Université de Montréal

Conception et implantation d'une bibliothèque pour la simulation de centres de contacts

par
Eric Buist

Département d'informatique et recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en d'informatique et recherche opérationnelle

Août, 2005

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé:

Conception et implantation d'une bibliothèque pour la simulation de centres de contacts

présenté par:

Eric Buist

a été évalué par un jury composé des personnes suivantes:

Guy Lapalme,	président-rapporteur
Pierre L'Écuyer,	directeur de recherche
Yann-Gaël Guéhéneuc,	membre du jury

Mémoire accepté le: 22 septembre 2005

RÉSUMÉ

ContactCenters est une bibliothèque que nous avons développée et pour construire des simulateurs de centres de contacts en utilisant le langage de programmation Java et la bibliothèque de simulation SSJ. Cet outil supporte des centres de contacts en mode mixte avec plusieurs types de contacts et groupes d'agents, des politiques de routage et de composition d'appels sortants complexes et divers processus d'arrivées pour les contacts entrants. Chaque contact étant représenté par une entité distincte pendant la simulation, le programmeur dispose d'une flexibilité maximale pour ce qui est du routage et des calculs statistiques. ContactCenters peut aider le programmeur à implanter un simulateur de centres de contacts complexe en réduisant la quantité de code à écrire et de bogues à corriger. Grâce aux optimisations des machines virtuelles Java contemporaines, un programme ContactCenters devrait être plus performant qu'un modèle équivalent construit sous l'un des logiciels à interface graphique disponibles sur le marché et se fondant sur un langage complètement interprété et peu répandu. L'utilisation de la bibliothèque permet de disposer de toute la puissance de la plate-forme Java et d'interagir avec d'autres outils logiciels d'une façon simple et portable.

Dans ce mémoire, nous présentons l'architecture générale de la bibliothèque en décrivant les différentes composantes disponibles et une interface de haut niveau permettant à des programmes d'optimisation ou d'analyse statistique d'accéder à un simulateur ContactCenters ou à une implantation de formule d'approximation d'une façon transparente. Nous enchaînons avec un exemple de simulateur complet utilisant la bibliothèque ainsi qu'un exemple d'utilisation de l'interface de haut niveau. Nous comparons également les performances de ContactCenters avec celles d'Arena Contact Center Edition de Rockwell, un logiciel commercial largement utilisé dans l'industrie. Finalement, nous proposons quelques extensions pour SSJ, un outil utilisé par ContactCenters pour gérer la simulation.

Mots-clés : centres d'appels, Java, application SSJ, génie logiciel

ABSTRACT

ContactCenters is a library we developed for writing contact center simulators using the Java programming language and the SSJ simulation library. It supports multi-skill and blend contact centers with complex routing and dialing policies as well as various arrival processes for inbound contacts. Each contact being represented by its own entity during simulation, the programmer benefits from a maximal flexibility, especially for routing and statistical collecting. The library can help the programmer in developing complex contact center simulators. It can reduce the amount of code to be written as well as debugging time. Thanks to optimizations in modern Java virtual machines, we expect ContactCenters simulators to be faster than equivalent models constructed using one of the commercially available graphical user interface-based simulation systems using fully-interpreted and not widely-used programming languages. Using the library gives access to the power and richness of the Java platform, permitting the interaction with many third-party libraries with simplicity and portability.

In this thesis, we present the architecture of the library by describing its components as well as an high-level interface permitting external programs to use a ContactCenters simulator or an approximation formula transparently, for optimization or statistical analysis. We then give a complete example of a simulator using the library, and a second example using the high-level interface. We also compare the performance of ContactCenters with Rockwell's Arena Contact Center Edition, a widely-used simulation program. Finally, we propose some extensions for SSJ, a simulation tool used by ContactCenters.

Keywords: call centers, Java, SSJ application, software engineering

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES CODES SOURCE

LISTE DES NOTATIONS ET DES SYMBOLES

Acronymes

ACD Automatic Call Distributor, effectue le routage dans un centre de contacts

DTD Document Type Definition, format permettant de définir le contenu permis dans un type particulier de document XML

i.i.d. Indépendant et identiquement distribué, caractérise un ensemble de variables aléatoires dont la loi de probabilité de chaque élément est identique et indépendante des autres éléments.

IVR Interactive Voice Response, système de traitement automatisé d'appels téléphoniques

JNI Java Native Interface, interface permettant à des applications Java d'interagir avec des programmes écrits dans d'autres langages de programmation comme C/C++

JRE Java Runtime Environment, permet d'exécuter des applications Java

JVM Java Virtual Machine, machine virtuelle capable d'interpréter les classes Java et compiler le byte-code afin de l'exécuter

PABX Private Automatic Branch eXchange, commutateur par lequel tous les appels téléphoniques transitent avant d'atteindre un centre d'appels

SSJ Stochastic Simulation in Java, bibliothèque utilisée par ContactCenters pour effectuer la simulation

VBA Visual Basic for Applications, langage de programmation intégré à divers produits pour permettre l'écriture de macros par l'utilisateur

XML eXtensible Markup Language, méta-langage permettant de représenter des données complexes et hiérarchiques

Symboles mathématiques

ε_i Facteur d'efficacité des agents dans le groupe i pendant toute la journée

$\varepsilon_i(t_1,t_2)$ Efficacité des agents dans le groupe i pendant l'intervalle de temps $[t_1,t_2]$

$g_1(s), g_2(s)$ Niveau de service

i Indice d'un groupe d'agents

I Nombre de groupes d'agents

k Indice d'un type de contact

K Nombre de types de contact

m Nombre de lots dans une simulation sur horizon infini

n Taille d'un échantillon produit par simulation

$N_i(t)$ Nombre total d'agents dans un groupe i au temps t

$N_{\mathrm{b},i}(t)$ Nombre d'agents occupés (*busy*) dans un groupe i au temps t

$N_{\mathrm{b},i,k}(t)$ Nombre d'agents occupés dans un groupe i à servir des contacts de type k, au temps t

$N_{\mathrm{f},i}(t)$ Nombre d'agents libres (*free*) pour servir un contact dans un groupe i au temps t

$N_{\mathrm{g},i}(t)$ Nombre d'agents fantômes (*ghost*) du groupe i, c'est-à-dire nombre d'agents devant quitter le système après avoir terminé le service d'un contact en cours, au temps t

$N_{\mathrm{i},i}(t)$ Nombre d'agents inoccupés dans un groupe i, pouvant ou non servir des contacts au temps t

$o_i(t_1,t_2)$ Taux d'occupation des agents du groupe i pendant l'intervalle de temps $[t_1,t_2]$

$o_{i,k}(t_1,t_2)$ Taux d'occupation des agents du groupe i servant des contacts de type k, pendant l'intervalle de temps $[t_1,t_2]$

p Indice d'une période, entre 0 et $P+1$

P Nombre de périodes principales

s Temps d'attente acceptable pour tous les contacts

$s_{k,\cdot}$ Temps d'attente acceptable pour les contacts de type k

$s_{\cdot,p}$ Temps d'attente acceptable pendant la période p

$s_{k,p}$ Temps d'attente acceptable pour les contacts de type k pendant la période p

t Temps de simulation

t_p Temps de fin de la période p

T Temps de fin de la simulation

W_{X} Somme des temps d'attente des contacts servis

W_{Y} Somme des temps d'attente des contacts ayant abandonné

X Nombre de contacts servis

$X_{\mathrm{b}}(s)$ Nombre de contacts servis après un temps d'attente supérieur ou égal à s (*bad contacts*)

$X_{\mathrm{g}}(s)$ Nombre de contacts servis après un temps d'attente inférieur à s (*good contacts*)

Y Nombre d'abandons

$Y_{\mathrm{b}}(s)$ Nombre d'abandons après un temps d'attente supérieur ou égal à s

$Y_{\mathrm{g}}(s)$ Nombre d'abandons après un temps d'attente inférieur à s

REMERCIEMENTS

Je remercie tout d'abord Pierre L'Écuyer pour m'avoir proposé ce projet et m'avoir soutenu en tant que directeur de recherches tout au long de sa réalisation. Je le remercie pour ses nombreuses suggestions qui ont contribué à améliorer la bibliothèque, sa documentation et ce mémoire, ainsi que pour l'aide financière qu'il m'a offerte durant toute la durée de ma maîtrise.

Je remercie Bell Canada et le Conseil de Recherche en Sciences Naturelles et en Génie (CRSNG) pour avoir contribué au financement d'un projet de recherche sur les centres d'appels dans lequel s'inscrit ce mémoire.

Je remercie Athanassios Avramidis, Mehmet Tolga Ceẓik et Wyean Chan pour avoir utilisé la bibliothèque pendant son développement et proposé diverses améliorations.

Je remercie également mes parents, Pauline et Réal, pour avoir pris soin de mon éducation sans laquelle ce mémoire n'aurait pas pu être écrit.

CHAPITRE 1

INTRODUCTION

L'importance économique des centres de contacts a déjà clairement été démontrée [10]. De tels centres constituent l'interface entre une entreprise et sa clientèle. Ils doivent répondre à un nombre croissant de requêtes et constituent une importante source de revenu.

Un *centre d'appels* [12, 25] constitue un ensemble de ressources telles que des lignes téléphoniques, des commutateurs, des routeurs, des employés et des ordinateurs servant d'interface de communication entre une entreprise et des clients. Puisque les clients apprécient de pouvoir communiquer par le biais de plusieurs médias, ces centres se sont généralisés, donnant naissance aux *centres de contacts*. Désormais, les clients peuvent communiquer en utilisant le téléphone, la télécopie, le courrier et l'Internet.

Un *contact* consiste en une requête de communication entre un client et une entreprise. Les contacts *entrants* sont générés par des clients tentant d'entrer en communication pour obtenir un service tel qu'une réservation ou du support technique. Les contacts *sortants* sont initiés de façon proactive par les employés ou, dans le cas des appels téléphoniques, par un système spécialisé appelé *composeur*. Les communications sortantes permettent par exemple la vente à distance ainsi que le rappel de clients. Les centres capables de traiter les deux types de contacts sont dits *mixtes*.

Afin de simplifier le routage, les contacts entrants et sortants sont classés selon la raison pour laquelle une communication est établie. Cette raison est représentée sous la forme d'une valeur numérique k entre 0 et $K-1$, où K est le nombre total de types de contacts supportés par un système particulier. Le type de contact est déterminé en utilisant sa provenance (numéro de téléphone de l'appelant, site Web utilisé, etc.) et le système de traitement automatisé.

Le *service* d'un contact constitue un traitement destiné à satisfaire la requête d'un client. De nos jours, plusieurs requêtes peuvent être traitées entièrement par des systèmes automatisés, mais parfois, un client peut manifester le besoin ou le désir de parler à un

être humain. Le service comprend la phase de traitement automatique, le travail d'un employé pendant la communication avec le client et le travail que l'employé doit parfois effectuer après le dialogue.

Chaque employé, aussi appelé *agent*, fait partie d'un groupe $i \in \{0, \ldots, I-1\}$ définissant ses compétences. Il possède également certaines particularités qui peuvent affecter son efficacité et son horaire de travail. Au temps t de la journée, le groupe i contient $N_i(t)$ membres connectés dont $N_{\mathrm{b},i}(t)$ sont en train de servir des contacts, $N_{\mathrm{f},i}(t)$ sont libres et disponibles pour de nouveaux services et $N_i(t) - N_{\mathrm{b},i}(t) - N_{\mathrm{f},i}(t)$ sont inoccupés mais non disponibles. Le nombre d'agents $N_i(t)$ est souvent plus petit que le nombre planifié en raison de retards, de pauses prolongées, etc.

Un client peut être servi par plusieurs agents avant d'obtenir satisfaction. Par exemple, un réceptionniste peut demander à parler au département de comptabilité avant de pouvoir répondre à la requête d'un client. Un utilisateur éprouvant des problèmes techniques avec un logiciel peut devoir parler à différents techniciens avant d'obtenir une solution. Un *retour* se produit lorsqu'un client servi doit recontacter l'entreprise pour obtenir un nouveau service ou tenter de nouveau de satisfaire sa requête initiale.

Dans le cas de communications différées comme les courriers électroniques, le service peut même être préemptif, c'est-à-dire qu'un agent peut arrêter de traiter un contact pour se charger d'une tâche plus prioritaire. Par exemple, un agent en train de répondre à un courrier électronique peut mettre son message de côté pour traiter un appel téléphonique. Lorsque l'appel est terminé, l'agent reprend sa tâche initiale si cette tâche n'a pas déjà été confiée à un autre employé. Ainsi, en raison des retours et du service préemptif, les agents traitent parfois plusieurs clients simultanément.

Il arrive souvent qu'un client ne peut être servi immédiatement et doit attendre en file. Un tel client peut devenir impatient et décider d'abandonner, quittant le système sans recevoir de service. Dans ce cas, il peut tenter de recontacter le centre plus tard ou abandonner bel et bien, selon l'importance du service qu'il désire recevoir. Certains centres de contacts permettent également de laisser un message dans le but d'être rappelé ultérieurement. Dans le cas d'un appel téléphonique, un client peut également recevoir un signal occupé et être *bloqué* sans pouvoir attendre en file ou être servi.

1.1 Éléments constitutifs d'un centre de contacts

Dans un centre de contacts, la communication passe par un certain nombre de canaux et est gérée par des systèmes de traitement automatisés. Lorsqu'un client doit contacter un agent, un routeur est utilisé pour établir la connexion. Nous examinons maintenant de plus près les éléments fondamentaux constituant un centre de contacts.

1.1.1 Canaux de communication

La communication s'effectue toujours à l'aide d'un certain nombre de canaux dont la nature dépend du médium utilisé. Lorsqu'un client tente de joindre un centre d'appels téléphoniques, la compagnie de téléphone le connecte à un commutateur appelé *Private Automatic Branch eXchange* (PABX). Le nombre de lignes reliant la compagnie téléphonique au PABX étant limité, certains appels peuvent arriver à un moment où toutes les lignes sont occupées. Dans un système réel, il est difficile d'obtenir le nombre d'appels ainsi bloqués, car aucun appareil ne permet de les compter. Un centre de contacts fournit parfois plusieurs banques distinctes de lignes téléphoniques dont certaines sont dédiées à des appels prioritaires ou à des télécopies tandis que d'autres sont utilisées pour des appels ordinaires. L'identification de la banque de lignes téléphoniques à utiliser s'effectue grâce au numéro de téléphone composé par le client ou le numéro d'où provient l'appel. Un coût de location fixe ou horaire peut être associé à ces lignes, qu'elles soient occupées ou libres.

Parfois, le coût d'une ligne occupée est supérieur à celui d'une ligne libre, par exemple pour des numéros 1-800 gratuits pour les clients. Dans ce cas, les gestionnaires de centres d'appels doivent éviter que des clients attendent trop longtemps. Dans cette optique, certains équipements permettent de faire varier le nombre de lignes disponibles pendant la journée, autorisant les gestionnaires à imposer davantage d'appels bloqués que d'attentes prolongées.

Dans le cas des médias Internet, il est raisonnable de considérer que le nombre de canaux de communication est illimité puisque la transmission d'un courrier électronique est très rapide comparativement au traitement d'un appel téléphonique.

1.1.2 Système de traitement automatisé

Plusieurs centres d'appels disposent de nos jours d'un système de traitement automatisé appelé *Interactive Voice Response* (IVR). Grâce à ce système, le client reçoit des messages enregistrés ou générés par synthèse vocale lui demandant de choisir des options, d'entrer un numéro de compte, etc. Il communique avec l'IVR en utilisant les touches de son téléphone ou sa voix, selon ses préférences ou le système utilisé. Beaucoup d'appels téléphoniques se terminent à l'intérieur d'un tel système, allégeant le travail des agents. Par exemple, plusieurs banques et caisses proposent désormais un service automatisé de consultation de son solde qui permet aussi d'effectuer diverses transactions sans le recours à un être humain. Contrairement aux agents qui constituent une ressource limitée, un système automatisé peut supporter un très grand nombre de requêtes simultanées.

Une application Web constitue une seconde forme de système de traitement automatisé offrant davantage de flexibilité pour l'interaction avec le client que l'IVR. Il est possible d'afficher des images, des boutons, des menus, demander la saisie de texte, etc. Une requête ne pouvant être complètement traitée de façon automatique est acheminée à un agent par l'intermédiaire d'un courrier électronique. Les applications Web sont par exemple utilisées pour le support technique et la vente.

1.1.3 Routeur

Le *routeur*, aussi appelé distributeur automatique d'appels (*Automatic Call Distributor* ou ACD) dans le cas des centres d'appels téléphoniques, constitue l'élément central d'un centre de contacts permettant d'acheminer les clients lui parvenant à des agents capables de les servir. Lorsqu'un client ne peut être traité immédiatement, il est placé dans une file d'attente vérifiée lorsqu'un agent devient libre.

Pour prendre une décision, le routeur applique une *politique de routage* qui considère en général tout l'état du centre de contacts. De nos jours, les agents ne disposent pas tous de la même formation : certains peuvent traiter tous les types de requêtes, d'autres un sous-ensemble seulement. Lorsque le centre de contacts sert une clientèle multi-lingue,

la spécialisation des agents est pratiquement inévitable. C'est pourquoi les routeurs modernes peuvent être programmés avec un haut niveau de flexibilité pour gérer différentes configurations tenant compte d'une telle spécialisation.

Chaque jour, le routeur recueille des statistiques qui peuvent être utilisées pour analyser le centre de contacts. Souvent, seules les valeurs moyennes pour P périodes de temps $p = 1, \ldots, P$ (quinze minutes, une demi-heure ou une heure) sont conservées afin d'économiser l'espace de stockage. De nos jours, cet espace n'est plus un problème majeur, mais beaucoup de vieux équipements sont encore en service et continuent d'agréger les statistiques recueillies.

1.1.4 Composeur d'appels sortants

Afin d'augmenter l'activité des agents, certains systèmes tentent d'établir des communications avec des clients, par téléphone, courrier ou par l'Internet. Pour les appels sortants, un *composeur* est souvent utilisé pour effectuer plusieurs numérotations simultanément. Cet appareil tente d'anticiper le nombre d'agents libres à un moment donné et de leur affecter des appels sortants s'ils peuvent les servir. La liste des numéros à composer peut être obtenue en consultant des annuaires téléphoniques ou des listes de clients à rappeler. Le composeur achemine les appels réussis au routeur qui les affecte à des agents capables de les servir. L'agent répondant à l'appel détermine ensuite si la bonne personne a été rejointe.

Parfois, le composeur a contacté trop de clients à la fois et le nombre d'agents libres n'est plus suffisant au moment de leur connexion. Le client contacté doit alors être mis en file d'attente ou être déconnecté. Un appel sortant ainsi traité est nommé *mismatch* et est souvent perdu, car rares sont les clients qui attendront après avoir décroché le téléphone.

Les conditions de composition et le nombre de numéros à composer simultanément sont déterminés par une *politique de numérotation*. Une bonne politique doit fournir suffisamment de travail aux agents tout en maintenant une qualité de service acceptable. En particulier, elle se doit de limiter le nombre de *mismatchs*.

1.2 Principales statistiques des centres de contacts

L'analyse des centres de contacts vise à prévoir leur comportement en cas de changements de paramètres tels que le nombre d'agents en service et la politique de routage ou de variations de conditions comme le nombre d'arrivées. Les gestionnaires tentent de trouver les paramètres optimaux donnant une qualité de service acceptable tout en minimisant les coûts d'opération.

Une foule de statistiques, aussi nommées mesures de performance ou de qualité de service, sont disponibles pour effectuer cette analyse. Nous présentons ici celles qui sont le plus largement utilisées, mais il est possible d'en imaginer d'autres. Soit $X_{k,p}$ le nombre de contacts servis de type k étant entrés dans le centre pendant la période p. Soit $X_{k,\cdot} = \sum_{p=0}^{P-1} X_{k,p}$ le nombre de contacts de type k servis pendant toute la journée et soit $X_{\cdot,p} = \sum_{k=0}^{K-1} X_{k,p}$ le nombre de contacts servis étant entrés dans le centre pendant la période p. Soit $X = \sum_{k=0}^{K-1} X_{k,\cdot} = \sum_{p=0}^{P-1} X_{\cdot,p} = \sum_{k=0}^{K-1} \sum_{p=0}^{P-1} X_{k,p}$ le nombre total de contacts servis. De façon similaire, il est possible de définir $Y_{k,p}$, $Y_{k,\cdot}$, $Y_{\cdot,p}$ et Y comme étant le nombre d'abandons et $B_{k,p}$, $B_{k,\cdot}$, $B_{\cdot,p}$ et B, le nombre de contacts bloqués.

Nous définissions également $X_{\mathrm{g}}(s)$ comme le nombre de contacts servis après un temps d'attente inférieur à $s > 0$ et $X_{\mathrm{b}}(s) = X - X_{\mathrm{g}}(s)$ le nombre de contacts servis après un temps d'attente supérieur ou égal à s. Si $s_{k,\cdot} > 0$ représente le temps d'attente acceptable pour les contacts de type k, $s_{\cdot,p} > 0$ le temps d'attente acceptable pour les contacts arrivés durant la période p et $s_{k,p} > 0$, le temps pour les contacts de type k arrivés pendant la période p, de la même façon que $X_{k,\cdot}$, $X_{\cdot,p}$, etc., il est possible de définir $X_{\mathrm{g},k,\cdot}(s_{k,\cdot})$, $X_{\mathrm{b},k,\cdot}(s_{k,\cdot})$, etc. Nous définissons également $Y_{\mathrm{g}}(s)$ comme le nombre d'abandons après un temps d'attente inférieur à s et $Y_{\mathrm{b}}(s) = Y - Y_{\mathrm{g}}(s)$. Encore une fois, nous pouvons définir $Y_{\mathrm{g},k,\cdot}(s_{k,\cdot})$, $Y_{\mathrm{b},\cdot,p}(s_{\cdot,p})$, etc.

W_{X} est défini comme la somme des temps d'attente de tous les contacts servis tandis que W_{Y} représente la somme des temps d'attente des contacts ayant abandonné sans être servis. $W_{\mathrm{X},\cdot,p}$ représente la somme des temps d'attente pour tous les contacts servis arrivés pendant la période p. Nous pouvons également définir $W_{\mathrm{X},k,p}$, $W_{\mathrm{X},k,\cdot}$, $W_{\mathrm{Y},k,p}$, $W_{\mathrm{Y},k,\cdot}$ et $W_{\mathrm{Y},\cdot,p}$.

Toutes ces quantités sont des *variables aléatoires* qui prennent des valeurs suivant une loi de probabilité [29]. Plusieurs de ces variables sont dites *discrètes*, car elles prennent un nombre dénombrable de valeurs. Par exemple, avec probabilité $P(X = x)$, x contacts sont servis pendant une journée. Nous nous intéressons souvent à la valeur *espérée* d'une variable aléatoire qui est définie, dans le cas discret, par

$$E[X] = \sum_{x=-\infty}^{\infty} xp(x) = \sum_{x=0}^{\infty} xp(x) \text{ puisque } X \geq 0 \tag{1.1}$$

où $p(x)$ est la *fonction de masse* de X telle que

$$\begin{aligned} p(x) &= P(X = x), \\ \sum_{x=-\infty}^{\infty} p(x) &= 1. \end{aligned}$$

Dans le cas du nombre de contacts servis, $p(x) = 0$ pour $x < 0$. La variable aléatoire W_{X} est dite *continue*, car elle peut prendre un nombre infini et non dénombrable de valeurs. Son espérance est définie par

$$E[W_{\mathrm{X}}] = \int_{-\infty}^{\infty} wf(w)\, dw = \int_{0}^{\infty} wf(w)\, dw \text{ puisque } W_{\mathrm{X}} \geq 0 \tag{1.2}$$

où $f(w)$ est la *fonction de densité* de W_{X} telle que

$$\begin{aligned} P(a \leq W_{\mathrm{X}} \leq b) &= \int_{a}^{b} f(w)\, dw, \\ \int_{-\infty}^{\infty} f(w)\, dw &= 1. \end{aligned}$$

Encore une fois, puisque W_{X} correspond à un temps d'attente, $f(w) = 0$ pour $w < 0$. La *variance*, utilisée pour étudier la variation de X autour de son espérance, est définie comme

$$\mathrm{Var}(X) = E\left[(X - E[X])^2\right] = E\left[X^2\right] - E[X]^2. \tag{1.3}$$

L'espérance de chaque variable aléatoire définie précédemment peut représenter une mesure de performance. D'autres mesures se définissent quant à elles comme des rap-

ports d'espérances. Par exemple, le *niveau de service* global est défini par

$$g_1(s) = \frac{E[X_{\mathrm{g}}(s)]}{E[X + Y_{\mathrm{b}}(s)]} \tag{1.4}$$

ou encore par

$$g_2(s) = \frac{E[X_{\mathrm{g}}(s) + Y_{\mathrm{g}}(s)]}{E[X + Y]}. \tag{1.5}$$

De façon similaire, il est possible de définir $g_{1,\mathrm{K},k}(s_{k,\cdot})$, $g_{2,\mathrm{P},p}(s_{\cdot,p})$, etc. Le nombre de contacts bloqués peut également être pris en compte dans le niveau de service, par exemple en additionnant $E[B]$ au dénominateur du rapport.

Le *taux d'occupation des agents* dans un groupe i, durant un intervalle de temps $[t_1, t_2]$, est défini par

$$o_i(t_1, t_2) = \frac{E\left[\int_{t_1}^{t_2} N_{\mathrm{b},i}(t)\, dt\right]}{E\left[\int_{t_1}^{t_2} (N_{\mathrm{b},i}(t) + N_{\mathrm{f},i}(t))dt\right]} \tag{1.6}$$

Cette mesure, souvent calculée pour toute la durée d'ouverture du centre de contacts ou pour une période, correspond à l'espérance du temps d'activité des agents sur celle du temps pendant lequel ils peuvent servir des contacts. Il est possible d'obtenir le taux d'occupation global en remplaçant $N_{\mathrm{b},i}(t)$ par $N_{\mathrm{b}}(t) = \sum_{i=0}^{I-1} N_{\mathrm{b},i}(t)$ et $N_{\mathrm{f},i}(t)$ par $N_{\mathrm{f}}(t) = \sum_{i=0}^{I-1} N_{\mathrm{f},i}(t)$. Il est aussi possible d'estimer le *facteur d'efficacité des agents* dans le groupe i, défini par

$$\varepsilon_i(t_1, t_2) = \frac{E\left[\int_{t_1}^{t_2} (N_{\mathrm{f},i}(t) + N_{\mathrm{b},i}(t))dt\right]}{E\left[\int_{t_1}^{t_2} N_i(t)\, dt\right]}. \tag{1.7}$$

Soit $N_{\mathrm{b},i,k}(t)$ le nombre d'agents dans le groupe i occupés à servir des contacts de type k. Si le groupe ne peut pas servir de tels contacts, $N_{\mathrm{b},i,k}(t) = 0$ pour tout t. Nous avons $N_{\mathrm{b},i}(t) = \sum_{k=0}^{K-1} N_{\mathrm{b},i,k}(t)$. Le taux d'occupation des agents de groupe i par des con-

tacts de type k durant l'intervalle $[t_1, t_2]$ est alors défini par

$$o_{i,k}(t_1,t_2) = \frac{E\left[\int_{t_1}^{t_2} N_{\mathrm{b},i,k}(t)\ dt\right]}{E\left[\int_{t_1}^{t_2} (N_{\mathrm{b},i}(t) + N_{\mathrm{f},i}(t))dt\right]}. \tag{1.8}$$

Cette mesure permet d'évaluer quels types de contacts occupent la plus grande partie du temps des agents.

Le *temps de réponse moyen* constitue le délai moyen nécessaire pour qu'un contact en file soit servi. Il est défini par

$$w_{\mathrm{X}} = \frac{E[W_{\mathrm{X}}]}{E[X]}. \tag{1.9}$$

Le *temps d'attente moyen* est le délai moyen nécessaire pour qu'un contact soit servi ou abandonne. Il est défini par

$$w = \frac{E[W_{\mathrm{X}} + W_{\mathrm{Y}}]}{E[X+Y]}. \tag{1.10}$$

Ces temps peuvent être globaux ou comptés seulement pour les contacts ayant à attendre. Il est aussi fréquent d'estimer le taux d'abandon, le taux de *mismatch* et la taille moyenne de la file d'attente. Parfois, le nombre moyen de contacts de type k dans la file est également estimé.

Toutes ces quantités ne peuvent pas être calculées de façon exacte en général, car la loi de probabilité des variables aléatoires impliquées est trop complexe. Il nous faut donc employer des méthodes approximatives telles que la simulation. Lorsque la simulation est mise en œuvre pour estimer ces différentes mesures de performance, pour chaque variable aléatoire, un échantillon de n copies indépendantes et identiquement distribuées (i.i.d.) est généré. La méthode la plus simple pour obtenir ces échantillons consiste à répéter l'expérimentation n fois de façon indépendante. La moyenne arithmétique est ensuite utilisée comme estimateur des espérances. Par exemple, $E[X]$ est estimé par

$$E[X] \approx \bar{X}_n = \frac{1}{n}\sum_{r=0}^{n-1} X_r \tag{1.11}$$

où X_r correspond à la r^{e} copie de la variable aléatoire X. De la même façon, le niveau de service est estimé par

$$g_1(s) \approx \bar{g}_{1,n}(s) = \frac{\bar{X}_{\text{g},n}(s)}{\bar{X}_n + \bar{Y}_{\text{b},n}(s)} = \frac{\frac{1}{n}\sum_{r=0}^{n-1} X_{\text{g},r}(s)}{\frac{1}{n}\sum_{r=0}^{n-1} X_r + \frac{1}{n}\sum_{r=0}^{n-1} Y_{\text{b},r}(s)}. \tag{1.12}$$

Pour estimer la variance $\sigma^2 = \text{Var}(X)$, il est possible d'utiliser la variance empirique

$$\sigma^2 \approx S_n^2 = \frac{1}{n-1}\sum_{r=0}^{n-1}(X_r - \bar{X}_n)^2. \tag{1.13}$$

1.3 La simulation des centres de contacts

Avec les premiers centres ne supportant qu'un seul type d'appel téléphonique, des formules analytiques fondées sur la théorie des files d'attente étaient utilisées pour effectuer l'analyse sous des hypothèses simplificatrices fortes. Avec l'accroissement de la complexité des systèmes, ces formules s'éloignent de plus en plus de la réalité, car elles tiennent difficilement compte de la non-stationnarité, c'est-à-dire de la variation par rapport au temps des lois de probabilité pour les temps inter-arrivées, les durées de service, etc. Les abandons, les files multiples de taille limitée, le routage complexe, les appels sortants, etc. posent aussi des difficultés pour les formules analytiques. Lorsque les lois de probabilité ne sont plus exponentielles, les résultats de leur utilisation deviennent encore plus incertains. Seule la simulation peut fournir des évaluations précises tenant compte de toute la complexité.

Un centre de contacts pourrait bien entendu être modélisé grâce à un logiciel de simulation générique, mais cette tâche nécessiterait un énorme travail de conception et même de programmation. Il existe heureusement des logiciels spécialisés qui supportent la simulation de la plupart des centres de contacts d'aujourd'hui. Ces logiciels étendent un outil de simulation plus général se présentant comme une interface graphique ou un langage dédié. Toutefois, de nouveaux cas qui n'étaient pas prévus initialement peuvent survenir à n'importe quel moment et s'avérer difficiles à traiter. Par exemple, un nouvel algorithme de routage pourrait imposer une politique de sélection des agents ne pouvant

être implantée de façon simple par une extension du logiciel. Un gestionnaire devant simuler un système non supporté par son logiciel commercial doit attendre la prochaine version en espérant que le nouvel aspect sera pris en charge ou utiliser des primitives de bas niveau pour l'implanter lui-même, ce qui peut exiger la programmation dans un langage dédié à la simulation.

Ce langage, qui est souvent interprété, réduit la performance des simulateurs produits. Parfois trop spécialisé, il impose également des limites de flexibilité. Par exemple, il est souvent difficile d'implanter des techniques de réduction de variance telles que les variables aléatoires communes étant donné que le logiciel n'offre pas un contrôle suffisant des générateurs de variables aléatoires. Étant commercial et très spécialisé, il est moins connu, répandu et supporté qu'un langage générique et les outils associés sont eux aussi commerciaux. Ce langage déjà de très haut niveau est mis en œuvre pour construire des primitives comme des entités et des branchements conditionnels. Parfois, ces primitives déjà de haut niveau, représentées par des éléments graphiques, sont utilisées pour construire des objets plus complexes. Ces multiples couches permettent la flexibilité et la simplicité d'utilisation, mais la performance est grandement réduite. De plus, certaines éditions de base des logiciels de simulation n'autorisent la construction des modèles que par l'intermédiaire des objets graphiques, limitant par le fait même l'extensibilité ; aucune extension construite par l'utilisateur n'est aussi performante que les modules fournis par le fabricant.

Ainsi, une extension d'un outil commercial pour les centres de contacts peut être mise au point, écrite dans un langage dédié à la simulation et adaptée aux modèles futurs par son fabricant, mais elle hérite de toutes les limitations de l'outil choisi. Plutôt qu'un langage dédié à la simulation, il est souvent préférable de recourir à un langage de programmation générique, bien supporté et largement utilisé, surtout si la vitesse d'exécution est un critère important.

Il serait possible de construire un simulateur ou une bibliothèque pour la simulation de centres de contacts directement dans un tel langage, mais plusieurs tâches fondamentales telles que la génération des variables aléatoires et la gestion de la liste d'événements devraient être prises en charge. Une bibliothèque de support à la simulation s'avère alors

indispensable pour éviter de compliquer inutilement le simulateur de centres de contacts.

Le langage utilisé doit être choisi judicieusement, car plusieurs sont de trop bas niveau pour permettre l'écriture d'un système simple et réutilisable. Par exemple, grâce au langage C et à la bibliothèque de simulation SSC [17], un simulateur de centres d'appels a pu être construit [10]. Il ne traitait qu'un seul modèle avec un type d'appel entrant et un type d'appel sortant. Il est difficile de lui ajouter de nouvelles fonctionnalités sans modifier et recompiler tous ses fichiers, car le langage C ne dispose pas de primitives de niveau suffisant pour gérer le couplage entre les éléments de façon simple. En effet, un programme C est constitué de plusieurs fonctions qu'il n'est pas possible de redéfinir sans tout recompiler. Cette limitation peut être contournée par un usage astucieux de pointeurs sur des fonctions, mais de telles pratiques diminuent la lisibilité du code de la bibliothèque et des applications en faisant usage. Le support de la programmation orientée objet semble essentiel à la construction d'une bibliothèque suffisamment extensible et réutilisable.

1.4 La bibliothèque ContactCenters

Dans ce mémoire, nous présentons la bibliothèque *ContactCenters* que nous avons développée et qui permet de construire des simulateurs de centres de contacts utilisant l'approche par événements. La bibliothèque est implantée sous la forme d'un ensemble de classes Java utilisant Stochastic Simulation in Java (SSJ) [18, 20, 22] ainsi que Colt [13]. SSJ fournit un système de simulation rapide et robuste tandis que Colt facilite la gestion de matrices et de certains calculs statistiques. ContactCenters fournit un certain nombre de composantes de base pouvant être combinées pour modéliser un centre de contacts de façon très détaillée. Sa flexibilité permet la construction d'un grand nombre de modèles généraux et certains modèles spécifiques sont déjà disponibles. Chaque contact est défini comme une entité, c'est-à-dire un objet, avec un ensemble d'attributs prédéfinis que l'utilisateur peut étendre si nécessaire.

Java constitue un langage puissant, largement utilisé et très bien supporté. Grâce à l'héritage, les classes de la bibliothèque peuvent facilement être étendues sans les ré-

crire en entier. Un simulateur peut tirer parti de Java pour accéder à un grand nombre de bibliothèques d'optimisation et d'analyse statistique, ainsi qu'à des outils de construction d'interfaces graphiques. Grâce aux optimisations des récentes machines virtuelles Java, un simulateur écrit avec ContactCenters s'exécute beaucoup plus rapidement qu'un modèle conçu grâce aux outils commerciaux les plus utilisés et fondés sur un langage complètement interprété et peu répandu.

La simulation est souvent utilisée pour effectuer une optimisation. Par exemple, il est possible de rechercher le nombre optimal d'agents dans chaque groupe ou la politique de routage la plus appropriée. Puisque plusieurs simulations sont souvent nécessaires pour accomplir cette tâche, la performance constitue un aspect crucial.

Les logiciels commerciaux calculent un très grand nombre de mesures de performance qui ne sont pas toutes nécessaires pour une analyse donnée. La bibliothèque ContactCenters, quant à elle, offre un contrôle total sur les mesures de performance estimées. Étant donné que l'utilisateur choisit lui-même quoi évaluer et comment implanter ses estimateurs, il peut maîtriser tous les aspects de ses expérimentations. La possibilité d'estimer seulement les mesures requises augmente la performance tandis que le contrôle des estimateurs utilisés rend la réduction de variance possible.

L'utilisation d'une bibliothèque confère suffisamment de flexibilité pour tester diverses techniques d'estimation de sous-gradients. Un sous-gradient permet par exemple d'estimer la variation du niveau de service en fonction du vecteur d'affectation des agents et constitue un outil essentiel pour l'optimisation [2, 8].

Les systèmes de simulation commerciaux sont souvent liés à un seul module d'optimisation générique dont la logique est cachée tandis qu'un algorithme spécialisé pour les centres de contacts est souvent plus flexible et efficace. Plusieurs optimiseurs peuvent utiliser un même simulateur ContactCenters pour évaluer des mesures de performance, permettant la comparaison de divers algorithmes.

Par rapport aux outils commerciaux, nous disposons ainsi d'une performance, d'une flexibilité et d'une interopérabilité accrues. Malheureusement, la simplicité d'utilisation caractéristique à une interface graphique est perdue. Toutefois, rien n'empêche une telle interface d'être éventuellement créée. Pour ce faire, un simulateur générique pourrait

être construit et une interface pourrait permettre d'en fixer ses paramètres. Contrairement aux logiciels commerciaux, l'interface graphique étant totalement indépendante de la bibliothèque proprement dite, son utilisation serait facultative et plusieurs interfaces concurrentes pourraient être construites.

1.5 Plan du mémoire

Le chapitre suivant examine plus en détails les solutions actuelles pour simuler des centres de contacts. Nous traitons brièvement des possibilités d'un logiciel de planification intégrant la simulation pour ensuite nous concentrer sur les deux outils dédiés aux centres de contacts.

Le chapitre 3 présente l'architecture de la bibliothèque ContactCenters et ses différents constituants qu'il est possible de combiner pour créer des simulateurs. Nous y abordons les composantes élémentaires que nous avons élaborées, le mode de fonctionnement du routeur et différents problèmes rencontrés pendant la conception avec les solutions que nous avons trouvées. Nous y présentons également une interface permettant la communication entre un simulateur de centres de contacts et d'autres programmes afin de favoriser l'écriture de simulateurs génériques. Nous abordons finalement le simulateur générique que nous avons construit et qui tire parti de cette interface.

Le chapitre 4 présente divers exemples utilisant la bibliothèque afin de démontrer ses possibilités en pratique. Nous présentons d'abord un exemple complet et commenté de simulateur. Nous comparons ensuite la performance de ContactCenters avec celle d'un logiciel commercial largement utilisé, en exécutant plusieurs exemples de simulateurs. Nous expliquons comment le simulateur générique peut utiliser XML pour la lecture de paramètres et interagir avec d'autres logiciels par le biais de l'interface unifiée.

Le chapitre 5 présente un ensemble d'extensions proposées pour la bibliothèque SSJ utilisée par ContactCenters. Ces extensions incluent des collecteurs statistiques de haut niveau et la gestion d'expérimentations utilisant la méthode des moyennes par lots.

Le guide de l'utilisateur de ContactCenters [6] contient davantage d'informations à propos de la bibliothèque. Outre la documentation détaillée de toutes les classes et les

méthodes, de nombreux exemples additionnels sont disponibles.

CHAPITRE 2

OUTILS DE SIMULATION EXISTANTS

Nous avons vu dans le chapitre précédent que la simulation constitue le seul outil permettant de tenir compte de toute la complexité des centres de contacts contemporains. Quelques solutions logicielles sont disponibles pour cette tâche, mais toutes souffrent de limitations. Plusieurs logiciels de planification intègrent maintenant cette technique, permettant d'obtenir des résultats avec une grande simplicité. Toutefois, ils se limitent à des modèles très spécifiques. Des outils tels qu'*Arena Contact Center Edition* de Rockwell [28] et *ccProphet* de NovaSim [26] sont beaucoup plus génériques tout en demeurant spécialisés pour les centres de contacts. Tous deux permettent à l'utilisateur de définir les paramètres d'un modèle de simulation en quelques clics de souris, d'utiliser les animations pour le débogage et une meilleure compréhension du système et de produire des rapports statistiques complets. De façon interne, un modèle de centre de contacts complexe, générique et dont les paramètres sont ajustables par l'usager est implanté. Malheureusement, la taille des systèmes qui peuvent être simulés de façon efficace est plutôt limitée. À notre connaissance, aucune bibliothèque spécialisée pour les centres de contacts ne tire parti de toute la puissance d'un langage de programmation générique.

2.1 Simulateurs intégrés

Il existe de nombreux outils permettant de planifier l'affectation des agents et les horaires dans les centres de contacts. De nos jours, la complexité accrue des centres de contacts force les fabricants à utiliser la simulation pour raffiner les approximations employées à l'origine. Le simulateur intégré dans leurs logiciels est spécialisé pour les centres de contacts et très simple d'utilisation, mais il est lié à l'outil de planification et ne peut que difficilement être utilisé de façon indépendante. Il n'est pas toujours bien documenté, ses fonctionnalités sont limitées et son code source est caché ; sa personnalisation est difficile, voire impossible. En particulier, il ne peut souvent pas estimer des

mesures de performance définies par l'usager.

Par exemple, *Call Center Designer* de Portage Communications Inc. est un outil de planification à faible coût pour les centres d'appels de petite ou moyenne taille utilisant des approximations. Un module d'extension appelé *SimACD* [27] permet de raffiner les approximations en simulant chaque appel individuellement. Malheureusement, contrairement à ce qu'on pourrait penser, le module **SimACD** de Call Center Designer n'offre pas réellement une plus grande flexibilité que les autres modules du logiciel : le modèle de simulation ne comprend qu'un seul type d'appel et un seul groupe d'agents, le routage ne peut être paramétré et les lois de probabilité ne peuvent être modifiées. La simulation a ainsi été limitée en fonction des autres fonctionnalités offertes par le logiciel.

Grâce à la fenêtre principale présentée sur la figure 2.1, l'utilisateur de Call Center Designer peut déterminer la date de début de la semaine à prédire, le début et la fin des heures d'ouverture et la durée des périodes. Contrairement à la version de démonstration que nous avons testée, le produit commercial permet de modifier tous les champs des boîtes de dialogue.

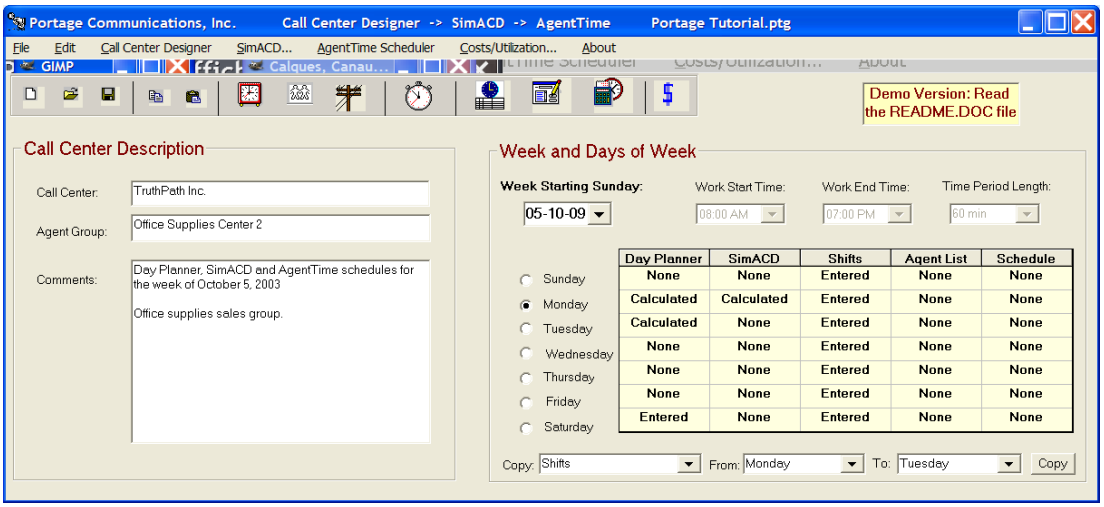

Figure 2.1 – Fenêtre principale de Call Center Designer

Pour chaque jour à planifier, l'utilisateur peut employer le **Day Planner** présenté à la figure 2.2 afin de prévoir le nombre d'agents nécessaires. Pour chaque période, le nombre moyen d'appels et les durées de communication et de travail après la communication

doivent être entrés. Le bouton **Calculate** détermine le nombre d'agents, le nombre de lignes téléphoniques, le temps de réponse moyen, la taille moyenne de la file et le taux d'occupation des agents. Aucune loi de probabilité ne peut être spécifiée et l'algorithme exact d'évaluation est caché.

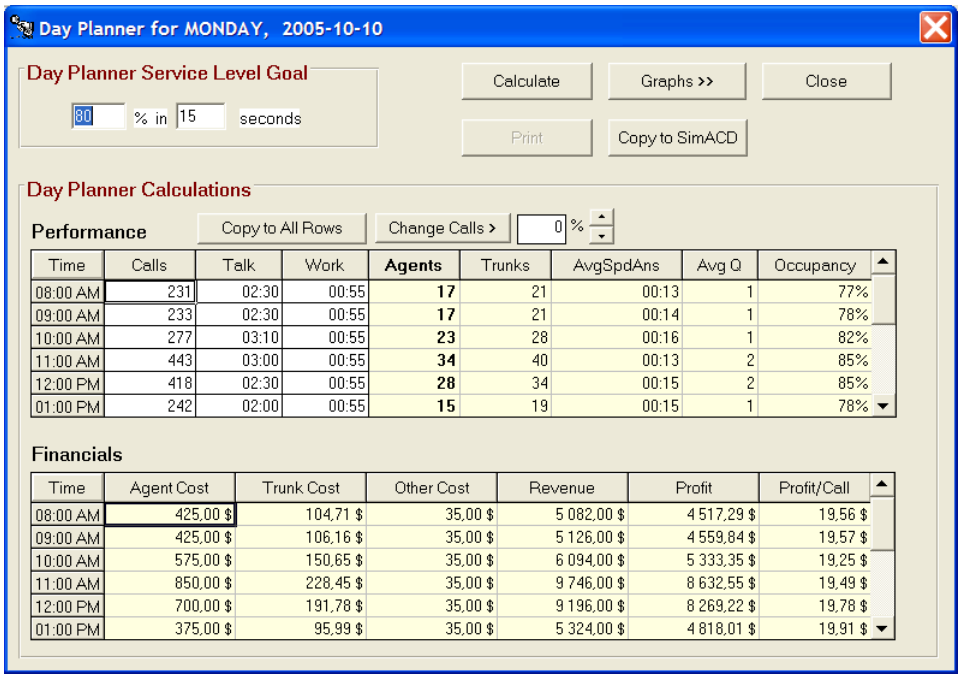

Figure 2.2 – Fenêtre **Day Planner** de Call Center Designer

La figure 2.3 présente le module **Staffing** de ce même logiciel. Pour exploiter ce module, l'utilisateur entre divers paramètres, spécifie un niveau de service à atteindre et le programme peut calculer des statistiques pour divers nombres d'agents. La ligne mise en évidence du tableau de résultats indique le nombre minimal d'agents grâce auquel le niveau de service voulu a été atteint.

La figure 2.4 présente le module **SimACD** de Call Center Designer. La simulation permet d'ajuster le délai entre l'arrivée d'un appel dans le centre et son routage vers un agent ou une file d'attente ainsi que le pourcentage de clients tentant un nouveau contact après abandon.

Call Center Designer comprend également un module de planification du nombre de lignes téléphoniques et des horaires pour les agents, mais ces derniers ne sont pas

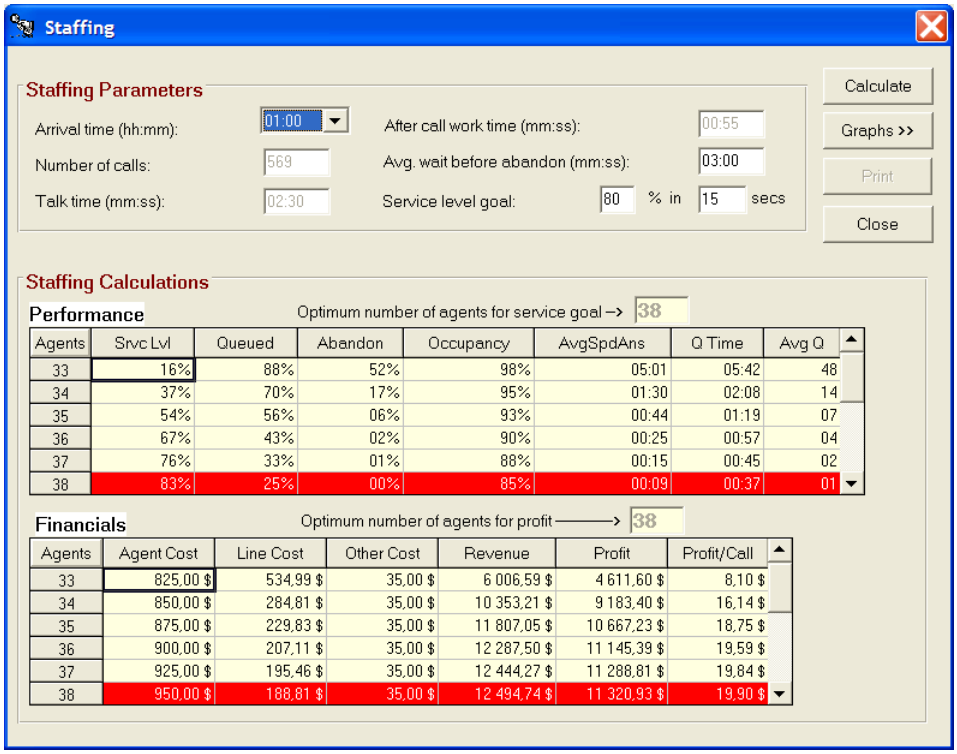

Figure 2.3 – Fenêtre **Staffing** de Call Center Designer

utilisables dans la version de démonstration. Ainsi, ce logiciel très spécialisé permet d'obtenir approximativement des paramètres pour un centre d'appels, mais il ne s'adapte pas aux centres modernes et aux besoins de la recherche.

2.2 Arena Contact Center Edition de Rockwell

Arena de Rockwell [4, 28] est l'un des outils de simulation les plus employés dans l'industrie. Il utilise de façon interne le langage SIMAN, mais l'utilisateur n'a que rarement accès au code. Il fournit un ensemble de modules regroupés en panneaux et adaptés à diverses situations de modélisation. Le panneau de base fournit les primitives nécessaires pour modéliser un système en utilisant des processus. Dans un modèle développé sous Arena, des entités telles que des clients sont créées et circulent à travers différents processus permettant de les diriger en fonction de décisions, de les traiter par l'intermédiaire de ressources, de les séparer, les regrouper et les détruire. Les modules sont insérés

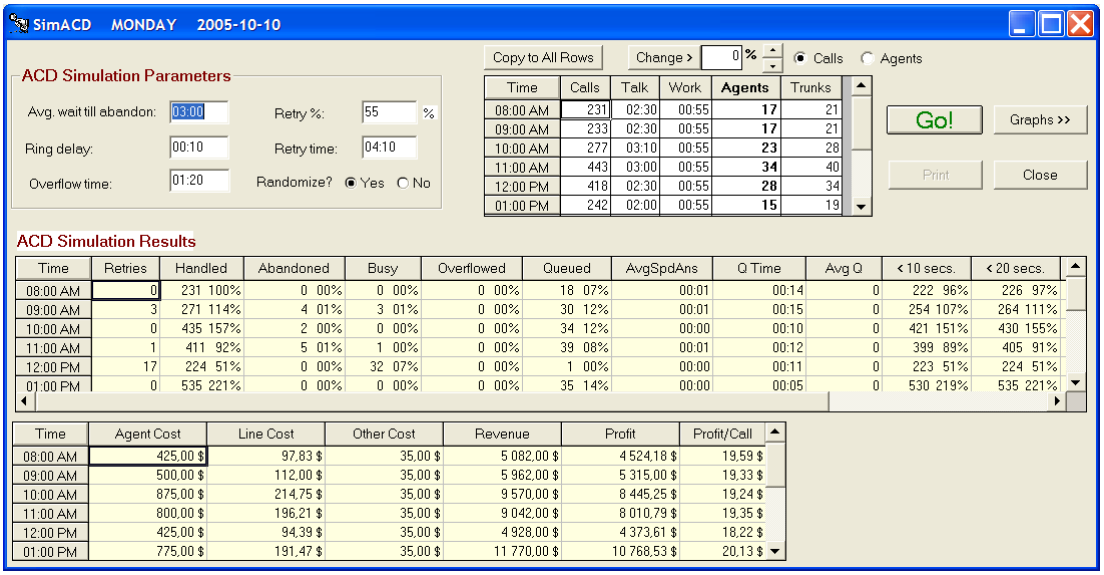

Figure 2.4 – Fenêtre **SimACD** de Call Center Designer

dans le modèle en les glissant depuis leur panneau d'origine vers la zone d'édition et des outils sont utilisés pour les interconnecter. Une extension Arena, appelée *template*, fournit un certain nombre de panneaux additionnels souvent dédiés à un domaine spécifique.

Arena peut produire des rapports statistiques sous forme de fichiers ASCII, de documents Microsoft Access ou utiliser le logiciel Crystal Report intégré pour afficher ces rapports, les imprimer ou les enregistrer en format PDF. Le contenu des rapports dépend des modules utilisés pour construire le modèle de simulation.

De nouveaux modules peuvent être construits en utilisant la version Professionnelle du produit. Malheureusement, cette version est également nécessaire pour visualiser le contenu des modules. Rockwell fournit un certain nombre d'extensions spécialisées pour les chaînes de montage ainsi que les centres de contacts. Ces extensions se présentent sous la forme de panneaux fournissant des modules de plus haut niveau à l'utilisateur et parfois des interfaces graphiques pour les configurer plus facilement.

Arena Contact Center Edition consiste en une version d'Arena avec une extension permettant de modéliser la plupart des centres de contacts existants en quelques clics de souris. L'utilisateur place différents modules correspondant aux éléments du système, fixe leurs paramètres et les connecte entre eux si nécessaire. Le logiciel est livré avec

plusieurs exemples commentés de centres de contacts permettant son apprentissage.

Le logiciel crée une entité pour chaque contact simulé et lui associe un ensemble d'attributs pouvant être étendu par l'utilisateur. Les animations permettent de suivre le parcours des contacts dans le système afin de vérifier visuellement si le modèle correspond bien à la réalité. Il est également possible de les désactiver pour accroître la vitesse d'exécution de la simulation.

La figure 2.5 présente l'exemple *Bank* issu du manuel d'utilisation d'Arena 8.0. Dans ce modèle, chaque agent peut gérer tous les contacts entrants, mais il traite sa spécialité plus efficacement. Les clients appellent pour des vérifications (*Checking*), des épargnes (*Savings*) ou pour obtenir un solde (*Account Balance*). Des agents spécialisés pour la vérification et les épargnes sont disponibles tandis que tous peuvent vérifier un solde avec la même efficacité. Lorsque le routeur parvient à associer un contact avec un agent spécialisé, le temps de service est multiplié par 0.75. Pour les soldes, le choix des agents est fait de façon aléatoire puisqu'aucune priorité n'est définie.

Les panneaux et modules se trouvent à gauche de la fenêtre du logiciel tandis que la partie centrale affiche le modèle de simulation. Le panneau **Contact Data** contient les modules pour modéliser le centre de contacts tandis que le panneau **Script** permet de construire la logique de routage. Nous allons examiner brièvement les éléments du modèle Bank sans détailler toutes les étapes de sa construction. Pour ces détails, voir le manuel d'utilisation d'Arena Contact Center Edition fourni avec le logiciel.

L'utilisateur doit d'abord insérer un module **Configuration**, dont les options sont présentées sur la figure 2.6, permettant de définir l'horizon de simulation (jour, semaine, mois), ainsi que les banques de canaux de communication. Dans cet exemple, une seule banque de quinze canaux est partagée par tous les contacts. Le bouton **Advanced** donne accès à une seconde fenêtre d'options permettant par exemple de définir le nombre de réplications à simuler, fixé à 1 par défaut.

Pour chaque type de contact, l'utilisateur ajoute un module **Contact** définissant un nom, une loi de probabilité pour les temps de service, les taux d'arrivée, la priorité, le script de routage à utiliser, etc. Les contacts arrivent toujours selon un processus de Poisson non homogène dont le taux d'arrivée constant pour chaque période est défini par

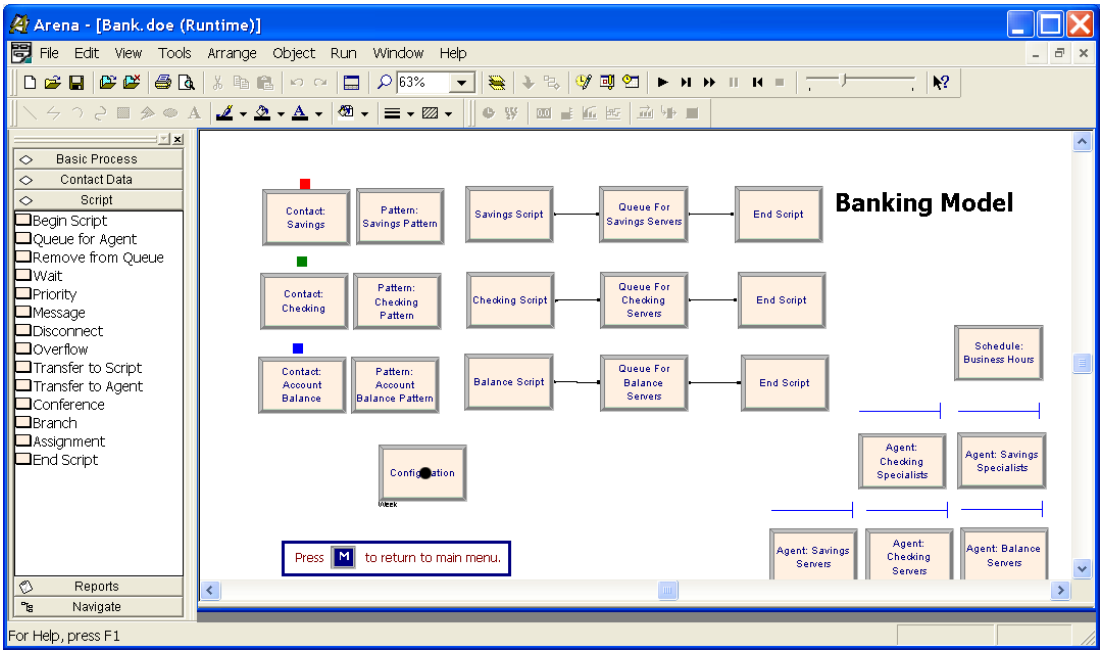

Figure 2.5 – Exemple de modèle sous Arena Contact Center Edition

un module **Pattern**, comme montré sur la figure 2.7. Dans cet exemple, chaque type de contact a son propre ensemble de taux d'arrivée.

Chaque groupe d'agents est représenté par un module **Agent Group** définissant le nombre de membres ainsi que les types de contacts pouvant être servis. À chaque type de contact peut être associé un degré de préférence et un multiplicateur de temps de service. Un module **Schedule** est associé à chaque groupe d'agents afin de déterminer les heures d'activité de ses membres. Dans l'exemple montré, chacun des deux groupes utilise le même horaire et peut servir les trois types de contacts. Les multiplicateurs de temps de service varient d'un groupe à l'autre afin de récompenser le traitement par les spécialistes. Par exemple, pour le groupe traitant les épargnes, le multiplicateur est fixé à 0.75 pour les épargnes et 1 pour les autres types de contacts. Tous les membres d'un groupe d'agents sont traités de façon identique par le logiciel. Si les agents sont différenciés, l'utilisateur doit créer plusieurs groupes ne contenant qu'un seul membre.

Le module **Parent Group** peut être utilisé pour représenter un ensemble de groupes d'agents tous capables de servir un contact donné. Chaque membre d'un groupe pa-

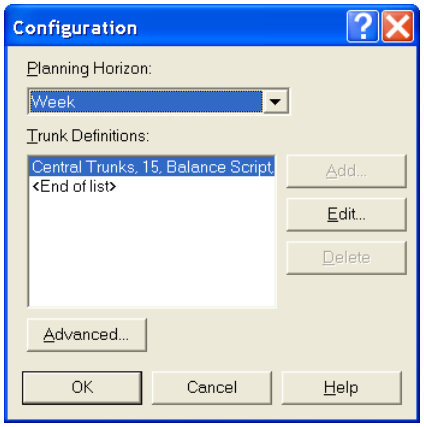

Figure 2.6 – Options du module **Configuration** d'Arena Contact Center Edition

rent reçoit une priorité affectant la sélection effectuée par le routeur. Dans cet exemple, un groupe parent est défini pour chaque type de contact et contient les deux groupes d'agents du modèle. Seule la préférence de sélection varie d'un groupe parent à l'autre.

Le routage associant les contacts aux agents est assuré par des scripts formés de modules interconnectés. Dans l'exemple Bank, un script distinct est construit pour chaque type de contact et contient trois modules : début du script, transfert vers le groupe parent correspondant au type, fin du script. Si un contact ne peut être servi immédiatement, il est placé dans une file d'attente associée au groupe parent et peut abandonner après un certain temps de patience. Le contact attend en fait dans toutes les files des groupes d'agents membres du groupe parent. Lorsqu'il est servi, le contact en attente est retiré de toutes ces files afin de ne pas être traité plusieurs fois par différents agents. Bien entendu, des scripts beaucoup plus complexes peuvent être construits. Par exemple, le routeur peut, après un certain temps d'attente, transférer le contact en attente vers une autre file, associée à un groupe d'agents différent, s'il n'a pas déjà été servi. Des branchements probabilistes ou conditionnels sont aussi possibles, permettant un routage stochastique ou dépendant de divers attributs du système.

Comme le montre la figure 2.8, Crystal Report peut afficher les statistiques de la plupart des aspects du système. Il est aussi possible de calculer des statistiques pendant différentes périodes de la simulation en ajoutant des modules **Report** dans le modèle.

Pour un modèle de cette petite taille, Arena Contact Center Edition est sans doute

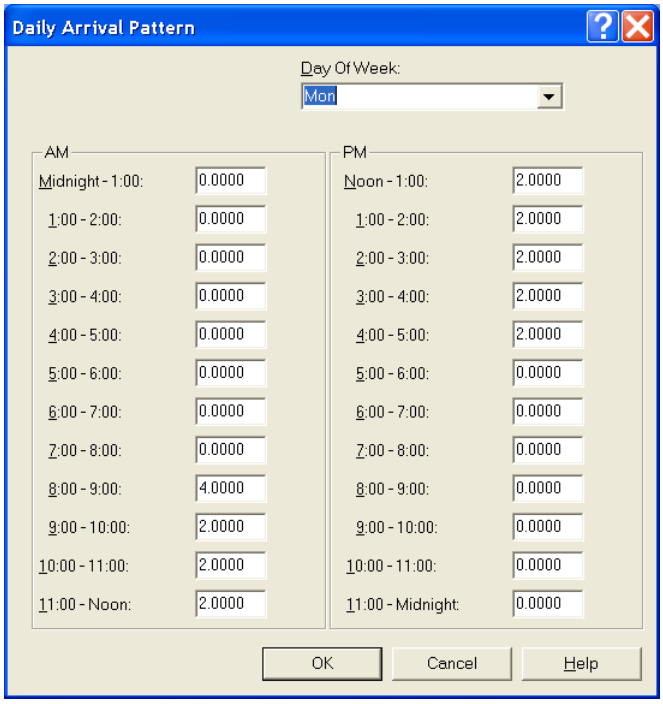

Figure 2.7 – Configuration des taux d'arrivée par période sous Arena Contact Center Edition

un choix judicieux en raison de sa simplicité d'utilisation. Les problèmes se manifestent lorsque la taille du centre devient grande. Lorsqu'il y a beaucoup de types de contacts avec des règles de routage dépendant du type, il devient nécessaire de construire un grand nombre de scripts de routage (au moins trois modules). Il est alors laborieux de créer ou mettre à jour un tel modèle à moins de construire des programmes Visual Basic for Application (VBA) pour automatiser les opérations. Lorsque le modèle est construit, il est très grand et peut être difficilement navigable.

Arena Contact Center Edition est livré avec un système d'optimisation nommé OptQuest faisant appel à un algorithme générique combinant diverses approches telles que les méthodes tabous et les réseaux de neurones. Sa logique exacte demeure cachée à l'utilisateur et aucun module d'optimisation spécialisé pour les centres de contacts n'est fourni. La simulation d'un petit exemple prend déjà quelques minutes si plus de cinquante réplications sont exigées. Le traitement de gros modèles étant très long, l'optimisation avec un algorithme itératif nécessitant d'exécuter la simulation plusieurs fois avec

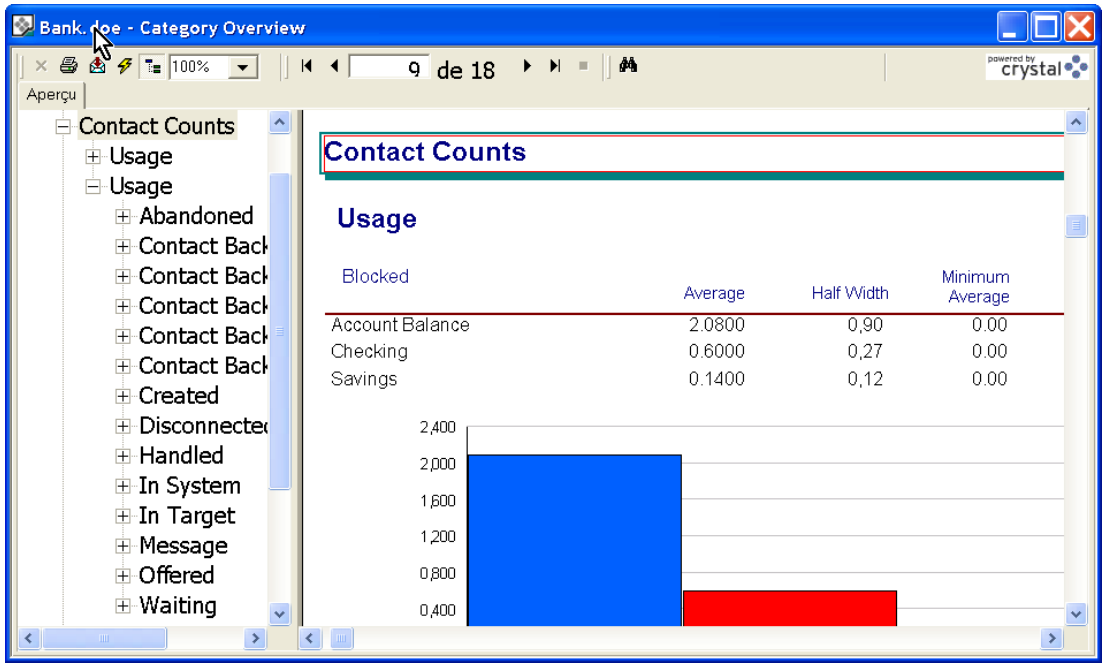

Figure 2.8 – Extrait d'un rapport statistique produit par Arena Contact Center Edition

des paramètres différents n'est envisageable qu'avec de petits systèmes.

L'extension Contact Center d'Arena ne modélise les arrivées de contacts que par le processus de Poisson. Certaines options sont offertes pour gérer des contacts sortants, mais elles sont peu documentées. Dans le module **Contact**, il est possible d'indiquer si un type de contact est entrant ou sortant. Lorsqu'un agent devient libre, s'il peut servir au moins un type de contact sortant, il tente automatiquement de communiquer avec des clients. Le module **Contact** comprend également un bouton **Return** permettant les rappels par les agents. Malheureusement, le logiciel ne prend pas en charge les politiques de numérotation complexes. Pour modéliser les appels sortants de façon plus générale, il est possible d'utiliser des modules de plus bas niveau pour simuler un composeur, mais cela accroît la complexité du modèle et son temps d'exécution.

2.3 ccProphet de NovaSim

ccProphet de NovaSim [26] constitue une alternative à Arena Contact Center Edition permettant la simulation d'un très grand nombre de centres de contacts. Malheu-

reusement, aucune version académique ou de démonstration n'est disponible et la seule source publique d'informations à propos du logiciel constitue la liste des questions fréquemment posées se trouvant sur le site Web de son fabricant.

Pour le traitement des données et la construction de graphiques, ccProphet se sert de Microsoft Excel, un logiciel largement utilisé avec lequel beaucoup de gestionnaires de centres de contacts sont familiers. Le logiciel de simulation fournit une interface graphique permettant de construire le centre de contacts sans entrer de code : l'utilisateur entre des données dans diverses boîtes de dialogue afin de spécifier les paramètres de son modèle et connecte divers modules entre eux. Les modules disponibles sous ccProphet sont très similaires à ceux trouvés sous Arena Contact Center Edition.

Chaque entité représentant un contact comporte un grand nombre d'attributs facilitant le débogage et permettant de retracer tout son chemin dans le système. Les animations permettent également de suivre visuellement le parcours des contacts dans le système. Les paramètres du modèle peuvent être modifiés en tout temps, sans exiger la recompilation d'un programme ou le redémarrage de la simulation tandis que sous Arena Contact Center Edition, le modèle ne peut être changé pendant son exécution, ce qui est un atout intéressant. Contrairement à son concurrent, ccProphet peut simuler le comportement du système de réponse interactif ainsi que les appels sortants.

Plutôt qu'étendre un système de simulation générique avec interface graphique, ccProphet implante un modèle de centre de contacts directement dans un langage interprété et dédié à la simulation appelé SIMUL8 (se prononce *Simulate*). D'après le site Web de NovaSim, ce langage est très connu dans l'industrie et sa syntaxe est simple à apprendre. Contrairement à Arena Contact Center Edition qui emploie l'approche par processus, ccProphet utilise celle par événements.

ccProphet est livré avec une licence complète de SIMUL8 incluant un éditeur et un débogueur tandis que sous Arena Contact Center Edition, le code SIMAN est à peine visible et éditable. Grâce à SIMUL8, il est possible de personnaliser un modèle en implantant de nouveaux éléments qui n'étaient pas prévus initialement. En particulier, il est possible de calculer des mesures de performance définies par l'utilisateur. Bien entendu, la flexibilité offerte par ccProphet dépend directement de celle de SIMUL8. L'utilisateur

désireux d'étendre le modèle de ccProphet doit impérativement écrire du code SIMUL8 ; il ne peut pas construire son extension à l'aide d'objets graphiques.

En conclusion, les deux logiciels commerciaux comparés offrent des fonctionnalités mais aussi des limitations similaires. En tenant compte des informations disponibles au sujet de ces logiciels, déterminer lequel satisfait le mieux aux besoins est difficile.

CHAPITRE 3

ARCHITECTURE DE LA BIBLIOTHÈQUE

La bibliothèque ContactCenters, contrairement aux outils commerciaux présentés au chapitre précédent, constitue un ensemble de classes écrites dans le langage Java plutôt que dans un langage dédié à la simulation. L'architecture interne n'étant pas dissimulée derrière une interface graphique mais plutôt visible et exploitable par l'usager, il est crucial que la bibliothèque soit bien conçue.

La conception d'une bibliothèque est une tâche difficile, car plusieurs objectifs, parfois contradictoires, doivent être pris en compte. La flexibilité s'avère un but fondamental de la conception, car il est important de pouvoir simuler la plus large gamme possible de centres de contacts existants mais également à venir. Lors de la conception, il faut ainsi limiter le nombre d'hypothèses simplificatrices afin d'éviter de rencontrer des contraintes dans l'avenir. Le plus important besoin de flexibilité se trouve au niveau des politiques de routage des contacts et de composition des appels sortants, ainsi que des variables aléatoires générées pour les temps inter-arrivées, les durées de service, les temps de patience, etc.

Puisqu'un simulateur écrit avec ContactCenters est destiné à être appelé plusieurs fois avec des paramètres différents, par exemple pour l'optimisation, la performance est un élément essentiel. Il doit être possible de simuler de très gros systèmes, avec des dizaines de types de contacts et de groupes d'agents. Le simulateur doit ainsi limiter la création d'objets et effectuer efficacement le couplage entre ses diverses composantes. La performance inclut le temps d'exécution ainsi que l'utilisation mémoire des simulateurs, mais étant donné la grande quantité de mémoire des machines contemporaines et sa libération automatique par le ramasse-miettes de Java, il est judicieux de nous concentrer sur le premier facteur.

La modularité constitue un troisième objectif de conception que nous avons jugé primordial. Chaque élément de la bibliothèque doit être le plus indépendant possible des autres de façon à pouvoir être modifié ou remplacé sans affecter de façon majeure

le comportement de tout le système. Chaque élément doit contenir son état interne et le mettre lui-même à jour afin que cet état demeure consistant. La modularité permet de tester unitairement le plus grand nombre possible de composantes afin de limiter les bogues dans la bibliothèque et de réduire la complexité de leur recherche.

L'extensibilité est une force indéniable pour une bibliothèque et un facteur important pour déterminer sa réutilisabilité. L'utilisateur comme le programmeur doivent être en mesure d'étendre les fonctionnalités de ContactCenters sans devoir adapter tous les simulateurs existants. Il doit être possible d'ajouter de nouveaux processus d'arrivée, de nouvelles politiques de routage ou de composition mais également de supporter de nouveaux types de rapports statistiques, sans recompiler toute la bibliothèque. L'héritage fournit un puissant mécanisme permettant de réaliser cet objectif puisque l'utilisateur peut modifier le comportement d'une méthode en la redéfinissant dans une sous-classe, sans même avoir accès au code source de la classe parent.

La simplicité d'utilisation représente un dernier objectif crucial puisqu'elle détermine l'utilisabilité de la bibliothèque. Chaque élément de ContactCenters doit le plus possible correspondre à un élément d'un centre de contacts réel afin de simplifier la vie du gestionnaire désireux d'utiliser l'outil. L'écriture d'un simulateur doit être relativement simple et éviter la redondance propice à l'erreur afin de faciliter la tâche du programmeur.

Tous ces objectifs sont à la fois complémentaires et contradictoires et un compromis doit être établi. Lors de la conception, il faut veiller à ce que l'accroissement de la performance ne se fasse pas au détriment de la flexibilité et de la modularité. Par exemple, il n'est pas souhaitable de remplacer les objets représentant les contacts par de simples compteurs. Outre la vitesse d'exécution du code, l'accroissement de la performance peut être obtenu en réduisant la variance dans les estimateurs. La bibliothèque doit ainsi offrir un degré suffisant de flexibilité pour permettre d'implanter de telles optimisations dans le futur. Par contre, une trop grande flexibilité peut nuire à la simplicité d'utilisation. Pour le programmeur, le nombre d'objets à créer et de méthodes à appeler peut devenir trop grand, accroissant la taille des simulateurs. Pour le gestionnaire, une généralisation trop forte peut entraîner la confusion. La simplicité maximale d'utilisation nécessite

quant à elle la création de simulateurs précompilés ou d'interfaces graphiques diminuant la nécessité de la programmation pour l'utilisateur mais également la flexibilité et la performance.

Dans ce chapitre, nous présentons l'architecture en composantes indépendantes que nous avons choisie pour la bibliothèque. Après avoir expliqué les principes fondamentaux de notre architecture, nous exposons les différentes composantes de la bibliothèque ainsi que leur mode d'interconnexion permettant la construction d'un simulateur. Nous présentons le fonctionnement du composeur d'appels sortants et du routeur, deux éléments centraux du système. Nous traitons des problèmes divers que nous avons dû résoudre pendant la conception, pour la gestion du temps de simulation et la génération des variables aléatoires suivant des lois de probabilité non stationnaires. Finalement, nous présentons une interface de haut niveau permettant l'interaction de simulateurs génériques avec d'autres programmes.

3.1 Grandes lignes de l'architecture

Comme présenté à la figure 3.1, la bibliothèque définit un certain nombre de composantes élémentaires indépendantes les unes des autres et correspondant à des classes. Ceci comprend les contacts, les agents, les files d'attente et les processus d'arrivée. Sur la figure ainsi que les suivantes, chaque rectangle représente une classe disponible dans la bibliothèque tandis que chaque ovale dénote des données ou des opérations qui ne sont pas liées à une seule classe. Les flèches indiquent une relation entre les éléments. Le modèle des observateurs, décrit plus en détails dans la section 3.1.1, est utilisé pour connecter les éléments entre eux d'une façon flexible.

Le composeur d'appels sortants constitue une composante de second niveau, car elle interagit avec divers éléments du système. Il est mis en fonction chaque fois qu'un service se termine et il consulte l'état du système, par exemple le nombre d'agents libres, pour décider des appels à composer.

Le routeur constitue une seconde composante de deuxième niveau : il reçoit les contacts produits et les achemine vers des agents ou des files d'attente. Il permet également à

des systèmes de comptage d'événements d'obtenir des informations à propos de chaque contact dont le traitement est terminé.

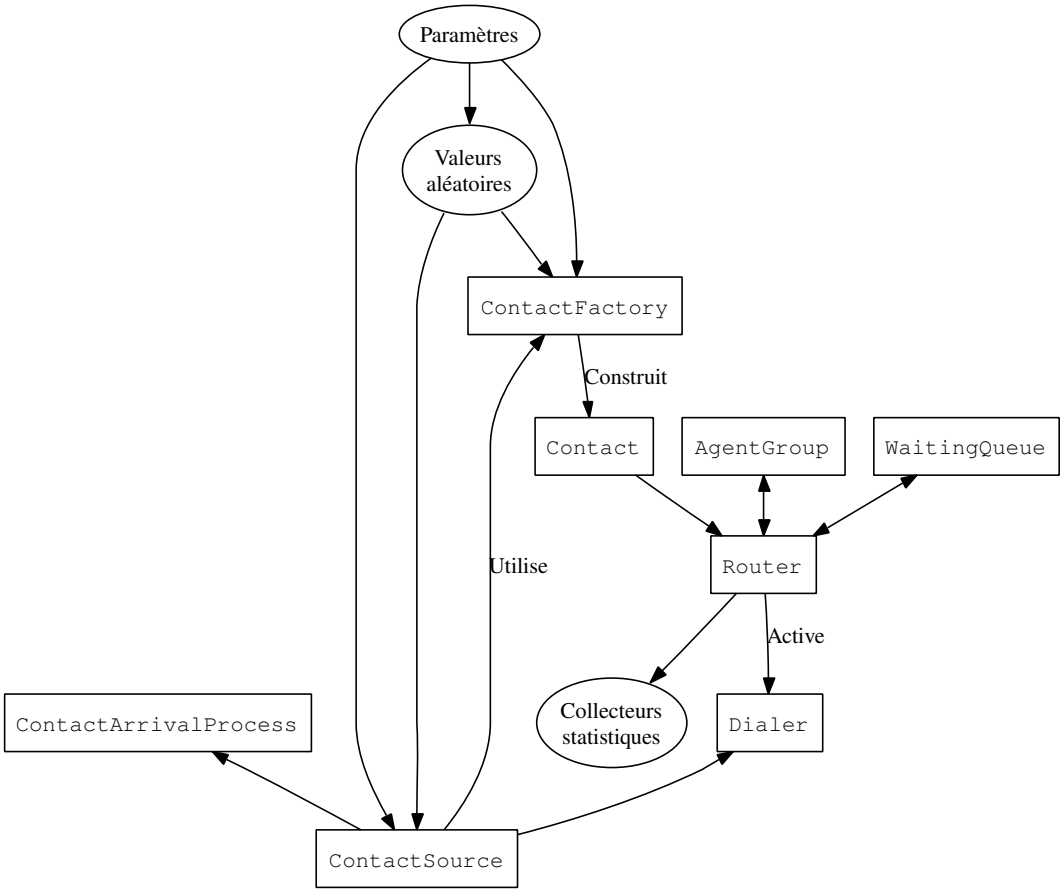

Figure 3.1 – Architecture de la bibliothèque

Les classes de ContactCenters se trouvent dans le paquetage `umontreal.iro.lecuyer.contactcenters`. Un sous-paquetage est offert pour les contacts et processus d'arrivée, les files d'attente, les groupes d'agents, les composeurs et les routeurs. Un dernier sous-paquetage contient une interface de haut niveau que nous aborderons à la section 3.6 et qui permet d'accéder au simulateur depuis d'autres programmes. Pour plus d'informations à propos des classes contenues dans ces différents paquetages, voir le guide d'utilisateur de la bibliothèque ContactCenters [6].

3.1.1 Mécanisme d'interaction entre éléments

Dans un simulateur de centres de contacts, une forte interaction entre les divers éléments est nécessaire. Par exemple, la fin d'un service par un agent doit provoquer la sélection d'un contact en file et possiblement la composition d'appels sortants. D'un autre côté, nous souhaitons disposer du plus grand nombre possible de composantes indépendantes pour maximiser la modularité. Il est ainsi nécessaire de mettre en place un mécanisme permettant ces interactions sans renforcer inutilement le couplage entre composantes.

Du point de vue performance, l'appel direct de méthodes est sans doute le meilleur choix pour faire interagir des objets. Malheureusement, son utilisation exige que chaque composante dispose d'informations à propos des méthodes des autres composantes. En particulier, chaque groupe d'agents doit contenir une référence vers le routeur et appeler une méthode spécifique pour annoncer la fin d'un service. Avec de tels appels directs, il devient difficile d'ajouter de nouvelles actions lorsque des événements se produisent, par exemple imprimer une trace pour le débogage ou mettre à jour une interface graphique à la fin des services. L'interaction par appels de méthodes favorise, dans le cas de la simulation de centres de contacts, un couplage fort qui pourrait mener à un système devant être adapté et recompilé en entier pour chaque simulateur particulier. Un tel système ne répond pas à l'objectif principal d'une bibliothèque qui consiste à fournir un ensemble de primitives réutilisables.

Nous avons plutôt opté pour le mécanisme des observateurs [11] pour gérer l'interaction entre les divers éléments de la bibliothèque. Dans ce modèle, un objet *observable* comporte une liste d'*observateurs* dont le contenu exact n'est connu qu'à l'exécution. Chaque observateur constitue un objet implantant une interface spécifique, par exemple `NewContactListener` pour recevoir de nouveaux contacts. Lorsque l'information doit être diffusée, la liste est parcourue et une méthode spécifiée par l'interface est appelée pour chaque observateur. L'objet observable fournit aussi des méthodes pour ajouter et supprimer des observateurs plutôt que laisser l'utilisateur manipuler sa liste interne, évitant l'apparition de doublons ou d'objets invalides. Par convention, ces méthodes sont

nommées selon l'interface d'observation, par exemple `addNewContactListener (NewContactListener l)` et `removeNewContactListener (NewContact-Listener l)`.

Avec ce mécanisme, il devient toutefois difficile de conserver des interfaces de communication stables sans utiliser d'objets temporaires pour transmettre des messages. Par exemple, l'interface `AgentGroupListener` servant à annoncer les changements au sein d'un groupe d'agents a dû être modifiée à plusieurs reprises au cours du développement de la bibliothèque pour ajouter différentes informations transmises. Heureusement, ce problème a pu être résolu en encapsulant les message à transmettre dans des objets qui doivent obligatoirement être construits pour le bon fonctionnement de simulateurs, à savoir les contacts et les structures représentant les événements de simulation.

Nous n'avons pas utilisé les classes `Observable` et `Observer` de Java, qui implantent le modèle choisi, car elles sont trop générales et réduisent la performance et la clarté des programmes. En effet, lors de la diffusion d'une information, `Observable` fait une copie interne de la liste des observateurs pour un comportement plus cohérent avec plusieurs fils d'exécution. Cette copie, qui diminue la performance, n'est pas nécessaire lorsqu'un seul fil d'exécution est utilisé, ce qui est toujours le cas lors de la simulation par événements avec SSJ. De plus, les observateurs de Java reçoivent l'information diffusée sous la forme d'un objet générique qui doit être transtypé selon les besoins, diminuant la lisibilité du code.

3.1.2 Simulation avec l'approche par événements

Pour la simulation avec la bibliothèque ContactCenters, nous avons opté pour l'approche par événements [16] dans laquelle des *événements* se produisent à des instants précis et déclenchent un certain nombre d'actions, incluant la transmission de messages à des observateurs. Le moteur de simulation, implanté par SSJ, maintient une liste d'événements futurs ordonnés selon l'instant de leur exécution. Tant que la liste n'est pas vide ou que le simulateur n'est pas arrêté, le premier événement est retiré puis exécuté. Pendant cette exécution, des actions ont lieu et de nouveaux événements peuvent être planifiés. Par exemple, l'arrivée d'un contact constitue un événement qui engendre son

routage et la planification de la prochaine arrivée. Tous les événements de simulation définis dans ContactCenters provoquent, à leur exécution, la transmission d'informations à des observateurs, mais SSJ permet de définir des événements qui échappent à cette règle.

Sous SSJ, un événement de simulation est défini en créant une sous-classe de `Event` et en implantant sa méthode `actions` responsable de son exécution. Dans ContactCenters, ces implantations sont pour la plupart situées à l'intérieur de classes internes des diverses composantes du système. Par exemple, la classe représentant les événements de fin de service est définie à l'intérieur de celle représentant les groupes d'agents. Initialement, ces classes internes étaient privées, c'est-à-dire cachées aux yeux de l'usager. Nous avons ensuite décidé de rendre la plupart d'entre elles publiques afin que les observateurs puissent utiliser leurs instances comme transmetteurs d'informations.

L'approche par processus constitue une alternative possible aux événements. Contrairement à un événement dont toutes les actions se produisent au même temps de simulation, un *processus* est un objet s'exécutant à divers instants. Plusieurs processus existent de façon concurrente, mais un seul est en cours d'exécution à un moment donné. Le processus courant peut allouer des ressources pour effectuer une tâche donnée et peut être suspendu pour céder le contrôle à un autre processus. La simulation s'arrête lorsqu'un processus en fait la requête ou lorsque plus aucun processus ne peut être exécuté ou relancé. L'approche par processus donne souvent des programmes plus concis et intuitifs, ce qui accroît la simplicité d'utilisation. Malheureusement, des expérimentations ont montré qu'un programme utilisant SSJ et construit avec les processus tourne de 12 à 700 fois plus lentement qu'un programme équivalent employant l'approche par événements [20]. Pour cette raison, nous avons choisi l'approche par événements pour notre bibliothèque.

3.2 Composantes élémentaires

Examinons maintenant de plus près chacune des composantes de la bibliothèque représentant des éléments du centre de contacts relativement indépendants entre eux.

3.2.1 Contacts

Pour simuler des centres de contacts de façon générale, une structure de données est nécessaire pour héberger les informations relatives à chaque contact. Un contact, représenté par un objet de la classe `Contact`, est toujours associé à un client avec lequel une communication est désirée. Toutefois, dans un système complexe, un client peut effectuer plusieurs contacts avant d'obtenir satisfaction ou d'abandonner définitivement. Un contact est caractérisé par un ensemble d'attributs tels que le temps d'arrivée, le temps passé en file d'attente ou avec un agent, etc. Il peut également mémoriser l'ensemble des étapes de son passage dans le système, mais par défaut, ce mécanisme est inhibé pour minimiser l'utilisation de la mémoire.

Dans les logiciels de simulation commerciaux, l'utilisateur dispose de la possibilité d'ajouter des attributs personnalisés aux entités représentant les contacts. La programmation orientée objet fournit cette possibilité puisqu'il est possible de créer une sous-classe de `Contact` et d'y ajouter de nouveaux champs. Il est ainsi possible de définir de nouveaux attributs tels que des coûts, des indications pour guider le routage, des variables aléatoires, etc.

La création d'une sous-classe de `Contact` permet également de redéfinir certaines méthodes appelées lorsque le contact entre ou sort d'une file d'attente, lorsque le service débute ou se termine, etc. Pour le moment, ce mécanisme en grande partie supplanté par les observateurs permet seulement de calculer le temps total d'attente et celui passé avec un agent.

3.2.2 Canaux de communication

De façon générale, puisque la capacité du système peut être limitée, il est nécessaire de disposer d'objets `TrunkGroup` représentant des groupes de canaux de communication. Un tel groupe comporte une capacité limitée qui peut être changée au cours de la simulation, un canal peut être alloué ou libéré en tout temps et des statistiques sur le nombre d'unités libres et occupées peuvent être calculées. Les canaux de communication sont similaires aux ressources de l'approche par processus, mais un contact désirant

un canal n'attend pas si aucun n'est disponible. Il se trouve plutôt bloqué et quitte le système.

Par défaut, aucun groupe de canaux n'est associé aux contacts, si bien que la capacité du système est infinie. De plus, un simulateur simple n'imposant pas de limite de capacité n'a pas besoin de créer d'instances de `TrunkGroup`. Si un groupe de canaux est associé à un contact, un canal est alloué pendant tout son cycle de vie et un contact arrivant à un moment où tous les canaux de son groupe sont occupés se voit bloqué. Puisque seul le routeur est en mesure de savoir avec précision quand un contact entre dans le système et en ressort, il est responsable d'allouer et de libérer les canaux.

3.2.3 Processus d'arrivée

Une *source de contacts* détermine à quels moments les contacts parviennent dans le système et est représentée par une implantation de l'interface `ContactSource`. Deux types de sources sont supportées par la bibliothèque : les processus d'arrivée dont nous traitons ici et le composeur d'appels sortants qui sera abordé dans la section 3.3.

Un *processus d'arrivée* concret définit un algorithme permettant de générer des durées inter-arrivées pour les contacts entrants. Ces temps peuvent dépendre de tout l'état du système, mais dans un modèle réaliste, ils ne sont habituellement affectés que par le temps de simulation et les arrivées précédentes.

Lors de chaque arrivée, le processus doit construire un objet représentant un nouveau contact, mais un appel direct à un constructeur explicite empêcherait tout usage de sous-classes de `Contact`. Pour résoudre ce problème, qui se présente également pour le composeur, chaque source contient une *usine abstraite*. Un tel objet fournit une méthode destinée à construire des instances issues de plusieurs classes partageant un ancêtre commun. Par opposition, un constructeur crée des instances d'une classe seulement. L'interface `ContactFactory`, implantée par tout objet construisant des contacts, applique ce modèle en spécifiant une méthode `newInstance` qui construit, initialise et retourne un nouveau contact. Ainsi, il est possible de remplacer la sous-classe de `Contact` (et le constructeur invoqué) sans modifier l'implantation des processus d'arrivée et des composeurs.

Pour permettre à un simulateur utilisant ContactCenters d'employer des sous-classes de `Contact`, plutôt qu'utiliser une usine abstraite, il serait possible de définir, dans les sources de contacts, une méthode chargée de la construction et pouvant être redéfinie au besoin. Mais, avec cette solution, l'emploi d'une sous-classe de `Contact` forcerait l'utilisateur à définir, pour chaque source de contacts, une sous-classe redéfinissant la méthode de construction par défaut. Cela accroîtrait la taille et la redondance des programmes testant par exemple différents processus d'arrivée.

Une autre solution pour remplacer l'usage de l'usine abstraite est la *réflexion*, un mécanisme permettant à un programme de manipuler des objets sans connaître leur type exact à la compilation. L'utilisateur passe à une source de contacts une référence vers la sous-classe appropriée de `Contact` et le programme peut utiliser cette sous-classe, connue à l'exécution seulement, pour créer des instances. L'usage de la réflexion pour effectuer l'instanciation a été exclu, car il diminuait la performance du système en raison du très grand nombre de contacts à instancier.

Lorsqu'un processus d'arrivée est activé, il planifie une première arrivée sous la forme d'un événement SSJ. À partir de l'exécution de cet événement et jusqu'à la fin de la simulation ou la désactivation du processus, le cycle présenté à la figure 3.2 est parcouru. Lorsqu'une arrivée se produit, le processus d'arrivée utilise l'usine abstraite qui lui est associée pour créer un contact et le diffuser à tous les observateurs enregistrés. Chaque observateur de nouveaux contacts doit implanter l'interface `NewContactListener` afin de recevoir les objets diffusés par l'intermédiaire de la méthode `newContact`. Lorsque tous les observateurs ont été informés du nouveau contact, un nouveau temps inter-arrivées est généré pour planifier un futur événement.

La logique de base est contenue dans une classe abstraite nommée `ContactArrivalProcess` et spécifiant une méthode `nextTime` générant les temps inter-arrivées. Tout processus d'arrivée peut ainsi être implanté par une sous-classe concrète, du moment qu'il est possible de générer des temps inter-arrivées. Plusieurs algorithmes se fondent sur le processus de Poisson et sont ainsi implantés dans des sous-classes de `PoissonArrivalProcess`. Cette classe représente un processus de Poisson pour lequel le taux d'arrivée λ est fixe par défaut. Toutefois, à tout instant pendant la simu-

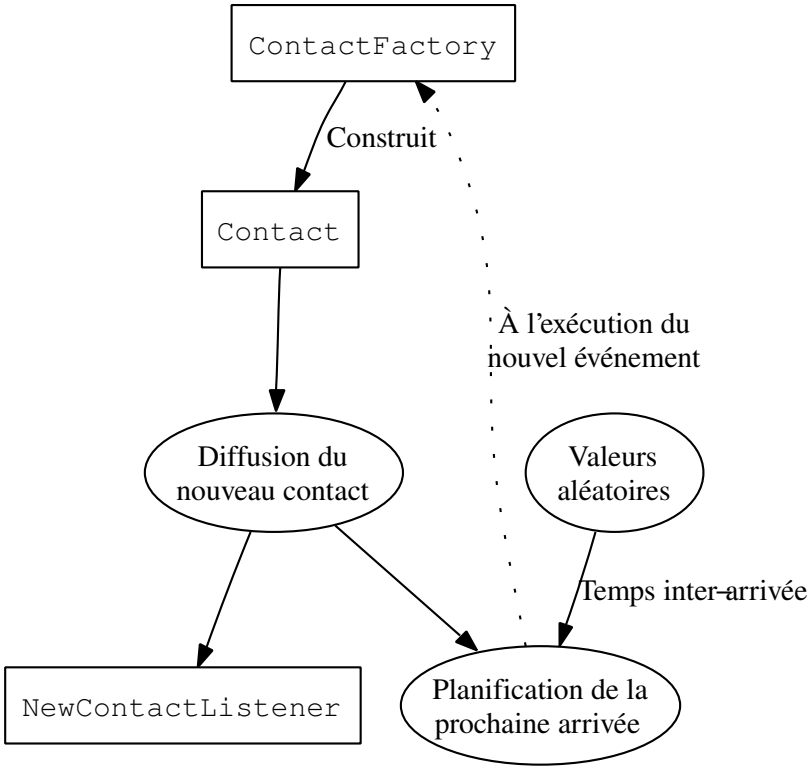

Figure 3.2 – Processus d'arrivée

lation, le taux peut être altéré et le temps de la prochaine arrivée est automatiquement modifié en conséquence. Une sous-classe est disponible, permettant le changement automatique de λ au début de chaque période. Le taux d'arrivée par période peut également être stochastique, suivant par exemple la loi gamma. Pour le moment, le système ne supporte pas un processus de Poisson avec taux d'arrivée variant de façon continue dans le temps, mais il serait facile de l'implanter dans une nouvelle sous-classe.

Il arrive parfois qu'un processus stochastique génère un nombre espéré d'arrivées A_p pour différentes périodes de temps p sans que le processus d'arrivée correspondant ne soit Poisson. Par exemple, ce cas se présente avec le modèle Dirichlet décrit dans [3]. Comme si chaque valeur A_p était générée depuis la loi de Poisson, A_p arrivées sont planifiées pendant la période p, à des temps uniformément distribués. Les temps d'arrivée générés pour chaque période sont ensuite triés de façon à pouvoir obtenir des durées inter-arrivées. La classe `PoissonUniformArrivalProcess` implante un tel pro-

cessus avec des valeurs A_p déterministes. En pratique, une sous-classe est construite pour ajouter un algorithme générant des nombres espérés d'arrivées aléatoires.

Actuellement, tous les processus d'arrivée présentés dans [3] sont implantés. Cela inclut le processus de Poisson non homogène couramment utilisé en plus de variantes doublement stochastiques telles que Poisson-gamma. Le système de gestion des arrivées semble suffisamment générique pour que nous envisagions éventuellement de le séparer de la bibliothèque ContactCenters pour qu'il soit utilisable pour d'autres types de simulation.

3.2.4 Files d'attente

Une file d'attente, représentée par une instance de `WaitingQueue`, constitue une structure de données dans laquelle peuvent résider des objets représentant des contacts ne pouvant être servis immédiatement. Chaque file d'attente supporte une généralisation des abandons que nous avons appelée *sorties automatiques*. Dans ce modèle, plutôt que des contacts, une file contient des objets `DequeueEvent` représentant des événements de sortie. Lorsqu'un contact est ajouté à la fin de la file, un tel événement est planifié si un temps maximal d'attente est disponible. Lorsque l'événement de sortie se produit, le contact associé quitte automatiquement la file sans être servi. Ceci inclut le cas particulier d'un abandon mais aussi la déconnexion du centre de contacts pour diverses raisons (panne de la ligne téléphonique, fermeture du centre de contacts, etc.) ou son transfert automatique vers une autre file.

Un contact mis en file peut bien entendu être retiré manuellement, c'est-à-dire par l'intermédiaire d'un événement, autre que celui de sortie de file, appelant une méthode appropriée de `WaitingQueue`. L'événement de sortie est alors annulé et retiré de la structure de données représentant la file. Par exemple, lorsqu'un agent devient libre, le premier contact d'une file peut être retiré par le routeur pour être servi. Les contacts en attente peuvent également être énumérés pour être affichés à l'écran, testés ou retirés de la file.

Une valeur numérique est associée à chaque sortie de file, qu'elle soit automatique ou manuelle. Cette valeur, stockée à l'intérieur de l'événement de sortie, permet aux divers

éléments du simulateur de déterminer si un contact a abandonné, a été servi, déconnecté, etc. La valeur numérique 0 est réservée au cas où le contact quitte la file pour être servi par un agent tandis que la valeur 1 est réservée pour dénoter un abandon.

Il serait possible de remplacer cet indicateur numérique par un objet d'état [11]. Toutefois, la classe de cet objet, déclarée dans le même paquetage que la classe `WaitingQueue`, devrait définir une méthode pour chaque action dépendant du type de sortie de file tandis que ces actions dépendent de la structure du routeur et des mécanismes de collecte statistique.

Une file d'attente contient une liste d'observateurs implantant l'interface `WaitingQueueListener` et informés lorsqu'un contact entre ou sort. L'ensemble des informations est transmis par le biais des événements de sortie de file, maximisant la stabilité à long terme de l'interface sans affecter la performance.

Deux types de structures de données utilisant le cadre des Collections Java sont disponibles pour stocker les contacts en attente. Chaque structure propose une implantation par défaut, mais l'utilisateur est en mesure de fournir sa propre implantation s'il le désire. La premier type constitue une liste de contacts ordonnée par temps d'entrée dans la file et prise en charge par la sous-classe `StandardWaitingQueue`. Par défaut, une liste doublement chaînée, proposée par Java, est mise en œuvre. Chaque élément d'une telle liste est placé dans un nœud (un objet) contenant un pointeur sur son voisin de gauche et son voisin de droite. Étant donné que la recherche du premier élément ainsi que l'insertion et la suppression d'un élément quelconque se font en temps constant, cette liste s'avère un excellent choix pour implanter une file d'attente premier arrivé premier servi qui est en général le modèle utilisé pour les centres de contacts. Pour des modèles supportant plusieurs types de contacts, il est souvent possible de combiner plusieurs de ces files pour obtenir un système de priorités. Une liste simplement chaînée, dont chaque nœud contient un pointeur sur le voisin suivant seulement, suffirait pour une file d'attente de centre de contacts, mais Java ne propose pas d'implantation pour une telle liste.

Le second type de structure de données est un ensemble ordonné pris en charge par la sous-classe `PriorityWaitingQueue`. Par défaut, un arbre rouge-noir [9] est employé pour trier les contacts. Cette structure constitue un arbre binaire s'équilibrant auto-

matiquement lors de sa modification. La recherche, l'ajout et la suppression s'effectuent dans $O(\lg q)$, où q est la taille de la file. L'utilisateur peut fournir un comparateur afin de spécifier comment chaque paire de contacts doit être ordonnée. Toutefois, une telle file de priorités s'avère beaucoup moins efficace qu'une file utilisant une liste ordonnée. Elle a été créée par souci de généralité uniquement ; elle n'a pas encore été nécessaire dans l'implantation d'un simulateur.

3.2.5 Agents

Au temps de simulation t, un groupe d'agents i, représenté par une instance de `AgentGroup`, comprend $N_i(t) \in \mathbb{N}$ membres dont $N_{\mathrm{b},i}(t)$ sont occupés et $N_{\mathrm{i},i}(t)$ sont libres. Parmi les $N_{\mathrm{i},i}(t)$ agents inoccupés, seuls $N_{\mathrm{f},i}(t) \leq N_{\mathrm{i},i}(t)$ sont disponibles pour servir de nouveaux contacts. De façon générale, ces quantités peuvent varier à tout moment pendant la journée. Seul $N_i(t)$ peut être librement ajusté par l'utilisateur tandis que les autres quantités constituent des variables observées.

Toutefois, dans la réalité, $N_i(t)$ constitue également une variable observée dont le comportement est conditionné par les horaires individuels des agents. En pratique, cette variable doit pouvoir être fixée afin de permettre la modélisation d'agents non différenciés pour un groupe i, mais il faut prévoir le cas où $N_i(t)$ est inférieur à $N_{\mathrm{b},i}(t)$. Puisque par convention, tous les agents terminent les services qu'ils ont commencés avant de quitter le système, il arrive parfois que $N_{\mathrm{b},i}(t) > N_i(t)$. Dans ce cas, aucun service ne doit être interrompu, si bien que $N_{\mathrm{g},i}(t) = N_{\mathrm{b},i}(t) - N_i(t)$ agents dits *fantômes* doivent quitter le centre après avoir terminé leur service courant. Ainsi, le nombre réel d'agents dans un groupe i au temps t est donné par $N_i(t) + N_{\mathrm{g},i}(t)$. Aucun nouveau contact n'est accepté par le groupe lorsque $N_{\mathrm{b},i}(t) \geq N_i(t)$. Étant donné que $N_i(t)$ contient les $N_{\mathrm{i},i}(t)$ agents inoccupés et les $N_{\mathrm{b},i}(t) - N_{\mathrm{g},i}(t)$ agents occupés qui feront encore partie du groupe après la fin de leur service, en tout temps durant la simulation, nous avons l'invariant

$$N_{\mathrm{b},i}(t) + N_{\mathrm{i},i}(t) = N_i(t) + N_{\mathrm{g},i}(t). \tag{3.1}$$

Parfois, certains agents sont branchés au routeur mais ne sont pas disponibles pour

servir des contacts. La bibliothèque permet de simuler le comportement de chaque agent particulier afin de modéliser cet aspect, mais les informations nécessaires ne sont pas toujours disponibles. Dans ce cas, la non disponibilité de certains agents inoccupés peut être modélisée approximativement par un facteur d'efficacité global ε_i. Ce facteur peut être utilisé pour compenser la sur-planification des agents [10]. Il est la plupart du temps initialisé à 1 de façon à ce que tous les agents puissent servir des contacts. Si les agents ne sont pas différenciés, il peut alors arriver que le niveau de service obtenu par simulation soit supérieur à celui obtenu avec le centre de contacts réel tandis que le taux d'occupation des agents est plus petit. Ceci s'explique par le fait que les agents simulés sont trop « efficaces » par rapport aux agents réels. En réduisant ε_i, il est possible de modéliser approximativement cette efficacité réduite puisque les agents ne sont disponibles que pendant une fraction ε_i du temps. Le facteur d'efficacité est ajusté empiriquement de façon à faire correspondre le taux d'occupation des agents et les autres mesures de performance avec les données d'un centre de contacts réel. Ce modèle, bien que primitif, a été implanté, car il est le seul qui puisse être employé si les agents ne sont pas différenciés.

Plus précisément, si $N_{\mathrm{b},i}(t) = 0$, le nombre d'agents disponibles est approximé par $N_{\mathrm{f},i}(t) = \mathrm{round}(\varepsilon_i N_i(t))$, où $\mathrm{round}(\cdot)$ arrondit son argument à l'entier le plus près et ε_i est le facteur d'efficacité durant toute la simulation. Lorsque $N_{\mathrm{b},i}(t) > 0$, le nombre d'agents occupés encore membres du groupe après leur service est soustrait de $N_{\mathrm{f},i}(t)$, ce qui donne un second invariant pour un groupe d'agents :

$$N_{\mathrm{b},i}(t) + N_{\mathrm{f},i}(t) = \mathrm{round}(\varepsilon_i N_i(t)) + N_{\mathrm{g},i}(t). \tag{3.2}$$

Lorsque $\varepsilon_i = 1$ (la plupart du temps), les équations (3.1) et (3.2) sont équivalentes et $N_{\mathrm{i},i}(t) = N_{\mathrm{f},i}(t)$. Dans ce cas, le taux d'occupation donné par l'équation (1.6) peut être défini par

$$o_i(t_1, t_2) \quad = \quad \frac{E\left[\int_{t_1}^{t_2} N_{\mathrm{b},i}(t)\, dt\right]}{E\left[\int_{t_1}^{t_2} (N_{\mathrm{b},i}(t) + N_{\mathrm{f},i}(t)) dt\right]}$$

$$
\begin{aligned}
&= \frac{E\left[\int_{t_1}^{t_2} N_{\mathrm{b},i}(t)\ dt\right]}{E\left[\int_{t_1}^{t_2} (N_{\mathrm{b},i}(t) + N_{\mathrm{i},i}(t))dt\right]} \\
&= \frac{E\left[\int_{t_1}^{t_2} N_{\mathrm{b},i}(t)\ dt\right]}{E\left[\int_{t_1}^{t_2} (N_i(t) + N_{\mathrm{g},i}(t))dt\right]}.
\end{aligned} \tag{3.3}
$$

Si les agents ne sont pas différenciés, lorsque $N_i(t)$ diminue, les agents inoccupés disparaissent en premier afin d'éviter de produire des agents fantômes inutilement. Cette décision n'a que peu d'impact sur la simulation puisque, dans tous les cas, les agents terminent le service des contacts avant de quitter. Ainsi,

$$N_{\mathrm{i},i}(t) > 0 \Rightarrow N_{\mathrm{g},i}(t) = 0. \tag{3.4}$$

Cet invariant ne tient plus lorsque les agents sont différenciés puisqu'un agent occupé peut en général être marqué pour quitter le système avant que tous les agents inoccupés n'aient quitté.

Le service d'un contact est divisé en deux phases dont la fin est déclenchée par un événement : la communication (première phase) et le travail après la communication (seconde phase). Il est nécessaire de distinguer ces deux phases afin de permettre la limitation du nombre de lignes téléphoniques. La ligne allouée par un contact doit être libérée à la fin de la communication avec un agent et disponible pour d'autres clients pendant le travail après la communication. Si le nombre de lignes est illimité (ou très grand), le temps de travail après la communication peut simplement être inclus dans le temps de service. Chacune de ces deux phases peut être terminée automatiquement par l'exécution d'un événement de fin de service ou manuellement, par l'appel d'une méthode d'`AgentGroup`. À la fin de la communication, le contact associé à l'événement de fin de service quitte le centre ou est transféré vers une autre partie du système. L'agent, de son côté, demeure occupé et débute son travail après la communication dont la durée peut être nulle, par exemple si le nombre de canaux de communication est illimité. Un événement de simulation est planifié une seconde fois seulement si la durée du travail est non nulle. À la fin de ce travail, l'agent redevient libre et peut traiter un nouveau contact.

La simulation du service d'un contact par un agent est prise en charge par un objet `EndServiceEvent` qui représente à la fois l'événement de fin de communication et celui de fin de service. Par souci de cohérence et de généralité, bien que cette particularité n'a pas encore été utilisée dans les simulateurs implantés, une valeur numérique est associée à chaque fin de communication et fin de service. Cette valeur permet à des observateurs de déterminer la raison de la fin de communication.

Ces mécanismes permettent le support du service préemptif, mais aucun exemple n'a encore été implanté pour le moment. Lorsqu'un contact prioritaire tel qu'un appel téléphonique doit être servi par un agent en train de répondre à un courrier électronique, cet agent interrompt le service en cours et stocke le contact, ainsi que le temps de service restant, dans une file d'attente afin de pouvoir répondre à l'appel. L'interruption s'effectue par l'appel d'une méthode de `AgentGroup` plutôt que par l'exécution de l'événement de fin de service. Lorsqu'un agent devient libre, il examine la file de contacts non prioritaires et peut reprendre le service interrompu. Afin que le service puisse être repris par le même agent qui l'a interrompu (cas le plus courant et le plus logique), une file peut être allouée pour chaque agent susceptible d'accomplir des services préemptifs ou pour chaque groupe d'agents si leurs membres ne sont pas différenciés. En conservant l'information appropriée, il est même possible d'imposer une pénalité, par exemple un temps de service plus long, lorsqu'un service interrompu par un agent x est repris par un agent y ou si la durée d'interruption excède quelques jours. Grâce à l'indicateur de fin de service, le routeur peut distinguer les fins de service des interruptions pour la collecte statistique et la gestion des files d'attente.

Par défaut, la classe `AgentGroup` ne construisant pas un objet distinct pour chaque agent, les membres du groupe ne sont pas différenciés. Il est certes possible de différencier les agents en construisant des groupes contenant un seul membre, mais il est utile de regrouper les agents pour simplifier l'implantation des politiques de routage. La sous-classe `DetailedAgentGroup` permet quant à elle de modéliser le comportement de chaque agent, représenté par un objet de classe `Agent`. Chaque agent peut être ajouté ou retiré d'un groupe en tout temps, mais un agent ne peut pas faire partie de plusieurs groupes simultanément. La classe `DetailedAgentGroup` dispose d'une

méthode permettant de démarrer le service d'un contact par un agent particulier. Si la méthode de début de service définie dans `AgentGroup` est utilisée, l'agent avec le plus long temps d'inactivité parmi les membres du groupe est choisi.

Jusqu'à présent, `DetailedAgentGroup` n'a servi qu'à déterminer l'agent avec le plus long temps d'inactivité dans un groupe pour implanter certaines politiques de routage, mais il est possible d'utiliser cette sous-classe pour gérer des agents de même groupe ayant des horaires différents ou servant des contacts à des vitesses différentes. Puisque chaque membre d'un groupe contient son propre état, il devient possible de modéliser les agents libres mais non disponibles avec précision. Par exemple, les agents peuvent être rendus non disponibles à des moments et pendant des durées aléatoires déterminés en fonction des horaires. En contrepartie, considérer chaque agent individuellement réduit la performance, car `DetailedAgentGroup` doit maintenir à jour des listes internes contenant les agents libres, les agents occupés et les agents fantômes.

Tous les événements relatifs à un groupe d'agents sont transmis à des observateurs enregistrés et implantant l'interface `AgentGroupListener`. Comme dans le cas de la file d'attente, l'événement de fin de service sert de transmetteur d'informations plutôt qu'un objet temporaire. Une seconde interface nommée `AgentListener` est également disponible pour recevoir des informations relatives à un agent individuel, chaque objet `Agent` ayant sa propre liste d'observateurs.

3.3 Composeur d'appels sortants

De façon similaire à un processus d'arrivée, le composeur d'appels sortants, représenté par la classe `Dialer`, constitue une source fournissant de nouveaux contacts à des observateurs, incluant habituellement un routeur chargé d'acheminer les contacts à des agents. Toutefois, le composeur utilise l'état du système pour prendre sa décision. Par exemple, le nombre d'appels composés peut dépendre du nombre d'agents dans certains groupes.

Lorsqu'un certain événement, par exemple la fin d'un service, se produit, le composeur peut être déclenché pour tenter de contacter des clients. Les différentes étapes

alors effectuées sont présentées à la figure 3.3. Le composeur interroge d'abord sa politique de numérotation afin de déterminer combien d'appels il doit essayer. Si des appels doivent être effectués, il extrait les objets `Contact` les représentant d'une liste de composition souvent infinie et dont les éléments sont créés par une usine abstraite `ContactFactory`. La liste de composition peut aussi être finie et construite par un autre processus stochastique, comme des appelants laissant un message. Le contenu de la liste pourrait même être totalement généré par la politique de numérotation et alors dépendre de l'état du système.

Avec une certaine probabilité, l'appel sortant réussit. La probabilité de succès peut dépendre du temps de simulation, du contact extrait de la liste, etc. Le composeur définit deux listes distinctes d'observateurs pour annoncer les appels réussis et échoués indépendamment. Le routeur n'est intéressé que par les appels réussis, mais certains collecteurs statistiques peuvent également compter les appels échoués.

En utilisant ce composeur, il a été possible d'implanter toutes les politiques de composition utilisées dans le centre d'appels mixte d'Alexandre Deslauriers [10]. Toutefois, il a été nécessaire de généraliser ces politiques afin qu'elles fonctionnent dans un système avec plusieurs types de contacts.

Par exemple, une politique à base de seuil est fournie par `ThresholdDialerPolicy`. Elle nécessite la définition de deux ensembles de groupes d'agents. L'*ensemble de test* est utilisé pour décider si des appels seront effectués tandis que l'*ensemble de destination* détermine le nombre d'appels à tenter. Souvent, l'ensemble de test comprend tous les groupes d'agents du centre de contacts tandis que l'ensemble de destination contient tous les agents capables de servir les appels sortants qui seront générés. Un ensemble de groupes d'agents est représenté par une instance de la classe `AgentGroupSet` qui encapsule une liste de groupes d'agents et définit des opérations pour obtenir le nombre total d'agents dans l'ensemble. L'utilisateur doit lui-même construire ces ensembles, car la politique de numérotation ne dispose pas des informations nécessaires pour le faire automatiquement.

Pour que le composeur effectue des appels, le nombre d'agents libres $N_{\mathrm{f}}^{\mathrm{t}}(t)$ dans l'ensemble de test doit être supérieur ou égal à un seuil s_{t}. Lorsque cette condition s'ap-

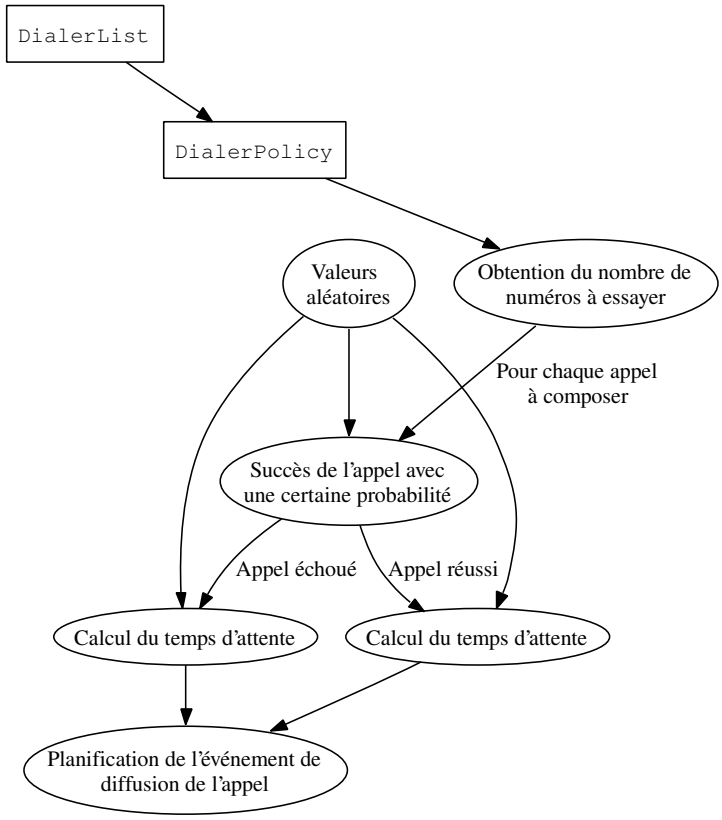

Figure 3.3 – Composeur d'appels sortants

plique, le composeur détermine $N_{\mathrm{f}}^{\mathrm{d}}(t)$, le nombre d'agents libres dans l'ensemble de destination. Si $N_{\mathrm{f}}^{\mathrm{d}}(t)$ est supérieur ou égal à un second seuil s_{d}, le composeur effectue $\max\{\mathrm{round}(\kappa N_{\mathrm{f}}^{\mathrm{d}}(t))+c,0\}$ appels, où $\kappa \in \mathbb{R}$ et $c \in \mathbb{N}$ sont des constantes et la fonction $\mathrm{round}(\cdot)$ arrondit son argument à l'entier le plus près. Si le nombre d'agents libres est insuffisant, aucun appel n'est tenté. Par exemple, pour qu'un seul appel soit composé à la fois, nous devons fixer $\kappa = 0$ et $c = 1$. Pour que le nombre d'appels composés corresponde au double du nombre d'agents libres, nous devons fixer $\kappa = 2$ et $c = 0$. Tous les paramètres de la politique peuvent être modifiés à tout moment pendant la simulation, si bien que les seuils et les constantes pourraient varier dans le temps.

La politique de composition la plus complexe actuellement implantée tient compte du niveau de service et du taux de *mismatch* pendant les dix dernières minutes d'opération du système pour prendre sa décision. Bien entendu, des politiques encore plus complexes

pourraient être mises en place au besoin.

Le moment auquel le composeur est activé est important s'il utilise l'une des politiques précédentes. Par exemple, si le composeur est activé par un observateur enregistré auprès d'un groupe d'agents, il verra un certain nombre d'agents libres à la fin d'un service. Toutefois, au même instant de simulation, dès que tous les observateurs ont reçu l'annonce de la fin de service, le routeur effectue la sélection de contacts et réduira peut-être le nombre d'agents libres. C'est pourquoi un mécanisme a été prévu dans le routeur pour activer des composeurs après la sélection de contacts. Des difficultés similaires apparaissent si plusieurs composeurs sont utilisés dans un même simulateur si bien que l'utilisateur peut contrôler l'ordre d'activation des composeurs par le routeur.

3.4 Routeur

Assurant la connexion entre les contacts, les agents et les files d'attente, le routeur, représenté par la classe `Router`, constitue le point central de tout simulateur de centres de contacts. Afin de maximiser la flexibilité du système, il doit pouvoir accéder à un maximum d'informations pour prendre ses décisions. Cette information est disponible par le biais des files d'attente, groupes d'agents ainsi que des événements SSJ représentant les sorties de file et les fins de service. Nous abordons à présent la structure générale du routeur de ContactCenters ainsi que les éléments nécessaires à l'implantation d'une politique de routage.

3.4.1 Structure générale

La figure 3.4 présente les interactions entre le routeur et les différents éléments du centre de contacts. Chaque rectangle représente une classe ou une interface du système tandis que les flèches indiquent une relation entre des éléments. Dans le cas des interfaces d'observateurs, l'objet transmetteur d'informations est indiqué.

Le routeur reçoit l'ensemble des contacts créés par les processus d'arrivée et réussis par les composeurs. Il doit connaître et gérer les groupes d'agents ainsi que les files d'attente pour assurer la liaison entre tous ces éléments. Deux types de routage sont à

prendre en compte : la sélection d'un agent pour un nouveau contact et la sélection d'un contact en attente pour un agent libre.

Pour assurer le premier type de routage, le routeur est défini comme un observateur de nouveaux contacts. Lorsque le routeur reçoit un nouveau contact, la méthode `newContact` de la classe `Router` lui alloue un canal de communication si un groupe de canaux lui est associé et le bloque si un tel canal n'est pas disponible. il tente ensuite de consulter les groupes d'agents connectés et prend une décision en fonction de la politique de routage implantée. S'il ne peut être servi immédiatement, le contact peut être inséré dans une file d'attente connectée si la capacité des files le permet. La sélection de la file d'attente de destination dépend, encore une fois, de la politique de routage implantée.

Pour prendre en charge le second type de routage, le routeur doit être informé de chaque fin de service et ajout d'agents. Lorsqu'un agent devient libre ou se connecte au système, le routeur tente de consulter les files d'attente connectées afin d'en extraire un contact à lui affecter.

En raison de ces deux types de routage, une relation bidirectionnelle est établie entre le routeur et les groupes d'agents : le routeur déclenche des services tandis que les groupes d'agents annoncent les membres devenus libres. Grâce à l'utilisation des observateurs, les groupes d'agents n'ont pas besoin d'implanter une partie de la politique de routage ou même de connaître la structure exacte du routeur.

Une relation similaire existe entre le routeur et les files d'attente : le routeur insère les contacts ne pouvant être servis dans des files tandis que les files d'attente annoncent les contacts sortants au routeur. Ce système facilite la collecte statistique et permet d'implanter le transfert d'une file à une autre. Encore une fois, nous évitons que la file d'attente n'ait à implanter une partie de la politique de routage.

Puisque tous les événements du centre de contacts transitent par le biais du routeur, ce dernier dispose des informations nécessaires pour savoir quand un contact quitte le système. En utilisant le modèle des observateurs, le routeur annonce tous ces contacts, qu'ils soient bloqués, servis ou sortis d'une file d'attente, par le biais de l'interface `ExitedContactListener`. Les informations sont transmises par un contact et une valeur numérique indiquant la raison du blocage, un événement de fin de service ou un

événement de sortie de file, selon le type de sortie du contact. Comme son homologue réel, le routeur simulé peut ainsi effectuer la collecte d'observations statistiques.

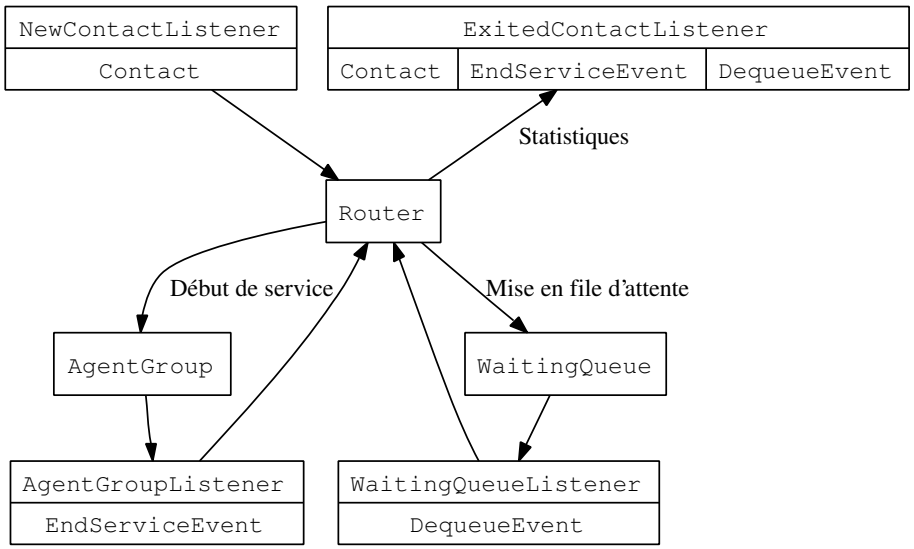

Figure 3.4 – Architecture du routeur

Si, dans le futur, cette conception ne s'avère pas assez flexible ou performante, il est possible de construire de nouvelles architectures de routeurs sans modifier les composantes élémentaires de la bibliothèque. Plusieurs architectures pourraient même exister de façon parallèle. Par exemple, un routeur utilisant une logique interprétée, construite par interface graphique et codée en XML, pourrait exister en parallèle avec le routeur actuel. La première solution favoriserait la simplicité d'utilisation tandis que la seconde privilégierait la performance.

Une autre implantation de routeur que nous envisageons dans le futur se fonde sur le patron des stratégies [11]. Avec ce modèle, pour chaque opération abordée à la section suivante (sélection d'agent, sélection de file, etc.), une interface est définie, avec une implantation distincte pour chaque algorithme supporté. La construction du routeur sélectionne alors une implantation pour chacune des différentes opérations. Cette technique permet d'obtenir un routeur très flexible, mais elle duplique le nombre de classes dans le système. Nous prévoyons implanter ce patron de conception si la flexibilité des politiques actuelles devient insuffisante.

3.4.2 Éléments constitutifs d'une politique de routage

L'implantation d'une politique de routage se fait par la création d'une sous-classe de `Router` définissant divers éléments par l'intermédiaire de champs ou de méthodes. Tout d'abord, une structure de données est habituellement nécessaire pour stocker des informations de routage, par exemple les conditions pour qu'un agent puisse servir un contact et les priorités à mettre en place. Ensuite, divers algorithmes utilisant cette structure sont nécessaires pour sélectionner les agents, les files d'attente et les contacts, vider les files d'attente et définir le comportement du routeur lorsque certains événements se produisent.

Un premier algorithme de sélection affecte un agent libre à un nouveau contact. Le choix est souvent restreint à des agents dans un ensemble de groupes dépendant du type de contact à acheminer. Par exemple, le routeur peut sélectionner l'agent ayant le plus long temps d'inactivité ou n'importe quel agent de façon aléatoire.

Un second algorithme est utilisé pour sélectionner une file d'attente lorsqu'un contact ne peut pas être servi immédiatement. Souvent, la file sélectionnée correspond au type de contact à acheminer, mais toute autre politique de sélection pourrait être mise en place.

Un troisième algorithme de sélection est employé lorsqu'un agent devient libre ou se branche au routeur afin de lui affecter un contact en file. L'ensemble des files à vérifier dépend souvent du groupe dans lequel l'agent libre se trouve. L'algorithme peut par exemple parcourir toutes les files accessibles à l'agent et retirer le contact ayant le plus long temps d'attente ou se trouvant dans la file contenant le plus grand nombre de clients.

Une politique de routage peut définir un quatrième algorithme pour supporter le vidage automatique des files d'attente afin d'éviter qu'un client arrivé peu de temps avant la fermeture n'attende la réouverture pour être servi. Lorsque ce mécanisme est activé et un contact ne peut être servi que par des agents dont le groupe est vide, il est automatiquement déconnecté. Par défaut, cet algorithme n'étant pas utilisé, seuls les abandons permettent de faire automatiquement en sorte qu'un client n'attendra pas indéfiniment après la fermeture du centre.

Finalement, il est possible de redéfinir le comportement du routeur lorsqu'un contact

termine un service ou quitte la file. Ceci permet, par exemple, de transférer un contact à un nouvel agent pour modéliser des services successifs ou de transférer un contact d'une file vers une autre après un temps d'attente trop long. Par défaut, le modèle le plus simple est considéré : les contacts bloqués, servis ou sortis d'une file quittent le système.

3.4.3 Structures de données pour le routage

Les politiques de routage actuellement implantées supportent deux types de structure de données : les listes ordonnées et la matrice de rangs. Chaque structure comporte ses avantages et est préférée par certaines politiques. Ces structures ont permis d'implanter diverses politiques de routage inspirées de [15, 31], mais elles ne permettent pas de couvrir toutes les situations possibles. Par exemple, des informations de routage distinctes pourraient être associées à chaque agent plutôt qu'à chaque groupe. De telles politiques peuvent heureusement être implantées en définissant de nouvelles structures de données.

3.4.3.1 Listes ordonnées

Pour chaque type de contact k, nous avons une liste ordonnée de groupes d'agents $i_{k,0}, i_{k,1}, \ldots$. De la même façon, pour chaque groupe d'agents i, nous avons une liste de types de contacts $k_{i,0}, k_{i,1}, \ldots$. Ces listes déterminent quels agents peuvent servir les contacts et définissent les priorités. L'importance de l'ordre induit dépend de la politique de routage choisie.

Le tableau 3.1 présente un exemple d'un tel ensemble de listes ordonnées inspiré de [15]. Le centre de contacts considéré possède trois types de contacts ($K = 3$) et deux groupes d'agents ($I = 2$). Les contacts de type 0 sont servis par des agents du groupe 0 seulement tandis que les contacts de type 2 sont servis par des agents du groupe 1. Les contacts de type 1 peuvent quant à eux être servis par tous les agents mais préfèrent ceux du groupe 0. Les agents dans le groupe 0 servent des contacts de types 1 et 0, accordant la priorité au type 1. Les agents du groupe 1 servent quant à eux des contacts de types 2 et 1 avec priorité pour le type 2. L'efficacité d'un tel routage dépend de la politique utilisée ainsi que des autres paramètres du système, par exemple le taux d'arrivée, les durées de

service, etc.

Tableau 3.1 – Exemple de listes ordonnées pour le routage

Type vers groupe

k	Liste
0	0
1	0, 1
2	1

Groupe vers type

i	Liste
0	1, 0
1	2, 1

`QueuePriorityRouter` constitue un exemple d'utilisation de ces structures de données. Lorsqu'un contact de type k est annoncé au routeur utilisant cette politique, ce dernier teste la présence d'agents dans le groupe $i_{k,0}$. Si au moins un agent est libre, le contact est servi par ce dernier. Dans le cas contraire, le routeur teste les groupes $i_{k,1}$, $i_{k,2}$ et ainsi de suite. Si aucun agent ne peut servir le contact, ce dernier est mis dans une file Q_k correspondant à son type.

Lorsqu'un agent dans le groupe i devient libre, la liste des types de contacts est consultée de façon similaire. Chaque indice $k_{i,j}$ correspond à une file d'attente $Q_{k_{i,j}}$ dont le premier contact présent est retiré si la file n'est pas vide. Si aucune file ne contient un contact, l'agent demeure libre jusqu'à une prochaine arrivée.

Il arrive parfois que seul un sous-ensemble des listes ordonnées soit disponible. Par exemple, les $i_{k,j}$ pourraient être donnés et non les $k_{i,j}$. Il est certes possible de générer les listes manquantes, mais l'ordre obtenu est alors arbitraire et le routeur qui en résulte peut parfois s'avérer mauvais. Pour diminuer l'impact de cet ordre sur la qualité de service, l'algorithme de sélection des contacts peut être altéré, en prenant par exemple la file contenant le plus grand nombre de contacts ou celle dont le premier contact a le plus long temps d'attente.

Les listes ordonnées permettent de bien visualiser l'ordre dans lequel les groupes d'agents et les files d'attente sont testés lors du routage. Les algorithmes les utilisant sont habituellement linéaires par rapport à leur taille. Malheureusement, elles interdisent le partage de priorités. Par exemple, si nous souhaitons que le routeur affecte à un agent libre du groupe 0 un contact de types 1 ou 0 ayant attendu le plus longtemps tout en donnant priorité aux contacts de type 2 pour les agents du groupe 1, il est nécessaire

de créer une politique de routage personnalisée puisque la structure de données ne peut contenir toute l'information de priorité.

Les listes ordonnées peuvent également causer des problèmes de cohérence. Dans l'exemple du tableau 3.1, si la liste associée au groupe 0 ne contenait que 1, la politique serait incohérente puisqu'elle affecterait des agents du groupe 0 aux contacts de type 0 nouvellement arrivés sans affecter des contacts de type 0 en attente aux agents du groupe 0 devenus libres. Dans cet exemple particulier, une telle politique ferait en sorte qu'un contact de type 0 mis en file y resterait jusqu'à abandonner. Pour éviter ce problème, des méthodes de détection des incohérences ont été implantées, mais elles doivent parcourir toutes les listes séquentiellement, ce qui prend un certain temps. Plusieurs routeurs peuvent être construits avec les mêmes paramètres, par exemple pour tester différentes politiques de routage, si bien que ces méthodes ne sont pas appelées automatiquement lors de la construction des routeurs.

3.4.3.2 Matrice de rangs

La matrice de rangs $I \times K$ associe, pour tous $i = 0, \ldots, I-1$ et tous $k = 0, \ldots, K-1$, une valeur $r(i,k)$ donnant le rang, c'est-à-dire un entier indiquant une priorité, du type de contact k pour tous les agents du groupe i. Si $r(i,k) = \infty$, les agents dans le groupe i ne peuvent pas servir les contacts de type k. Dans le cas contraire, plus le rang est petit, plus la préférence pour les contacts de type k est grande pour les agents du groupe i.

Sous Arena Contact Center Edition, chaque groupe d'agents doit posséder une liste de types de contacts pouvant être servis. À chaque type de contact dans cette liste est associée une priorité. Le routage disponible avec ce produit commercial ressemble ainsi à une matrice de rangs dont les rangées sont réparties à l'intérieur de différents modules.

Le tableau 3.2 présente un exemple d'une matrice de rangs pour un centre de contacts avec $K = 4$ et $I = 4$. Le modèle, que nous examinons plus en détails à la section 4.3.1, comprend deux types de contacts entrants, deux types sortants, deux groupes d'agents spécialisés pour les contacts entrants ou sortants ainsi que deux groupes mixtes. Chaque colonne de la matrice permet de construire des listes de priorités pour un type de contact afin d'effectuer la sélection d'un agent. Par exemple, les contacts entrants du premier

type sont servis en priorité par des agents dans le groupe entrant et en second lieu par des agents mixtes, car le rang pour les agents du premier groupe est inférieur à celui du second. Le routage pour les trois autres types de contacts est défini de façon similaire, avec des groupes d'agents différents. Pour cet exemple particulier, chaque colonne de la matrice permet d'obtenir des listes type vers groupe uniques, mais la conversion n'est pas toujours unique en général.

Une rangée de la matrice de rangs permet de dresser une liste de priorités de types de contacts pour un groupe d'agents précis afin d'effectuer la sélection d'un contact. Par exemple, les agents du groupe entrant peuvent servir des contacts du premier et du second type entrant avec le même niveau de priorité puisque les rangs sont égaux. Un algorithme secondaire, dépendant de la politique de routage, devra choisir dans quelle file prendre un contact en attente. Pour les deux premiers groupes d'agents, il n'est alors pas possible de construire des listes groupe vers type uniques puisqu'une priorité unique ne peut être associée à chaque type.

Tableau 3.2 – Exemple d'une matrice de rangs

	Type entrant 1	Type entrant 2	Type sortant 1	Type sortant 2
Groupe entrant	1	1	∞	∞
Groupe sortant	∞	∞	1	1
Groupe mixte 1	3	∞	2	∞
Groupe mixte 2	∞	3	∞	2

Le choix d'un agent peut s'effectuer par un parcours séquentiel des groupes pouvant servir le nouveau contact, comme le fait la politique implantée par `AgentsPrefRouter`. L'algorithme de ce routeur dresse une liste initiale de candidats et applique différents tests sur chacun d'eux jusqu'à ce qu'il reste au plus un agent. La liste initiale des candidats constitue tous les agents disponibles pouvant servir le contact à acheminer. Le routeur sélectionne un agent dans le groupe ayant le plus petit rang, c'est-à-dire la plus haute priorité. Si plusieurs groupes partagent le même rang minimal pour un type de contact donné, le routeur prend l'agent avec le plus long temps d'inactivité. Un contact ne pouvant être servi immédiatement est inséré dans une file correspondant à son type.

Lorsqu'un agent devient libre, une liste de types de contacts est dressée et restreinte

par plusieurs tests de façon semblable à la section d'agent. Le routeur sélectionne tout d'abord la file avec le plus petit rang. Si plusieurs files partagent le même rang minimal, le contact ayant le plus long temps d'attente est sélectionné.

La matrice de rangs élimine les problèmes de flexibilité des listes ordonnées, mais les routeurs qui l'utilisent sont plus complexes. Il arrive également qu'il soit plus facile de visualiser le comportement d'une politique de routage utilisant des listes ordonnées plutôt qu'une matrice de rangs.

3.5 Problèmes divers

3.5.1 Subdivision de l'horizon de simulation

Afin de simplifier la modélisation et l'estimation des paramètres des centres de contacts, l'horizon simulé (jour, semaine, mois, etc.) est habituellement divisé en périodes de quinze à soixante minutes entre lesquelles les taux d'arrivée, le nombre d'agents dans chaque groupe, les durées de service, etc. peuvent varier. Parfois, des statistiques sont également recueillies pour chacune des périodes séparément. Des événements sont utilisés pour marquer le début des périodes afin de provoquer la mise à jour du système. Dans ContactCenters, le modèle des observateurs est utilisé pour gérer cette mise à jour : chaque changement de période est annoncé à une liste d'observateurs enregistrés ajustant les paramètres ou les collecteurs statistiques sous leur contrôle. Le système supporte des périodes ayant des durées fixes ou variables.

Le centre de contacts comprend P périodes dites *principales* représentant les heures d'ouverture. Chaque période principale $p = 1,\ldots,P$ correspond à l'intervalle de temps $[t_{p-1},t_p)$, où $t_0 < \cdots < t_P$. Dans le cas fréquent où chacune de ces périodes a une durée fixe d, $t_p = t_0 + pd$ pour $p = 1,\ldots,P$. Souvent, la simulation doit débuter avant l'ouverture du centre de contacts, pour gérer des clients appelant juste avant l'arrivée d'agents, et se terminer après la fermeture, pour traiter les services en cours et vider les files d'attente. C'est pourquoi nous définissons deux périodes additionnelles : la *période préliminaire* $[0,t_0)$ pendant laquelle le centre de contacts n'est pas encore ouvert et la *période de fermeture* $[t_P,T]$ pendant laquelle aucune arrivée ne se produit et les agents

terminent leurs services.

Nous avons choisi de toujours faire débuter la période préliminaire à 0 afin que l'indice p de la période courante puisse être obtenu en tout temps pendant la simulation. Cette décision ne limite pas la flexibilité puisque les processus d'arrivée peuvent être démarrés ou arrêtés à tout moment. Ainsi, le centre de contacts peut demeurer inactif jusqu'au temps $0 < t_{-1} \leq t_0$ auquel les arrivées débutent.

Il est important que le temps t_{p-1} soit inclus dans la période p afin de ne pas compliquer inutilement l'ajustement des paramètres. Si tel n'était pas le cas, les observateurs de changement de période tentant de déterminer la période courante obtiendraient $p-1$ plutôt que p. L'ajustement des paramètres ne conviendrait alors pas à la nouvelle période sur le point de débuter.

Plutôt qu'une période préliminaire, l'utilisateur pourrait employer $P+1$ périodes principales dont seules les P dernières correspondraient à des heures d'ouverture. Par contre, un tel choix réduirait la performance puisqu'il serait moins souvent possible d'utiliser des périodes de durée fixe. Pour la collecte d'observations, il est parfois nécessaire d'obtenir la période $p(t)$ correspondant à un temps t arbitraire, par exemple pour compter une fin de service dans la période durant laquelle le contact est arrivé. Si les périodes ont une durée fixe,

$$p(t) = \left\lfloor \frac{t - t_0}{d} \right\rfloor \tag{3.5}$$

est obtenu en temps constant. Par contre, si les durées sont variables, une recherche binaire dans $O(\lg P)$ est nécessaire. Dans plusieurs modèles, les périodes principales ont une durée fixe, mais la période préliminaire a une durée différente. La différenciation de cette période permet d'utiliser (3.5) plus souvent et ainsi d'améliorer la performance. La période de fermeture, quant à elle, doit toujours être traitée différemment puisque sa durée est aléatoire, contrairement aux autres périodes.

Malheureusement, ces périodes additionnelles ne sont appropriées que si chaque réplication correspond à une journée. La solution la plus simple pour simuler une semaine ou un mois consiste à diviser l'horizon en périodes de durée fixe et à fixer $t_0 = 0$. Par ex-

emple, chaque heure de la semaine, y compris les heures de fermeture, peut correspondre à une période principale avec ses propres paramètres. Pendant les heures de fermeture, les taux d'arrivée sont fixés à 0 et aucun agent n'est disponible. À la fin de chaque période terminant une journée, si $N_i(t)$ est remis à 0, aucun agent du groupe i ne restera pour servir des contacts en file. Il est alors nécessaire de s'assurer que le centre se vide à ce moment-là, c'est-à-dire qu'aucun client n'attend indéfiniment. Ce comportement peut être obtenu en activant le vidage des files par le routeur ou en supportant les abandons. Cette approche est utilisée par Arena Contact Center Edition et a été adoptée pour simuler les exemples issus de ce logiciel (voir section 4.2.1).

3.5.2 Génération des variables aléatoires

Sous SSJ, deux approches sont disponibles pour générer des variables aléatoires. L'utilisateur peut employer une méthode statique spécifique à la loi de probabilité choisie ou construire un objet représentant cette loi. Dans tous les cas, des difficultés apparaissent pour traiter des lois de probabilité non stationnaires. De plus, les valeurs aléatoires peuvent dépendre de l'état du système.

3.5.2.1 Lois de probabilité dépendant de la période

L'approche la plus simple pour obtenir un générateur de variables aléatoires suivant une loi de probabilité non stationnaire consiste à construire un générateur de base et à modifier ses paramètres au besoin, par exemple au début de chaque période. Pour mettre le générateur à jour, il faut malheureusement utiliser une méthode et des paramètres dépendant de la loi de probabilité particulière mise en œuvre. Un changement de loi de probabilité implique alors des modifications à plusieurs endroits dans le programme. Avec cette approche, pour écrire un simulateur générique, il est nécessaire d'utiliser la réflexion pour appeler des méthodes connues à l'exécution seulement ou ne supporter qu'un nombre limité de lois de probabilité, ce qui diminue la performance, la clarté ou la fonctionnalité du programme, selon la solution choisie.

Si les paramètres d'une loi de probabilité varient de façon continue dans le temps, la

seule solution disponible consiste à utiliser les méthodes statiques des classes représentant les lois de probabilité. Cela conduit aux mêmes problèmes qu'avec un générateur de base mis à jour périodiquement, avec en plus une réduction de performance pour certaines lois précalculant des tables lorsqu'un objet est utilisé.

Pour le cas où les paramètres sont des fonctions constantes définies par morceaux, un générateur de variables aléatoires peut être construit pour chacune des périodes pendant laquelle la loi de probabilité ne change pas. Lorsqu'une valeur aléatoire est requise, la période courante doit alors être déterminée à partir du temps de simulation et le générateur adéquat est utilisé. De cette façon, la loi de probabilité peut être changée en modifiant uniquement le code construisant les générateurs et les variables peuvent être générées, dans un simulateur générique, sans recourir à la réflexion. Afin de faciliter l'emploi de cette solution, ContactCenters définit une classe de support permettant d'encapsuler un tableau définissant un générateur pour chaque période et se chargeant de sa sélection automatiquement.

3.5.2.2 Lois de probabilité dépendant d'autres paramètres

Pour générer des variables aléatoires, les composantes élémentaires, pour demeurer génériques, doivent utiliser des objets fournis par l'utilisateur plutôt qu'appeler des méthodes spécifiques à une loi de probabilité. Ainsi, plusieurs composantes peuvent encapsuler une référence vers un générateur de variables aléatoires qui peut être changé à tout moment. Malheureusement, selon l'endroit où se trouve cette référence, des limitations apparaissent. Par exemple, prenons le temps de service qui peut dépendre à la fois du type de contact k et du groupe d'agents i. En définissant le générateur approprié comme variable d'instance de `Contact`, il est impossible d'obtenir des temps de service dépendant du groupe de l'agent choisi. De la même façon, si le générateur est encapsulé dans `AgentGroup`, le temps de service ne peut plus dépendre du type de contact.

La flexibilité peut être améliorée en remplaçant la référence par un tableau. Par exemple, encore une fois pour le temps de service, chaque contact pourrait encapsuler un générateur pour chacun des I groupes d'agents ou chaque groupe d'agents pourrait con-

tenir un générateur pour chacun des K types de contacts. Toutefois, cette solution accroît la complexité des composantes élémentaires tout en ne couvrant pas d'autres aspects pouvant influencer le temps de service, par exemple l'agent choisi parmi les membres du groupe pour le service, un attribut du contact défini par l'utilisateur, etc.

Une solution plus générale au problème précédent consiste à ajouter des méthodes aux composantes élémentaires afin de générer les variables aléatoires. Par exemple, `AgentGroup` pourrait inclure une méthode nommée `getContactTime` retournant une durée de service. Chacune de ces méthodes de génération prend un contact en argument et retourne une valeur numérique. Habituellement, la logique de toutes ces méthodes est très semblable : soit une constante est retournée ou extraite du contact passé en argument, soit un générateur de variables aléatoires est sélectionné et une valeur est générée. Personnaliser le mécanisme de génération exige de créer une sous-classe et de redéfinir la méthode appropriée, ce qui peut facilement mener à la nécessité de créer, dans un simulateur, une sous-classe pour chaque composante élémentaire du système.

Pour combiner les avantages des approches précédentes, il est possible de définir une nouvelle interface semblable à celle de SSJ mais adaptée aux centres de contacts. De façon analogue à la classe de base `RandomVariateGen`, l'interface `ValueGenerator` que nous avons définie fournit une méthode `nextDouble` retournant une valeur numérique. Contrairement à son équivalent sous SSJ, cette méthode prend un contact en argument de façon à ce que ses attributs puissent affecter la valeur retournée. Par exemple, un générateur de valeurs est utilisé en lieu et place d'un générateur de variables aléatoires classique pour obtenir la probabilité de réussite d'un appel sortant ainsi que le délai nécessaire pour contacter le client.

Dans le cas des files d'attente, un générateur de valeurs peut être enregistré pour toute valeur de l'indicateur de sortie. Lorsqu'un temps d'attente maximal est nécessaire, tous les générateurs enregistrés sont utilisés pour obtenir des valeurs. Puisque chaque type de sortie de file correspond à un événement et qu'une seule sortie se produit, il est logique de définir le temps d'attente maximal comme le minimum de toutes les valeurs générées.

Par exemple, une sortie de type 1, représentant un abandon, pourrait survenir après

un temps de patience exponentiel tandis qu'une sortie de type 5 pourrait correspondre à une déconnexion ayant lieu après deux minutes. Si le temps de patience est inférieur à deux minutes, le temps d'attente maximal correspond au temps de patience et la sortie automatique est de type 1. Dans le cas contraire, le temps d'attente maximal est de deux minutes et la sortie de file prévue est de type 5. Si le temps de patience est exactement deux minutes, ce qui est peu probable, la première occurrence du minimum est utilisée, si bien que le type de sortie prévu est 1. Bien entendu, le type de sortie effectif peut différer du type prévu si l'événement représentant le contact en file est manipulé. Par exemple, il sera 0 si le contact est retiré par le routeur pour être servi.

Le même mécanisme est appliqué pour les temps de communication et les temps de travail après la communication dans le cas des groupes d'agents. Chaque groupe d'agents dispose d'une liste de générateurs pour chacun des deux types de durées. De même, chaque agent dispose de ses propres listes de générateurs qui ont priorité sur celles de son groupe si elles contiennent des éléments enregistrés.

Afin d'éviter que cette généralisation complique inutilement les simulateurs simples, des implantations par défaut de `ValueGenerator` sont disponibles. Un temps de patience, un temps de service et un temps de travail après la communication sont définis comme attributs de tout contact. Dans le cas des files d'attente, un générateur par défaut retournant le temps de patience d'un contact est associé au type de sortie 1. De façon similaire, des générateurs retournant le temps de communication et celui de travail après la communication d'un contact donné sont liés à l'indicateur 0 de tous les groupes d'agents. Par défaut, les listes de générateurs pour chaque agent sont vides afin que celles de son groupe soient utilisées pour obtenir les temps de service.

Les variables aléatoires stockées comme attributs des contacts peuvent être générées lors de l'instanciation des objets `Contact` plutôt qu'au moment où elles sont nécessaires. Ceci permet d'augmenter la synchronisation des variables aléatoires puisque toutes les variables sont toujours générées dans le même ordre. Si, lors de l'instanciation du contact, l'information nécessaire à la génération d'une variable aléatoire n'est pas disponible, l'utilisateur peut alors implanter l'interface `ValueGenerator` et affecter le générateur à la file d'attente, au groupe d'agents et à l'agent adéquats.

3.5.3 Gestion des observations statistiques

Dans un simulateur écrit avec ContactCenters, la gestion statistique se passe en deux temps : le traitement des événements qui se produit tout au long d'une réplication et la collecte des observations qui a lieu à sa toute fin. Pendant une réplication, plusieurs événements mettent à jour de nombreux compteurs. Par exemple, un simulateur peut calculer le nombre de contacts servis, la somme des temps d'attente, l'intégrale de la taille de la file en fonction du temps, etc. À la fin d'une réplication, les valeurs obtenues constituent des observations qui peuvent être transformées avant leur collecte. Par exemple, certaines valeurs numériques comme le nombre de contacts servis peuvent être normalisées en utilisant la durée de simulation. Les événements sont traités à divers moments pendant la simulation et les compteurs sont réinitialisés entre chaque réplication. Les observations sont quant à elles recueillies à la fin des réplications et les collecteurs ne sont pas réinitialisés.

Les composantes élémentaires de la bibliothèque n'effectuent aucun traitement d'événements statistiques. En particulier, les groupes d'agents ne conservent pas une trace du nombre de membres en fonction du temps et les files d'attente ne relèvent aucune statistique sur leurs tailles. Cette décision a été prise, car il existe un trop grand nombre de mesures pouvant être calculées. Tenir compte de toutes ces mesures réduirait la performance ou la modularité tandis que se limiter à un sous-ensemble affecterait la flexibilité.

Par exemple, en plus de l'intégrale de $N_{\mathrm{b},i}(t)$, il est possible de calculer celles de $N_{\mathrm{b},i,k}(t)$ pour $k = 0, \ldots, K-1$. Pour calculer efficacement ces intégrales, les groupes d'agents doivent connaître K, une information dont ils n'ont pas réellement besoin. Ce nombre devrait être donné lors de la construction du groupe d'agents et mis à jour s'il est modifié. Il est possible d'éviter cette complication en créant une classe statique contenant l'information, mais cette classe rendrait plus difficile le test unitaire des groupes d'agents.

Le traitement d'événements est ainsi effectué par des observateurs enregistrés au niveau des composantes appropriées. À priori, pour gérer le traitement des événements statistiques, il semble suffisant d'observer les groupes d'agents et les files d'attente di-

rectement. Malheureusement, ce mécanisme répartit le traitement des événements dans plusieurs classes et ne fonctionne bien que lorsque les contacts sont servis par un seul groupe d'agents et attendent dans une seule file. Dans le cas contraire, il devient difficile d'éviter de compter un contact plusieurs fois.

Seul le routeur est en mesure de déterminer quand un contact quitte effectivement le système. Le système de traitement d'événements doit ainsi s'enregistrer auprès de ce dernier pour obtenir les informations adéquates. En employant un observateur de contacts sortants, l'utilisateur évite de compter un contact plusieurs fois ou d'omettre les contacts bloqués. De plus, la majeure partie du code de traitement se trouve concentrée dans une même classe, améliorant la lisibilité du programme.

Pour les intégrales sur le nombre d'agents et la taille des files d'attente, ContactCenters fournit des classes encapsulant des collecteurs statistiques pour automatiser leur calcul. La classe `GroupVolumeStat` définit un observateur qui permet d'obtenir les intégrales de $N_i(t)$, $N_{\mathrm{b},i}(t)$, $N_{\mathrm{b},i,k}(t)$, $N_{\mathrm{i},i}(t)$, $N_{\mathrm{f},i}(t)$ et $N_{\mathrm{g},i}(t)$ pour tout groupe d'agents. Pour chaque file d'attente, `QueueSizeStat` peut quant à elle calculer l'intégrale de la taille et les intégrales du nombre de contacts en attente de chaque type.

De la même façon que le traitement d'événements, il n'existe aucun mécanisme prédéfini de collecte d'observations, car pour une même mesure de performance, il peut exister de multiples définitions, par exemple (1.4) et (1.5) pour le niveau de service. Tous les cas ne peuvent être couverts par la bibliothèque et de nouveaux cas peuvent surgir à tout moment. Le traitement des observations est alors effectué par le simulateur écrit par l'utilisateur.

3.6 Interface de haut niveau

Lorsque les composantes élémentaires sont interconnectées et les différents problèmes de gestion sont résolus, nous obtenons un simulateur de centres de contacts à part entière. Puisque tous ces simulateurs partagent beaucoup de code en commun, il devient intéressant de créer des systèmes génériques couvrant le plus grand nombre possible d'aspects. Comme les logiciels commerciaux, un simulateur générique implante

un modèle complexe dont les paramètres sont ajustés par l'utilisateur. Il devient alors possible d'éviter la programmation pour l'utilisateur, voire même, à très long terme, de proposer une interface graphique lui permettant de définir les paramètres.

Idéalement, un seul simulateur universel devrait exister, mais si le modèle change trop profondément, un nouveau programme devra être créé. En effet, un simulateur complexe et universel devient vite difficile à maintenir et une modification visant à changer le modèle peut introduire des bogues ou réduire les performances. Un système plus spécifique à un modèle peut de son côté bénéficier d'optimisations améliorant la performance. Un simulateur simple facilite les expérimentations avec divers estimateurs dans le but, par exemple, de réduire la variance. Plusieurs simulateurs pourraient ainsi exister et, sans une interface de communication uniformisée, passer de l'un à l'autre exigerait l'adaptation d'une grande partie du programme d'application. Cette interface peut également fournir le niveau d'abstraction nécessaire pour permettre à une application d'utiliser la simulation ou des formules analytiques de façon interchangeable.

La première tentative d'intégration d'un simulateur utilisant ContactCenters à un autre programme visait à optimiser l'affectation des agents dans un centre d'appels tenant compte de leurs compétences [8]. Le programme d'optimisation calcule un nombre d'agents N_i pour chaque groupe $i = 0, \ldots, I-1$ en utilisant la programmation en nombres entiers ou linéaire. Il déclenche ensuite une simulation sur horizon infini avec $N_i(t) = N_i$ pour $i = 0, \ldots, I-1$ et corrige le vecteur $(N_0, \ldots, N_{I-1})$ afin que le niveau de service soit suffisamment élevé. L'algorithme itératif utilisé appelle le simulateur plusieurs fois avec des vecteurs d'affectation différents mais en utilisant des variables aléatoires communes afin d'obtenir le niveau de service et son sous-gradient par différences finies.

La première tentative d'interface de haut niveau visait à rendre le simulateur accessible depuis un programme d'optimisation écrit en C et implantant l'algorithme précédent. Il accédait à la bibliothèque commerciale CPLEX [14] pour la programmation en nombres entiers et devait appeler le simulateur Java pour estimer le niveau de service. Pour ce faire, le programme C créait une machine virtuelle Java et accédait au simulateur par le biais de Java Native Interface (JNI, [24]). Cette interface causait plusieurs difficultés, car lorsque le simulateur était étendu, il fallait modifier le programme d'application

en C pour passer de nouveaux paramètres à son constructeur. JNI allourdit beaucoup le code si bien que la maintenance de l'interface nécessitait beaucoup de travail afin de permettre un accès simple depuis le programme C l'utilisant.

Cette tentative a permis de cerner les différents problèmes à résoudre pour gérer les paramètres des modèles et faire interagir un simulateur avec un programme d'optimisation. Lorsqu'une interface de communication suffisamment générale a pu être construite, il a été possible de mettre au point un simulateur générique s'adaptant à un grand nombre de systèmes.

3.6.1 Gestion des paramètres

Dans les programmes d'exemple comme celui que nous verrons à la section 4.1, les paramètres du modèle sont encodés dans des champs de la classe implantant le simulateur. Cette approche ne peut pas être utilisée dans un système générique, car il faudrait recompiler le simulateur à chaque changement des paramètres.

Pour éviter ces recompilations, il vaut mieux séparer le code implantant la logique de simulation des paramètres du modèle. Pour y parvenir, les paramètres sont lus depuis un fichier et stockés dans une structure de données intermédiaire passée au constructeur du simulateur. De cette façon, son appel nécessite un nombre limité d'arguments et demeure inchangé même si de nouveaux paramètres sont ajoutés dans le futur. La structure de données pour stocker les paramètres consiste en un objet d'une classe de données définissant des champs pour stocker les valeurs et la logique nécessaire pour leur lecture. Ainsi, le simulateur et le programme d'optimisation peuvent utiliser les paramètres tout en évitant de dupliquer le mécanisme de lecture.

Cette approche exige de spécifier un format binaire ou textuel pour le fichier de données lu par l'objet de paramètres. Un fichier binaire est très facile à analyser pour un programme et largement utilisé par les logiciels commerciaux. Les valeurs numériques peuvent être lues sans conversion depuis des chaînes de caractères et les erreurs de syntaxe sont plutôt rares puisque le fichier n'est pas manipulé directement par l'utilisateur. Java fournit plusieurs mécanismes pour faciliter la lecture, le plus élaboré étant la sérialisation qui permet de convertir un objet en une chaîne d'octets et vice versa. Un format

binaire exige toutefois la création d'un éditeur spécialisé afin de manipuler les fichiers. Pour permettre à l'utilisateur de modifier le fichier efficacement, cet éditeur doit inévitablement proposer une interface graphique. L'interface doit être mise à jour lors de chaque ajustement du format et il n'est pas possible d'utiliser un simulateur employant le fichier de paramètres binaire tant qu'elle n'est pas terminée.

Un format textuel est de son côté plus facile à lire pour l'utilisateur et les fichiers peuvent être édités manuellement ou générés par un script ou un programme. Le simulateur utilisant le format peut être construit plus rapidement tandis qu'un éditeur dédié accroissant la simplicité d'utilisation peut être écrit plus tard. Toutefois, la lecture par un programme est beaucoup plus complexe : des conversions depuis des chaînes de caractères sont nécessaires et plusieurs variations syntaxiques sont possibles. Par exemple, l'utilisateur peut insérer un ou plusieurs blancs pour séparer les valeurs, laisser des lignes vides, commettre des erreurs de syntaxe, etc. L'analyseur doit pouvoir détecter les erreurs et en indiquer la source afin d'aider l'utilisateur à les corriger.

Afin de pouvoir nous concentrer sur l'écriture d'un simulateur plutôt que celle d'un éditeur de fichiers binaires, nous avons préféré un format textuel pour tous les fichiers de configuration. Pour des cas simples, un fichier séquentiel avec un paramètre par ligne suffit, mais il devient vite difficile à gérer et les erreurs sont indétectables. Par exemple, si le simulateur nécessite de lire le nombre de types de contacts, le nombre de groupes d'agents puis le nombre de périodes, une valeur par ligne, et l'utilisateur omet l'une des lignes, le programme n'en saura rien et lira des valeurs incorrectes. Puisqu'aucun paramètre optionnel n'est possible, étendre le format devient difficile.

Plutôt que le format précédent, il serait possible d'utiliser les fichiers de propriétés pour la lecture des paramètres. Chaque ligne d'un tel fichier contient une paire (clé, valeur) définissant une propriété ou débute par un # pour représenter un commentaire qui est ignoré. La syntaxe de ces fichiers est simple et Java fournit un mécanisme pour les lire efficacement. L'utilisateur peut modifier l'ordre d'apparition des propriétés dans le fichier et certains paramètres peuvent être optionnels. Toutefois, puisque les fichiers de propriétés ne sont pas hiérarchiques, leur syntaxe ne met pas clairement en évidence les groupes que forment les types de contacts, les groupes d'agents, etc. Des propriétés

complexes telles que les tables de routage sont aussi difficiles à représenter sous forme de chaînes de caractères tenant sur une seule ligne. Ce système serait convenable pour des modèles simples, mais il l'est moins pour des cas plus complexes.

Nous avons choisi d'utiliser *eXtensible Markup Language* (XML, [32]), car ce métalangage permet la création de documents complexes et hiérarchiques. Sa syntaxe indique comment encoder des éléments, leurs attributs ainsi que leur contenu tandis que l'auteur d'un format est libre de déterminer quelles structures sont autorisées.

Un *élément* XML est caractérisé par un nom, un certain nombre d'attributs et un contenu. Un *attribut* constitue une paire (clé, valeur) rattachée à un élément précis. Le *contenu* d'un élément peut être formé de texte ou d'autres éléments. Un *document* est composé d'une en-tête et d'un élément appelé *racine*. Par exemple, le code XML

```
<el attr="val">texte<enfant/></el>
```

définit l'élément `el` possédant l'attribut `attr` ayant la valeur `val` et dont le contenu est composé du texte `texte` suivi de l'élément `enfant`.

Malgré les redondances imposées par son format, un fichier XML étant textuel, il peut être manipulé avec n'importe quel éditeur. Pour améliorer la simplicité d'utilisation, il vaut mieux utiliser un logiciel spécialement adapté pour le XML, par exemple XMLSpy d'Altova [1], car un tel outil permet d'éviter les erreurs de syntaxe comme l'omission d'un marqueur de fermeture, le formatage incomplet d'un élément, etc.

3.6.2 Interaction avec le simulateur

Grâce au système de gestion des paramètres de la section précédente, la construction d'un objet représentant un simulateur est nettement facilitée, la longue liste d'arguments passés au constructeur étant dès lors remplacée par un seul objet. À présent, nous définissons une interface implantée par les simulateurs et permettant leur interaction avec d'autres programmes. L'interface a été conçue en premier lieu pour permettre à un programme d'optimisation d'utiliser un simulateur ou une formule d'approximation de façon interchangeable, sans nécessairement connaître tout le format de paramètres, ce qui

nécessite un certain niveau d'abstraction.

Définissons d'abord une *évaluation* comme un processus permettant d'estimer un certain nombre de mesures de performance en fonction de paramètres réglables par l'usager. L'estimation peut être obtenue par une formule analytique ou par simulation tandis que les paramètres proviennent d'un fichier ou sont fournis par un programme. Un tel système d'évaluation spécialisé pour les centres de contacts est représenté par l'interface `ContactCenterEval` qui définit deux types d'entrées : les paramètres internes et les options externes.

Par exemple, dans le cas d'un simulateur, les paramètres internes sont contenus dans un fichier XML et convertis en un objet. Les options d'évaluation incluent le vecteur d'affectation des agents et la condition d'arrêt. Chaque évaluation déclenche un certain nombre de réplications de la simulation et réinitialise les germes des générateurs de variables aléatoires. Un système d'évaluation faisant appel à une formule d'approximation utilisera le même objet de paramètres que le simulateur, mais il n'emploiera qu'un sous-ensemble des valeurs.

De cette façon, un optimiseur peut évaluer le niveau de service du système avec différents vecteurs d'affectation des agents et utiliser les variables aléatoires communes lorsque la simulation est mise en œuvre. Seule une très petite partie du code, excluant idéalement l'algorithme d'optimisation, connaît la classe exacte implantant le système d'évaluation. Contrairement aux solutions commerciales, il est dès lors possible d'évaluer, lors d'une optimisation, l'impact d'une approximation puisque le même programme peut fonctionner indifféremment avec un simulateur ou une formule analytique.

Les mesures de performance sont regroupées en matrices dont chaque ligne correspond à une condition d'observation et chaque colonne, à une période. Par exemple, la matrice de mesures pour le niveau de service comprend une ligne pour chaque type de contact ainsi qu'une ligne pour le niveau global. Dans le cas d'une formule d'approximation ou d'une simulation fournissant des résultats pour une période à l'état stationnaire, les matrices de mesures ne contiennent qu'une colonne.

Il existe un très grand nombre de groupes de mesures de performance et de nouveaux groupes peuvent s'ajouter à tout moment. Définir une méthode d'accès pour ob-

tenir une estimation pour chaque groupe de mesures rendrait l'interface `ContactCenterEval` très difficile à implanter et l'ajout d'un nouveau type de mesure causerait la modification de toutes les implantations existantes. Nous avons ainsi décidé de définir une seule méthode appelée `getPerformanceMeasure` et prenant en argument une clé indiquant la matrice de mesures à obtenir. Plutôt qu'une valeur numérique ou une chaîne de caractères, cette clé consiste en une instance d'un type énuméré [5] nommé `PerformanceMeasureType` et définissant un champ pour chaque type de mesure supporté. L'interface d'évaluation définit également des méthodes permettant de déterminer quels groupes de mesures sont disponibles pour une implantation donnée et pour formater les matrices de mesures dans le but de les afficher à l'écran.

Un système similaire a été défini pour les options d'évaluation. La méthode `setEvalOption` accepte ainsi une clé de type `EvalOptionType` et un objet fournissant la valeur de l'option. La classe exacte de la valeur dépend de l'option particulière, par exemple un tableau d'entiers pour le vecteur d'affectation des agents. L'interface définit également des méthodes permettant de tester si un système d'évaluation supporte une option donnée.

Cette interface peut facilement être étendue sans en modifier sa spécification : de nouveaux groupes de mesures de performance peuvent être définis et de nouvelles options d'évaluation peuvent être mises en place. Le programme dispose de la possibilité de tester si le système d'évaluation mis en œuvre dispose des fonctionnalités nécessaires pour ses besoins.

Toutefois, la simulation fournit des résultats qui ne sont pas disponibles avec les formules d'approximation, par exemple la variance empirique et des intervalles de confiance pour chaque mesure de performance estimée. Encore une fois, l'information est regroupée en matrices pour en faciliter le traitement. Pour tenir compte de ces aspects, l'interface `ContactCenterSim`, qui étend `ContactCenterEval`, fournit les méthodes additionnelles nécessaires pour obtenir ces résultats d'une façon indépendante du simulateur. Chacune de ces méthodes, comme `getPerformanceMeasure`, prend une clé en argument et retourne une matrice.

3.6.3 Construction d'un simulateur générique

La meilleure façon de maximiser la simplicité d'utilisation d'un système de simulation consiste sans doute à minimiser la programmation à effectuer pour l'utilisateur. C'est pourquoi, après avoir conçu l'interface de communication présentée à la section précédente, nous avons tenté d'implanter un simulateur générique. Ce simulateur permet de traiter un grand nombre de centres d'appels dont les paramètres sont stockés dans des fichiers de données XML et son utilisation est nettement plus simple que l'écriture d'un programme directement avec la bibliothèque. Malheureusement, un tel simulateur ne couvre pas tous les centres de contacts possibles et la prise en charge d'un aspect non supporté nécessite de la programmation.

La classe `MSKCallCenterSim` implante un modèle de centre d'appels mixte supportant K_{I} types d'appels entrants et K_{O} types sortants. La plupart des paramètres sont spécifiés seulement pour les périodes principales. Pendant la période préliminaire, aucun agent n'est en service et pour les autres paramètres, les valeurs de la première période principale sont utilisées. Pendant la période de fermeture, les paramètres de la dernière période principale sont employés.

Les indices $0,\ldots,K_{\mathrm{I}}-1$ correspondent à des types d'appels entrants tandis que les indices $K_{\mathrm{I}},\ldots,K-1$ réfèrent à des types sortants. Chaque source de contacts produit des appels d'un seul type et peut être activée ou désactivée en tout temps durant la simulation. Les processus d'arrivée pour les appels entrants sont choisis parmi une liste prédéfinie, mais l'utilisateur peut définir les paramètres des arrivées.

Chaque type d'appel sortant dispose de son propre composeur avec sa politique de numérotation et ses paramètres, incluant la probabilité de succès durant chaque période. Il est également possible de définir, pour chaque période, la loi de probabilité pour les temps entre la composition et le succès ou l'échec de l'appel. Les composeurs en fonction sont déclenchés chaque fois qu'un service est terminé.

Le routeur reçoit tous les appels entrants et les appels sortants réussis. L'utilisateur doit sélectionner une politique de routage parmi une liste de politiques prédéfinies et la paramétrer à l'aide de listes ordonnées ou d'une matrice de rangs. Selon la politique

choisie, le simulateur préférera une structure de données plutôt qu'une autre.

Les appels ne pouvant être servis immédiatement sont insérés dans une file d'attente et peuvent abandonner immédiatement avec une probabilité dépendant à la fois de leur type et de leur période d'arrivée. Les appelants décidant d'attendre peuvent devenir impatients et abandonner. Pour les appels de type k arrivant pendant la période p, les temps de patience sont indépendants et identiquement distribués (i.i.d.) et suivent une loi de probabilité quelconque. Les durées de service pour les appels de type k arrivés pendant la période p et servis par des agents du groupe i sont elles aussi i.i.d. et suivent une loi de probabilité définie par l'utilisateur. Pour chaque groupe d'agents défini, il est possible de fixer le nombre de membres pour chaque période ainsi que l'efficacité ε_i.

Le simulateur calcule différentes statistiques comme le nombre d'appels produits, servis, bloqués ou ayant abandonné. Il calcule également le taux d'occupation des agents, la taille moyenne des files d'attente et le nombre d'appels servis par chaque groupe d'agents. Chaque statistique est calculée pour chaque période p ainsi que pour tout l'horizon. Chaque événement relatif à un appel est compté dans la période de son arrivée et non dans celle où il se produit. Ceci est nécessaire pour éviter d'introduire un biais dans les estimateurs de rapports d'espérances. Par exemple, si plusieurs appels arrivaient pendant la période p et étaient servis pendant la période $p+1$, la valeur du niveau de service dans la période $p+1$ pourrait dépasser 1 si tous les événements étaient comptés dans la période où ils se produisent.

La simulation peut être effectuée de façon stationnaire pour une seule période de durée supposément infinie dans le modèle, en utilisant la méthode des moyennes par lots pour obtenir des intervalles de confiance (voir section 5.3.2), ou pour tout l'horizon, avec un nombre donné de réplications indépendantes (voir section 5.3.1). Dans le cas stationnaire (horizon infini), le simulateur est initialisé avec les paramètres pour une période et ces paramètres demeurent fixes tout au long de l'expérience. Dans le cas non stationnaire (horizon fini), les paramètres peuvent changer d'une période à l'autre. Simuler sur horizon infini ne semble pas naturel pour des centres de contacts, mais il peut s'avérer utile de le faire pour comparer les résultats de simulation avec des approximations considérant souvent le système comme stationnaire.

CHAPITRE 4

EXEMPLES

Afin de démontrer les possibilités de la bibliothèque, nous présentons maintenant un exemple complet et commenté de centre de contacts. Nous enchaînons avec une comparaison de performance entre la bibliothèque et un logiciel commercial largement utilisé, effectuée en mesurant le temps d'exécution sur plusieurs exemples. Nous montrons ensuite comment l'interface de haut niveau présentée au chapitre précédent peut être mise en œuvre.

Les exemples présentés dans ce chapitre ne couvrent pas toutes les possibilités de la bibliothèque. La documentation de ContactCenters comprend un guide en format PDF contenant plusieurs autres exemples complets et incluant les programmes de test utilisés pour les comparaisons de performance de la section 4.2.

4.1 Exemple de simulateur écrit avec ContactCenters

Le système modélisé dans cet exemple, que nous avons présenté pour la première fois dans [7], comprend trois types de contacts, deux groupes d'agents et les journées sont divisées en trois périodes de deux heures. Les contacts de type k arrivent selon un processus de Poisson à un taux aléatoire $B\lambda_{k,p}$ durant la période p. Les taux de base $\lambda_{k,p}$ sont déterministes tandis que B constitue une variable aléatoire dont la loi de probabilité est gamma avec paramètres (α_0, α_0). Généré au début de chaque jour, B a une moyenne de 1, une variance de $1/\alpha_0$ et représente le facteur d'occupation du système [3]. Si $B > 1$, le taux d'arrivée des contacts est plus élevé que d'habitude. Si $B < 1$, il est inférieur à la normale.

Lorsqu'une arrivée se produit, le contact entrant dans le système est affecté à un agent sélectionné en fonction de son type k. Comme dans l'exemple du tableau 3.1, les contacts de type 0 ne peuvent être servis que par des agents dans le groupe 0 tandis que ceux de type 2 ne sont servis que par des agents dans le groupe 1. Les contacts de type 1

sont servis par des agents dans le groupe 0, mais si $N_{\mathrm{f},0}(t) = 0$, ils peuvent être envoyés à des agents dans le groupe 1. Les temps de service pour des contacts arrivant durant la période p sont i.i.d. et suivent la loi exponentielle de moyenne $1/\mu_p$.

Les agents dans les groupes ne sont pas différenciés et leur nombre $N_i(t)$ est constant durant une période mais peut varier d'une période à l'autre. Si $N_{\mathrm{b},i}(t) \geq N_i(t)$, tous les services en cours sont terminés et le groupe i n'accepte aucun nouveau contact jusqu'à ce que $N_{\mathrm{b},i}(t) < N_i(t)$.

Un contact ne pouvant être servi immédiatement est inséré dans une file d'attente correspondant à son type et supportant les abandons. Pour les contacts arrivés durant la période p, les temps de patience sont i.i.d. et suivent la loi exponentielle de moyenne $1/\nu_p$. Aucune priorité n'est appliquée lors de la sélection d'un contact dans une file.

Le programme calcule certaines statistiques définies dans la section 1.2. Nous souhaitons estimer le niveau de service pour tous les contacts et le taux d'occupation des agents du groupe 0, $o_0(t_0, T)$, où $T \geq t_P$ est le temps de simulation où le centre de contacts est fermé et toutes les files d'attente sont vides. Ces mesures sont définies par (1.4) et (3.3), respectivement. Nous estimons aussi $E[X_{\mathrm{g},k,p}(s)]$, le nombre espéré de contacts de type k arrivés pendant la période p et servis après un temps d'attente inférieur à s, pour tout type de contact k et période p. Pour ce faire, la simulation est répétée n fois de façon indépendante afin d'obtenir des copies i.i.d. des variables aléatoires nécessaires. Les espérances sont ensuite estimées en les remplaçant par des moyennes.

Le listing 4.1 présente le programme de simulation complet. La première partie définit des constantes et des variables pour ensuite créer différents objets. La seconde partie contient des méthodes et des classes internes utilisant les éléments créés pour implanter la logique de simulation. La méthode `main` se trouvant à la toute fin et appelée lors de l'exécution du programme construit un simulateur grâce à `new SimpleMSK()`, déclenche la simulation à l'aide de `simulate` et affiche un rapport statistique en utilisant `printStatistics`. La méthode `simulate` appelle `simulateOneDay` n fois tandis que cette dernière méthode initialise le système pour une nouvelle réplication, déclenche la simulation et recueille un certain nombre d'observations. Examinons maintenant plus en détails les différentes parties de ce programme.

Listing 4.1 – Exemple de simulateur écrit avec ContactCenters

```
import umontreal.iro.lecuyer.rng.MRG32k3a;
import umontreal.iro.lecuyer.probdist.ExponentialDist;
import umontreal.iro.lecuyer.probdist.GammaDist;
import umontreal.iro.lecuyer.randvar.RandomVariateGen;
import umontreal.iro.lecuyer.randvar.ExponentialGen;
import umontreal.iro.lecuyer.randvar.GammaGen;
import umontreal.iro.lecuyer.stat.Tally;
import umontreal.iro.lecuyer.stat.RatioTally;
import umontreal.iro.lecuyer.stat.matrix.MatrixOfTallies;
import umontreal.iro.lecuyer.contactcenters.PeriodChangeEvent;
import umontreal.iro.lecuyer.contactcenters.MultiPeriodGen;
import umontreal.iro.lecuyer.contactcenters.contact.Contact;
import umontreal.iro.lecuyer.contactcenters.contact.ContactFactory;
import umontreal.iro.lecuyer.contactcenters.contact.
       PiecewiseConstantPoissonArrivalProcess;
import umontreal.iro.lecuyer.contactcenters.server.AgentGroup;
import umontreal.iro.lecuyer.contactcenters.server.GroupVolumeStat;
import umontreal.iro.lecuyer.contactcenters.queue.WaitingQueue;
import umontreal.iro.lecuyer.contactcenters.queue.StandardWaitingQueue;
import umontreal.iro.lecuyer.contactcenters.router.Router;
import umontreal.iro.lecuyer.contactcenters.router.SingleFIFOQueueRouter;
import umontreal.iro.lecuyer.contactcenters.router.ExitedContactListener;
import umontreal.iro.lecuyer.simevents.Event;
import umontreal.iro.lecuyer.simevents.Sim;

public class SimpleMSK {
   // Tous les temps sont en minutes
   static final int K                  = 3;        // Nombre de types de contacts
   static final int I                  = 2;        // Nombre de groupes d'agents
   static final int P                  = 3;        // Nombre de périodes
   static final double PERIODDURATION  = 120.0;    // Deux heures
   // LAMBDA[k][p] donne le taux d'arrivée pour le type k durant la période p
   static final double[][] LAMBDA      =
      { { 0, 4.2, 5.3, 3.2, 0 }, { 0, 5.1, 4.3, 4.8, 0 }, { 0, 6.3, 5.2, 4.8, 0 } };
   // Paramètre gamma pour le facteur d'occupation
   static final double ALPHA0          = 28.7;
   // Taux de service pour chaque période
   static final double[] MU   = { 0.5, 0.5, 0.6, 0.4, 0.4 };
   // Taux d'abandon pour chaque période
   static final double[] NU   = { 0.3, 0.3, 0.4, 0.2, 0.2 };
   // Temps d'attente acceptable (20s)
   static final double AWT    = 20/60.0;
   // NUMAGENTS[i][p] donne le nombre d'agents dans le groupe i,
   // pendant la période p
   static final int[][] NUMAGENTS = { { 0, 12, 18, 9, 9 }, { 0, 15, 20, 11, 11 } };
   // Table de routage, TYPETOGROUPMAP[k] et GROUPTOTYPEMAP[i] contiennent
   // des listes ordonnées
   static final int[][] TYPETOGROUPMAP  = { { 0 }, { 0, 1 }, { 1 } };
   static final int[][] GROUPTOTYPEMAP = { { 1, 0 }, { 2, 1 } };
```

```
static final double LEVEL   = 0.95;      // Niveau des intervalles de confiance
static final int NUMDAYS    = 10000;     // Nombre de réplications

PeriodChangeEvent pce;      // Événement marquant le début des périodes
PiecewiseConstantPoissonArrivalProcess[] arrivProc
   = new PiecewiseConstantPoissonArrivalProcess[K];
AgentGroup[] groups = new AgentGroup[I];
WaitingQueue[] queues = new WaitingQueue[K];
Router router;
RandomVariateGen sgen;                   // Générateur de temps de service
RandomVariateGen pgen;                   // Générateur de temps de patience
RandomVariateGen bgen;                   // Générateur du facteur d'occupation

// Compteurs
int numGoodSL, numServed, numAbandoned, numAbandonedAfterAWT;
double[][] numGoodSLKP = new double[K][P];
GroupVolumeStat vstat;                   // Intégrales pour le taux d'occupation

// Collecteurs statistiques
Tally served = new Tally ("Number of served contacts");
Tally abandoned = new Tally ("Number of contacts having abandoned");
MatrixOfTallies goodSLKP = new MatrixOfTallies
   ("Number of contacts meeting target service level",
    new String[] { "Type 0", "Type 1", "Type 2" },
    new String[] { "Period 0", "Period 1", "Period 2" });
RatioTally serviceLevel = new RatioTally ("Service level");
RatioTally occupancy = new RatioTally ("Occupancy ratio");

SimpleMSK() {
   // Une période préliminaire de durée 0, P périodes principales et
   // une période de fermeture, les périodes principales commencent à 0.
   pce = new PeriodChangeEvent (PERIODDURATION, P + 2, 0);
   for (int k = 0; k < K; k++)                    // Pour tout type de contact
      arrivProc[k] = new PiecewiseConstantPoissonArrivalProcess
         (pce, new MyContactFactory (k), LAMBDA[k], new MRG32k3a());
   bgen = new GammaGen (new MRG32k3a(), new GammaDist (ALPHA0, ALPHA0));
   for (int i = 0; i < I; i++) groups[i] = new AgentGroup (pce, NUMAGENTS[i]);
   for (int q = 0; q < K; q++) queues[q] = new StandardWaitingQueue();
   sgen = MultiPeriodGen.createExponential (pce, new MRG32k3a(), MU);
   pgen = MultiPeriodGen.createExponential (pce, new MRG32k3a(), NU);
   router = new SingleFIFOQueueRouter (TYPETOGROUPMAP, GROUPTOTYPEMAP);
   for (int k = 0; k < K; k++) arrivProc[k].addNewContactListener (router);
   for (int i = 0; i < I; i++) router.setAgentGroup (i, groups[i]);
   for (int q = 0; q < K; q++) router.setWaitingQueue (q, queues[q]);
   router.addExitedContactListener (new MyContactMeasures());
   vstat = new GroupVolumeStat (groups[0]);
}

// Crée les nouveaux contacts
class MyContactFactory implements ContactFactory {
```

```
   int type;
   MyContactFactory (int type) { this.type = type; }
   public Contact newInstance() {
      Contact contact = new Contact (type);
      contact.setDefaultServiceTime (sgen.nextDouble());
      contact.setDefaultPatienceTime (pgen.nextDouble());
      return contact;
   }
}

// Met à jour les compteurs quand un contact sort
class MyContactMeasures implements ExitedContactListener {
   public void blocked (Router router, Contact contact, int bType) {}
   public void dequeued (Router router, WaitingQueue.DequeueEvent ev) {
      ++numAbandoned;
      if (ev.getContact().getTotalQueueTime() >= AWT) ++numAbandonedAfterAWT;
   }
   public void served (Router router, AgentGroup.EndServiceEvent ev) {
      ++numServed;
      Contact contact = ev.getContact();
      if (contact.getTotalQueueTime() < AWT) {
         ++numGoodSL;
         int period = pce.getPeriod (contact.getArrivalTime()) - 1;
         if (period >= 0 || period < P)
            ++numGoodSLKP[contact.getTypeId()][period];
      }
   }
}

void simulateOneDay() {
   Sim.init();    pce.init();
   double b = bgen.nextDouble();
   for (int k = 0; k < K; k++) arrivProc[k].init (b);
   for (int i = 0; i < I; i++) groups[i].init();
   for (int q = 0; q < K; q++) queues[q].init();
   numGoodSL = numServed = numAbandoned = numAbandonedAfterAWT = 0;
   vstat.init();
   for (int k = 0; k < K; k++) for (int p = 0; p < P; p++) numGoodSLKP[k][p] = 0;
   for (int k = 0; k < K; k++) arrivProc[k].start();
   pce.start();    Sim.start();     // La simulation s'exécute ici
   pce.stop();
   served.add (numServed);  abandoned.add (numAbandoned);
   goodSLKP.add (numGoodSLKP);
   serviceLevel.add (numGoodSL, numServed + numAbandonedAfterAWT);
   double Nb = vstat.getStatNumBusyAgents().sum();    // Intégrale de N_{b,0}(t)
   double N = vstat.getStatNumAgents().sum();         // Intégrale de N_0(t)
   double Ng = vstat.getStatNumGhostAgents().sum();   // Intégrale de N_{g,0}(t)
   occupancy.add (Nb, N + Ng);
}
```

```
   void simulate (int n) {
      served.init();          abandoned.init();    goodSLKP.init();
      serviceLevel.init();    occupancy.init();
      for (int r = 0; r < n; r++) simulateOneDay();
   }

   public void printStatistics() {
      System.out.println (served.reportAndCIStudent (LEVEL, 3));
      System.out.println (abandoned.reportAndCIStudent (LEVEL, 3));
      System.out.println (serviceLevel.reportAndCIDelta (LEVEL, 3));
      System.out.println (occupancy.reportAndCIDelta (LEVEL, 3));
      for (int k = 0; k < goodSLKP.rows(); k++)
         System.out.println (goodSLKP.rowReportAndCIStudent (k, LEVEL, 3));
   }

   public static void main (String[] args) {
      SimpleMSK s = new SimpleMSK();  s.simulate (NUMDAYS);  s.printStatistics();
   }
}
```

Le programme importe d'abord les classes nécessaires pour accéder à la bibliothèque ContactCenters et définit une classe `SimpleMSK` représentant le simulateur. Dans le but de simplifier le programme, aucun mode d'accès n'est spécifié pour les champs et méthodes. Ces champs et méthodes sont alors accessibles par toute classe du paquetage par défaut puisque nous n'avons placé notre exemple dans un paquetage explicite. Dans un programme réel, les champs seront privés (mot-clé `private`) tandis que les méthodes seront privées ou parfois publiques (mot-clé `public`). Pour plus de simplicité, les paramètres du modèle sont déclarés dans des champs statiques, mais dans un système réel, ces derniers devraient être lus depuis un fichier externe pour éviter de recompiler le programme avant chaque expérimentation. Dans cet exemple, tous les temps sont en minutes.

Pour chaque type de composante du centre de contacts simulé, un tableau est déclaré dans un champ de la classe `SimpleMSK`. Par exemple, les processus d'arrivée sont représentés par un tableau de K éléments nommé `arrivProc`. Les compteurs statistiques permettant de calculer des valeurs spécifiques à chaque réplication sont ensuite déclarés. Pour les variables aléatoires X, $X_{\mathrm{g}}(s)$, Y et $Y_{\mathrm{b}}(s)$, des scalaires suffisent, mais dans le cas de $X_{\mathrm{g},k,p}(s)$, une matrice de $K \times P$ est nécessaire. Pour chaque mesure de performance

à estimer, un collecteur statistique est déclaré pour recueillir les observations calculées par chaque réplication. La classe `RatioTally`, que nous décrirons plus en détails à la section 5.2.3, permet de calculer les rapports de moyennes, avec intervalles de confiance pour le rapport des espérances correspondantes. Pour l'estimation de $E[X_{g,k,p}(s)]$ par la moyenne, nous utilisons le collecteur matriciel que nous présenterons à la section 5.2.2.

Le constructeur de `SimpleMSK` crée les différentes composantes déclarées dans des champs et les connecte entre elles. L'événement de changement de période est créé de façon à définir trois périodes principales de 120 minutes, avec une période préliminaire de durée 0 et une période de fermeture de durée aléatoire.

Pour chacun des types de contacts, une usine abstraite et un processus d'arrivée sont créés. Le processus d'arrivée s'enregistre automatiquement comme observateur de changement de période afin de mettre le taux d'arrivée à jour au début des périodes. Le générateur du facteur d'occupation `bgen` est ensuite construit de façon à retourner des variables gamma par inversion. La construction des groupes d'agents nécessite l'événement de changement de période ainsi qu'un tableau contenant le nombre d'agents pour chaque période. Comme les processus d'arrivée, les groupes d'agents s'enregistrent comme observateurs de changement de période afin de mettre $N_i(t)$ à jour automatiquement. Il existe également un second constructeur prenant uniquement $N_i(t)$ en paramètre et permettant de changer le nombre d'agents manuellement pendant la simulation. La construction des files d'attente n'exige quant à elle aucun paramètre.

Les temps de service et de patience sont générés en utilisant respectivement `sgen` et `pgen`, qui constituent des générateurs adaptés aux périodes multiples (voir la section 3.5.2.1). De façon générale, la construction d'un tel objet exige de créer un tableau de générateurs de variables aléatoires et d'initialiser manuellement chacun d'eux. Ce tableau, ainsi que l'événement de changement de période, doivent ensuite être passés au constructeur de `MultiPeriodGen`. Toutefois, dans le cas exponentiel, une méthode statique de support nommée `createExponential` est disponible pour accélérer la construction. Cette méthode est utilisée dans le constructeur pour créer `sgen` et `pgen`.

La construction du routeur exige de choisir une politique de routage et ses paramètres, qui sont ceux donnés en exemple au tableau 3.1. La politique de routage est

déterminée par la sous-classe de `Router` mise en œuvre. Les processus d'arrivée, groupes d'agents et files d'attente sont ensuite liés au routeur et le système de comptage des événements est connecté en sortie afin de recevoir tous les contacts sortant du centre.

Le compteur `vstat` permet de calculer les intégrales nécessaires à l'estimation du taux d'occupation dans le premier groupe d'agents. `GroupVolumeStat` observe le comportement d'un seul groupe d'agents et définit des compteurs internes pour les différentes quantités. Si nous calculions le taux d'occupation pour les deux groupes d'agents, il serait nécessaire de définir un second compteur ou un tableau de deux éléments.

Le cœur du programme réside dans la méthode `simulateOneDay` chargée d'effectuer une réplication. Cette méthode est semblable pour tout simulateur de centre de contacts, mais elle peut être personnalisée par l'usager, par exemple pour planifier des événements additionnels comme l'activation d'un composeur. Après avoir remis l'horloge de simulation à zéro grâce à `Sim.init()`, le simulateur initialise les éléments du système afin d'éviter tout effet de bord produit par les réplications précédentes. La période courante est remise à 0 par `pce.init()` et le facteur d'occupation B pour la journée est généré et utilisé pour initialiser les processus d'arrivée. Après l'initialisation des groupes d'agents et des files d'attente, les compteurs d'événements sont remis à 0 et `vstat` est initialisé, de même que ses compteurs statistiques internes.

Avant de démarrer la simulation, les processus d'arrivée sont activés et planifient alors leurs premiers contacts. L'événement de changement de période est activé par la méthode `pce.start()` qui planifie le début de la première période principale au temps 0. Enfin, `Sim.start()` est utilisée pour démarrer le traitement de la liste d'événements.

Lorsqu'une arrivée se produit, le processus crée un nouveau contact en appelant la méthode `newInstance` de l'usine abstraite appropriée. Dans cet exemple, ces objets `MyContactFactory` ne diffèrent que par la valeur de leur champ `type` qui est utilisée, dans `newInstance`, comme identificateur de type pour le nouveau contact. Un temps de service et un temps de patience sont générés et le contact construit et initialisé est retourné pour être transmis au routeur.

Si le routeur reçoit un contact de type 0, il obtient une liste ordonnée contenant 0

seulement ; il tente alors de lui affecter un agent dans le groupe 0. Si $N_{\mathrm{f},0}(t) = 0$, le contact est inséré dans la file 0. Les contacts de type 2 sont traités de façon similaire. Dans le cas du type 1, puisque la liste ordonnée est $\{0,1\}$, le routeur vérifie si $N_{\mathrm{f},0}(t) > 0$ et si tel est le cas, affecte le contact à un agent de groupe 0. Si $N_{\mathrm{f},0}(t) = 0$, il tente d'affecter le contact de type 1 à un agent de groupe 1. Si $N_{\mathrm{f},1}(t) = 0$, le contact est inséré dans la file 1.

Lorsqu'un agent du groupe 0 devient libre, le routeur obtient la liste ordonnée $\{1,0\}$ et tente de retirer le contact de la file 1 ou 0 ayant attendu le plus longtemps. Dans le cas du groupe 1, la liste ordonnée étant $\{2,1\}$, le routeur consulte les files 2 et 1. Contrairement à celle des agents, la sélection des contacts ne tient pas compte de l'ordre des indices dans les listes en raison du type de routeur que nous avons choisi.

Lorsqu'un contact sort du système, il est annoncé à l'instance de `MyContact-Measures` que nous avons créée pendant la construction du simulateur. La méthode `blocked` ne contient aucun code, car la capacité du système est infinie par défaut si bien qu'aucun contact n'est bloqué. La méthode `dequeued` compte un abandon et, si le temps d'attente est supérieur ou égal à s, un abandon après le temps d'attente acceptable. La méthode `served` compte une fin de service ainsi qu'un contact acceptable pour le niveau de service si le temps d'attente est inférieur à s.

Le comptage d'un contact dans `numGoodSLKP` exige la connaissance de son type k ainsi que de la période principale pendant laquelle il est arrivé. L'identificateur de type est contenu dans l'objet représentant le contact tandis que la période doit être inférée à partir du temps d'arrivée. Les contacts ne mémorisent pas la période de leur arrivée, car un modèle pourrait définir plusieurs événements de changement de période. La méthode `getPeriod` est utilisée pour obtenir $p(t)$, la période correspondant à un temps de simulation quelconque t, mais l'indice retourné est entre 0 et $P+1$, pouvant représenter la période préliminaire ou de fermeture en plus des périodes principales. Puisque les arrivées se produisent pendant les périodes principales uniquement, $p(t) \in \{1,\ldots,P\}$. Un est alors soustrait de $p(t)$ afin que l'indice obtenu soit dans $\{0,\ldots,P-1\}$. Si $p(t) < 1$ ou $p(t) > P$, t correspond à un temps ne se trouvant pas dans une période principale et l'événement est ignoré.

Au temps t_P, le centre de contacts doit être fermé. Par conséquent, les processus d'arrivée Poisson changent automatiquement leurs taux d'arrivée à 0 lors du début de la période de fermeture, empêchant toute nouvelle arrivée. Les contacts terminent alors leur service et, puisque $N_i(t) > 0$ pour $t \geq t_P$, chaque contact en attente est servi ou abandonne. Si $N_i(t) = 0$ lorsque $t \geq t_P$, c'est-à-dire que le dernier élément du tableau `NUMAGENTS[i]` était 0 pour tous i, tous les contacts en file abandonneraient après un certain temps, à moins d'activer le vidage automatique des files par le routeur.

Lorsque la simulation est terminée, c'est-à-dire lorsque la liste des événements est vide, `Sim.start()` retourne et les observations obtenues sont placées dans les collecteurs statistiques appropriés. Il est recommandé d'appeler la méthode `pce.stop()` afin que tous les objets soient avertis de la fin de la période de fermeture qui n'est pas planifiée comme événement. L'appel à `pce.stop()` pourrait également se trouver dans la méthode `actions` du dernier événement de la simulation, mais dans cet exemple, cet événement est une fin de service et est défini dans ContactCenters et non dans notre programme.

Dans le cas du taux d'occupation des agents, pour obtenir les intégrales $\int_{t_0}^{T} N_0(t)\,dt$, $\int_{t_0}^{T} N_{\mathrm{g},0}(t)\,dt$ et $\int_{t_0}^{T} N_{\mathrm{b},0}(t)\,dt$, les compteurs statistiques correspondants sont retrouvés et la méthode `sum` est appelée sur chaque compteur. Si le taux d'occupation global devait être calculé, il faudrait itérer sur chacun des groupes et calculer les sommes

$$\begin{aligned} N(t) &= \sum_{i=0}^{I-1} \int_0^T N_i(t)\,dt, \\ N_{\mathrm{g}}(t) &= \sum_{i=0}^{I-1} \int_0^T N_{\mathrm{g},i}(t)\,dt, \\ \text{et } N_{\mathrm{b}}(t) &= \sum_{i=0}^{I-1} \int_0^T N_{\mathrm{b},i}(t)\,dt. \end{aligned}$$

Malheureusement, ces estimateurs tiennent compte des statistiques des agents pendant la période de fermeture, ce qui n'est pas toujours souhaitable. En effet, le taux d'occupation pendant la période de fermeture n'est pas très intéressant, car tous les agents sont occupés à terminer des services. Pour obtenir des estimateurs pour les heures d'ou-

verture seulement, il faudrait définir un observateur de changement de période obtenant les intégrales au temps t_P plutôt qu'au temps T. Nous ne l'avons pas fait afin de garder le programme simple, mais certains exemples dans [6] emploient cette technique.

Lorsque l'exécution des réplications est terminée, un rapport statistique semblable à la figure 4.1 est affiché à l'écran. Dans ce rapport, « `num. obs.` » représente le nombre d'observations, c'est-à-dire le nombre de réplications `NUMDAYS`, tandis que le niveau de confiance des intervalles est 95%.

Le théorème limite centrale est utilisé pour le calcul des intervalles de confiance sur les espérances. En particulier, le programme doit calculer un tel intervalle $[I_1, I_2]$ sur $\mu = E[X]$, avec un niveau de confiance $1-\alpha$ de façon à ce que

$$P(I_1 \leq \mu \leq I_2) \approx 1-\alpha.$$

Pour déterminer I_1 et I_2, des variables aléatoires, le programme considère que pour $r = 0, \ldots, n-1$, le nombre de contacts servis X_r pendant la r^{e} réplication suit la loi normale. Alors, sachant que $\mu = E[X]$ et $\sigma^2 = \text{Var}(X) \approx S_n^2$,

$$\frac{\sqrt{n}(\bar{X}_n - \mu)}{S_n}$$

suit la loi de probabilité Student-t avec $n-1$ degrés de liberté [16, 19]. L'hypothèse de normalité est raisonnable, car X_r représente la somme d'un grand nombre de valeurs qui, par le théorème limite centrale, converge vers la distribution normale.

Soit Z une variable aléatoire suivant la loi de Student-t avec $n-1$ degrés de liberté et soit $t_{n-1,1-\alpha/2}$ la valeur de la fonction inverse de Student-t avec $n-1$ degrés de liberté et évaluée à $1-\alpha/2$. Nous avons

$$\begin{aligned} 1-\alpha &= P(-t_{n-1,1-\alpha/2} \leq Z \leq t_{n-1,1-\alpha/2}) \\ &\approx P(\bar{X}_n - t_{n-1,1-\alpha/2} S_n/\sqrt{n} \leq \mu \leq \bar{X}_n + t_{n-1,1-\alpha/2} S_n/\sqrt{n}) \end{aligned}$$

Ainsi, le centre de l'intervalle de confiance sur μ est donné par $\bar{X}_n$ tandis que son rayon est $t_{n-1,1-\alpha/2} S_n/\sqrt{n}$.

Si n est grand, il n'est pas nécessaire que X_r suive la loi normale. Par le théorème limite centrale, si n est grand, $\bar{X}_n$ suit approximativement la loi normale si bien que $\sqrt{n}(\bar{X}_n - \mu)/\sigma$ peut être considéré comme normal standard. Puisque nous ne connaissons pas la variance σ^2, nous devons la remplacer par son estimateur S_n^2 donné par (1.13). De même, la loi Student-t est presque normale si n est grand. Ainsi, si n est grand ou X_r suit la loi normale, nous pouvons utiliser la formule précédente pour calculer un intervalle de confiance approximatif sur μ. La probabilité de couverture de l'intervalle, c'est-à-dire $P(\bar{X}_n - t_{n-1,1-\alpha/2}S_n/\sqrt{n} \leq \mu \leq \bar{X}_n + t_{n-1,1-\alpha/2}S_n/\sqrt{n})$, n'est pas exactement $1-\alpha$, car l'hypothèse de normalité n'est pas totalement vraie en pratique.

Pour obtenir un intervalle de confiance sur les autres espérances, le programme procède de façon identique. Pour les rapports d'espérances tels que le niveau de service, le théorème Delta, décrit à la section 5.2.3, est mis en œuvre pour calculer des intervalles de confiance.

Chaque intervalle est calculé de façon indépendante, sans tenir compte des autres intervalles calculés. Par conséquent, le niveau de confiance global de tous les intervalles n'est pas $1-\alpha$. L'inégalité de Bonferroni est un outil simple et utile pour estimer le niveau de confiance global. Soit $\boldsymbol{\mu} = (\mu_0, \ldots, \mu_{d-1})$ un vecteur d'espérances et soit $\mathscr{I}$ un intervalle de confiance sur $\boldsymbol{\mu}$ et soit $\mathscr{I}_j$ un intervalle de confiance sur μ_j de niveau $1-\alpha_j$. L'inégalité de Bonferroni est définie comme suit :

$$P(\boldsymbol{\mu} \in \mathscr{I}) = P(\mu_j \in \mathscr{I}_j \, \forall j) = 1 - P(\exists j | \mu_j \notin \mathscr{I}_j) \geq 1 - \sum_{j=0}^{d-1} P(\mu_j \notin \mathscr{I}_j) = 1 - \sum_{j=0}^{d-1} \alpha_j. \tag{4.1}$$

Par exemple, la probabilité que $E[X]$ et le niveau de service $g_1(s)$ soient contenus dans les intervalles de confiance calculés est au moins $1-2\alpha$.

4.2 Comparaison avec un logiciel commercial

Afin de comparer ContactCenters avec Arena Contact Center Edition de Rockwell, nous avons implanté quatre exemples issus du manuel d'utilisation de la version 8.0 du produit et couvrant à peu près toutes ses possibilités. Outre tester la performance,

```
REPORT on Tally stat. collector ==> Number of served contacts
         min         max      average    standard dev.   num. obs.
     2411.000    5429.000   4288.541      431.645         10000
  95.0% confidence interval for mean: (  4280.080,  4297.002 )

REPORT on Tally stat. collector ==> Number of contacts having abandoned
         min         max      average    standard dev.   num. obs.
       9.000    4436.000    899.808      556.094         10000
  95.0% confidence interval for mean: (   888.908,   910.709 )

REPORT on Tally stat. collector ==> Service level
    func. of averages    standard dev.   num. obs.
                0.505         0.131         10000
  95.0% confidence interval for function of means: (     0.503,     0.508 )

REPORT on Tally stat. collector ==> Occupancy ratio
    func. of averages    standard dev.   num. obs.
                0.880         0.057         10000
  95.0% confidence interval for function of means: (     0.878,     0.881 )

Report for Number of contacts meeting target service level, Type 0
              min         max      average     std. dev.    95.0% conf. int.
Period 0      3.000     367.000      173.469      96.612    [171.575, 175.363]
Period 1    274.000     815.000      597.814      82.872    [596.189, 599.438]
Period 2      0.000     180.000       16.319      27.170    [15.786, 16.851]
Number of observations: 10000

Report for Number of contacts meeting target service level, Type 1
              min         max      average     std. dev.    95.0% conf. int.
Period 0      8.000     505.000      266.483     137.257    [263.792, 269.174]
Period 1    231.000     750.000      510.206      87.609    [508.489, 511.924]
Period 2      0.000     324.000       36.150      54.841    [35.075, 37.225]
Number of observations: 10000

Report for Number of contacts meeting target service level, Type 2
              min         max      average     std. dev.    95.0% conf. int.
Period 0     10.000     574.000      265.899     158.950    [262.783, 269.014]
Period 1    265.000     887.000      612.188      99.717    [610.233, 614.143]
Period 2      0.000     286.000       25.346      42.301    [24.517, 26.176]
Number of observations: 10000
```

Figure 4.1 – Exemple de résultats donnés par `SimpleMSK`

l'implantation de ces exemples a parfois inspiré des améliorations dans ContactCenters afin d'accroître la flexibilité et la simplicité d'utilisation. De plus, la comparaison des résultats produits par ces exemples avec ceux donnés par Arena Contact Center Edition constitue une technique de validation complémentaire aux tests unitaires que nous avons pu effectuer.

Ces comparaisons ont pu être réalisées grâce à une version académique d'Arena Contact Center Edition qui ne peut simuler que des modèles de taille réduite. Au moment où nous avons effectué ces tests, NovaSim ne fournissait malheureusement pas de version académique ou de démonstration de ccProphet.

4.2.1 Modèles implantés

Dans cette section, nous présentons brièvement les exemples d'Arena Contact Center Edition que nous avons implantés. Nous ne montrons pas le code source de ces exemples ici, car il ressemble beaucoup à celui de la section précédente. Pour plus d'informations à propos de ces exemples, voir le manuel d'utilisation d'Arena Contact Center Edition [28] et le guide d'exemples de la bibliothèque ContactCenters [6]. Tous les exemples utilisent des processus d'arrivée de Poisson dont le taux d'arrivée est fixe tout au long de la journée, à l'exception de la première heure d'ouverture.

Telethon. Cet exemple implante un modèle de centre de contacts pour un téléthon d'une durée de cinq jours. Entre 6h et 10h, des donateurs téléphonent à un groupe de douze volontaires tandis que le nombre de lignes téléphoniques se trouve limité à 24. Lorsqu'un contact ne peut être servi immédiatement, il est mis en file d'attente. Un contact peut sortir de la file après un temps de patience exponentiel d'une durée moyenne de deux minutes (abandon), après une durée maximale de deux minutes (le client doit laisser un message), lors de la fermeture du centre de contacts (déconnexion) ou lorsqu'un volontaire devient libre (service).

Cet exemple a inspiré l'ajout des indicateurs de sortie de file d'attente afin de pouvoir distinguer les déconnexions des abandons et des messages laissés lors du comptage des contacts. La modélisation de la déconnexion à la fin des journées a inspiré l'im-

plantation du vidage automatique des files par le routeur. Le support du temps d'attente maximal de deux minutes a nécessité l'ajout de la méthode plus élaborée de génération des temps d'attente présentée à la section 3.5.2.2. Dans notre implantation de Telethon, le générateur de temps de patience par défaut est associé au type de sortie 1 tandis qu'un générateur retournant toujours 2 est lié au type 5. Ce nombre a été choisi arbitrairement ; n'importe quelle valeur autre que 1 ou 0 aurait pu être employée. Le temps d'attente maximal est ainsi le minimum entre le temps de patience et 2 et le type de sortie de file permet de distinguer les messages des abandons et déconnexions.

Bilingual. Ce système supporte des clients parlant Anglais ou Espagnol tandis que des agents anglais, espagnols et bilingues sont disponibles. Lors de l'arrivée d'un contact, le routeur choisit de façon aléatoire parmi les agents libres parlant sa langue ou bilingues. Si aucun agent adéquat n'est libre, le contact est mis en file d'attente. En cas d'abandon, avec probabilité 0.75, le client rappelle après vingt minutes.

Ce système, un peu plus complexe que le modèle précédent, n'a pas posé de problèmes majeurs pour l'implantation. Une file d'attente a été définie pour chaque type de contact et le routeur `SingleFIFOQueueRouter` que nous avons utilisé dans la section précédente a été étendu pour implanter la sélection aléatoire des agents ainsi que les rappels. Pour simuler les rappels, lorsqu'un abandon se produit, le routeur planifie, avec probabilité 0.75, un événement traitant le contact comme une nouvelle arrivée.

Bank. Ce modèle a déjà été décrit à la section 2.2. L'implantation, semblable à celle du modèle précédent, a nécessité la définition d'un routeur personnalisé. Ce dernier hérite encore une fois de `SingleFIFOQueueRouter` et ne modifie la politique de sélection des agents que pour l'obtention d'un solde. De cette façon, des priorités sont appliquées pour la vérification et les épargnes tandis qu'un agent est choisi de façon aléatoire pour les clients désirant un solde. Pour la sélection des contacts, un agent libre consulte d'abord la file contenant des contacts correspondant à sa spécialité. Lorsque cette file est vide, le routeur applique la politique de `SingleFIFOQueueRouter`.

Teamwork. Lorsqu'un contact entre dans ce système, il est acheminé vers un des deux réceptionnistes ou mis en file d'attente s'il ne peut être servi. Après le service, avec probabilité 0.2, une *conférence* avec un agent du département de comptabilité est demandée : si un agent de la comptabilité est disponible, le client reste en ligne avec le réceptionniste tout en dialoguant avec la comptabilité. À la fin du service, la communication avec le réceptionniste, et la comptabilité si une conférence a eu lieu, est terminée et le réceptionniste transfère le contact servi vers le gestionnaire avec probabilité 0.05 ou un agent du support technique avec probabilité 0.95. Le gestionnaire dispose d'une file d'attente tandis que les contacts destinés au support technique sont déconnectés s'ils ne peuvent être servis immédiatement. Après un dialogue avec le technicien, un contact peut entrer en conférence, avec probabilité 0.2, avec un développeur. Les réceptionnistes, le gestionnaire et les techniciens effectuent un travail après communication lorsque le dialogue est terminé. Pour le réceptionniste, ce travail débute immédiatement après le transfert du client et non pas à sa sortie du système.

Pour cet exemple, une sous-classe de `Contact` a dû être créée pour contenir de nouveaux attributs. Un routeur personnalisé a également été nécessaire pour simuler le chemin parcouru par les contacts. Cet exemple nous a incité à déclarer, dans la classe `Router`, des méthodes permettant d'effectuer des actions lors de chaque étape d'un contact (mise en file d'attente, début d'un service, etc.). Il a aussi été nécessaire de mettre à jour `AgentGroup` pour supporter les services en deux étapes afin de bien reproduire le modèle Arena Contact Center Edition. Malheureusement, il n'a jamais été possible de parvenir aux mêmes résultats que le logiciel commercial, car il semble que son module Contact Center souffre de certains bogues. Par exemple, il a été possible, en ajustant les paramètres du modèle, d'obtenir un niveau de service supérieur à 100% avec le logiciel commercial.

4.2.2 Méthode expérimentale et résultats

Pour évaluer la performance de notre bibliothèque et la comparer avec celle d'Arena Contact Center Edition, nous devons calculer le temps nécessaire à l'exécution des divers exemples. Nous souhaitons déterminer le temps nécessaire au processeur pour exécuter

les exemples et non la durée réelle d'exécution qui peut varier en fonction des autres tâches exécutées par la machine.

Sous Arena Contact Center Edition, il a été nécessaire de modifier les paramètres des modèles testés afin d'augmenter le nombre de réplications, de désactiver les animations et d'activer le mode *batch* pour une performance maximale. Le logiciel ne fournissant aucun moyen de mesurer le temps de processeur, il a été nécessaire de l'estimer par le temps système. Cette approximation est raisonnable si les temps d'exécution sont longs et si la machine n'est pas sollicitée par d'autres applications pendant les simulations. Afin de maximiser la précision des estimations, nous avons dû créer un programme externe pour appeler Arena Contact Center Edition, charger les modèles et les démarrer, prenant le temps système au début et à la fin des simulations.

Malheureusement, avant la version 1.5, Java ne disposait d'aucun mécanisme intégré pour le calcul du temps de processeur ; seul le temps système était disponible. SSJ fournit heureusement une classe `Chrono` utilisant une méthode native permettant d'obtenir ce temps de processeur au niveau du système d'exploitation. Un nombre élevé de réplications a été simulé afin que les temps d'exécution ContactCenters s'élèvent à quelques secondes.

Les temps d'exécution pour Arena Contact Center Edition, Java Runtime Environment (JRE) 1.4.2 et 1.5.0 de Sun sont rapportés au tableau 4.1. Pour chaque entrée, la valeur de gauche indique le temps nécessaire à l'exécution tandis que celle de droite donne le nombre de contacts traités par seconde, obtenu en divisant le nombre de contacts espérés $nE[A]$ par le temps d'exécution. Le nombre espéré d'arrivées $E[A]$ pendant une réplication dépend du modèle de simulation. La dernière colonne donne le quotient du temps d'exécution obtenu avec Arena Contact Center Edition et celui obtenu avec JRE 1.5. Les temps ont été obtenus avec un processeur AMD Athlon Thunderbird 1GHz, sous Arena 8.0 de Rockwell et JRE de Sun, tournant sous Windows XP.

Nous observons que les temps d'exécution des exemples sous Arena Contact Center Edition sont environ 30 fois plus longs que ceux obtenus avec ContactCenters et que Java 1.5 n'apporte pas un gain de performance significatif par rapport à Java 1.4.2. C'est pourquoi dans le tableau, nous n'avons pas rapporté les facteurs pour les deux versions

Tableau 4.1 – Temps d'exécution pour 1 000 réplications

Exemple	$E[A]$	Arena		JRE1.4		JRE1.5		Arena/JRE
Telethon	1 000	4min23s	3 802/s	7s	142 857/s	7s	142 857/s	38
Bilingual	5 000	23min39s	3 523/s	37s	135 135/s	37s	135 135/s	38
Bank	3 600	23min57s	2 505/s	42s	85 714/s	40s	90 000/s	36
Teamwork	7 000	22min56s	5 087/s	1min32s	76 086/s	1min30s	77 777/s	15

de JRE. En général, plus le modèle est complexe, plus le nombre de contacts traités par seconde est petit. Toutefois, nous observons un comportement étrange avec Teamwork qui définit une logique de simulation plus complexe que les autres exemples. À première vue, il devrait être plus long à exécuter que tous les autres exemples, mais il est plus rapide que Telethon sous Arena Contact Center Edition. Ce phénomène s'est aussi produit avec ContactCenters, pour d'anciennes versions de Teamwork [7]. Ce comportement s'explique en tenant compte du chemin parcouru par les contacts dans le système. Dans Teamwork, beaucoup de contacts abandonnent avant de recevoir du service ou sont déconnectés au niveau du support technique, réduisant la taille moyenne des files d'attente. Sous Arena Contact Center Edition, le nombre de processus en mémoire, de même que les allocations de ressources pour gérer les services, est réduit. Sous ContactCenters, le nombre d'événements planifiés pour gérer les abandons ainsi que les services est réduit. Dans les autres modèles, le taux d'abandon est beaucoup moins élevé, car plus de deux agents peuvent traiter les contacts.

Ainsi, le temps d'exécution dépend du nombre de contacts à traiter ainsi que de leur chemin à travers le système. Le temps de traitement d'un seul contact dépend des variables aléatoires générées ainsi que du routage qui dépend de la taille du système.

4.3 Exemple d'utilisation du simulateur générique

Lorsqu'il emploie le simulateur générique, l'utilisateur est en mesure, par le biais de fichiers de configuration XML, de spécifier l'ensemble des paramètres du modèle. Le programme peut ensuite lire ces fichiers et être utilisé depuis la ligne de commandes ou

une autre application Java. Puisque les données en sont séparées, il n'est pas nécessaire de recompiler le programme pour chacune des expériences effectuées.

4.3.1 Fichiers de configuration

Dans cette section, nous allons commenter l'exemple de fichier de configuration du listing 4.2 sans détailler tous les paramètres possibles. L'ensemble des options supportées par le modèle est spécifié dans le manuel de référence [6].

Dans cet exemple, le centre d'appels ouvre à 9h et comporte cinq périodes d'une heure. Deux types d'appels entrants et deux types sortants sont supportés. Chaque type d'appel entrant est simulé par un processus d'arrivée indépendant et les appelants abandonnent après un certain temps de patience. Un composeur distinct est utilisé pour produire les appels sortants de chacun des deux types. En cas de *mismatch*, le client abandonne immédiatement dans la plupart des cas. Parfois, il attend quelques secondes avant de raccrocher.

Le premier des quatre groupes d'agents est spécialisé pour les appels entrants tandis que le second l'est pour les appels sortants. Les deux autres groupes sont mixtes, mais ils ne peuvent servir que la moitié des types d'appels. Lorsqu'un appel doit être acheminé vers un agent, le routeur préfère utiliser les spécialistes avant les généralistes. Les agents mixtes préfèrent quant à eux servir les appels de type sortant lorsque certains attendent en file.

Le fichier comprend un élément racine nommé `mskccparams` comportant des attributs fixant les aspects globaux du centre d'appels et des sous-éléments pour chaque type d'appel, chaque groupe d'agents, le routeur et les paramètres pour estimer le niveau de service.

Listing 4.2 – Exemple de fichiers de paramètres pour le simulateur générique

```
<?xml version="1.0" encoding="iso-8859-1"?>
<?import umontreal.iro.lecuyer.probdist.*?>
<?import umontreal.iro.lecuyer.randvar.*?>
<mskccparams defaultUnit="HOUR"                    queueCapacity="infinity"
             periodDuration="1.0h"                 numPeriods="5"
```

```
          startTime="9h">
<!— Type d'appel 0 -->
<inboundType name="First Inbound Type">
   <sourceToggleTimes>9h, 14h</sourceToggleTimes>
   <probAbandon>0.08, 0.01, 0.1, 0.09, 0.07</probAbandon>
   <patienceTime distributionClass="ExponentialDist" unit="HOUR">
         12.0</patienceTime>
   <serviceTime distributionClass="ExponentialDist"
               group="0"              unit="HOUR">
         60.0</serviceTime>
   <serviceTime distributionClass="ExponentialDist"
               group="2"              unit="HOUR">
         35.0</serviceTime>
   <arrivalProcess type="PIECEWISECONSTANTPOISSON" normalize="true">
      <arrivals>60.0, 50.0, 40.0, 45.0, 49.0</arrivals>
   </arrivalProcess>
</inboundType>
<!-- Type d'appel 1 —>
<inboundType name="Second Inbound Type">
   <sourceToggleTimes>9h, 10.2h,
                     10.8h, 14h</sourceToggleTimes>
   <probAbandon>0.06, 0.12, 0.23, 0.18, 0.15</probAbandon>
   <patienceTime distributionClass="ExponentialDist" unit="HOUR">
         6.0</patienceTime>
   <serviceTime distributionClass="ExponentialDist"
               group="0"              unit="HOUR">
         50.0</serviceTime>
   <serviceTime distributionClass="ExponentialDist"
               group="3"              unit="HOUR">
         29.0</serviceTime>
   <arrivalProcess type="POISSONGAMMA" normalize="true">
      <poissonGammaParams>
          <row repeat="5">25.2,  0.975</row>
      </poissonGammaParams>
   </arrivalProcess>
</inboundType>
<!— Type d'appel 2 -->
<outboundType name="First Outbound Type">
   <probAbandon>
      <row repeat="5">0.95</row>
   </probAbandon>
   <patienceTime distributionClass="ExponentialDist" unit="SECOND">
         0.33333</patienceTime>
   <serviceTime distributionClass="ExponentialDist"
```

```
                group="1"                unit="HOUR">
          8.178</serviceTime>
      <serviceTime distributionClass="ExponentialDist"
                group="2"                unit="HOUR">
          4.5</serviceTime>
      <sourceToggleTimes>
         <row>12.0h</row>
         <row>14.0h</row>
      </sourceToggleTimes>
      <dialer dialerPolicy="DIALXFREE"  dropMismatches="false"
              kappa="2.0"         c="0">
         <minFreeAgentsTest>
            <row repeat="5">4</row>
         </minFreeAgentsTest>
         <probReach>
            <row repeat="2">0.27</row>
            <row>0.28</row>
            <row repeat="2">0.29</row>
          </probReach>
      </dialer>
   </outboundType>
   <!-- Type d'appel 3 -->
   <outboundType name="Second Outbound Type">
      <probAbandon>
         <row repeat="5">0.98</row>
      </probAbandon>
      <patienceTime distributionClass="ExponentialDist" unit="SECOND">
          0.2</patienceTime>
      <serviceTime distributionClass="ExponentialDist"
                group="1"                unit="HOUR">
          9.2</serviceTime>
      <serviceTime distributionClass="ExponentialDist"
                group="3"                unit="HOUR">
          8.2</serviceTime>
      <sourceToggleTimes>
         <row>11.5h</row>
         <row>13.5h</row>
      </sourceToggleTimes>
      <dialer dialerPolicy="DIALXFREE" dropMismatches="false"
              kappa="2.5"         c="1">
         <minFreeAgentsTest>
            <row repeat="5">3</row>
         </minFreeAgentsTest>
         <probReach>
```

```
            <row>0.3</row>
            <row>0.33</row>
            <row>0.37</row>
            <row>0.4</row>
            <row>0.38</row>
         </probReach>
      </dialer>
   </outboundType>

   <!— Groupe d'agents 0 -->
   <agentGroup name="Inbound-only agents">
      <staffing>1, 5, 6, 9, 3</staffing>
   </agentGroup>
   <!-- Groupe d'agents 1 —>
   <agentGroup name="Outbound-only agents">
      <staffing>2, 7, 6, 3, 9</staffing>
   </agentGroup>
   <!— Groupe d'agents 2 -->
   <agentGroup name="Blend agents 1">
      <staffing>3, 5, 5, 4, 4</staffing>
   </agentGroup>
   <!-- Groupe d'agents 3 —>
   <agentGroup name="Blend agents 2">
      <staffing>2, 4, 6, 4, 5</staffing>
   </agentGroup>

    <router routerPolicy="AGENTSPREF">
      <ranks>
         <row>       1,        1, infinity, infinity</row>
         <row>infinity, infinity,        1,        1</row>
         <row>       3, infinity,        2, infinity</row>
         <row>infinity,        3, infinity,        2</row>
      </ranks>
   </router>
   <serviceLevel>
      <awt value="20s"/>
      <awt value="25s" k="0"/>
      <awt value="30s" k="1"/>
      <target value="0.8"/>
   </serviceLevel>
</mskccparams>
```

La première ligne du fichier spécifie une en-tête indiquant la version XML à utiliser

et l'encodage des caractères. Les deux lignes suivantes constituent des directives de traitement qui sont définies dans notre système de lecture des paramètres et qui permettent d'importer des paquetages Java pour rendre les noms de classes SSJ plus compacts.

Le nombre de périodes est fixé à cinq par l'attribut `numPeriods` tandis que leur durée est ajustée à une heure par `periodDuration`. Puisque `defaultUnit` est fixé à `HOUR`, une unité de temps de simulation correspond à une heure dans cet exemple.

Chaque type d'appel entrant est donné par un élément `inboundType` imbriqué dans `mskccparams`. Par exemple, pour le premier type d'appel montré au début du fichier, le processus d'arrivée démarre à 9h et s'arrête à 14h en raison de l'élément `sourceToggleTimes`. L'élément `probAbandon` fixe la probabilité d'abandon immédiat à 0.08 pendant la première période, 0.01 pendant la seconde et ainsi de suite.

La probabilité d'abandon ainsi que les autres paramètres sont donnés pour les périodes principales seulement. Pendant la période préliminaire, la probabilité d'abandon est celle de la première période principale tandis que durant la période de fermeture, elle correspond à celle de la dernière période principale. Cette règle s'applique pour tous les autres paramètres à l'exception des vecteurs d'affectation des agents. Durant la période préliminaire, le nombre d'agents est 0 tandis qu'il correspond au nombre pendant la dernière période principale durant la période de fermeture. Les paramètres sont donnés de cette façon afin de faciliter la création de fichiers de paramètres pour des systèmes stationnaires. Si les paramètres étaient donnés pour les périodes préliminaires et de fermeture, il faudrait au minimum trois périodes, même si le système était toujours simulé de façon stationnaire.

L'élément `patienceTime` est utilisé pour fixer un temps d'attente maximal avant abandon. L'attribut `distributionClass` accepte le nom d'une classe correspondant à une loi de probabilité SSJ tandis que le contenu doit correspondre aux paramètres donnés au constructeur de la classe choisie. Pour obtenir des temps de patience exponentiels, il suffit ainsi de consulter la documentation de SSJ [18] afin de connaître la paramétrisation de cette loi. Un objet `ExponentialDist` du paquetage `probdist` est construit avec un paramètre d'échelle λ qui correspond au taux de l'exponentielle. Il n'est pas nécessaire de spécifier le nom de paquetage de la classe `ExponentialDist` en rai-

son de la directive `import` placée au début du fichier. Puisque `unit` est `HOUR`, les valeurs aléatoires générées sont en heures, l'unité de temps par défaut si bien qu'aucune conversion d'unité n'est nécessaire. Avec ce taux fixé à 12, le temps de patience moyen correspond à $\frac{1}{12}\text{h} = 60\text{min}/12 = 5\text{min}$. L'élément `serviceTime` pour fixer la durée de service fonctionne de façon similaire, à l'exception de l'attribut optionnel `group` indiquant le groupe d'agents pour lequel le paramètre s'applique. Si `group` n'est pas spécifié, le temps de service donné s'applique à tout agent.

L'élément `arrivalProcess` fixe le processus d'arrivée de Poisson ainsi que les taux pour chaque période principale. Puisque `normalize` est `true`, chaque taux donné dans l'élément `arrivals` correspond au nombre espéré d'arrivées durant une période.

Le second type d'appel, configuré par le second élément `inboundType` du fichier, est associé à un processus d'arrivée qui démarre à 9h, s'arrête à 10h12 ($0.2\text{h} = 12\text{min}$), redémarre à 10h48 et s'arrête à 14h. Les autres paramètres sont fixés de façon similaire au type d'appel précédent, à l'exception du processus d'arrivée. Pour ce type, un processus de Poisson avec taux d'arrivée stochastiques est mis en œuvre : au début de chaque réplication, un taux d'arrivée est généré suivant la loi gamma avec paramètre de forme 25.2 et paramètre d'échelle 0.975 (moyenne $25.2/0.975 = 25.846$) pour chaque période.

Les deux éléments `outboundType` suivant les éléments `inboundType` définissent les paramètres pour les deux types d'appels sortants. La syntaxe des éléments est la même que pour les types entrants, à l'exception du composeur qui remplace le processus d'arrivée. L'élément `dialer` fixe la politique de composition et la probabilité de connexion pour chaque période principale. Dans cet exemple, la politique de composition correspond à celle décrite dans la section 3.3. Pour le premier type sortant, lorsqu'au moins quatre agents (élément `minFreeAgentsTest`) sont libres, le composeur tente d'effectuer $2N_{\text{f}}^{\text{d}}(t)$ appels. Les paramètres pour le second type d'appel sortant sont très similaires, car la même politique de composition est utilisée avec des options différentes. Pour que des appels soient composés, la valeur minimale par défaut de $N_{\text{f}}^{\text{d}}(t)$ est fixée à 1. L'élément `minFreeAgentsTarget` permet de modifier cette valeur si nécessaire.

L'élément `router`, au bas du fichier, spécifie la politique de routage ainsi que ses paramètres. La matrice de rangs donnée par l'élément `ranks` correspond à l'exemple

donné au tableau 3.2 et la politique de routage correspond à `AgentsPrefRouter` que nous avons décrite dans la section 3.4.3.2.

Finalement, l'élément `serviceLevel` à la toute fin du fichier utilise l'élément `awt` pour fixer le temps d'attente acceptable à 20s pour le niveau de service global et à 30s pour le type d'appel 1. Le niveau de service à atteindre, qui n'est pas utilisé par le simulateur, est fixé à 0.8 par l'élément `target`.

Les paramètres de l'expérience sont déterminés par un second fichier XML dont le format dépend de la méthode d'expérimentation choisie. Le listing 4.3 présente un exemple de fichier de paramètres pour une simulation sur horizon fini utilisant des réplications indépendantes. L'attribut `minReplications` fixe le nombre de réplications à 300 et la durée de simulation est déterministe puisque `targetError` contient une valeur négative. Le niveau de confiance de tous les intervalles calculés est fixé à 95% par l'attribut `level`. L'attribut `normalizeToDefaultUnit`, fixé à `false`, indique de ne pas normaliser les taux par la durée des périodes. Par exemple, grâce à ce réglage, le taux d'appels servis rapporté estime le nombre espéré d'appels servis par période plutôt que par unité de temps.

Listing 4.3 – Exemple de fichier de paramètres pour horizon fini

```
<repsimparams targetError="-1"   level="0.95"   minReplications="300"
              normalizeToDefaultUnit="false"/>
```

Le listing 4.4 présente quant à lui un exemple de fichier pour la simulation sur horizon infini utilisant les moyennes par lots (voir section 5.3.2). Puisque `initNonEmpty` est fixé à `true` et `targetInitOccupancy` est réglé à 0.9, le système est d'abord initialisé de façon à ce que 90% des agents soient occupés à servir des appels. Ensuite, puisque `warmupBatches` est fixé à 5 et `batchSize` est fixé à 20h, cinq intervalles de vingt heures sont simulés sans aucune collecte d'observations. Le nombre de lots à simuler est fixé à trente par l'attribut `minBatches` et la durée de simulation est encore une fois déterministe. Puisque `minBatches` exclut le réchauffement, le temps total de

simulation est fixé à `batchSize*(warmupBatches + minBatches)`.

Les éléments `printedStat` indiquent quelles mesures de performance doivent être affichées dans le rapport statistique imprimé par défaut. Les niveaux de service global et détaillé pour chaque type d'appel et période sont affichés en raison du premier élément `printedStat`. Le second élément indique d'afficher le taux d'occupation global pour tous les agents seulement en raison de l'attribut `detailed` fixé à `false`. Des éléments `printedStat` pourraient également être inclus dans un fichier de configuration pour une simulation sur horizon fini.

Listing 4.4 – Exemple de fichier de paramètres pour horizon infini

```
<batchsimparams targetError="-1"        level="0.95"
                initNonEmpty="true"     targetInitOccupancy="0.9"
                batchSize="20h"         minBatches="30"
                warmupBatches="5">
   <printedStat measure="SERVICELEVEL"/>
   <printedStat measure="OCCUPANCY" detailed="false"/>
</batchsimparams>
```

4.3.2 Exécution du simulateur

Le simulateur `MSKCallCenterSim` peut être exécuté depuis la ligne de commande ou depuis un programme Java. La ligne de commande se présente comme suit :

`java umontreal.iro.lecuyer.contactcenters.app.MSKCallCenterSim` ⟨paramètres du centre d'appels⟩ ⟨paramètres de simulation⟩

Pour exécuter le simulateur, la commande doit être inscrite sur une seule ligne ; les sauts de ligne ont été ajoutés pour le formatage seulement. Le programme prend deux noms de fichiers XML en argument. Le premier fichier doit contenir un élément `mskccparams` tandis que le second doit contenir l'élément `batchsimparams` ou `repsimparams`. Les deux fichiers sont lus, un simulateur est construit et une expérimentation est déclenchée. Les résultats sont ensuite affichés à l'écran. L'exécution depuis

la ligne de commande constitue un moyen simple de tester un fichier de paramètres ou obtenir des statistiques sur un centre d'appels sans écrire de code.

Il est également possible d'accéder au simulateur depuis un autre programme Java pour traiter les résultats obtenus en vue d'une optimisation, d'une analyse statistique, d'un formatage personnalisé, etc. Le listing 4.5 présente un programme Java appelant le simulateur pour obtenir le niveau de service pour tous les appels et toutes les périodes. Le programme n'est composé que d'une méthode `main` effectuant la lecture des paramètres, la construction du simulateur et la simulation proprement dite.

Listing 4.5 – Exemple d'utilisation de l'interface de haut niveau

```
import umontreal.iro.lecuyer.xmlconfig.ParamReader;
import umontreal.iro.lecuyer.contactcenters.app.ContactCenterSim;
import umontreal.iro.lecuyer.contactcenters.app.MSKCallCenterSim;
import umontreal.iro.lecuyer.contactcenters.app.MSKCallCenterParams;
import umontreal.iro.lecuyer.contactcenters.app.SimParams;
import umontreal.iro.lecuyer.contactcenters.app.BatchSimParams;
import umontreal.iro.lecuyer.contactcenters.app.RepSimParams;
import umontreal.iro.lecuyer.contactcenters.app.PerformanceMeasureType;
import javax.xml.parsers.*;
import org.xml.sax.*;
import java.io.*;
import cern.colt.matrix.DoubleMatrix2D;
import cern.colt.list.DoubleArrayList;
import umontreal.iro.lecuyer.stat.matrix.MatrixOfStatProbes;
import umontreal.iro.lecuyer.stat.matrix.MatrixOfTallies;

public class CallSim {
   public static void main (String[] args) throws
      ParserConfigurationException, SAXException, IOException {
      if (args.length != 2) {
         System.err.println ("Usage: java CallSim <call center params>"
                             + " <simulation params>");
         System.exit (1);
      }
      String ccPsFn = args[0];
      String simPsFn = args[1];
      ParamReader reader = new ParamReader();
      // Définition de l'élément racine pour les paramètres du modèle
      reader.elements.put ("mskccparams", MSKCallCenterParams.class);
```

```
        // Définition des éléments racine pour les paramètres de l'expérience
        reader.elements.put ("batchsimparams", BatchSimParams.class);
        reader.elements.put ("repsimparams", RepSimParams.class);
        // Lecture des paramètres du modèle
        MSKCallCenterParams ccPs = (MSKCallCenterParams)reader.read (ccPsFn);
        // Lecture des paramètres de l'expérience.
        // Le programme accepte des paramètres pour horizon infini ou fini.
        SimParams simPs = (SimParams)reader.read (simPsFn);
        // Vérification des paramètres
        ccPs.check();
        simPs.check();
        // Construction du simulateur
        ContactCenterSim sim = new MSKCallCenterSim (ccPs, simPs);

        // Simulation
        sim.eval();
        DoubleMatrix2D sl = sim.getPerformanceMeasure
            (PerformanceMeasureType.SERVICELEVEL);
        System.out.println ("Service level = " +
                            sl.get (sl.rows() - 1, sl.columns() - 1));

        if (simPs.getKeepObs()) {
            // Afficher toutes les observations pour le niveau de service,
            // si elles sont disponibles.
            MatrixOfStatProbes m = sim
                .getMatrixOfStatProbes (PerformanceMeasureType.SERVICELEVELREP);
            // Le transtypage ne fonctionne que si la mesure de performance
            // retournée correspond à une moyenne. Cela ne fonctionne pas
            // pour un rapport.
            MatrixOfTallies mta = (MatrixOfTallies)m;
            DoubleArrayList obs = mta.getArray (mta.rows() - 1,
                                                mta.columns() - 1);
            System.out.println ("\nObservations for the service level");
            for (int i = 0; i < obs.size(); i++)
                System.out.println (obs.get (i));
        }
    }
}
```

Après avoir vérifié que leur nombre est adéquat, le programme copie les arguments passés par l'utilisateur dans des variables. La construction des objets de paramètres est ensuite effectuée à l'aide d'un objet `ParamReader`. Le système de lecture des para-

mètres étant générique, l'utilisateur doit lui-même établir la correspondance entre le nom des éléments racine et les classes d'objets de paramètres.

Pour la lecture des paramètres du modèle, l'élément racine `mskccparams` est associé à la classe `MSKCallCenterParams`. Pour les paramètres de l'expérience, l'élément `batchsimparams` est associée à la classe `BatchSimParams` tandis que l'élément `repsimparams` est lié à `RepSimParams`. Ces associations sont nécessaires afin de guider le système de lecture de paramètres.

La méthode `read` est ensuite utilisée pour convertir le fichier dont le nom est contenu dans `ccPsFn` vers l'objet de paramètres `ccPs` de classe `MSKCallCenterParams`. De façon similaire aux paramètres du modèle, `simPs` est initialisé comme une instance de `SimParams`, classe de base qu'étendent `RepSimParams` et `BatchSimParams`. De cette façon, le programme peut accepter des paramètres pour des réplications indépendantes ou pour une simulation stationnaire avec moyennes par lots.

La méthode `check` est ensuite utilisée sur chaque objet de paramètres afin de vérifier sa cohérence. Le simulateur `MSKCallCenterSim` peut finalement être construit en utilisant les paramètres chargés.

Dans la suite du code, `sim` peut dès lors être considéré comme une référence de type `ContactCenterSim` ou même `ContactCenterEval` ; l'utilisation du simulateur se fait indépendamment de l'implantation construite. La méthode `eval` est utilisée pour déclencher la simulation tandis que `getPerformanceMeasure` retourne une matrice de nombres décimaux.

La documentation du simulateur indique que la matrice de niveaux de service contient une ligne par type de contact ainsi qu'une ligne pour le niveau global. Chaque colonne correspond à une période tandis que la dernière correspond à toute la simulation. Ainsi, l'élément de la dernière rangée et de la dernière colonne contient le niveau de service recherché.

En second lieu, le programme affiche toutes les observations nécessaires au calcul de l'estimateur du niveau de service à court terme. Le niveau de service précédent estime un rapport de deux espérances par un rapport de deux moyennes, si bien qu'une seule copie du niveau de service est disponible. Le niveau à court terme, quant à lui, estime

l'espérance d'un rapport si bien que plusieurs observations sont disponibles. Obtenir les observations peut être utile pour par exemple déterminer un quantile ou afficher un histogramme.

Tout d'abord, le programme doit vérifier si les observations sont disponibles. Par défaut, le simulateur ne les conserve pas, car il ne calcule que des statistiques de base. Pour conserver les observations, dans les paramètres de simulation (éléments racine `repsimparams` ou `batchsimparams`), l'utilisateur doit fixer l'attribut `keepObs` à `true`. La méthode `getKeepObs` est utilisée dans le programme pour tester la valeur de cet attribut. Si les observations sont disponibles, le programme obtient une matrice de collecteurs statistiques généraux pour le groupe de mesures de performance qui nous intéresse. Nous abordons plus en détails les collecteurs matriciels à la section 5.2.2. La matrice générale doit ensuite être convertie en matrice de collecteurs d'observations individuelles puisque seuls ces collecteurs permettent d'extraire les observations. Si le type de mesure de performance demandé était `SERVICELEVEL` plutôt que `SERVICELEVELREP`, la matrice retournée contiendrait des collecteurs adaptés à des fonctions de plusieurs moyennes (voir section 5.2.3) qui ne permettent pas l'extraction des observations. La méthode `getArray` est ensuite appelée pour obtenir un tableau contenant les observations pour l'élément inférieur droit de la matrice de niveaux de service qui contient le niveau global recherché. Finalement, chaque observation du tableau est affichée sur une seule ligne.

4.3.3 Performance du simulateur générique

Si nous exécutons 1 000 réplications en utilisant le simulateur de la section 4.1, nous obtenons, sur un AMD Thunderbird 1GHz, un temps de processeur de 47s. Il est possible de simuler le même modèle à l'aide du programme générique en créant le fichier XML adéquat, dont le code peut être trouvé dans le guide fourni avec la bibliothèque ContactCenters [6]. Sur la même machine, si nous simulons 1 000 réplications, nous obtenons un temps de processeur de 1min22s. Le temps d'exécution est plus long en raison du plus grand nombre de statistiques à calculer et des autres éléments supportés par le modèle quoiqu'inutilisés dans l'exemple. Cette comparaison montre qu'un simulateur

adapté à un modèle particulier est plus performant qu'un système générique.

CHAPITRE 5

EXTENSIONS À SSJ

Lors de la construction du premier simulateur utilisant la bibliothèque ContactCenters, il a été nécessaire d'écrire plusieurs méthodes pour gérer différents aspects non pris en charge par SSJ. Plutôt que d'écrire des classes spécifiques à la bibliothèque, nous avons tenté de les généraliser afin qu'elles soient utiles au plus grand nombre de développeurs possible. Ces extensions à SSJ incluent une classe pour regrouper des générateurs de nombres aléatoires dans une liste, des collecteurs statistiques de plus haut niveau et le support de la méthode des moyennes par lots.

5.1 Extensions pour les générateurs de nombres aléatoires

Afin de faciliter l'utilisation des variables aléatoires communes dans les systèmes complexes, nous avons défini une classe permettant de gérer une liste de générateurs de nombres aléatoires uniformes. Nous proposons également une usine abstraite permettant de faciliter les expérimentations avec plusieurs générateurs de variables aléatoires uniformes.

5.1.1 Liste de générateurs

Pour obtenir des nombres aléatoires sous SSJ, l'utilisateur construit un générateur de variables aléatoires uniformes représenté par l'interface `RandomStream`. Les valeurs obtenues se trouvent dans l'intervalle $[0, 1)$ et sont converties en variables non uniformes suivant n'importe quelle loi de probabilité supportée. Un nombre possiblement élevé de générateurs peut être construit et ces générateurs sont souvent disséminés un peu partout dans les objets du simulateur.

Lors de la comparaison de systèmes par simulation, il est possible de réduire la variance en utilisant les mêmes variables aléatoires lors de la simulation de chaque système. Pour faciliter l'implantation de ces variables aléatoires communes, les générateurs de va-

riables uniformes de SSJ comportent un mécanisme de réinitialisations des germes fonctionnant de la manière suivante. La période du générateur choisi, c'est-à-dire la suite de tous les nombres qu'il peut produire, est divisée en intervalles successifs appelés *streams* [21, 23] et suffisamment longs pour ne pas se chevaucher. Chaque fois qu'un générateur, c'est-à-dire un objet de toute classe implantant `RandomStream`, est construit, il utilise un intervalle, et donc un germe, distinct, si bien que chaque générateur peut être considéré comme indépendant des autres. Ces germes multiples permettent d'associer un générateur indépendant de variables aléatoires à chaque portion du système, comme les temps inter-arrivées, les durées de service, etc. Cela permet d'éviter, par exemple, que la variable uniforme pour le temps de service soit utilisée, dans un autre système, pour un temps inter-arrivée.

Les intervalles, utilisés pour maximiser la synchronisation des variables aléatoires, sont eux-mêmes divisés en sous-intervalles (*substreams*) encore une fois successifs et suffisamment longs pour éviter les chevauchements. Afin d'utiliser les variables aléatoires communes, pour chaque système comparé, les générateurs sont initialisés au début de leurs sous-intervalles courants et la simulation est exécutée. De cette façon, chaque système est simulé avec les mêmes variables aléatoires. Avant de passer à la réplication suivante, un saut aux sous-intervalles suivants est effectué de façon à ce que la nouvelle réplication utilise de nouvelles valeurs.

Ces réinitialisations et ces sauts doivent être effectués pour tous les générateurs et il est parfois difficile de les retrouver à travers la structure du simulateur et facile d'en oublier lorsque le système est complexe. Pour résoudre ce problème, la classe `RandomStreamManager` que nous avons définie permet de constituer une liste à laquelle tout générateur de nombres aléatoires construit peut être ajouté. Le gestionnaire de générateurs fournit des méthodes semblables à celles de `RandomStream` pour effectuer la même opération sur tous les générateurs de la liste. Ainsi, en un seul appel de méthode, il est dès lors possible de réinitialiser les germes de tous les générateurs.

Le listing 5.1 présente une version adaptée de l'exemple *Inventory* de SSJ [18] utilisant les listes de générateurs pour gérer les variables aléatoires communes. Le modèle simulé considère un inventaire pour un produit dont le nombre d'unités demandées quo-

tidiennement correspond à des variables aléatoires indépendantes suivant la loi de Poisson avec taux λ. Si X_j est le niveau de l'inventaire au début de la journée j et D_j est la demande pour ce même jour, $\min(X_j, D_j)$ ventes ont lieu et $\max(0, D_j - X_j)$ ventes sont perdues. À la fin de la journée, le niveau d'inventaire est $Y_j = \max(X_j - D_j, 0)$. Un profit c est associé à tout article vendu et une perte h est imposée pour tout produit non vendu. Une politique (s, S) est mise en place pour contrôler le niveau d'inventaire : si $Y_j < s$, commander $S - Y_j$ articles et ne rien faire dans le cas contraire. Avec probabilité p, une commande faite à la fin de la journée arrive au début du jour suivant. Avec probabilité $1 - p$, les articles commandés n'arrivent jamais. Lors d'une commande réussie, un coût fixe K est imposé en plus d'un coût marginal k pour chaque article commandé. Au début du premier jour, l'inventaire est fixé à $X_0 = S$.

Le programme du listing 5.1 étend la classe `Inventory` qui implante le modèle précédent et qui est présentée dans le guide d'exemples de SSJ [18]. Nous ne présentons pas le code de `Inventory`, car il n'est pas nécessaire pour la compréhension de notre exemple. Notre programme d'exemple crée d'abord une instance de `InventoryCRN` et déclenche trois expériences : une avec des variables aléatoires indépendantes, une avec des variables aléatoires communes sans liste de générateurs et une troisième avec les listes. Chaque expérience vise à évaluer l'effet sur le profit si S passe de 198 à 200, avec $\lambda = 100$, $c = 2$, $h = 0.1$, $K = 10$, $k = 1$ et $p = 0.95$. Chacune des trois expériences simule les deux systèmes pendant 200 jours et évalue la différence de profit. Afin d'obtenir un intervalle de confiance à 90% sur cette différence, chaque paire de simulations est répétée 5 000 fois de façon indépendante.

Listing 5.1 – Exemple d'utilisation des variables aléatoires communes

```
import umontreal.iro.lecuyer.rng.*;
import umontreal.iro.lecuyer.randvar.*;
import umontreal.iro.lecuyer.probdist.PoissonDist;
import umontreal.iro.lecuyer.stat.Tally;
import umontreal.iro.lecuyer.util.*;

public class InventoryCRN extends Inventory {
   Tally statDiff = new Tally ("stats on difference");
```

```
public InventoryCRN (double lambda, double c, double h,
                     double K, double k, double p) {
   super (lambda, c, h, K, k, p);
}

public void simulateDiff (int n, int m,
                          int s1, int S1, int s2, int S2) {
   double value1, value2;
   statDiff.init();
   for (int i=0; i<n; i++) {
      value1 = simulateOneRun (m, s1, S1);
      value2 = simulateOneRun (m, s2, S2);
      statDiff.add (value2 - value1);
   }
}

public void simulateDiffCRN (int n, int m,
                             int s1, int S1, int s2, int S2) {
   double value1, value2;
   statDiff.init();
   streamDemand.resetStartStream();
   streamOrder.resetStartStream();
   for (int i=0; i<n; i++) {
      value1 = simulateOneRun (m, s1, S1);
      streamDemand.resetStartSubstream();
      streamOrder.resetStartSubstream();
      value2 = simulateOneRun (m, s2, S2);
      statDiff.add (value2 - value1);
      streamDemand.resetNextSubstream();
      streamOrder.resetNextSubstream();
   }
}

public void simulateDiffCRNMan (int n, int m,
                                int s1, int S1, int s2, int S2) {
   double value1, value2;
   statDiff.init();
   RandomStreamManager rsm = new RandomStreamManager();
   rsm.add (streamDemand);
   rsm.add (streamOrder);
   rsm.resetStartStream();
   for (int i=0; i<n; i++) {
      value1 = simulateOneRun (m, s1, S1);
      rsm.resetStartSubstream();
```

```
            value2 = simulateOneRun (m, s2, S2);
            statDiff.add (value2 - value1);
            rsm.resetNextSubstream();
         }
      }

      public static void main (String[] args) {
         InventoryCRN system = new InventoryCRN
             (100.0, 2.0, 0.1, 10.0, 1.0, 0.95);

         Chrono timer = Chrono.createForSingleThread();
         system.simulateDiff (5000, 200, 80, 198, 80, 200);
         System.out.println (system.statDiff.reportAndCIStudent (0.9, 3));
         double varianceIndep = system.statDiff.variance();
         System.out.println ("Total CPU time: " + timer.format() + "\n");

         timer.init();
         system.simulateDiffCRN (5000, 200, 80, 198, 80, 200);
         System.out.println (system.statDiff.reportAndCIStudent (0.9, 3));
         double varianceCRN = system.statDiff.variance();
         System.out.println ("Total CPU time: " + timer.format());

         timer.init();
         system.simulateDiffCRNMan (5000, 200, 80, 198, 80, 200);
         System.out.println (system.statDiff.reportAndCIStudent (0.9, 3));
         System.out.println ("Total CPU time: " + timer.format());

         System.out.println
            ("Variance ratio:  " + PrintfFormat.format
             (7, 4, 1, varianceIndep / varianceCRN));
      }
}
```

La méthode `simulateDiff` effectue l'expérience avec des variables aléatoires indépendantes. La méthode `simulateDiffCRN`, très similaire à la précédente, utilise les variables aléatoires communes. Pour ce faire, chacun des deux générateurs est d'abord réinitialisé au début de leurs premiers sous-intervalles. 200 jours sont simulés avec la politique (s_1, S_1) et les générateurs sont réinitialisés pour simuler avec (s_2, S_2) de façon à obtenir les mêmes variables aléatoires. La différence de profit est ensuite recueillie et les générateurs sont réinitialisés au début de leurs sous-intervalles suivants. Malheureu-

sement, ajouter une nouvelle variable aléatoire au modèle impliquerait la modification de trois endroits dans cette méthode.

La troisième méthode, `simulateDiffCRNMan`, utilise une liste de générateurs pour réduire la redondance. D'abord, une instance `rsm` de `RandomStreamManager` est construite et les deux générateurs sont ajoutés dans sa liste interne. Si nous adaptions `Inventory` à ce nouveau mécanisme, il serait possible de simplifier le code en enregistrant les générateurs au moment de leur création. Lorsque les deux générateurs sont ajoutés à la liste, un seul appel de méthode suffit pour effectuer les différences réinitialisations.

5.1.2 Usine abstraite

Dans un programme de simulation complexe, il est nécessaire de construire plusieurs générateurs de variables aléatoires uniformes pour les différentes variables aléatoires du modèle. Des appels au constructeur de la classe choisie sont alors répétés sur plusieurs lignes, voire répartis dans plusieurs classes. S'il est nécessaire de remplacer le générateur de variables aléatoires uniformes pour effectuer des tests, il faut retrouver chacune des occurrences et les modifier. Par exemple, il peut être nécessaire de tester le même programme avec un générateur MRG32k3a et LFSR113 [18].

Pour résoudre ce problème, plutôt que construire directement les générateurs de variables aléatoires, l'utilisateur emploie une usine abstraite. Il crée alors une instance d'une classe implantant `RandomStreamFactory` et définissant une méthode `newInstance` construisant et retournant le générateur. Au lieu de construire le générateur directement, l'utilisateur appelle alors la méthode `newInstance` de l'objet implantant `RandomStreamFactory`.

`BasicRandomStreamFactory`, implantation de l'interface précédente, permet de construire un générateur quelle que soit sa classe, pourvu qu'elle définisse un constructeur ne prenant aucun argument, ce qui est le cas de tous les générateurs fournis par SSJ.

Dans l'exemple du listing 5.1, si nous souhaitions comparer plusieurs générateurs, nous pourrions modifier `Inventory` de façon à ce que son constructeur prenne comme

argument une usine abstraite utilisée pour créer les générateurs. Le listing 5.2, contrairement à la classe originale de [18], construit les générateurs dans le constructeur plutôt qu'au moment de leur déclaration. Un programme d'application ou une sous-classe peut alors changer le type des générateurs de variables aléatoires en passant une implantation différente de `RandomStreamFactory`. La méthode `main` modifiée utilise cette possibilité pour construire deux instances de la classe afin de comparer les résultats obtenus avec les générateurs MRG32k3a et LFSR113. Le `Chrono` est employé afin d'évaluer le temps de processeur nécessaire à l'exécution du programme dans les deux cas.

Listing 5.2 – Modification de `Inventory` pour utiliser une usine abstraite

```
public class Inventory {
// ...
   RandomStream streamDemand;
   RandomStream streamOrder;
// ...
   public Inventory (double lambda, double c, double h,
                     double K, double k, double p,
                     RandomStreamFactory rsf) {
      streamDemand = rsf.newInstance();
      streamOrder = rsf.newInstance();
      this.lambda = lambda;
      this.c = c;  this.h = h;  this.K = K;  this.k = k;  this.p = p;
      genDemand = new PoissonGen (streamDemand, new PoissonDist (lambda));
   }

   // Le reste de la classe Inventory, inchangé

   public static void main (String[] args) {
      Chrono timer = Chrono.createForSingleThread();
      RandomStreamFactory rsf1 =
         new BasicRandomStreamFactory (MRG32k3a.class);
      RandomStreamFactory rsf2 =
         new BasicRandomStreamFactory (LFSR113.class);
      Inventory system1 = new Inventory (100.0, 2.0, 0.1,
                  10.0, 1.0, 0.95, rsf1);
      Inventory system2 = new Inventory (100.0, 2.0, 0.1,
                  10.0, 1.0, 0.95, rsf2);
      system1.simulateRuns (500, 2000, 80, 200);
      System.out.println (system1.statProfit.reportAndCIStudent (0.9, 3));
```

```
        System.out.println ("Total CPU time: " + timer.format());

        timer.init();
        system2.simulateRuns (500, 2000, 80, 200);
        System.out.println (system2.statProfit.reportAndCIStudent (0.9, 3));
        System.out.println ("Total CPU time: " + timer.format());
    }
```

L'utilisateur peut vouloir créer une implantation personnalisée de l'interface pour automatiquement ajouter les générateurs à une liste ou en réutiliser certains déjà créés au lieu d'en construire de nouveaux.

5.2 Nouveaux collecteurs statistiques

SSJ fournit la classe `StatProbe` qui permet la collecte statistique. La sous-classe `Tally` permet de recueillir un certain nombre d'observations individuelles afin de calculer une moyenne, une variance empirique et un intervalle de confiance sous l'hypothèse de normalité, présentée à la section 4.1. La classe `Accumulate` permet quant à elle de calculer l'intégrale d'une fonction constante par morceaux et relative au temps de simulation. Chaque fois que la fonction à calculer change, un tel accumulateur doit être mis à jour et ajuster la valeur de l'intégrale. Ces collecteurs statistiques supportent le modèle des observateurs pour diffuser les observations ajoutées ou les mises à jour effectuées à un ensemble d'objets enregistrés. Par défaut, ce système est désactivé pour une performance maximale.

Ces fonctionnalités conviennent à de petits programmes ne servant qu'à un petit ensemble d'expérimentations et qui sont suffisamment simples pour être souvent modifiés et recompilés avec un risque d'erreur acceptable. Par contre, un simulateur générique et complexe doit pouvoir estimer un grand nombre de mesures. Bien qu'un petit sous-ensemble de valeurs est utilisé lors d'une analyse, toutes doivent être calculées pour répondre au plus grand nombre possible de besoins sans imposer de multiples recompilations. Puisque souvent, l'analyste n'examinera que les résultats produits par le programme sans adapter ce dernier à ses besoins, il est important que ceux-ci soient formatés

de façon compacte et claire. Il est donc nécessaire d'étendre le système de statistiques pour mieux gérer cette complexité.

Parfois, plusieurs statistiques similaires peuvent être calculées, par exemple le temps d'attente moyen pour différents types de clients. Malheureusement, les collecteurs statistiques proposés par SSJ gèrent des observations scalaires seulement. Il faut alors utiliser plusieurs collecteurs indépendants, ce qui peut poser certains problèmes. En effet, chaque champ doit être géré individuellement, accroissant la taille des programmes et rendant leur maintenance plus difficile. Par exemple, l'oubli de l'instruction d'initialisation d'un collecteur faussera complètement les résultats obtenus. Pour diminuer le nombre de variables à gérer et éviter des erreurs, certains collecteurs similaires peuvent être regroupés dans des tableaux. Même avec cette solution, la création des collecteurs, l'ajout des observations et la génération des rapports nécessitent tous des boucles `for` encombrant le code et la fonctionnalité de rapports est adaptée au cas de collecteurs scalaires seulement. La génération d'un rapport consiste à produire une chaîne de caractères dans laquelle sont encodées des informations destinées à être affichées à l'écran ou sauvegardées dans un fichier texte. Le rapport produit par défaut n'est pas suffisamment compact pour être appliqué à un groupe de collecteurs reliés si bien qu'il sera difficile d'y localiser l'information adéquate. Par exemple, la figure 5.1 montre un rapport présentant l'estimation de $E[X_{\mathrm{g},k,0}(s)]$, le nombre espéré de contacts de type k arrivés pendant la période 0 et servis après un temps d'attente inférieur à s, pour $k = 0, 1, 2$ dans le cas de l'exemple présenté à la section 4.1. Sur la figure, le caractère \ à la fin des lignes d'en-tête indique une coupure typographique ; dans le rapport produit par SSJ, l'en-tête se présente sur une seule ligne. Ce rapport présente chaque mesure comme si elle était indépendante, répétant plusieurs informations inutilement. S'il couvrait plusieurs autres mesures de performance, retrouver l'information serait plutôt difficile.

5.2.1 Collecteurs vectoriels

La classe `ArrayOfStatProbes` représente un tableau de collecteurs statistiques unis par une certaine relation logique. Contrairement à un tableau de collecteurs scalaires, un tel objet peut être initialisé en une seule instruction et permet d'obtenir un

```
REPORT on Tally stat. collector ==> Number of contacts meeting target \
   service level (Type 0/Period 0)
        min        max      average    standard dev.  num. obs.
      3.000    367.000    173.469       96.612        10000
  95.0% confidence interval for mean: (   171.575,   175.363 )

REPORT on Tally stat. collector ==> Number of contacts meeting target \
   service level (Type 1/Period 0)
        min        max      average    standard dev.  num. obs.
      8.000    505.000    266.483      137.257        10000
  95.0% confidence interval for mean: (   263.792,   269.174 )

REPORT on Tally stat. collector ==> Number of contacts meeting target \
   service level (Type 2/Period 0)
        min        max      average    standard dev.  num. obs.
     10.000    574.000    265.899      158.950        10000
  95.0% confidence interval for mean: (   262.783,   269.014 )
```

Figure 5.1 – Exemple de rapport statistique qui n'est pas suffisamment compact

vecteur contenant les sommes ou les moyennes. Tout objet de cette classe contient un tableau de collecteurs scalaires qui est géré automatiquement.

Pour ce qui est des rapports, il est bien entendu possible de retrouver les objets `StatProbe` du tableau et recourir à la fonctionnalité originale de SSJ. Pour obtenir des rapports plus compacts, `ArrayOfStatProbes` définit une nouvelle méthode de génération. Pour produire un rapport pour un tableau complet, elle crée une en-tête unique suivie d'un rapport d'une seule ligne pour chaque collecteur scalaire. Les rapports produits par cette fonctionnalité ressemblent à ceux qui sont affichés sur la figure 4.1.

La sous-classe `ArrayOfTallies` gère un tableau de `Tally` et permet l'ajout de vecteurs d'observations en une seule instruction. Il est également possible d'afficher un rapport compact incluant un intervalle de confiance de Student-t pour chaque élément du vecteur, comme calculé à l'exemple de la section 4.1. Une sous-classe nommée `ArrayOfTalliesWithCovariance` est disponible pour estimer la covariance entre chaque paire de compteurs du tableau, sans conserver toutes les observations. Il est ainsi possible d'obtenir les matrices de covariance et de corrélation empirique, quelle

que soit la longueur du tableau de collecteurs statistiques.

La *covariance* entre deux variables aléatoires X et Y est définie par

$$\text{Cov}(X,Y) = E[(X - E[X])(Y - E[Y])] = E[XY] - E[X]E[Y] \tag{5.1}$$

tandis que la *corrélation* est définie par

$$\text{Cor}(X,Y) = \frac{\text{Cov}(X,Y)}{\sqrt{\text{Var}(X)\text{Var}(Y)}}. \tag{5.2}$$

La covariance est estimée en utilisant la covariance empirique

$$\text{Cov}(X,Y) \approx S_{X,Y,n} = \frac{1}{n-1}\sum_{r=0}^{n-1}(X_r - \bar{X}_n)(Y_r - \bar{Y}_n) \tag{5.3}$$

tandis que la corrélation est estimée en remplaçant, dans (5.2), chaque quantité par son estimateur :

$$\text{Cor}(X,Y) \approx \hat{\rho}_{X,Y,n} = \frac{S_{X,Y,n}}{S_{X,n}S_{Y,n}}. \tag{5.4}$$

Nous supposons ici que X_r suit la même loi de probabilité que X et que $E[X] \approx \bar{X}_n$ et $E[Y] \approx \bar{Y}_n$.

Habituellement, puisque des vecteurs d'observations sont ajoutés à une instance de `ArrayOfTallies` plutôt que des observations individuelles directement dans les `Tally` internes, chaque élément d'un tableau contient un nombre identique d'observations. C'est pourquoi par défaut, dans les rapports, nous avons choisi de ne pas afficher le nombre d'observations pour chaque collecteur. Par contre, de façon générale, chaque collecteur d'un tableau pourrait contenir un nombre distinct d'observations. Par exemple, cela peut se produire si un simulateur considère des vecteurs aléatoires $\mathbf{X}_r = (X_{0,r},\ldots,X_{d-1,r})$ et $\mathbf{Y}_r$ pour tenter d'estimer $(E[X_{0,r}/Y_{0,r}],\ldots,E[X_{d-1,r}/Y_{d-1,r}])$ avec $X_{i,r} \leq Y_{i,r}$ et $Y_{i,r}$ proche de 0. Ce cas s'est produit lors de l'estimation du niveau de service dans un centre de contacts, pour divers types de contacts pendant diverses périodes. Pour certaines réplications r, nous pouvons avoir $X_{i,r} = Y_{i,r} = 0$, ce qui génère une observation invalide (*NaN*) qui est automatiquement ignorée. Ainsi, le nombre d'observations dans

chaque composante du vecteur peut varier. Pour tenir compte de cette possibilité, nous avons ajouté une option qui peut être activée pour afficher le nombre d'observations pour chaque collecteur individuel dans les rapports.

La classe `ArrayOfAccumulates` permet quant à elle de regrouper des compteurs calculant des intégrales en fonction du temps de simulation. Contrairement à `ArrayOfTallies`, cette classe ne comporte pas de méthode de mise à jour prenant un vecteur en paramètre, car les mises à jour ne se produisent pas simultanément pour tous les collecteurs. Encore une fois, des méthodes sont disponibles pour afficher des rapports statistiques plus compacts que ceux obtenus avec les classes de base de SSJ.

5.2.2 Collecteurs matriciels

Parfois, un vecteur de collecteurs statistiques ne suffit pas. Par exemple, une statistique peut être calculée pour chaque type de contact et chaque période. Il est bien entendu possible d'utiliser un tableau de taille $a * b$ pour représenter une matrice de $a \times b$ ou de $b \times a$, mais de nouvelles fonctionnalités sont nécessaires pour afficher un rapport pour une rangée et une colonne seulement. De plus, il peut être commode de manipuler les observations sous une forme matricielle plutôt que vectorielle.

C'est pourquoi nous avons défini une classe de collecteur matriciel pour chaque collecteur vectoriel. Grâce à ces nouveaux collecteurs, il est possible d'obtenir des matrices de sommes, de moyennes, etc. La classe `MatrixOfStatProbes` constitue l'ancêtre de tous les collecteurs matriciels. Chacune de ses instances contient un tableau interne de collecteurs statistiques scalaires et un rapport peut être obtenu pour chaque ligne ou chaque colonne de la matrice, encore une fois avec une seule ligne par scalaire.

5.2.3 Fonctions de plusieurs moyennes

Soit $\{\mathbf{Y}_n = (Y_{0,n}, \ldots, Y_{d-1,n}), n \geq 0\}$ une suite de vecteurs convergeant vers un vecteur $\boldsymbol{\mu}$ lorsque $n \to \infty$. Pour une fonction $g : \mathbb{R}^d \to \mathbb{R}$ continue, $g(\mathbf{Y}_n)$ converge vers $g(\boldsymbol{\mu})$. Considérant que les $\mathbf{Y}_n$ sont des vecteurs aléatoires, nous nous intéressons à la loi de probabilité de $g(\mathbf{Y}_n)$ pour n grand. La fonction de plusieurs moyennes $g(\mathbf{Y}_n)$ est utili-

sée pour estimer $\nu = g(\boldsymbol{\mu})$. Bien que les classes des sections précédentes permettent déjà de calculer les moyennes nécessaires au calcul de $g(\mathbf{Y}_n)$, certains outils sont nécessaires pour obtenir un intervalle de confiance sur ν.

Le théorème Delta [19, 30] que nous présentons ici est un outil important pour le calcul d'une intervalle de confiance sur une fonction de plusieurs moyennes. Supposons que $\mathbf{Y}$ est un vecteur aléatoire et que

$$r(n)(\mathbf{Y}_n - \boldsymbol{\mu}) \Rightarrow \mathbf{Y} \text{ si } n \to \infty$$

avec $r(n) \to \infty$ si $n \to \infty$. Le symbole $\Rightarrow$ représente ici la convergeance en loi de probabilité. Lorsque $\mathbf{Y}_n$ est un vecteur aléatoire obtenu par une moyenne

$$\mathbf{Y}_n = \frac{1}{n}\sum_{r=0}^{n-1} \mathbf{X}_r = \frac{1}{n}\sum_{r=0}^{n-1} (X_{0,r}, \ldots, X_{d-1,r})$$

suivant le théorème limite centrale, nous avons $r(n) = \sqrt{n}$ et $\mathbf{Y}$ suit la loi normale à d dimensions avec moyenne $\mathbf{0}$ et matrice de covariance $\boldsymbol{\Sigma}$.

Avec les définitions précédentes, si $g(\boldsymbol{\mu})$ est continue et dérivable autour de $\boldsymbol{\mu}$, nous pouvons déterminer son gradient $\nabla g(\boldsymbol{\mu}) = (\partial g(\boldsymbol{\mu})/\partial \mu_0, \ldots, \partial g(\boldsymbol{\mu})/\partial \mu_{d-1})'$. Le théorème Delta indique que

$$r(n)(g(\mathbf{Y}_n) - g(\boldsymbol{\mu})) \Rightarrow (\nabla g(\boldsymbol{\mu}))'\mathbf{Y}. \tag{5.5}$$

Lorsque $\mathbf{Y}_n$ est une moyenne suivant le théorème limite centrale, puisque $\mathbf{Y}$ est multinormale, $Z = (\nabla g(\boldsymbol{\mu}))'\mathbf{Y}$ est une combinaison linéaire de variables aléatoires normales qui suit ainsi elle-même la loi normale. La moyenne de Z est 0 étant donné que $E[\mathbf{Y}] = (E[Y_0], \ldots, E[Y_{d-1}]) = \mathbf{0}$ et sa variance est

$$\begin{aligned} \sigma^2 &= n\mathrm{Var}(g(\mathbf{Y}_n)) = \mathrm{Var}((\nabla g(\boldsymbol{\mu}))'\mathbf{Y}) \\ &= \sum_{i=0}^{d-1}\sum_{j=0}^{d-1} (\nabla g(\boldsymbol{\mu}))_i (\nabla g(\boldsymbol{\mu}))_j \Sigma_{i,j} = (\nabla g(\boldsymbol{\mu}))'\boldsymbol{\Sigma}\nabla g(\boldsymbol{\mu}). \end{aligned}$$

L'intervalle de confiance sur ν est calculé en utilisant le fait que

$$\sqrt{n}(g(\mathbf{Y}_n) - g(\boldsymbol{\mu}))/\sigma \Rightarrow N(0,1)$$

qui découle du théorème Delta. La variance σ^2, inconnue, est estimée en remplaçant $\boldsymbol{\mu}$ par $\mathbf{Y}_n$ et la matrice $\boldsymbol{\Sigma}$ par une matrice de covariance empirique $\mathbf{S}_n$ dont les éléments sont calculés par

$$S_{X_{i,r},X_{j,r},n} = \frac{1}{n-1}\sum_{r=0}^{n-1}(X_{i,r} - \bar{X}_{i,n})(X_{j,r} - \bar{X}_{j,n})$$

pour $i = 0,\ldots,d-1$ et $j = 0,\ldots,d-1$. L'estimateur de variance est donné par

$$\hat{\sigma}_n^2 = (\nabla g(\mathbf{Y}_n))'\mathbf{S}_n \nabla g(\mathbf{Y}_n).$$

Le centre de l'intervalle est alors $g(\mathbf{Y}_n)$ tandis que son rayon est $z_{1-\alpha/2}\hat{\sigma}_n/\sqrt{n}$, où $z_{1-\alpha/2} = \Phi^{-1}(1-\alpha/2)$, la fonction inverse de la loi normale, évaluée à $1-\alpha/2$.

Par exemple, cette technique peut être utilisée pour obtenir des intervalles de confiance sur des rapports d'espérances. Soit $((X_0,Y_0),\ldots,(X_{n-1},Y_{n-1}))$ des vecteurs aléatoires indépendants. La moyenne $\bar{X}_n = \frac{1}{n}\sum_{r=0}^{n-1} X_r$ est un estimateur de l'espérance $\mu_1 = E[X]$ et $\bar{Y}_n$ estime l'espérance $\mu_2 = E[Y]$. Nous souhaitons estimer le rapport d'espérances $\nu = g(\mu_1,\mu_2) = \mu_1/\mu_2$ en supposant que $\mu_2 \neq 0$ et en utilisant $\bar{\nu}_n = \bar{X}_n/\bar{Y}_n$. Par le théorème Delta, nous avons

$$\sqrt{n}(\bar{\nu}_n - \nu)/\sigma \Rightarrow N(0,1).$$

Pour obtenir σ^2, nous avons besoin du gradient

$$\nabla g(\mu_1,\mu_2) = \left(\frac{1}{\mu_2}, -\frac{\mu_1}{\mu_2^2}\right)$$

et de la matrice de covariance

$$\boldsymbol{\Sigma} = \begin{pmatrix} \mathrm{Var}(X) & \mathrm{Cov}(X,Y) \\ \mathrm{Cov}(X,Y) & \mathrm{Var}(Y) \end{pmatrix}.$$

Nous avons alors

$$\begin{aligned}\sigma^2 &= \mathrm{Var}(X)/\mu_2^2 - 2\mathrm{Cov}(X,Y)\mu_1/\mu_2^3 + \mathrm{Var}(Y)\mu_1^2/\mu_2^4 \\ &= (\mathrm{Var}(X) - 2\mathrm{Cov}(X,Y)\mu_1/\mu_2 + \mathrm{Var}(Y)\mu_1^2/\mu_2^2)/\mu_2^2 \\ &= (\mathrm{Var}(X) + \nu^2\mathrm{Var}(Y) - 2\nu\mathrm{Cov}(X,Y))/\mu_2^2.\end{aligned}$$

Encore une fois, la variance σ^2 est estimée en remplaçant les espérances et variances par leurs estimateurs.

Ce théorème est implanté par la classe `FunctionOfMultipleMeansTally` pour une fonction quelconque en dimension d. Un tel collecteur comprend un tableau interne de collecteurs `Tally` afin de traiter des vecteurs d'observations $\mathbf{X}_r$. En utilisant `ArrayOfTalliesWithCovariance`, il est possible d'obtenir la valeur du vecteur de moyennes $\mathbf{Y}_n$ ainsi que la matrice de covariance empirique $\mathbf{S}_n$ sans mémoriser toutes les observations.

Pour utiliser l'implantation, une sous-classe concrète de `FunctionOfMultipleMeansTally` doit implanter des méthodes pour calculer $g(\mathbf{Y}_n)$ et $\nabla g(\mathbf{Y}_n)$. Par exemple, la classe `RatioTally` supporte le cas courant des rapports de moyennes.

Des collecteurs vectoriels et matriciels pour regrouper ces nouveaux collecteurs de fonctions sont également proposés. Ils peuvent par exemple permettre d'estimer le niveau de service pour chaque type de contact et chaque période en plus du niveau global, dans un centre de contacts.

5.3 Gestion des expérimentations

Comme dans la section précédente, nous avons une suite de vecteurs $\{\mathbf{Y}_n, n \geq 0\}$ qui converge vers un vecteur $\boldsymbol{\mu}$ que nous souhaitons estimer. $\mathbf{Y}_n$ est obtenu en faisant une moyenne sur des vecteurs aléatoires $\mathbf{X}_r$ représentant des résultats de simulation. Les éléments de $\mathbf{X}_r$ sont obtenus en calculant le nombre d'occurrences d'un certain événement, une somme de valeurs ou une intégrale sur le temps de simulation, durant une étape r de l'expérience. La plupart du temps, $\mathbf{X}_r$ n'est pas représenté directement par un

vecteur à l'intérieur d'un programme mais plutôt par un groupe de scalaires, de vecteurs et de matrices. Par exemple, dans le programme de la section 4.1, $\mathbf{X}_r$ contient $X_{\mathrm{g}}(s)$, X, Y, $Y_{\mathrm{b}}(s)$, $X_{\mathrm{g},k,p}(s)$ pour $k = 0,\ldots,K-1$ et $p = 0,\ldots,P-1$, $\int_0^T N_{\mathrm{b},i}(t)\,dt$, $\int_0^T N_i(t)\,dt$ et $\int_0^T N_{\mathrm{g},i}(t)\,dt$.

De façon générale, une simulation produit un échantillon $(\mathbf{X}_0,\ldots,\mathbf{X}_{n-1})$ utilisé pour calculer la moyenne

$$\bar{\mathbf{X}}_n = (\bar{X}_{0,n},\ldots,\bar{X}_{d-1,n}) = \frac{1}{n}\sum_{r=0}^{n-1}\mathbf{X}_r$$

et la covariance empirique $S_{X_{i,r},X_{j,r},n}$ de paires de composantes (i,j), où $i = 0,\ldots,d-1$ et $j = 1,\ldots,d-1$. La moyenne $\bar{\mathbf{X}}_n$ permet d'estimer l'espérance $\boldsymbol{\mu} = E[\mathbf{X}_r]$ tandis que l'estimateur $S_{X_{i,r},X_{j,r},n}$ approxime la covariance $\mathrm{Cov}(X_{i,r},X_{j,r})$. Finalement, pour chaque composante j, un intervalle de confiance $[I_{1,j},I_{2,j}]$ sur $\mu_j = E[X_{j,r}]$ peut être calculé.

Il arrive également que $L \geq 0$ fonctions de plusieurs moyennes $\bar{\nu}_{\ell,n} = g_\ell(\bar{\mathbf{X}}_n)$, où $\ell = 0,\ldots,L-1$, soient calculées dans le but d'estimer $\boldsymbol{\nu} = (\nu_0,\ldots,\nu_{L-1})$, où $\nu_\ell = g_\ell(\boldsymbol{\mu})$. Par exemple, un programme pourrait définir $g_0(\boldsymbol{\mu}) = \mu_0/\mu_2$ et $g_1(\boldsymbol{\mu}) = \mu_1/\mu_2$. Un intervalle de confiance sur ces fonctions peut être calculé en utilisant le théorème Delta présenté à la section précédente et nécessitant les covariances entre chaque paire de scalaires. Toutefois, si $g_\ell(\bar{\mathbf{X}}_n)$ ne dépend que de $\bar{X}_{j,n}$, pour $j = 0,\ldots,d-1$, le théorème Delta n'est pas nécessaire pour calculer l'intervalle puisque la fonction ne considère qu'une seule moyenne.

Les intervalles de confiance sont calculés sur une fonction de $\mathbf{Y}_n$ à la fois, car le calcul d'ellipsoïdes de confiance n'est pas supporté par SSJ pour le moment. Lors de l'analyse des résultats, nous pouvons considérer une seule statistique $\bar{X}_{j,n}$ ou $\bar{\nu}_{\ell,n}$ à la fois avec un intervalle de confiance unidimensionnel ou plusieurs statistiques avec un intervalle de confiance multidimensionnel en forme de boîte dont le niveau global minimal est déterminé par l'inégalité de Bonferroni définie par l'équation (4.1).

La méthode utilisée pour estimer $\boldsymbol{\mu}$ dépend du type d'horizon simulé. Dans cette section, nous traitons le cas de l'horizon fini et infini avant de présenter les classes que nous avons proposées pour faciliter la gestion des expérimentations.

5.3.1 Simulation sur horizon fini

Une simulation est sur horizon fini lorsque le temps d'arrêt est fini. Par exemple, dans le cas des centres de contacts, il est possible de simuler une journée, une semaine, un mois, etc. Afin d'estimer les covariances et d'obtenir des intervalles de confiance, la simulation est répétée n fois pour obtenir $(\mathbf{X}_0, \ldots, \mathbf{X}_{n-1})$.

Dans le cas d'une simulation par événements discrets, chaque fois qu'un événement E_k se produit pendant la réplication r, le vecteur $\mathbf{X}_r$ est mis à jour en ajoutant un vecteur aléatoire de coûts $\mathbf{C}_{k,r} \in \mathbb{R}^d$. Par exemple, si le k^{e} événement représente la sortie d'un contact du système, $C_{k,j,r}$ peut être 1 s'il a été servi et 0 dans le cas contraire tandis que les autres composantes de $\mathbf{C}_{k,r}$ sont 0. Soit maintenant $N_r(t)$ le nombre d'événements qui se sont produits pendant la réplication r, durant l'intervalle $[0,t]$. La simulation sur horizon fini vise à estimer le coût total pendant un temps T ou pour un certain nombre m d'événements.

Pour un horizon fini de durée T, nous avons, pour la réplication r,

$$\mathbf{X}_r = \sum_{k=0}^{N_r(T)-1} \mathbf{C}_{k,r}.$$

L'horizon peut également être borné par le nombre d'événements N ; le temps T est alors aléatoire et

$$\mathbf{X}_r = \sum_{k=0}^{N} \mathbf{C}_{k,r}.$$

L'implantation d'une méthode expérimentale pour un tel horizon est simple : il suffit de répéter l'expérimentation n fois, générant un ensemble de vecteurs aléatoires i.i.d. $(\mathbf{X}_0, \ldots, \mathbf{X}_{n-1})$. Cette méthode simple est utilisée à la section 4.1 pour l'exemple de centre de contacts.

Avec la technique précédente, le niveau $1-\alpha$ des intervalles de confiance peut être défini, mais leur largeur est aléatoire et parfois plus grande que souhaité. L'*échantillonnage séquentiel* peut être utilisé pour contrôler la largeur des intervalles de confiance avec un niveau de confiance donné. Lorsque cette technique est employée, n est rendu aléatoire en effectuant un contrôle d'erreur après certaines réplications. Tant qu'une cer-

taine erreur relative n'est pas atteinte pour un ensemble prédéterminé de mesures de performance, le nombre de réplications à simuler augmente. Malheureusement, l'échantillonnage séquentiel produit des estimateurs biaisés, car le nombre de réplications est aléatoire.

5.3.2 Simulation sur horizon infini

Une simulation est sur horizon infini lorsque le comportement à long terme d'un système à l'état stationnaire est analysé. Dans ce cas, nous souhaitons estimer le coût moyen par unité de temps

$$\boldsymbol{\mu} = \lim_{T\to\infty} \frac{1}{T} E\left[\sum_{k=0}^{N(T)-1} \mathbf{C}_k\right] \overset{\text{w.p. 1}}{=} \lim_{T\to\infty} \frac{1}{T} \sum_{k=0}^{N(T)-1} \mathbf{C}_k,$$

w.p. 1 signifiant « avec probabilité 1. » Nous pouvons aussi estimer le coût moyen par événement

$$\boldsymbol{\mu} = \lim_{N\to\infty} \frac{1}{N} E\left[\sum_{k=0}^{N-1} \mathbf{C}_k\right] \overset{\text{w.p. 1}}{=} \lim_{N\to\infty} \frac{1}{N} \sum_{k=0}^{N-1} \mathbf{C}_k.$$

Ces deux formulations sont semblables, si bien que nous allons nous concentrer sur la première qui est plus générale. Le temps de simulation T doit être borné si bien que tout estimateur de $\boldsymbol{\mu}$ est biaisé. Le biais est causé à la fois par l'horizon tronqué et par le fait que souvent, le système simulé ne devient stationnaire qu'après un certain temps.

Il est possible d'estimer $\boldsymbol{\mu}$ de la même façon qu'à la section précédente, en répétant l'expérimentation n fois et en calculant la somme des coûts sur un horizon tronqué. Pour réduire le biais, de façon générale, il vaut mieux choisir T grand et n petit. Le biais est la plupart du temps minimal lorsque $n = 1$, si bien qu'une seule copie de $\mathbf{X}_r$ est générée par cette méthode. Afin de réduire davantage le biais, les événements se produisant pendant l'intervalle $[0, T_0]$ sont éliminés, ce qui permet de diminuer l'effet des conditions initiales sur l'estimateur. Ainsi, nous avons

$$\boldsymbol{\mu} \approx \bar{\boldsymbol{\mu}}_{N(T),T_0} = \frac{1}{T - T_0} \sum_{k=N(T_0)}^{N(T)-1} \mathbf{C}_k \tag{5.6}$$

ou encore

$$\boldsymbol{\mu} \approx \bar{\boldsymbol{\mu}}_{N,N_0} = \frac{1}{N-N_0} \sum_{k=N_0}^{N-1} \mathbf{C}_k. \tag{5.7}$$

Le temps T_0 ou le nombre d'événements N_0 doivent être suffisamment élevés pour diminuer le biais tout en demeurant beaucoup plus petits que T ou N pour éviter de trop augmenter la variance. Malheureusement, lorsque $n = 1$, il est difficile d'estimer des covariances et de calculer des intervalles de confiance sur les composantes de $\boldsymbol{\mu}$ et $\boldsymbol{\nu}$.

Une méthode pour remédier à ce problème consiste à regrouper les $\mathbf{C}_k$ en lots. Cela revient souvent à diviser l'intervalle $[T_0, T]$ en n sous-intervalles de taille $(T - T_0)/n$. Chaque lot $r = 0, \ldots, n-1$ est formé par les coûts imposés pendant l'intervalle de temps $[T_r, T_{r+1})$ où $T_0 < \cdots < T_n$. En particulier, si la taille des lots est fixe, $T_r = (T - T_0)r/n + T_0$. Dans ce cadre, l'observation r est donnée par

$$\mathbf{X}_r = \sum_{k=N(T_r)}^{N(T_{r+1})-1} \mathbf{C}_k.$$

L'estimateur de $\boldsymbol{\mu}$ devient alors

$$\bar{\boldsymbol{\mu}}_{N(T),T_0} = \frac{\frac{1}{n}\sum_{r=0}^{n-1} \mathbf{X}_r}{\frac{1}{n}\sum_{r=0}^{n-1}(T_{r+1} - T_r)} = \frac{\sum_{r=0}^{n-1} \mathbf{X}_r}{T - T_0} = \frac{1}{T - T_0} \sum_{k=N(T_0)}^{N(T)-1} \mathbf{C}_k. \tag{5.8}$$

Nous revenons ainsi à l'équation (5.6). Si chaque lot a une longueur identique, il n'est pas nécessaire de considérer un rapport de moyennes pour estimer $\boldsymbol{\mu} = E[\mathbf{X}_r]/(T - T_0)$ ou $\boldsymbol{\mu} = E[\mathbf{X}_r]/(N - N_0)$. Nous verrons l'utilité de la formulation précédente lorsque la longueur des lots sera aléatoire.

La technique la plus simple pour estimer les covariances entre les éléments de $\mathbf{X}_r$ lorsque les lots ont une longueur fixe consiste à considérer les $\mathbf{X}_r$ comme i.i.d. et à utiliser l'estimateur habituel. Pour obtenir un intervalle de confiance sur une composante de $\boldsymbol{\mu}$, il suffit de calculer cet intervalle sur la composante correspondante de $\mathbf{X}_r$ et de diviser le résultat par $T - T_0$ ou $N - N_0$. Avec cette méthode, l'estimateur est biaisé, car les coûts, même regroupés, ne sont pas indépendants. Pour réduire le biais, les lots doivent être suffisamment longs afin de minimiser la corrélation entre les valeurs regroupées,

mais leur nombre doit être suffisant pour estimer la variance avec précision. Certains estimateurs tenant compte de la corrélation entre les lots ont été proposés [16, 19] et doivent pouvoir, dans un système générique, remplacer l'estimateur simple.

La taille des lots peut être exprimée en unités de temps de simulation mais également en terme du nombre d'occurrences d'un certain événement. Par exemple, chaque lot peut regrouper un nombre identique de clients traités. Dans ce cas, il faudra remplacer $\bar{\boldsymbol{\mu}}_{N(T),T_0}$ par $\bar{\boldsymbol{\mu}}_{N,N_0}$ pour estimer $\boldsymbol{\mu}$ sans devoir utiliser un rapport de moyennes.

La taille des lots peut même être aléatoire, par exemple si elle correspond à un cycle régénératif. Soit $\{\mathbf{C}(t), t \geq 0\}$ un processus stochastique, c'est-à-dire une famille de vecteurs aléatoires. L'instant T_1 est un *point de régénération* si $\{\mathbf{C}(t+T_1), t \geq 0\}$ est stochastiquement équivalent à $\{\mathbf{C}(t), t \geq 0\}$ et indépendant de T_1 et de $\{\mathbf{C}(t), t < T_1\}$. Le terme *stochastiquement équivalent* signifie que les processus sont soumis aux mêmes lois de probabilité. En d'autres mots, au temps T_1, le processus stochastique est remis à zéro et son état et sa progression ne dépendent pas de son état avant T_1. Un processus stochastique ayant un point de régénération est dit *régénératif* et le processus $\{\mathbf{C}(t+T_1), t \geq 0\}$ est lui aussi régénératif avec point de régénération T_2. Nous avons ainsi une suite de points de régénération $T_1 < \cdots < T_n$ et les intervalles $[T_r, T_{r+1})$ entre ces points, pour $r = 0, \ldots, n-1$, forment des *cycles régénératifs*. Le premier cycle régénératif qui nous intéresse débute au temps T_0 qui est souvent 0. Il arrive parfois que le début de la simulation est rejeté, ne correspondant pas à l'état stationnaire ; le premier cycle régénératif commence alors au temps $T_0 > 0$. Si $Y_r = T_{r+1} - T_r$ correspond à la longueur du cycle r, par le théorème de renouvellement [19],

$$\boldsymbol{\mu} = \frac{E[\mathbf{X}_r]}{E[Y_r]}$$

Cette quantité peut être estimée par $\bar{\boldsymbol{\mu}}_{N(T),T_0}$ de l'équation (5.8), si bien que les cycles régénératifs peuvent être gérés de la même façon que des moyennes par lots classiques, sauf que les lots ont une taille aléatoire et correspondent à des cycles régénératifs. Étant donné que la longueur des cycles est aléatoire, l'estimateur de $\boldsymbol{\mu}$ devient également un rapport de moyennes si bien que le théorème Delta doit être utilisé pour calculer les

intervalles de confiance. Toutefois, étant donné que les cycles sont indépendants, les estimateurs de $\boldsymbol{\mu}$, de variances et de covariances sont sans biais tant que l'échantillonnage séquentiel n'est pas utilisé.

Si l'échantillonnage séquentiel est utilisé, le temps total de simulation devient aléatoire. Deux options sont alors possibles : le nombre de lots peut être aléatoire tandis que leur taille ne dépend pas de T ou le nombre de lots est constant tandis que leur taille augmente avec T. Dans ce second cas, à divers moments pendant la simulation, le temps peut être redivisé de façon à toujours produire un nombre identique d'intervalles de plus en plus longs.

5.3.3 Problèmes à résoudre pour l'implantation

Étant donné le nombre de variantes possibles, surtout avec la méthode des moyennes par lots, la construction d'un système d'expérimentation générique est assez difficile. Nous avons tenté de le faire afin de permettre la simulation de centres de contacts sur horizon infini pour ensuite généraliser le système. Pour le cas d'un horizon fini, comme le montre l'exemple de la section 4.1, aucune classe de support n'est réellement nécessaire, surtout si l'échantillonnage séquentiel n'est pas utilisé. Pour effectuer l'expérience, un groupe de compteurs réinitialisés au début de chaque réplication est utilisé pour additionner les $\mathbf{C}_{k,r}$ pendant les réplications tandis qu'un groupe de collecteurs statistiques sert à recueillir les $\mathbf{X}_r$. Il n'est pas nécessaire de conserver les observations, même quand les covariances empiriques sont calculées. Une classe de support peut toutefois être utile pour standardiser la structure générale des simulateurs afin d'aider l'utilisateur à s'y retrouver et à étendre un programme.

La simulation sur horizon infini avec moyennes par lots devient plus complexe si l'échantillonnage séquentiel avec taille de lots dépendant du temps total est utilisé. À moins de conserver toutes les valeurs des coûts $\mathbf{C}_k$, ce qui nécessiterait trop de mémoire, il n'est pas possible d'effectuer de multiples regroupements de façon totalement arbitraire. Afin de résoudre ce problème, le temps de simulation est divisé en m intervalles de relativement petite taille appelés *lots réels* et ces lots réels sont regroupés pour former $n \leq m$ *lots effectifs*.

Pour chacun des m lots réels, un vecteur $\mathbf{V}_j \in \mathbb{R}^d$ peut être calculé en sommant les coûts dans l'intervalle $[T_j, T_{j+1})$. Si le temps total de simulation est constant, $m = n$,

$$\mathbf{X}_r = \mathbf{V}_r \text{ pour } r = 0, \ldots, n-1$$

et la gestion de l'expérimentation est pratiquement aussi simple que pour des réplications indépendantes. Puisque chaque lot effectif contient toujours un seul lot réel, aucun regroupement n'est nécessaire. Ainsi, au début de chaque lot r, la valeur $\mathbf{V}_{r-1}$ du lot précédent est stockée et des compteurs sont réinitialisés. Parfois, avant d'être stocké dans un collecteur statistique, $\mathbf{X}_r$ peut être normalisé en fonction de la taille du lot r, par exemple pour obtenir le nombre de contacts servis par heure plutôt que par lot. Si le temps total de simulation et n sont aléatoires, nous avons toujours $\mathbf{X}_r = \mathbf{V}_r$ et aucun problème ne surgit.

Par contre, si le nombre effectif de lots n doit être constant tandis que leur taille est aléatoire, les $\mathbf{V}_j$ doivent être conservés et regroupés lorsque la simulation est terminée. Le nombre m de lots réels doit toujours, au moment du regroupement, être un multiple h du nombre de lots effectifs n, de façon à ce que

$$\mathbf{X}_r = \sum_{j=0}^{h-1} \mathbf{V}_{rh+j} \text{ pour } r = 0, \ldots, n-1.$$

Ce regroupement s'effectue sans aucune perte d'information si bien que toute valeur de $h = 1, \ldots$ est possible au cours de la simulation.

Pour supporter ces modes d'expérimentation, un ensemble de collecteurs statistiques est nécessaire pour stocker les $\mathbf{V}_j$ tandis qu'un second ensemble permet de recueillir les $\mathbf{X}_r$ après le traitement des $\mathbf{V}_j$. Le premier groupe de collecteurs statistiques pour les lots réels doit mémoriser toutes les valeurs afin de permettre le regroupement si nécessaire. Dans le cas du second groupe, il n'est pas nécessaire de mémoriser les observations si seules des statistiques de base telles que la moyenne et les covariances empiriques sont nécessaires. Pour optimiser l'utilisation de la mémoire, il est possible de n'utiliser que le second groupe de collecteurs lorsqu'aucun regroupement de lots réels n'est à prévoir,

mais cela introduit un dédoublement de code dans le programme de l'utilisateur. En effet, selon que le regroupement est actif ou non, les observations doivent être placées dans des collecteurs différents.

5.3.4 Exemple de simulateur

Pour tenter de résoudre ces problèmes plus simplement dans un programme de simulation, nous avons tenté d'écrire des classes abstraites de support. La classe de base `SimUtils` propose diverses méthodes utiles pour l'échantillonnage séquentiel, `RepSim` est prévue pour le cas de l'horizon fini et `BatchMeansSim` permet de gérer l'horizon infini. Pour tirer parti de ce système, l'utilisateur doit créer une sous-classe et implanter un certain nombre de méthodes. Les classes contiennent également des listes dans lesquelles peuvent être ajoutés des collecteurs statistiques pour automatiser leur initialisation.

Pour illustrer le fonctionnement de notre système de support à l'expérimentation, nous allons utiliser une variante de l'exemple de file $M/M/1$ du guide d'exemples de SSJ [18]. Dans une file $M/M/1$, des clients arrivent selon un processus de Poisson avec taux λ et sont servis par un seul serveur pendant une durée exponentielle de moyenne $1/\mu$. Si un client arrive tandis que le serveur est occupé, il doit attendre en file. Les abandons ne sont pas permis si bien que tous les clients sont servis. Ce système est régénératif, car son comportement est indépendant du passé lorsqu'il se vide.

Le programme original, présenté dans [18], simule une seule réplication sur horizon fini de durée T. Le nouveau programme, présenté sur le listing 5.3, implante le même modèle que l'ancien tout en ajoutant du code pour être simulé avec des réplications indépendantes, des moyennes par lots et des cycles régénératifs.

Soit $W(t_1,t_2)$ la somme des temps d'attente pendant l'intervalle de temps $[t_1,t_2)$, $X(t_1,t_2)$ le nombre de clients servis pendant $[t_1,t_2)$ et $Q(t)$ la taille de la file au temps t. Soit également $Q(t_1,t_2) = \int_{t_1}^{t_2} Q(t)\,dt$ l'intégrale de la taille de la file pendant l'intervalle

$[t_1, t_2)$. Nous souhaitons estimer le temps d'attente moyen par client

$$w = \lim_{T \to \infty} \frac{E[W(0,T)]}{E[X(0,T)]}$$

et la taille moyenne de la file à long terme

$$q = \lim_{T \to \infty} \frac{1}{T} E\left[\int_0^T Q(t)\, dt\right].$$

Pour estimer w et q, nous allons tester trois approches : tronquer l'horizon et simuler des réplications indépendantes, simuler une seule réplication et la diviser en lots de taille égale et utiliser des cycles régénératifs. Un intervalle de confiance à 95% est également calculé sur w et q sous l'hypothèse de normalité.

Le programme utilise l'échantillonnage séquentiel afin d'imposer une borne supérieure sur l'erreur relative du temps d'attente moyen. Un intervalle de confiance à 95% est calculé sur le temps d'attente moyen et est donné par $\bar{w}_n \pm \delta_n$. Nous souhaitons que l'erreur relative $\delta_n / \bar{w}_n$ soit au plus 0.5%. Toutefois, la simulation peut être arrêtée après un certain temps même si l'erreur n'est pas suffisamment petite afin d'éviter que la procédure d'échantillonnage séquentiel prenne trop de temps.

Le programme du listing 5.3 définit d'abord une classe `QueueEvBatch` implantant le modèle de simulation voulu. À l'intérieur de cette classe se trouve une classe interne pour chacune des méthodes expérimentales. La méthode `main` crée d'abord une file d'attente avec $\lambda = 1$ et $\mu = 2$. À partir de cette file, quatre expériences sont effectuées. D'abord, un minimum de $n = 30$ réplications indépendantes sont simulées avec horizon tronqué de durée $T = 10\,000$. Ensuite, un minimum de $m = 30$ lots réels de durée 10 000 sont simulés après une période de réchauffement de 1 000. Peu importe la valeur finale (et aléatoire) de m, le nombre de lots effectifs sera de $n = 30$ puisque les lots sont regroupés pour satisfaire cette condition. La même expérience est ensuite effectuée sans regroupement des lots ; le nombre de lots n est aléatoire. Finalement, l'expérience est refaite avec un minimum de $n = 1\,000$ cycles régénératifs. La valeur minimale de n est plus grande que dans les cas précédents, car les cycles sont très petits dans ce modèle.

Listing 5.3 – Exemple d'utilisation des classes de support à l'expérimentation

```
import umontreal.iro.lecuyer.simexp.BatchMeansSim;
import umontreal.iro.lecuyer.simexp.RepSim;
import umontreal.iro.lecuyer.simevents.*;
import umontreal.iro.lecuyer.rng.*;
import umontreal.iro.lecuyer.probdist.ExponentialDist;
import umontreal.iro.lecuyer.randvar.RandomVariateGen;
import umontreal.iro.lecuyer.stat.*;
import java.util.LinkedList;
import java.util.Observable;
import java.util.Observer;
import umontreal.iro.lecuyer.util.Chrono;
import umontreal.iro.lecuyer.util.PrintfFormat;

public class QueueEvBatch extends Observable {
   RandomVariateGen genArr;
   RandomVariateGen genServ;
   LinkedList<Customer> waitList = new LinkedList<Customer>();
   LinkedList<Customer> servList = new LinkedList<Customer>();

   // Compteurs
   double custWaits    = 0;
   int nCust           = 0;
   Accumulate totWait  = new Accumulate ("Size of queue");

   void initCounters() {
      custWaits = 0;
      nCust = 0;
      totWait.init();
   }

   class Customer { double arrivTime, servTime; }

   QueueEvBatch (double lambda, double mu) {
      genArr = new RandomVariateGen
         (new MRG32k3a(), new ExponentialDist (lambda));
      genServ = new RandomVariateGen
         (new MRG32k3a(), new ExponentialDist (mu));
   }

   class Arrival extends Event {
      public void actions() {
         new Arrival().schedule (genArr.nextDouble()); // Prochaine arrivée
         Customer cust = new Customer();   // Client arrivé
```

```
      cust.arrivTime = Sim.time();
      cust.servTime = genServ.nextDouble();
      if (servList.size() > 0) {        // Doit joindre la file
         waitList.addLast (cust);
         totWait.update (waitList.size());
      } else {                          // Début du service
         custWaits += 0.0;
         ++nCust;
         servList.addLast (cust);
         new Departure().schedule (cust.servTime);
      }
   }
}

class Departure extends Event {
   public void actions() {
      servList.removeFirst();
      if (waitList.size() > 0) {
         // Début du service pour le prochain en file
         Customer cust = waitList.removeFirst();
         totWait.update (waitList.size());
         custWaits += Sim.time() - cust.arrivTime;
         ++nCust;
         servList.addLast (cust);
         new Departure().schedule (cust.servTime);
      }
      else {
         // Le système est vide
         setChanged();
         notifyObservers();
      }
   }
}

class EndOfSim extends Event {
   public void actions() {
      Sim.stop();
   }
}

// Expérience avec des réplications indépendantes
class RepExp extends RepSim {
   // Paramètres
   double timeHorizon;     // Durée T de l'horizon de simulation
```

```
double targetError;     // Erreur relative à atteindre pour w
double level;           // Niveau des intervalles de confiance

// Collecteurs pour les réplications
RatioTally custWait = new RatioTally ("Waiting time");
Tally queueSize     = new Tally ("Size of queue");

RepExp (int minReps, int maxReps, double timeHorizon,
        double targetError, double level) {
   super (minReps, maxReps);
   this.timeHorizon = timeHorizon;
   this.targetError = targetError;
   this.level = level;
}

// Au début de l'expérience, avant la première réplication
protected void initReplicationProbes() {
   custWait.init();
   queueSize.init();
}

// Initialisation, au début de chaque réplication
protected void initReplication (int r) {
   waitList.clear();
   servList.clear();
   initCounters();
   new EndOfSim().schedule (timeHorizon);
   new Arrival().schedule (genArr.nextDouble());
}

// Ajout des observations, à la fin de chaque réplication
protected void addReplicationObs (int r) {
   custWait.add (custWaits, nCust);
   queueSize.add (totWait.average());
}

public int getRequiredNewReplications() {
   if (targetError < 0) return 0;
   return getRequiredNewObservations (custWait, targetError, level);
}

public String report() {
   return "Total simulation time: " +
      PrintfFormat.f (10, 0, getCompletedReplications()*timeHorizon) +
```

```
            "      Number of replications: " + getCompletedReplications() +
            "\n" + custWait.reportAndCIDelta (level, 3) +
            queueSize.reportAndCIStudent (level, 3);
    }
}

 // Expérience avec des moyennes par lots
 class BatchExp extends BatchMeansSim {
    // Paramètres
    double targetError;     // Erreur relative à atteindre pour w
    double level;           // Niveau des intervalles de confiance

    // Collecteurs pour les lots réels
    TallyStore rcustWait  = new TallyStore();
    TallyStore rnCust     = new TallyStore();
    TallyStore rqueueSize = new TallyStore();

    // Collecteurs pour les lots effectifs
    RatioTally custWait = new RatioTally ("Waiting time");
    Tally queueSize     = new Tally ("Size of queue");

    BatchExp (int minBatches, int maxBatches, double batchSize,
              double warmupTime, double targetError, double level) {
       super (minBatches, maxBatches, batchSize, warmupTime);
       this.targetError = targetError;
       this.level = level;
    }

    protected void initRealBatchProbes() {
       rcustWait.init();
       rnCust.init();
       rqueueSize.init();
    }

    protected void initEffectiveBatchProbes() {
       custWait.init();
       queueSize.init();
    }

    protected void initBatchStat() {
       initCounters();
    }

    protected void initSimulation() {
```

```
      waitList.clear();
      servList.clear();
      new Arrival().schedule (genArr.nextDouble());
   }

   protected void addRealBatchObs() {
      rcustWait.add (custWaits);
      rnCust.add (nCust);
      rqueueSize.add (totWait.sum());
   }

   // Ajoute des observations pour un lot effectif dont la
   // durée est l unités de temps de simulation et regroupant les
   // lots réels s, s+1, ..., s+h-1.
   protected void addEffectiveBatchObs (int s, int h, double l) {
      double w = getSum (rcustWait.getArray(), s, h);
      double n = getSum (rnCust.getArray(), s, h);
      double q = getSum (rqueueSize.getArray(), s, h);
      custWait.add (w, n);
      queueSize.add (q/l);
   }

   public int getRequiredNewBatches() {
      if (targetError < 0) return 0;
      int n = getRequiredNewObservations (custWait, targetError, level);
      if (getBatchAggregation())    n *= getNumAggregates();
      return n;
   }

   public String report() {
      return "Total simulation time: " +
         PrintfFormat.f (10, 0, Sim.time()) +
         "      Number of real batches: " +
         getCompletedRealBatches() + "\n" +
         custWait.reportAndCIDelta (level, 3) +
         queueSize.reportAndCIStudent (level, 3);
   }
}

// Expérience avec des cycles régénératifs
class RegExp extends BatchExp implements Observer {
   RatioTally queueSize = new RatioTally ("Size of queue");

   RegExp (int minBatches, int maxBatches,
```

```
            double targetError, double level) {
        super (minBatches, maxBatches, 1, 0, targetError, level);
        QueueEvBatch.this.addObserver (this);
        setBatchAggregation (false);
    }

    // Redéfinition de la condition de fin des lots
    protected void beginFirstBatch() {}

    public void update (Observable o, Object arg) {
        // Début d'un cycle régénératif
        newBatch();
    }

    // Supprimer la logique nécessaire pour les lots effectifs
    protected void initEffectiveBatchProbes() {}
    protected void addEffectiveBatchObs (int s, int h, double l) {}

    protected void initRealBatchProbes() {
        custWait.init();
        queueSize.init();
    }

    protected void addRealBatchObs() {
        custWait.add (custWaits, nCust);
        double t = getRealBatchLength (getCompletedRealBatches() - 1);
        queueSize.add (totWait.sum(), t);
    }

    public String report() {
        return "Total simulation time: " +
            PrintfFormat.f (10, 0, Sim.time()) +
            "      Number of cycles: " +
            getCompletedRealBatches() + "\n" +
            custWait.reportAndCIDelta (level, 3) +
            queueSize.reportAndCIDelta (level, 3);
    }
}

public static void main (String[] args) {
    QueueEvBatch queue = new QueueEvBatch (1.0, 2.0);
    System.out.print ("Independent replications");
    RepExp simRep = queue.new RepExp (30, 900, 10000.0, 0.005, 0.95);
    Chrono timer = Chrono.createForSingleThread();
```

```
      simRep.simulate();
      System.out.println (" (CPU time: " + timer.format() + ")");
      System.out.println (simRep.report());

      System.out.print ("Batch means, with aggregation");
      BatchExp simBatch = queue.new BatchExp
         (30, 900, 10000, 1000.0, 0.005, 0.95);
      simBatch.setBatchAggregation (true);
       timer.init();
       simBatch.simulate();
       System.out.println (" (CPU time: " + timer.format() + ")");
       System.out.println (simBatch.report());

       simBatch.setBatchAggregation (false);
       simBatch.setTargetBatches (simBatch.getMinBatches());
       System.out.print ("Batch means, without aggregation");
       timer.init();
       simBatch.simulate();
       System.out.println (" (CPU time: " + timer.format() + ")");
       System.out.println (simBatch.report());

       System.out.print ("Regenerative cycles");
       RegExp simReg = queue.new RegExp
          (1000, Integer.MAX_VALUE, 0.005, 0.95);
       timer.init();
       simReg.simulate();
       System.out.println (" (CPU time: " + timer.format() + ")");
       System.out.println (simReg.report());
   }
}
```

Le modèle est fortement inspiré de [18]. Trois types d'événements peuvent se produire : arrivée d'un client, départ d'un client ou arrêt de la simulation. Dans le programme original, un collecteur statistique compte les temps d'attente tandis qu'un second collecteur calcule l'intégrale sur la taille de la file. Nous avons remplacé le collecteur `custWaits` recueillant les temps d'attente par une valeur numérique puisque seule la somme des temps d'attente nous intéresse ici. Un autre compteur, appelé `nCust`, est utilisé pour compter le nombre de clients servis. Le collecteur `totWait` sert quant à lui à calculer l'intégrale sur la taille de la file. Le reste du programme implante exactement

le modèle $M/M/1$ décrit dans [18].

La classe interne `RepExp` sert à effectuer l'expérience avec des réplications indépendantes. Un objet de cette classe doit connaître la longueur T de l'horizon de simulation, le niveau $1-\alpha$ des intervalles de confiance et l'erreur relative à atteindre. Chaque réplication produit un vecteur $\mathbf{X}_r$ contenant $W(0,T)$, $X(0,T)$ et $Q(0,T)$. Nous pourrions ajouter $W(0,T)/X(0,T)$ à un collecteur statistique pour estimer le temps d'attente moyen, mais cette approche estimerait $\lim_{T\to\infty} E[W(0,T)/X(0,T)] \neq w$. Puisque nous souhaitons estimer le temps à long terme, nous devons définir une fonction de plusieurs moyennes $g_0(W(t_1,t_2),X(t_1,t_2),Q(t_1,t_2)) = W(t_1,t_2)/X(t_1,t_2)$. Ainsi, un collecteur statistique sur un rapport est défini pour le temps d'attente tandis qu'un autre collecteur recueille la taille moyenne de la file pendant chaque réplication. Il n'est pas nécessaire de calculer un intervalle de confiance sur un rapport dans le cas de la taille de la file, car T est constant. Si l'horizon était borné par le nombre de clients plutôt que le temps, $X(0,N(T)) = N(T)$ serait constant tandis que T serait aléatoire ; il faudrait une fonction de plusieurs moyennes pour estimer q tandis qu'elle ne serait pas nécessaire pour w. Dans tous les cas, les estimateurs calculés sont biaisés, car l'horizon est tronqué.

La simulation est déclenchée par la méthode `simulate` appelée depuis la méthode `main`. Cette méthode, implantée dans `RepSim`, commence par initialiser les collecteurs statistiques enregistrés et appelle `initReplicationProbes` pour permettre à l'utilisateur de compléter l'initialisation si nécessaire. Ici, nous avons choisi de manipuler tous les collecteurs manuellement afin de mieux illustrer le fonctionnement de notre système.

Ensuite, pour chaque réplication, `simulate` appelle `performReplication`. Cette méthode interne vide la liste d'événements avec `Sim.init` et appelle la méthode `initReplication` pour initialiser le modèle. La méthode d'initialisation vide la file d'attente avec `waitList.clear` et le serveur avec `servList.clear` pour ensuite remettre les compteurs à 0 à l'aide de `initCounters`. Un événement est planifié pour terminer la simulation au temps T et la première arrivée est planifiée. Ensuite, la simulation démarre et commence à exécuter des événements. Au temps T, la simulation s'arrête et les variables aléatoires $W(0,T)$, $X(0,T)$ et $Q(0,T)$ sont disponibles et

peuvent être traitées par `addReplicationObs`. Les méthodes `initReplication` et `addReplicationObs` sont souvent combinées en une seule méthode dans un programme simple, par exemple `simulateOneDay` dans l'exemple de la section 4.1. Si nous souhaitons créer une extension de notre programme $M/M/1$ supportant les abandons, avec la séparation proposée ici, il sera possible de le faire sans complètement récrire la méthode simulant une réplication.

Ce processus est répété n fois après quoi `getRequiredNewReplications` est appelée pour appliquer l'échantillonnage séquentiel. Une méthode de support implantée dans la classe `SimUtils`, dont hérite `RepSim`, est appelée pour obtenir n^*, le nombre estimé de réplications additionnelles à simuler pour que l'erreur relative sur w soit inférieure ou égale à 0.5%. Si $n^* > 0$, de nouvelles réplications sont simulées et, au prochain contrôle d'erreur, un total $n + n^*$ réplications sont disponibles. La simulation continue jusqu'à ce que la méthode de test retourne 0 ou jusqu'à ce que le nombre maximal de réplications soit atteint. Ce maximum permet d'éviter que la procédure d'échantillonnage séquentiel prenne trop de temps. Une méthode `report` permet finalement de produire un rapport statistique portant sur les deux mesures de performance estimées.

La seconde classe interne, `BatchExp`, permet d'utiliser la méthode des moyennes par lots. Encore une fois, le niveau des intervalles de confiance et l'erreur relative sont nécessaires. Cette fois-ci, pour chaque compteur défini dans le modèle, nous avons un collecteur de lots réels capable de stocker des valeurs. Le vecteur $\mathbf{V}_j$ contient $W(T_j, T_{j+1})$, $X(T_j, T_{j+1})$ et $Q(T_j, T_{j+1})$ pour le lot réel j. Les collecteurs de lots effectifs correspondent quant à eux à ceux de la classe interne précédente. Encore une fois, la simulation est déclenchée par `simulate`.

Dans le cas de `BatchMeansSim`, la méthode `simulate` initialise l'état du simulateur et appelle une méthode interne nommée `runSimulation` et qui, par défaut, initialise la liste d'événements et appelle `initSimulation` pour initialiser le modèle. À la différence de `initReplication`, cette méthode ne remet pas les compteurs à 0 et ne planifie pas un événement au temps T pour la fin de la simulation. La simulation est démarrée par la méthode `runSimulation`. Après la période de réchauffement par défaut d'une durée fixée en temps de simulation, la méthode `initRealBatch-`

`Probes` est toujours appelée pour initialiser les collecteurs des $\mathbf{V}_j$. Par défaut, les lots ont une taille fixe exprimée en unités de temps de simulation. À chaque fin de lot, la valeur des compteurs est copiée dans les collecteurs statistiques pour les $\mathbf{V}_j$ par `addRealBatchObs`. La méthode `initBatchStat` est quant à elle utilisée pour remettre les compteurs à 0 au début de chaque lot.

La méthode `addEffectiveBatchObs` permet de calculer un vecteur $\mathbf{X}_r$ et de l'ajouter à des collecteurs de lots effectifs. Si les lots ne sont pas regroupés, cette méthode est appelée après `addRealBatchObs` avec $h = 1$ et une valeur de s correspondant au dernier lot ajouté. Dans le cas contraire, elle est appelée n fois en fin de simulation ou avant un contrôle d'erreur, avec n correspondant à la valeur de `minBatches`. Que le regroupement soit actif ou non, l'implantation donnée dans l'exemple fonctionne. La méthode `getSum` est fournie par `BatchMeansSim` et permet de sommer différents éléments consécutifs d'un tableau. Des variantes de cette méthode sont fournies pour des tableaux Java ordinaires et des matrices unidimensionnelles de Colt [13]. Une dernière méthode `getSum` permet de sommer des colonnes consécutives dans une matrice bidimensionnelle Colt et retourne un tableau de sommes.

Finalement, comme dans la classe interne précédente, nous utilisons l'échantillonnage séquentiel pour contrôler l'erreur relative sur w. La méthode `getRequiredNewBatches` effectue un contrôle d'erreur et retourne le nombre de lots réels additionnels à simuler. Le test effectué par notre méthode donne le nombre de lots effectifs additionnels à simuler. Alors, si le regroupement est actif, il faut multiplier cette valeur par h puisque la méthode doit retourner le nombre de lots réels additionnels. Encore une fois, la méthode `report` produit un rapport statistique et l'estimateur de variance utilisé pour calculer les intervalles de confiance est biaisé puisque les lots ne sont pas indépendants.

Pour les cycles régénératifs, nous devons tout d'abord prévoir un moyen d'avertir le simulateur lorsque le système est vide. Pour ce faire, le programme utilise le modèle des observateurs comme suit. Pour éviter de créer une interface d'observateurs servant uniquement pour cet exemple simple, nous utilisons les observateurs de Java. La classe `QueueEvBatch` étend `Observable` et les observateurs sont avertis chaque fois que le système se vide. De cette façon, le modèle demeure toujours indépendant de la mé-

thode expérimentale choisie.

La classe interne `RegExp` étend `BatchExp` pour modifier la condition de terminaison des lots et désactiver le stockage des $\mathbf{V}_j$. En effet, comme nous le verrons plus loin, les cycles obtenus avec le modèle $M/M/1$ sont très petits et utiliser l'échantillonnage séquentiel occasionne des débordements de mémoire. Le collecteur statistique `queueSize`, de type `Tally`, est également remplacé par un collecteur `RatioTally`, car la longueur des cycles est aléatoire, contrairement à la longueur des lots.

Le constructeur de `RegExp` appelle la superclasse avec des paramètres prédéfinis pour la durée des lots et la période de réchauffement. Il s'enregistre ensuite comme observateur de `QueueEvBatch` et désactive le regroupement qui n'est pas utile avec les cycles. Chaque fois que `update` est appelée, un nouveau cycle démarre et est annoncé au système de gestion de l'expérience de `BatchMeansSim` par la méthode `newBatch`.

Les méthodes de gestion des lots effectifs sont remplacées par des implantations vides puisque les cycles ne sont jamais regroupés. La méthode `addRealBatchObs` ajoute les observations directement dans les collecteurs des $\mathbf{X}_r$ plutôt que stocker des $\mathbf{V}_j$ temporaires.

La figure 5.2 présente les résultats produits par le programme d'exemple. Comme nous pouvons le constater, chaque méthode est en mesure de produire un résultat similaire. Toutefois, le temps total de simulation varie d'une méthode à l'autre. La méthode la plus efficace est bien entendu l'approche régénérative puisque le biais des estimateurs est le plus petit tandis que les réplications indépendantes nécessitent le plus long temps de simulation.

```
Independent replications (CPU time: 0:0:13.98)
Total simulation time:    4850000      Number of replications: 485
REPORT on Tally stat. collector ==> Waiting time
    func. of averages    standard dev.  num. obs.
                0.500          0.028          485
  95.0% confidence interval for function of means: (     0.498,     0.503 )
REPORT on Tally stat. collector ==> Size of queue
         min        max      average    standard dev.  num. obs.
       0.414      0.612        0.501          0.030          485
  95.0% confidence interval for mean: (     0.498,     0.503 )

Batch means, with aggregation (CPU time: 0:0:15.53)
Total simulation time:    5401000      Number of real batches: 540
REPORT on Tally stat. collector ==> Waiting time
    func. of averages    standard dev.  num. obs.
                0.500        5.5E-3           30
  95.0% confidence interval for function of means: (     0.498,     0.502 )
REPORT on Tally stat. collector ==> Size of queue
         min        max      average    standard dev.  num. obs.
       0.489      0.512        0.499        6.3E-3           30
  95.0% confidence interval for mean: (     0.497,     0.502 )

Batch means, without aggregation (CPU time: 0:0:13.49)
Total simulation time:    4701000      Number of real batches: 470
REPORT on Tally stat. collector ==> Waiting time
    func. of averages    standard dev.  num. obs.
                0.500          0.028          470
  95.0% confidence interval for function of means: (     0.498,     0.503 )
REPORT on Tally stat. collector ==> Size of queue
         min        max      average    standard dev.  num. obs.
       0.429      0.625        0.500          0.029          470
  95.0% confidence interval for mean: (     0.498,     0.503 )

Regenerative cycles (CPU time: 0:0:16.0)
Total simulation time:    4492936      Number of cycles: 2244754
REPORT on Tally stat. collector ==> Waiting time
    func. of averages    standard dev.  num. obs.
                0.500          1.907       2244754
  95.0% confidence interval for function of means: (     0.498,     0.503 )
REPORT on Tally stat. collector ==> Size of queue
    func. of averages    standard dev.  num. obs.
                0.501          2.065       2244754
  95.0% confidence interval for function of means: (     0.498,     0.503 )
```

Figure 5.2 – Exemple de résultats donnés par `QueueEvBatch`

CHAPITRE 6

CONCLUSION

La bibliothèque ContactCenters, contribution principale de ce mémoire, permet de construire divers simulateurs de centres de contacts complexes supportant un grand nombre de types de contacts et de groupes d'agents. Beaucoup de code est nécessaire pour écrire un simulateur, mais il est possible de construire des programmes génériques s'adaptant à plusieurs modèles. Grâce à une interface de communication, ces simulateurs sont accessibles d'une façon uniforme par des programmes d'optimisation et d'analyse statistique. ContactCenters couvre un grand nombre de cas de simulation, notamment l'implantation de politiques de routage complexes, la simulation d'appels sortants avec une liste de composition, le service d'un contact par plusieurs agents, etc. Ce mémoire fournit une base qui pourrait évoluer vers un produit commercial largement utilisé dans l'industrie. Nous avons également proposé un certain nombre d'extensions pour la bibliothèque de simulation SSJ, comme des collecteurs statistiques vectoriels et matriciels et la gestion des moyennes par lots pour simulation sur horizon infini

ContactCenters ne remplace pas totalement les outils avec interface graphique, car la construction d'un nouveau modèle exige souvent de la programmation Java qui n'est pas à la portée de tout gestionnaire de centres de contacts. Arena Contact Center Edition de Rockwell ou ccProphet de NovaSim demeurent des outils utiles pour tester différentes architectures de centres de contacts. L'utilisation des animations facilite le débogage et permet une compréhension plus intuitive des modèles. Toutefois, lorsque les modèles deviennent gros et doivent être optimisés, ContactCenters devient un outil intéressant.

Avec notre bibliothèque, les simulations sont suffisamment rapides pour autoriser des expérimentations multiples et l'utilisation d'algorithmes itératifs pour l'optimisation par simulation. Il a été possible d'implanter, avec ContactCenters, plusieurs exemples du manuel d'utilisation d'Arena Contact Center Edition et les programmes tournent environ 25 fois plus rapidement avec ContactCenters.

Dans le futur, il sera possible d'accroître davantage la performance des simulations

en réduisant la variance des différents estimateurs calculés. Pour le moment, seules les variables aléatoires communes sont mises en œuvre, mais il est possible d'expérimenter diverses autres techniques [16, 19]. Pour raffiner l'optimisation, il est prévu d'implanter des algorithmes alternatifs pour le calcul des sous-gradients. Toutes ces extensions sont possibles sans modifier profondément la bibliothèque car, grâce au modèle des observateurs, il est facile d'interagir avec chaque composante du simulateur et le système de collecte statistique peut être librement personnalisé.

Sur le long terme, pour simplifier la simulation pour un gestionnaire de centres de contacts, il serait nécessaire de disposer de simulateurs génériques dont les fichiers de configuration XML seraient cachés par un éditeur spécialisé ou une interface graphique. Le développement d'une telle interface, qui n'est pas prévu pour le moment, pourrait constituer un projet d'extension de la bibliothèque ContactCenters.

BIBLIOGRAPHIE

[1] Altova. Altova XMLSpy 2005 — XML editor, XSLT/XQuery debugger, XML Schema/WSDL designer, SOAP debugger, 2005. Voir `http://www.altova.com/products_ide.html`.

[2] J. Atlason, M. A. Epelman et S. G. Henderson. Call center staffing with simulation and cutting plane methods. *Annals of Operations Research*, 127:333–358, 2004.

[3] A. N. Avramidis, A. Deslauriers et P. L'Ecuyer. Modeling daily arrivals to a telephone call center. *Management Science*, 50(7):896–908, 2004.

[4] V. Bapat. The Arena product family : Enterprise modeling solutions. Dans S. Chick, P. J. Sánchez, D. Ferrin et D. J. Morrice, éditeurs, *Proceedings of the 2003 Winter Simulation Conference*, pages 210–217. IEEE Press, 2003.

[5] J. Bloch. *Effective Java Programming Language Guide*. Addison-Wesley, mai 2001.

[6] E. Buist et P. L'Ecuyer. *ContactCenters : A Java Library for Simulating Contact Centers*, 2005. Manuel de l'utilisateur, disponible sur `http://www-etud.iro.umontreal.ca/~buisteri/contactcenters`.

[7] E. Buist et P. L'Ecuyer. A Java library for simulating contact centers. Dans *Proceedings of the 2005 Winter Simulation Conference*. IEEE Press, 2005. Soumis.

[8] M. T. Ceẓik et P. L'Ecuyer. Staffing multiskill call centers via linear programming and simulation, 2004. Soumis.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest et C. Stein. *Introduction to Algorithms*. MIT Press, deuxième édition, septembre 2001.

[10] A. Deslauriers. Modélisation et simulation d'un centre d'appels téléphoniques dans un environnement mixte. Mémoire de maîtrise, Département d'informatique et de Recherche Opérationnelle, Université de Montréal, Montréal, Canada, 2003.

[11] E. Gamma, R. Helm, R. Johnson et J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., deuxième édition, 1998.

[12] N. Gans, G. Koole et A. Mandelbaum. Telephone call centers : Tutorial, review, and research prospects. *Manufacturing and Service Operations Management*, 5: 79–141, 2003.

[13] W. Hoschek. *The Colt Distribution : Open Source Libraries for High Performance Scientific and Technical Computing in Java*. CERN, Genèvre, 2004. Disponible sur `http://dsd.lbl.gov/~hoschek/colt`.

[14] ILOG, Inc. ILOG CPLEX : High-performance software for mathematical programming and optimization, 2005. Voir `http://www.ilog.com/products/cplex`.

[15] G. Koole, A. Pot et J. Talim. Routing heuristics for multi-skill call centers. Dans S. Chick, P. J. Sánchez, D. Ferrin et D. J. Morrice, éditeurs, *Proceedings of the 2003 Winter Simulation Conference*, pages 1813–1816. IEEE Press, 2003.

[16] A. M. Law et W. D. Kelton. *Simulation Modeling and Analysis*. McGraw-Hill, New York, troisième édition, 2000.

[17] P. L'Ecuyer. *SSC : A Library for Stochastic Simulation in C*, 2002. Manuel de l'utilisateur.

[18] P. L'Ecuyer. *SSJ : A Java Library for Stochastic Simulation*, 2004. Manuel de l'utilisateur, disponible sur `http://www.iro.umontreal.ca/~lecuyer`.

[19] P. L'Ecuyer. IFT-6561 — stochastic discrete-event simulation, 2005.

[20] P. L'Ecuyer et E. Buist. Simulation in Java with SSJ. Dans *Proceedings of the 2005 Winter Simulation Conference*, 2005. Soumis.

[21] P. L'Ecuyer et S. Côté. Implementing a random number package with splitting facilities. *ACM Transactions on Mathematical Software*, 17(1):98–111, 1991.

[22] P. L'Ecuyer, L. Meliani et J. Vaucher. SSJ : A framework for stochastic simulation in Java. Dans E. Yücesan, C.-H. Chen, J. L. Snowdon et J. M. Charnes, éditeurs, *Proceedings of the 2002 Winter Simulation Conference*, pages 234–242. IEEE Press, 2002.

[23] P. L'Ecuyer, R. Simard, E. J. Chen et W. D. Kelton. An object-oriented random-number package with many long streams and substreams. *Operations Research*, 50(6):1073–1075, 2002.

[24] S. Liang. *The Java(TM) Native Interface Programmer's Guide and Specification*. Addison-Wesley, 2002. Aussi disponible sur `http://java.sun.com/docs/books/jni`.

[25] V. Mehrotra et J. Fama. Call center simulation modeling : Methods, challenges, and opportunities. Dans S. Chick, P. J. Sánchez, D. Ferrin et D. J. Morrice, éditeurs, *Proceedings of the 2003 Winter Simulation Conference*, pages 135–143. IEEE Press, 2003.

[26] NovaSim. ccProphet — simulate your call center's performance, 2003. Voir `http://www.novasim.com/CCProphet`.

[27] Portage Communications Inc. SimACD — computer simulation software for inbound call centers, 2004. Voir `http://www.portagecommunications.com/simacd.htm`.

[28] Rockwell Automation, Inc. Arena simulation, 2005. Voir `http://www.arenasimulation.com`.

[29] S. M. Ross. *Introduction to Probability and Statistics for Engineers and Scientists*. Academic Press, 2000.

[30] R. J. Serfling. *Approximation Theorems for Mathematical Statistics*. Wiley, New York, 1980.

[31] W. Whitt et R. B. Wallace. A staffing algorithm for call centers with skill-based routing. Article en développement, disponible sur `http://www.columbia.edu/~ww2040/poolingMSOMrevR.pdf`, 2004.

[32] F. Yergeau, T. Bray, J. Paoli, C. M. Sperberg-McQueen et E. Maler. *Extensible Markup Language (XML) 1.0*. W3C Recommendation, troisième édition, février 2004. Aussi disponible sur `http://www.w3.org/TR/REC-xml`.