the mean energy consumption and the mean response time. Since only one option exists for the jobs that cannot be offloaded (as they will always be processed locally) our attention is focused on on the offloadable jobs, where the right decision must be taken whether or not to offload and which interface to use for offloading. The key elements for the considered offloading system are as shown inside the blue block (local execution) and the red block (remote execution), which are analysed separately in the following subsections.

B. Local Execution

As shown inside the blue dotted block in Fig. 1, there are two kinds of jobs (offloadable and unoffloadable) arriving to the processor of the mobile device. We adopt the preemptive scheduling policy here. That is, from the perspective of an offloadable job, the unoffloadable jobs do not exist, since service to the unoffloadable jobs is immediately interrupted upon the arrival of an offloadable job. To the offloadable jobs, the system behaves like an $M/M/1$ queue with arrival rate λ_m and service rate μ_m.

The workload, or utilisation, i.e. the fraction of time when the server is busy, is denoted as: $\rho_m = \lambda_m/\mu_m$. The mean number of jobs in an $M/M/1$ queue is given by:

$$\mathbb{E}[N_m] = \frac{\rho_m}{1-\rho_m}. \tag{1}$$

By Little's Law we obtain the mean response time as:

$$\mathbb{E}[T_m] = \frac{1}{\lambda_m}\mathbb{E}[N_m]. \tag{2}$$

We assume the mobile device consumes energy only when there are jobs in the system and that the mobile device operates at a constant power p_m whenever it is busy [14]. Since $P_m = \lambda_m \mathcal{E}_m$ is the consumed power, the mean energy consumption $\mathbb{E}[\mathcal{E}_m]$ can be more conveniently expressed as:

$$\begin{aligned}\mathbb{E}[\mathcal{E}_m] &= \frac{1}{\lambda_m}\cdot\mathbb{E}[P_m] = \frac{1}{\lambda_m}\cdot p_m \Pr\{N_m > 0\} \\ &= \frac{1}{\lambda_m}\cdot p_m \rho_m,\end{aligned} \tag{3}$$

where $\Pr$ denotes the probability operation.

C. Remote Execution

As shown inside the red dotted block in Fig. 1, the remote execution includes the transmission model and the cloud model. To facilitate the analysis of the mobile cloud offloading system, we assume that a cellular network is available to mobile users all the time while the availability of a WiFi network depends on the location. That is, mobile users move in and out of a WiFi coverage area. We model this time variation of the WiFi connection state by the ON-OFF alternating renewal process $\left(T_{\text{ON}}^{(j)}, T_{\text{OFF}}^{(j)}\right)$, $j \geq 1$, as shown in Fig. 2. The ON periods represent the presence of the WiFi connectivity, while the OFF periods represent the interruption of the WiFi connectivity. During the latter periods data is either not transmitted (because the interface is idle) or it is transmitted only through the cellular network. The duration of each ON period $T_{\text{ON}}^{(j)}$ or OFF period $T_{\text{OFF}}^{(j)}$, is assumed to be an exponentially distributed random variable and independent of the duration of other ON or OFF periods [9].

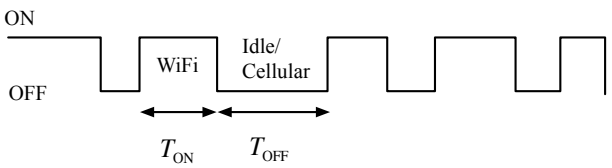


Figure 2. The WiFi network availability model

Accordingly, we build two different offloading strategies:

- **Uninterrupted Offloading Strategy**: we employ a single queue with two states to offload jobs to the cloud server. When there is a WiFi connection available, all jobs are sent over the WiFi network; otherwise, they are sent over the cellular interface as the cellular network is always available [15]. Therefore, the process of offloading jobs to the cloud cannot be interrupted.
- **Interrupted Offloading Strategy**: we assign jobs upon arrival to one of two parallel queues which describe cellular or WiFi transmission [5]. Offloading is interrupted during the periods when WiFi is disconnected. It is a dispatching problem where incoming jobs are splitted on arrival for service by two servers and join before departure. Only when all the jobs are transmitted and have rejoined can the cloud processing start.

We have two states (for the uninterrupted offloading strategy) or two parallel queues (for the interrupted offloading strategy). Upon arrival of a job an assignment decision is made and the job is placed into the corresponding queue (state).

A key assumption in our work is that each server (state) operates at a constant power p_i ($i \in \{1,2\}$) whenever it is busy. We use $e_i \in \{0,1\}$ to indicate whether Server (State) i is available or not. If $e_i = 1$, Server (State) i is available, otherwise it is unavailable. Since $P_i = \lambda_i \mathcal{E}_i$ is the consumed power, further by Little's Law, $\mathbb{E}[N_i] = \lambda_i \mathbb{E}[T_i]$, the mean energy consumption and mean response time due to offloading can be calculated as follows, respectively.

$$\begin{aligned}\mathbb{E}[\mathcal{E}_o] &= \frac{1}{\lambda_c}\sum_{i=1}^{2}\lambda_i\mathbb{E}[\mathcal{E}_i] = \frac{1}{\lambda_c}\sum_{i=1}^{2}\mathbb{E}[P_i] \\ &= \frac{1}{\lambda_c}\Big\{\sum_{i=1}^{2} p_i \Pr\{N_i > 0, e_i = 1\}\Big\},\end{aligned} \tag{4}$$

$$\begin{aligned}\mathbb{E}[T_o] &= \frac{1}{\lambda_c}\sum_{i=1}^{2}\lambda_i\mathbb{E}[T_i] + \mathbb{E}[T_c] \\ &= \frac{1}{\lambda_c}\Big\{\sum_{i=1}^{2}\mathbb{E}[N_i] + \mathbb{E}[N_c]\Big\},\end{aligned} \tag{5}$$

where λ_i is the mean rate of jobs arriving to Queue (State) i; $\mathbb{E}[\mathcal{E}_i]$ and $\mathbb{E}[T_i]$ are the mean energy consumption and mean response time in Queue (State) i, respectively, and $\mathbb{E}[T_c]$ is the expected execution time in the cloud server.

Since the probability that the corresponding server (state) is busy is equal to the load [14], we have $\Pr\{N_i > 0 | e_i = 1\} = \rho_i$ when the server (state) is always available. Further, we have:

$$\begin{aligned}\Pr\{N_i > 0, e_i = 1\} &= \Pr\{N_i > 0 | e_i = 1\}\cdot\Pr\{e_i = 1\} \\ &= \rho_i \cdot \Pr\{e_i = 1\}.\end{aligned} \tag{6}$$

D. Metrics

The general cost metric includes energy consumption related costs in addition to the usual performance metrics such as the response time. The response time is the time between the arrival of a job until it completes service and departs. The energy consumption is the energy spent on the mobile device in that period. We use queueing theory to model the offloading systems according to different response time energy metrics. We study the tradeoff between the mean energy consumption and mean response time, which is a non-trivial multi-objective optimization problem and define three different metrics in the following subsections.

1) ERWS: The Energy-Response time Weighted Sum (ERWS) is a metric to capture energy-performance tradeoffs and to compare different policies. It is defined by setting the cost function as the weighted sum of both average values, i.e. the ERWS metric:

$$ERWS = \omega\mathbb{E}[\mathcal{E}] + (1-\omega)\mathbb{E}[T], \tag{7}$$

where $\mathbb{E}[\mathcal{E}]$ and $\mathbb{E}[T]$ are the mean energy consumption and mean response time, respectively. ω (ranging between 0 and 1) is a weighting parameter that indicates the relative significance between the energy consumption and response time for the mobile device. Large ω favors energy consumption while small ω favors response time. In some special cases performance can be traded for power consumption and vice versa [16], therefore we can use the ω parameter to express such special cases preferences for different applications.

For Eq. (7), the mean time and energy are additive terms over time; therefore the ERWS metric has the advantage of being analytically tractable and can be optimized via Markov Decision Processes [17]. From the view of minimization, this metric allows comparing arbitrary offloading policies to the optimal offloading policy in our work. However, it has the disadvantage of a linear combination of two metrics on different scales.

2) ERP: The Energy-Response time Product (ERP) metric is widely accepted as a suitable metric to capture energy-performance tradeoffs. It is defined as:

$$ERP = \mathbb{E}[\mathcal{E}] \cdot \mathbb{E}[T]. \tag{8}$$

Minimizing ERP can be seen as maximizing the 'performance-per-joule', with performance being defined as the inverse of mean response time [17].

The energy response time product does not suffer from comparison of different scales. While the ERWS metric implies that a reduction in mean response time from 1000 sec to 999 sec is of the same value as a reduction from 2 sec to 1 sec, the ERP metric implies that a reduction in mean response time from 2 sec to 1 sec is much better than a reduction from 1000 sec to 999 sec, which is indeed a more realistic view on the actual system [17]. However, since ERP is the product of two expectations, it is a difficult metric to address analytically.

3) ERWP: To overcome the disadvantages mentioned above, we propose a new metric named Energy-Response time Weighted Product (ERWP), which combines the strengths of ERWS and ERP. It is defined as:

$$ERWP = \mathbb{E}[\mathcal{E}]^{\omega} \cdot \mathbb{E}[T]^{1-\omega}. \tag{9}$$

We can rewrite Eq. (9) as:

$$ERWP = e^{\omega\cdot\ln(\mathbb{E}[\mathcal{E}])+(1-w)\cdot\ln(\mathbb{E}[T])}. \tag{10}$$

Therefore, it inherits the characteristics of the ERWS metric that assigns different importance weights to energy and time, and has the advantage of being analytically tractable since the logarithmic expectation is additive over time.

Meanwhile, when $\omega = 0.5$, the mean energy consumption and mean response time have equal importance:

$$ERWP = \sqrt{\mathbb{E}[\mathcal{E}] \cdot \mathbb{E}[T]}, \tag{11}$$

which indicates that ERWP metric has the advantage of the ERP metric in being sensitive to difference of the scales of the component metrics.

To the best of our knowledge the ERWP metric has not been treated analytically before.

Based on the above analysis in the local execution and the remote execution for the offloadable jobs, we can derive the ERWP metric from Eq. (9) after combining the results in Eqs. (2)–(5), which is expressed by:

$$\begin{aligned} ERWP &= \Big\{\pi\mathbb{E}[\mathcal{E}_o] + (1-\pi)\mathbb{E}[\mathcal{E}_m]\Big\}^{\omega} \\ &\quad \cdot\Big\{\pi\mathbb{E}[T_o] + (1-\pi)\mathbb{E}[T_m]\Big\}^{1-\omega} \\ &= \frac{1}{\lambda}\Big\{\sum_{i=1}^{N}\mathbb{E}[P_i] + \mathbb{E}[P_m]\Big\}^{\omega} \\ &\quad \cdot\Big\{\sum_{i=1}^{N}\mathbb{E}[N_i] + \mathbb{E}[N_c] + \mathbb{E}[N_m]\Big\}^{1-\omega}. \end{aligned} \tag{12}$$

Similarly, we can derive the ERWS metric from Eq. (7) and ERP metric from Eq. (8) for the offloadable jobs.

III. Uninterrupted Offloading Strategy

In this section, we analyse the uninterrupted offloading strategy for remote execution. We formulate the queueing model based on the WiFi availability model, and then we use queueing analysis to derive the ERWP metric.

A. Problem Formulation

Figure 3 depicts an uninterrupted offloading strategy based on the WiFi network's availability model. The total cost for offloading a job is composed of the cost for sending the job to the cloud and idly waiting for the cloud to complete the job. We propose a Markov modulated queue for uplink transmission. A single-server queuing system that oscillates between two feasible states is denoted by f_{ON} and f_{OFF}. The persistence of the system at any state is governed by a random mechanism: if the system functions at state f_{ON} it tends 'to jump' to the other state with Poisson intensity ξ and if the system functions at state f_{OFF} it tends 'to jump' to the other state with Poisson intensity η [18].

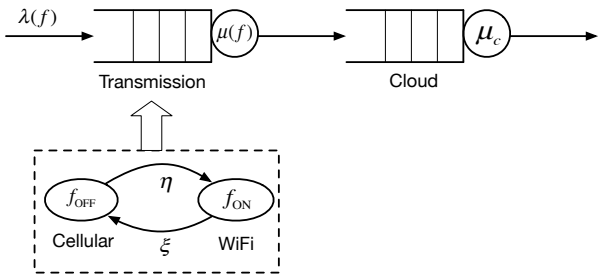


Figure 3. Uninterrupted offloading strategy with cellular and WiFi networks

From Fig. 3, the jobs are offloaded either via a cellular connection or a WiFi network to the cloud. We assume that the mean job size is $\mathbb{E}[X]$, the transmission speed of the cellular network is s_1 with service rate $\mu_1 = s_1/\mathbb{E}[X]$, and its operating power is p_1 when serving jobs and zero whenever idle. Similarly, WiFi runs at speed s_2, with service rate $\mu_2 = s_2/\mathbb{E}[X]$, and its operating power is p_2. We assume that jobs arrive according to

a Poisson process with rate $\lambda(f)$, and the modulating process $f \in \{f_{\text{ON}}, f_{\text{OFF}}\}$ determines the transition rates.

$$\lambda(f) = \begin{cases} \lambda_1, & \text{if } f = f_{\text{OFF}} \\ \lambda_2, & \text{if } f = f_{\text{ON}} \end{cases} \text{and} \, \mu(f) = \begin{cases} \mu_1, & \text{if } f = f_{\text{OFF}} \\ \mu_2, & \text{if } f = f_{\text{ON}} \end{cases}. \tag{13}$$

The original offloading jobs arrive at the system according to a Poisson process with rate λ_c, since the modulating process has two states, which results in independent Poisson processes with rate λ_i ($i \in \{1, 2\}$), we have $\lambda_1 + \lambda_2 = \lambda_c$.

It is well-known from the literature [19] that a product-form for the tandem queues between the modulated (the exponential queue) and the modulating processes exists if and only if the following condition holds:

$$\exists \rho \in R^+ \quad s.t. \, \forall f \in \{f_{\text{ON}}, f_{\text{OFF}}\}, \quad \frac{\lambda(f)}{\mu(f)} = \rho. \tag{14}$$

Then by substituting Eq. (13) into Eq. (14), we have:

$$\frac{\lambda_1}{\mu_1} = \frac{\lambda_2}{\mu_2}. \tag{15}$$

This is the case where the traffic intensities ρ_1 and ρ_2 are equal, though arrival intensities and service capacities need not be equal. Now this transition from one state to the second state will carry no influence on the random variable 'number of jobs present in the queue', since the traffic intensity $\lambda_i/\mu_i (= \rho)$ has not changed [20].

Figure 3 demonstrates that the uninterrupted offloading strategy is a conditional product-form model consisting of a tandem between a transmission queue (with two alternating states of cellular and WiFi) and an exponential queue representing the cloud. The former queue alternates its service by means of mutual resets according to the availability of WiFi, which is governed by an interrupted Poisson Process (IPP) with exponentially distributed ON-OFF periods. We model the intermittent availability of WiFi hotspots as a FCFS queue with occasional server break-down [7]. The queue works in mutual exclusive states and the reset occurs when the WiFi period alternates, either from ON-state to OFF-state or from OFF-state to ON-state. When WiFi becomes unavailable, the cellular network starts transmitting, otherwise when the WiFi connection becomes available, jobs are served only by the WiFi network. Specifically, offloading is not interrupted in this model, either in ON-state where the WiFi network is processing the existing jobs, or in the OFF-state during which the job is serving by the cellular network (the cellular connectivity is assumed to be always available). We assume that the sojourn time in a hotspot and the time to move from one hotspot to another are exponentially distributed with parameters ξ (failure rate), and η (recovery rate), respectively.

Since there is no waiting time before entering service, the $M/M/\infty$ queue of the cloud is occasionally referred to as a delay (sometimes pure delay) station, the probability distribution of the delay being that of the service time. Thus, the expected execution time taken in the cloud server can be calculated as $\mathbb{E}[T_c] = 1/\mu_c$. Since the application jobs are remotely executed in the cloud server rather than in the mobile device and we are only concerned with the energy consumption of the mobile device, we do not need to concern ourselves with calculating the energy consumption of the cloud server.

B. Metric-Based Analysis

We use queueing analysis to derive formulas for the average number of jobs in the $M/M/1$ queue. Given the previously stated assumptions, the uninterrupted offloading strategy can be modelled with a 2D Markov chain, as shown in Fig. 4.

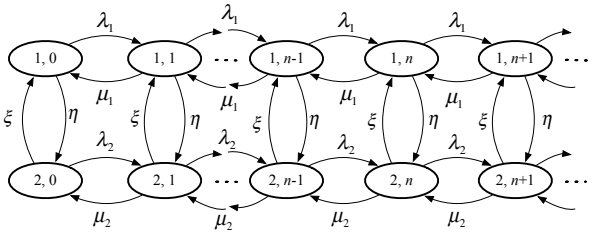


Figure 4. 2D Markov chain for the uninterrupted offloading strategy

The states with cellular network are denoted $\{1, n\}$, and the states with WiFi connectivity are denoted $\{2, n\}$. n corresponds to the number of jobs in the system (queuing and in service). Writing the balance equations for this chain gives:

$$\pi_{1,0}(\lambda_1 + \eta) = \pi_{1,1}\mu_1 + \pi_{2,0}\xi \tag{16a}$$

$$\pi_{2,0}(\lambda_2 + \xi) = \pi_{2,1}\mu_2 + \pi_{1,0}\eta \tag{16b}$$

$$\pi_{1,n}(\lambda_1 + \eta + \mu_1) = \pi_{1,n-1}\lambda_1 + \pi_{1,n+1}\mu_1 + \pi_{2,n}\xi \; (n > 0) \tag{16c}$$

$$\pi_{2,n}(\lambda_2 + \xi + \mu_2) = \pi_{2,n-1}\lambda_2 + \pi_{2,n+1}\mu_2 + \pi_{1,n}\eta \; (n > 0) \tag{16d}$$

The steady-state probability of finding the offloading system in some region with WiFi unavailability (with only cellular access) is $\pi_1 = \frac{\mathbb{E}[T_{\text{OFF}}]}{\mathbb{E}[T_{\text{ON}}]+\mathbb{E}[T_{\text{OFF}}]} = \frac{\xi}{\eta+\xi}$. Similarly, the steady-state probability for the periods with WiFi availability is $\pi_2 = \frac{\mathbb{E}[T_{\text{ON}}]}{\mathbb{E}[T_{\text{ON}}]+\mathbb{E}[T_{\text{OFF}}]} = \frac{\eta}{\eta+\xi}$.

Let two quantities λ^* and μ^* be defined as:

$$\lambda^* = \pi_1 \cdot \lambda_1 + \pi_2 \cdot \lambda_2, \tag{17}$$

$$\mu^* = \pi_1 \cdot \mu_1 + \pi_2 \cdot \mu_2 \,. \tag{18}$$

We define the probability generating functions for both Cellular and WiFi states as:

$$G_i(z) = \sum_{n=0}^{\infty} \pi_{i,n} z^n, \quad |z| \le 1, \forall i = 1, 2. \tag{19}$$

After some calculation, yield:

$$(\lambda_1+\eta+\mu_1)G_1(z) = \lambda_1 z G_1(z)+\xi G_2(z)+\frac{\mu_1}{z}[G_1(z)-\pi_{1,0}]+\pi_{1,0}\mu_1, \tag{20}$$

$$(\lambda_2+\xi+\mu_2)G_2(z) = \lambda_2 z G_2(z)+\eta G_1(z)+\frac{\mu_2}{z}[G_2(z)-\pi_{2,0}]+\pi_{2,0}\mu_2. \tag{21}$$

After solving the Eqs. (20) and (21), we have:

$$g(z)G_1(z) = \pi_{2,0}\xi\mu_2 z + \pi_{1,0}\mu_1\big[\xi z + \lambda_2 z(1-z) - \mu_2(1-z)\big],$$

where $g(z) = \lambda_1\lambda_2 z^3 - (\eta\lambda_2 + \xi\lambda_1 + \lambda_1\lambda_2 + \lambda_1\mu_2 + \lambda_2\mu_1)z^2 + (\eta\mu_2 + \xi\mu_1 + \mu_1\mu_2 + \lambda_1\mu_2 + \lambda_2\mu_1)z - \mu_1\mu_2$, and it is proven that $g(z)$ has only one root z_0 in the interval $(0, 1)$ [20].

After some algebraic manipulations, we obtain:

$$\pi_{1,0} = \frac{\xi(\mu^* - \lambda^*)z_0}{\mu_1(1-z_0)(\mu_2 - \lambda_2 z_0)}, \tag{22}$$

$$\pi_{2,0} = \frac{\eta(\mu^* - \lambda^*)z_0}{\mu_2(1-z_0)(\mu_1 - \lambda_1 z_0)}. \tag{23}$$

Once the values of $\pi_{1,0}$ and $\pi_{2,0}$ have been established, the probability generating functions can be calculated as:

$$G_1(z) = \frac{\xi(\mu^* - \lambda^*)z + \pi_{1,0}\mu_1(1-z)(\lambda_2 z - \mu_2)}{g(z)}, \quad (24)$$

$$G_2(z) = \frac{\eta(\mu^* - \lambda^*)z + \pi_{2,0}\mu_2(1-z)(\lambda_1 z - \mu_1)}{g(z)}. \quad (25)$$

By using $\mathbb{E}[N_i] = \sum_{n=0}^{\infty} n\pi_{i,n} = dG_i(z)/dz|_{z=1}$, we get the average number of jobs in the system [20]:

$$\begin{aligned}\mathbb{E}[N] &= \mathbb{E}[N_1] + \mathbb{E}[N_2] \\ &= \frac{\lambda^*}{(\mu^* - \lambda^*)} + \frac{\mu_1(\mu_2 - \lambda_2)\pi_{1,0} + \mu_2(\mu_1 - \lambda_1)\pi_{2,0}}{(\xi + \eta)(\mu^* - \lambda^*)} \\ &\quad - \frac{(\mu_1 - \lambda_1)(\mu_2 - \lambda_2)}{(\xi + \eta)(\mu^* - \lambda^*)}, \end{aligned} \quad (26)$$

which contains a root of third order equation, and thus is difficult to calculate. However, when taking the balance traffic condition in Eq. (15) into account, it can be further simplified.

In such a situation, let $\mu_1/\lambda_1 = \mu_2/\lambda_2 = \theta$, and then we substitute them into $\pi_{1,0}$, $\pi_{2,0}$ and $g(z)$, it is easy to prove that $\pi_{1,0}/\pi_{2,0} = \pi_1/\pi_2$ and $g(\theta) = 0$.

Therefore, we have the decomposition:

$$g(z) = \lambda_1\lambda_2(z - \theta)(z^2 - kz + \theta), \quad (27)$$

where $k = \eta/\lambda_1 + \xi/\lambda_2 + 1 + \theta$. The root of interest z_0, which is located in the interval $(0, 1)$, is equal to $z_0 = (k - \sqrt{k^2 - 4\theta})/2$.

Finally, we get $\pi_{i,0} = \pi_i \cdot (1 - 1/\theta)$ $(\forall i = 1, 2)$, then $\rho = 1 - (\pi_{1,0} + \pi_{2,0}) = \lambda^*/\mu^*$. By using induction, it is easy to prove $\pi_{i,n} = \pi_i \cdot (1 - \rho)\rho^n$ [20].

Therefore, the partial generating functions are derived as:

$$G_i(z) = \pi_i(1-\rho)\sum_{n=0}^{\infty}(z\rho)^n = \pi_i \cdot \frac{\mu^* - \lambda^*}{\mu^* - \lambda^* z}, \quad \forall i = 1, 2, \quad (28)$$

and then by using $\mathbb{E}[N_i] = \sum_{n=0}^{\infty} n\pi_{i,n} = dG_i(z)/dz|_{z=1}$, we obtain:

$$\mathbb{E}[N_1] = \pi_1 \cdot \frac{\lambda^*}{\mu^* - \lambda^*}, \quad (29)$$

$$\mathbb{E}[N_2] = \pi_2 \cdot \frac{\lambda^*}{\mu^* - \lambda^*}. \quad (30)$$

The mean number of jobs in cloud queue is calculated as:

$$\mathbb{E}[N_c] = \frac{\lambda_c}{\mu_c}. \quad (31)$$

Since $\Pr\{e_i = 1\} = \pi_i$, according to Eq. (6), we have:

$$\Pr\{N_i > 0, e_i = 1\} = \rho_i \cdot \pi_i, \quad \forall i = 1, 2. \quad (32)$$

Further, by substituting Eqs. (29)–(32) into Eq. (12), we can formulate the explicit expressions of the ERWP metric for the uninterrupted offloading strategy.

IV. Interrupted Offloading Strategy

In this section, we discuss the interrupted offloading strategy for remote execution. First, we formulate the queueing model with cellular and WiFi networks, and then we use queueing analysis to derive the ERWP metric based on an $M/M/1$ queue and an $M/M/1$ queue with intermittent server availability.

A. Problem Formulation

The interface selection problem in offloading systems is modelled as the decision to which queue an arriving job should be assigned. In this decision the offloading dispatcher takes both performance and energy costs into account. This seems natural for heterogeneous servers where a job needs a different amount of energy and time to be served by different servers. For example, whereas assigning each job to a low power server would be beneficial from the energy consumption perspective at low loads, such a strategy may end up in difficulties at higher loads as the response time increases rapidly [21]. Indeed, the energy-performance tradeoff takes on different explicit expressions under different offloading strategies.

As shown in Fig. 5, we consider a queuing model that consists of two parallel queues of cellular and WiFi with work conserving queuing disciplines. λ_1 and λ_2 are the mean rates of jobs into Queue 1 and Queue 2. Since the original offloading jobs arrive at the system according to a Poisson process with rate λ_c, one strategy would be random assignment into Queue i ($i \in \{1, 2\}$), which results in independent Poisson processes with rate λ_i, and we have $\lambda_1 + \lambda_2 = \lambda_c$.

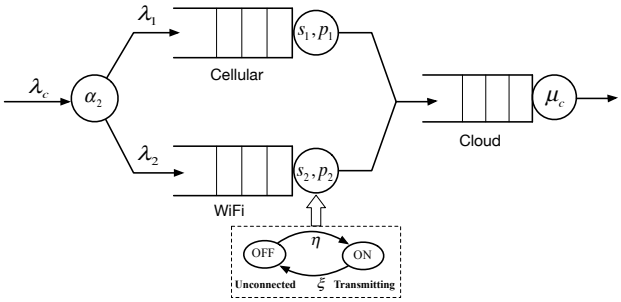


Figure 5. Interrupted offloading strategy with cellular and WiFi networks

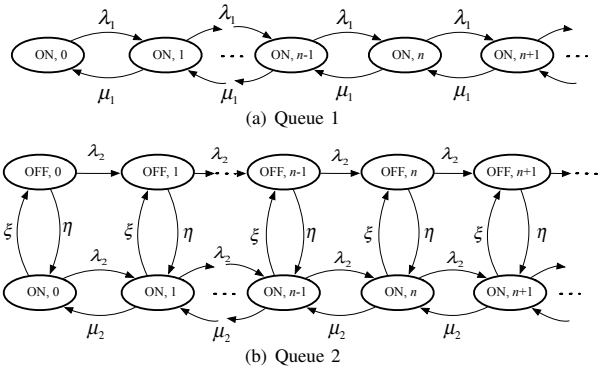
The two queues have the following behavior:

- **Queue 1**: When a job is offloaded to the cloud via a cellular network, there is queueing due to the transmission speed of the cellular link. Costs arise in terms of transmission delays (queueing and actual transmission time) and transmission energy consumption. Server 1 is always available since the cellular connection is always on.
- **Queue 2**: When a job is offloaded to the cloud via a WiFi network, there is queueing due to the transmission speed of the WiFi link. We model the intermittent availability of hotspots as a FCFS queue with occasional server breakdown. The availability of Server 2 is governed by an IPP with exponentially distributed ON-OFF periods. Specifically, the server is either in ON-state processing the existing jobs, or in OFF-state during which no job receives service. We can assume that the service station fails from time to time and resumes its operation after a random time. When a server recovers, it continues to serve the customer whose service has been interrupted, i.e. the work already completed is not lost (cf. data transfers with resume).

Similarly, the cloud queue is a pure delay station at which jobs spend an exponentially distributed amount of time with mean equal to $1/\mu_c$ time units.

B. Metric-Based Analysis

The Markov chains for the interrupted offloading strategy in Fig. 5 can be decomposed into two separate chains, as shown in Fig. 6.



(a) Queue 1

(b) Queue 2

Figure 6. Markov chains for interrupted offloading strategy with cellular and WiFi networks

1) Queue 1: the Markov chain is depicted in Fig. 6(a), and the expected number of jobs can be calculated as:

$$\mathbb{E}[N_1] = \frac{\rho_1}{1-\rho_1}, \tag{33}$$

where the workload for Queue 1 during the busy period is $\rho_1 = \lambda_1/\mu_1 = \lambda_1 \mathbb{E}[X]/s_1$.

2) Queue 2: refers to offloading jobs from the mobile device to the cloud via a WiFi network, which is modelled as an $M/M/1$-FCFS queue with intermittently available service. The Markov chain is depicted in Fig. 6(b), which is equivalent to assuming that $\lambda_1 = \lambda_2$ and $\mu_1 = 0$ in Fig. 4.

Writing the balance equations for this chain gives:

$$\pi_{\text{OFF},0}(\lambda_2 + \eta) = \pi_{\text{ON},0}\xi \tag{34a}$$

$$\pi_{\text{ON},0}(\lambda_2 + \xi) = \pi_{\text{OFF},0}\eta + \pi_{\text{ON},1}\mu_2 \tag{34b}$$

$$\pi_{\text{OFF},n}(\lambda_2 + \eta) = \pi_{\text{OFF},n-1}\lambda_2 + \pi_{\text{ON},n}\xi \tag{34c}$$

$$\pi_{\text{ON},n}(\lambda_2 + \xi + \mu_2) = \pi_{\text{ON},n-1}\lambda_2 + \pi_{\text{ON},n+1}\mu_2 + \pi_{\text{OFF},n}\eta \tag{34d}$$

After simple algebraic operations of equations in Eq. (34) yields:

$$(\pi_{\text{OFF},n} + \pi_{\text{ON},n}) \cdot \lambda_2 = \pi_{\text{ON},n+1}\mu_2 \ (n = 0, 1, 2, \dots). \tag{35}$$

Summation of Eq. (35) over all n, we obtain:

$$\pi_{\text{ON},0} = \pi_{\text{ON}} - \frac{\lambda_2}{\mu_2}, \tag{36}$$

where $\pi_{\text{ON},0} = \pi_{2,0}$, $\pi_{\text{ON}} = \pi_2$ and $\pi_{\text{OFF}} = \pi_1$.

According to Eqs. (17) and (18), we have $\lambda^* = \lambda_2$ and $\mu^* = \pi_2\mu_2$. After substituting the above values in Eq. (26), we derive the mean number of jobs in Queue 2 as:

$$\begin{aligned}\mathbb{E}[N_2] &= \mathbb{E}[N_{\text{OFF}}] + \mathbb{E}[N_{\text{ON}}] \\ &= \frac{\lambda^*}{(\mu^* - \lambda^*)} + \frac{\mu_2\lambda_2\pi_{2,0} - (-\lambda_2)(\mu_2 - \lambda_2)}{(\xi+\eta)(\mu^* - \lambda^*)} \\ &= \frac{\lambda_2}{\pi_2\mu_2 - \lambda_2} + \frac{\pi_1\lambda_2\mu_2}{(\pi_2\mu_2 - \lambda_2)(\xi+\eta)}.\end{aligned} \tag{37}$$

3) Optimization: Since Server 1 is always available, we have $\Pr\{e_1 = 1\} = 1$ and the fraction of time that Server 2 is available to process jobs is: $\Pr\{e_2 = 1\} = \frac{\eta}{\xi+\eta} = \pi_2$, where as the recovery rate $\eta \to \infty$, the availability of Server 2 tends to be 1. Then we have:

$$\Pr\{N_1 > 0, e_1 = 1\} = \rho_1, \tag{38}$$

$$\Pr\{N_2 > 0, e_2 = 1\} = \rho_2 \cdot \pi_2. \tag{39}$$

Further, by substituting Eqs. (33), (37)–(39) into Eq. (12), we can formulate the optimization of the ERWP metric for the offloading assignment as:

$$\lambda_i^{\min} = \arg\min_{\lambda_i} ERWP. \tag{40}$$

We seek the arrival rate $\lambda_i^{\min}$ to Queue i such that ERWP is minimised when both queues are in operation. We can apply Newton's method to find $\lambda_i^{\min}$ iteratively [22].

In other words, arriving jobs are assigned to Queue 1 and Queue 2 according to the optimized objective defined in Eq. (40), minimizing the ERWP metric.

V. Performance Evaluation

A. A Realistic Offloading Scenario

Different wireless network interfaces vary in many ways, which we have to capture in simplified form in just few parameters. Cellular networks such as EDGE and 3G, usually have much higher availability than WiFi, but the transmission rate of WiFi is higher. Besides, the WiFi interface is more energy-efficient than the cellular interface [8]. These imply that WiFi is much faster and more energy-efficient than the cellular interface for transmitting the same quantity of data. Therefore, we consider here a simple scenario where the transmission rate of the cellular network is smaller than that of WiFi, i.e. $s_1 < s_2$ and the power consumption when transmitting jobs via the cellular link is larger than the WiFi link, i.e. $p_1 > p_2$.

Using measurements from real traces in [15], the average data rates for the cellular network and WiFi are set as $s_1 = 800$ Kbps and $s_2 = 2$ Mbps, respectively. The mean job size is 125 KB. According to the power models developed in [23], we set the power coefficients $p_1 = 2.5$ W, $p_2 = 0.7$ W and $p_m = 2$ W, respectively. Besides, suppose that the total job arrival rate for offloading is $\lambda = 0.6$ packet/s, the mobile service rate $\mu_m = 2$ tasks/s, the cloud service rate $\mu_c = 5$ tasks/s and both the failure rate ξ and recovery rate η of Server 2 are equal to 1.

B. Performance Analysis

We first analyse the case when the offloading probability $\pi = 0.5$, indicating that half of the offloadable jobs are offloaded to the cloud, while the rest are executed on the mobile device.

From Fig. 7, we can observe that the uninterrupted offloading strategy performs significantly better than the interrupted one when ω is small, but as ω approaches to 1, the interrupted strategy performs much better. This means that when considering energy consumption more important than response time (for delay-tolerance applications), it is better to use the interrupted strategy; otherwise when considering response time more important (for delay-sensitive applications), the uninterrupted strategy is much more preferred, which fully uses the unavailable periods of WiFi by transmitting with a cellular network. Since energy

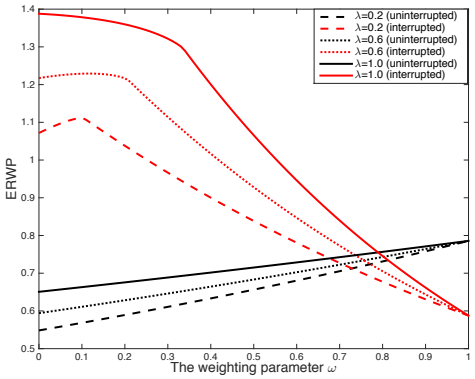


Figure 7. Comparison of two offloading strategies under arrival rate λ

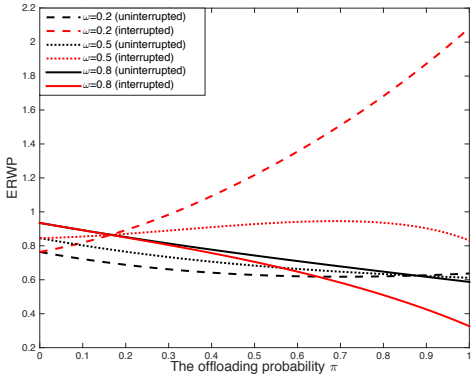


Figure 10. Comparison of two offloading strategies under ω parameter

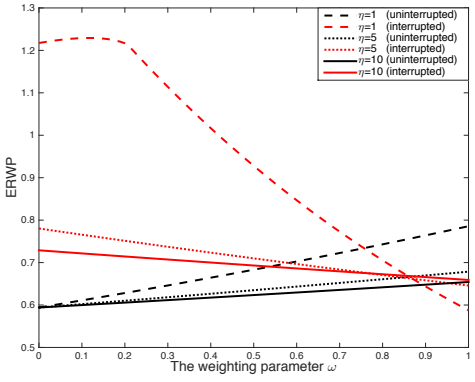


Figure 8. Comparison of two offloading strategies under recovery rate η

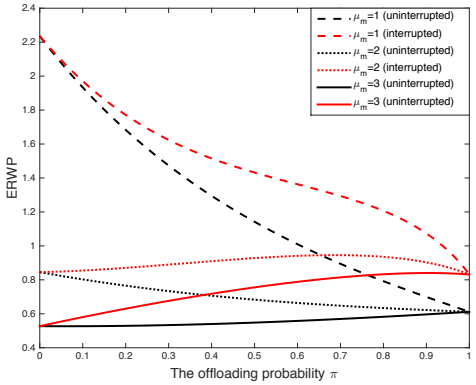


Figure 11. Comparison of two offloading strategies under different mobile service rate μ_m

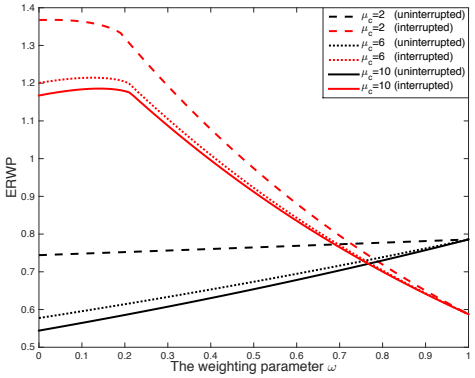
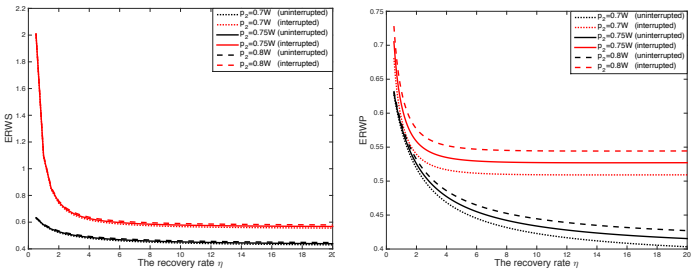


Figure 9. Comparison of two offloading strategies under cloud service rate μ_c



(a) ERWS (b) ERWP

Figure 12. Comparison of of two offloading strategies based on ERWS and ERWP metrics

is measured per job, when the weight is on energy consumption only the metric is for both strategies insensitive to the job arrival rate. As the arrival rate of the offloadable jobs λ increases, none of the offloading strategies can achieve a low ERWP value. However, the uninterrupted strategy varies less. The interrupted strategy is more sensitive to the job arrival rate.

Similar observations can be made from Fig. 8 or sensitivity to the recovery rate η. The interrupted strategy suffers more from long repair times than the uninterrupted strategy. This is reasonable, due to the lower WiFi availability, resulting in most of the jobs being offloaded through the slower and more energy consuming cellular network interface. When $\omega < 0.85$, (i.e. response time is more important) the interrupted strategy also performs much better as η increases. The reason is that arriving jobs to the WiFi queue have a higher probability to be offloaded to the cloud. However, with more importance being given to the energy consumption, this strategy performs much worse as η increases and the down-times become less.

In Fig. 9, it is observed that the faster the cloud serves, the lower the ERWP value is. The cloud service rate μ_c has a great influence on the mean response time since we have to wait for the cloud service to be finished. But the cloud service rate has little influence on the energy consumption since the jobs are processed remotely rather than locally on the mobile device.

Then, we dynamically change the offloading probability π to find the optimal offloading decision.

As shown in Fig. 10, when ω is small ($\omega = 0.2$), it is not worth offloading any job in the interrupted strategy, while it is better to offload half of the offloadable jobs to the cloud

in the uninterrupted strategy. However, with more focus on the energy consumption (ω approaches to 1), it is better to offload all the jobs for both strategies; meanwhile the interrupted offloading strategy obtains lower values for the metrics than the uninterrupted one.

The ERWP metric can be treated as the ERP metric when we set the weighting parameter $\omega = 0.5$, i.e. when both the energy consumption and the response time have the equal importance. From Fig. 11, it is observed that the uninterrupted strategy should always be preferred. When the mobile service rate μ_m is very small, it is worth offloading all the jobs to the cloud with both offloading strategies. Since the local execution is very slow it is beneficial to offload all jobs. However, when μ_m reaches some value, it is better not to offload since the cost saved from remote execution is not enough to cover the extra cost for offloading.

In Fig. 12 we compare the ERWP metric with the ERWS metric. When the power p_2 changes slightly, e.g. from 0.7 W to 0.8 W, it is difficult to tell the difference between the two offloading strategies according to the ERWS metric depicted in Fig. 12(a), while it is very clear to see the difference according to the ERWP metric shown in Fig. 12(b). It seems that the ERWP metric is much more sensitive to the large scale and can capture small changes when the ERWS metric has the disadvantage of a linear combination of two criteria on different scales.

VI. Conclusions

In this paper, we have developed two offloading strategies to leverage the complementary strength of WiFi and cellular networks. We have formulated queueing models to carry out optimality analysis of the energy-performance tradeoff for mobile cloud offloading systems based on the ERWP metric, which captures both energy and performance metrics and also intermittently available access links. The ERWP metric combines the advantages of both an additive metric (ERWS) and a product metric (ERP), i.e. it not only assigns different importance weights to energy consumption and response time, but also is insensitive to criteria on different scales.

We find that for delay-tolerant applications, it is better to use the interrupted strategy instead of the uninterrupted one, while for delay-sensitive applications, the uninterrupted strategy shows very good results and outperforms the interrupted offloading one by a significant margin. The offloading probability closely depends on the mobile service rate, the cloud service rate and the weighting parameter. Slow mobile service rate and fast cloud service rate will result in more jobs to be offloaded to the cloud. We can thus judiciously make the offloading decisions of whether to offload or not and how much to offload that optimise the ERWP metric.

The proposed queueing model can be used to describe complex and realistic offloading systems. So far, the assumption that the WiFi data rate and power consumption remain constant in all the regions within WiFi coverage is somehow unrealistic. Therefore, it is worth considering scenarios where they might be different at each connected access point in the future.

References

[1] H. T. Dinh, C. Lee, D. Niyato, and P. Wang, "A survey of mobile cloud computing: architecture, applications, and approaches," *Wireless communications and mobile computing*, vol. 13, no. 18, pp. 1587–1611, 2013.

[2] M.-R. Ra, J. Paek, A. B. Sharma, R. Govindan, M. H. Krieger, and M. J. Neely, "Energy-delay tradeoffs in smartphone applications," in *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pp. 255–270, ACM, 2010.

[3] H. Wu, Q. Wang, and K. Wolter, "Tradeoff between performance improvement and energy saving in mobile cloud offloading systems," in *Communications Workshops (ICC), 2013 IEEE International Conference on*, pp. 728–732, IEEE, 2013.

[4] H. Wu and K. Wolter, "Dynamic transmission scheduling and link selection in mobile cloud computing," in *Analytical and Stochastic Modeling Techniques and Applications*, pp. 61–79, Springer, 2014.

[5] H. Wu and K. Wolter, "Tradeoff analysis for mobile cloud offloading based on an additive energy-performance metric," in *Performance Evaluation Methodologies and Tools (VALUETOOLS), 2014 8th International Conference on*, ACM, 2014.

[6] A. Rahmati and L. Zhong, "Context-for-wireless: context-sensitive energy-efficient wireless data transfer," in *Proceedings of the 5th international conference on Mobile systems, applications and services*, pp. 165–178, ACM, 2007.

[7] E. Hyytiä, T. Spyropoulos, and J. Ott, "Offload (only) the right jobs: Robust offloading using the markov decision processes," in *IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, (Boston, MA, USA), Jun. 2015. to appear.

[8] P. Shu, F. Liu, H. Jin, M. Chen, F. Wen, Y. Qu, and B. Li, "etime: energy-efficient transmission between cloud and mobile devices," in *INFOCOM, 2013 Proceedings IEEE*, pp. 195–199, IEEE, 2013.

[9] F. Mehmeti and T. Spyropoulos, "Performance analysis of "on-the-spot" mobile data offloading," in *Global Communications Conference (GLOBECOM), 2013 IEEE*, pp. 1577–1583, IEEE, 2013.

[10] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pp. 49–62, ACM, 2010.

[11] K. Kumar, J. Liu, Y.-H. Lu, and B. Bhargava, "A survey of computation offloading for mobile systems," *Mobile Networks and Applications*, vol. 18, no. 1, pp. 129–140, 2013.

[12] A. Papoulis and S. U. Pillai, *Probability, random variables, and stochastic processes*. Tata McGraw-Hill Education, 2002.

[13] V. Cardellini, V. De Nito Personé, V. Di Valerio, F. Facchinei, V. Grassi, F. Lo Presti, and V. Piccialli, "A game-theoretic approach to computation offloading in mobile cloud computing," tech. rep., Technical Report, available online at http://www. optimizationonline. org/DB HTML/2013/08/3981. html, 2013.

[14] A. Wierman, L. L. Andrew, and A. Tang, "Power-aware speed scaling in processor sharing systems," in *INFOCOM 2009, IEEE*, pp. 2007–2015, IEEE, 2009.

[15] K. Lee, J. Lee, Y. Yi, I. Rhee, and S. Chong, "Mobile data offloading: How much can wifi deliver?," *Networking, IEEE/ACM Transactions on*, vol. 21, no. 2, pp. 536–550, 2013.

[16] Y.-W. Kwon and E. Tilevich, "Energy-efficient and fault-tolerant distributed mobile execution," in *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*, pp. 586–595, IEEE, 2012.

[17] A. Gandhi, V. Gupta, M. Harchol-Balter, and M. A. Kozuch, "Optimality analysis of energy-performance trade-off for server farm management," *Performance Evaluation*, vol. 67, no. 11, pp. 1155–1171, 2010.

[18] S. Balsamo, G.-L. dei Rossi, and A. Marin, "Queueing networks and conditional product-forms," in *Proceedings of the 7th International Conference on Performance Evaluation Methodologies and Tools*, pp. 204–213, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2013.

[19] J.-M. Fourneau, B. Plateau, and W. Stewart, "Product form for stochastic automata networks," in *Proceedings of the 2nd international conference on Performance evaluation methodologies and tools*, pp. 32–, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2007.

[20] U. Yechiali and P. Naor, "Queuing problems with heterogeneous arrivals and service," *Operations Research*, vol. 19, no. 3, pp. 722–734, 1971.

[21] A. Penttinen, E. Hyytiä, and S. Aalto, "Energy-aware dispatching in parallel queues with on-off energy consumption," in *Performance Computing and Communications Conference (IPCCC), 2011 IEEE 30th International*, pp. 1–8, IEEE, 2011.

[22] I. Tsimashenka and W. J. Knottenbelt, "Trading off subtask dispersion and response time in split-merge systems," in *Analytical and Stochastic Modeling Techniques and Applications*, pp. 431–442, Springer, 2013.

[23] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani, "Energy consumption in mobile phones: a measurement study and implications for network applications," in *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, pp. 280–293, ACM, 2009.