Master d'Informatique Fondamentale
École Normale Supérieure de Lyon

Sémantique du parallélisme
Chantal Keller

Service combinators for farming virtual machines

K. Bhargavan, A. D. Gordon, I. Narasamdya

Encadrant : Daniel Hirschkoff

12 juin 2008

Introduction

Cet article présente des mécanismes concurrents permettant de gérer des services (internet par exemple), dans le cadre particulier où ils sont implantés sur une ferme de machines virtuelles. Ces mécanismes ont été mis en place au sein du projet Baltic, qui a réalisé une interface de haut niveau pour gérer automatiquement ces services sur l'architecture choisie. Cette interface a également l'originalité de contrôler que l'implantation qu'elle réalise répond bien aux spécifications de l'utilisateur.

Nous allons présenter les points importants permettant de comprendre cet article, puis nous montrerons à travers un exemple comment des systèmes concurrents interviennent normalement pour la gestion d'une ferme de machines virtuelles orientées service. Nous verrons ensuite ce qui, dans l'architecture de la ferme, rend nécessaire ce système concurrent. Enfin, nous nous attacherons à décrire le modèle de langage utilisé pour décrire ces systèmes concurrents, et son implantation pratique dans le cadre du projet.

Table des matières

1 Présentation

Le but de l'article est d'expliquer comment créer une interface de haut niveau permettant de gérer des services sur une ferme de serveurs. Après avoir défini les termes-clés du problème, nous donnerons l'idée générale des auteurs concernant la mise en place de cette interface, et ce qui en fait son originalité. Nous présenterons enfin succinctement son fonctionnement, que l'on s'attachera ensuite à expliciter.

1.1 Définitions

Définition 1 *Virtualisation :* Ensemble des techniques permettant de faire fonctionner plusieurs machines virtuelles sur une même machine physique.

Une définition plus complète est disponible sur le site de Wikipédia [1].

Définition 2 *Ferme :* Ensemble de serveurs implantant ensemble certaines fonctions.

Définition 3 *Rôle :* Dans une ferme, ce qui donne sa tâche à un serveur.

En pratique, un serveur se lance à partir d'une image disque, et c'est cette image qui lui confère un certain rôle.

Définition 4 *Rôle orienté service :* Rôle consistant à importer et exporter des services.

Ici, on cherche à distribuer des rôles orientés services sur une ferme de processeurs, en utilisant la virtualisation : chaque processeur pourra implanter plusieurs services, et ces services communiqueront à travers un réseau virtuel. Le but du projet est d'automatiser cette distribution.

1.2 Le projet *Baltic*

Le projet *Baltic* est un projet qui a vu le jour très récemment, sous l'initiative d'Andrew D. Gordon et al. L'article à étudier est un rapport technique très détaillé de ce projet, qui présente à la fois :

- comment utiliser en pratique l'API développée par les auteurs ;
- les caractéristiques techniques de cette API ;
- comment elle est implantée ;
- les considérations théoriques ayant permis cette implantation.

Les deux premières parties ne seront pas présentées, car elle n'ont que peu de liens avec le thème qui nous occupe, à avoir les systèmes concurrents. La troisième partie, qui sera abordée en détail, permet de comprendre l'intérêt de l'utilisation de systèmes concurrents pour répondre aux problèmes du projet. Enfin, nous présenterons la dernière partie, qui permet de comprendre en quoi ces systèmes concurrents se basent sur les modèles étudiés par la communauté des chercheurs, comme le π-calcul, et en quoi ils en diffèrent dans le but de réaliser cette implantation bien spécifique.

1.3 Contribution du projet

De nombreux outils existent pour gérer "manuellement" une ferme de serveurs. Cependant, ils peuvent être difficiles à utiliser, en particulier s'ils demandent une bonne connaissance d'un langage de spécification de bas niveau pour décrire le fonctionnement de la ferme souhaitée.

En revanche, le projet *Baltic* nous propose une interface de haut niveau permettant d'implanter une ferme de services. Ainsi, l'utilisateur n'a plus qu'à décrire dans un langage de haut niveau (par exemple, XML) les spécifications de la ferme souhaitée. L'API distribue alors lui-même des tâches (souvent atomiques) aux différents serveurs virtuels. Cette distribution présente de nombreux avantages :

- l'automatisation : l'utilisateur ne gère plus le nombre de machines virtuelles à utiliser, sur quels serveurs physiques les placer, les liens à créer entre elles...
- la garantie de fonctionnement : les auteurs proposent un système de typage qui garantit que les spécifications que l'ont demande seront gérées sans conflit par le système en sortie de l'API ;
- une abstraction du réseau : les communications se feront sur un réseau virtuel indépendant du réseau physique.

1.4 Service combinators

Pour cela, les auteurs mettent un place ce qu'ils appellent des "services combinators", qui sont des serveurs installés sur chaque machine physique et faisant le lien entre les fichiers de contrôle, les services distants, et les machines virtuelles. Vus des processeurs distants, ce sont des "serveurs à fonctions" : on les appelle pour un certain service qu'ils ont à effectuer.

Au niveau de l'implantation, chaque service combinator est un module écrit dans le langage F#, et qui gère :

- les importations et exportations ;
- un contrôleur qui lui-même gère les machines virtuelles.

L'idée de ce système est que les machines virtuelles vont pouvoir travailler en concurrence, pour fournir les services le plus rapidement possible. Pour cela, on utilise une version concurrente du λ-calcul simplement typé, appelée λ-calcul partitionné, en référence aux différentes partitions sur lesquelles tournent les machines virtuelles.

Ce λ-calcul concurrent est conçu pour répondre aux besoins d'une ferme de services, mais se rapprochent d'autres modèles comme le π-calcul. En pratique, il est simulé en F#.

2 Mise en place d'un système concurrent

Nous allons présenter sur un exemple comment un système concurrent apparaît naturellement pour résoudre le problème qu'on s'est posé.

2.1 Exemple

Cet exemple, très simple, est appelé "enterprise order processing" (EOP), et correspond à la mise en place d'un service de paiement à distance. Il n'a qu'un seul rôle, qui consiste à gérer l'entrée d'ordres. Ces ordres demandent à effectuer un paiement ; il faut donc demander à un serveur distant si celui-ci est possible (voir Fig. 1).

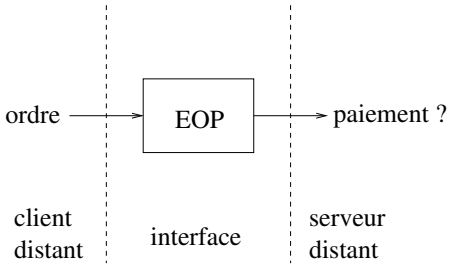

FIG. 1 – *Enterprise order processing vu comme une boîte noire*

Le rôle a donc une unique procédure, appelée `SubmitOrder`. À cette procédure, on va faire correspondre sur notre ferme une interface, appelée `ISubmitOrder`.

On choisit de découper cette interface en trois sous-interfaces :

- `IOrderEntry`, gérant un appel à `SubmitOrder` provenant d'un client distant ;
- `IPayment`, demandant à un serveur distant l'autorisation de paiement ;
- `IOrderProcessing`, effectuant l'ordre dès qu'il a reçu l'autorisation de paiement.

Ces trois interfaces vont être crées à chaque appel à `SubmitOrder` puis évoluer d'une certaine manière jusqu'à obtenir la réponse à renvoyer au client. Une configuration possible est donc que toutes les interfaces tournent sur le même serveur (voir FIG. 2).

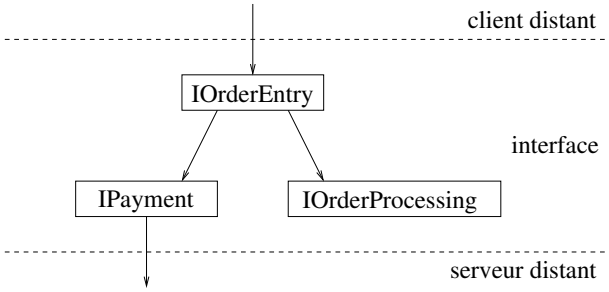

FIG. 2 – *Configuration possible*

On va maintenant voir différentes possibilités pour gérer ces interfaces, qui vont nous mener petit à petit à un système concurrent.

2.2 Première possibilité : ferme isolée

Toutes les fonctions se lancent sur la même ferme isolée (en supposant que le service d'autorisation de paiement n'est pas distant). On peut la représenter FIG. 3.

L'appel à `SubmitOrder` va faire un appel à la fonction `OrderEntry`, qui va elle-même appeler `Payment` et `OrderProcessing`.

2.3 Deuxième possibilité : ferme isolée et service de paiement distant

Cette fois, on ne crée pas de machine virtuelle pour exécuter la fonction `Payment`, mais on fait appel à un service distant (voir FIG. 4).

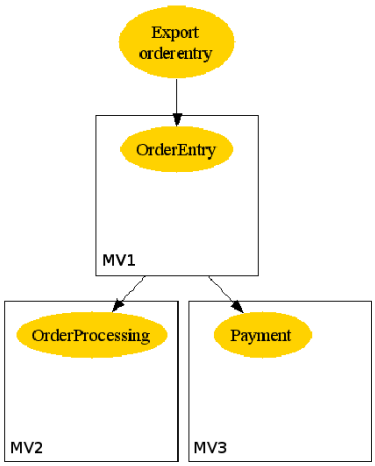

FIG. 3 – *Ferme isolée*

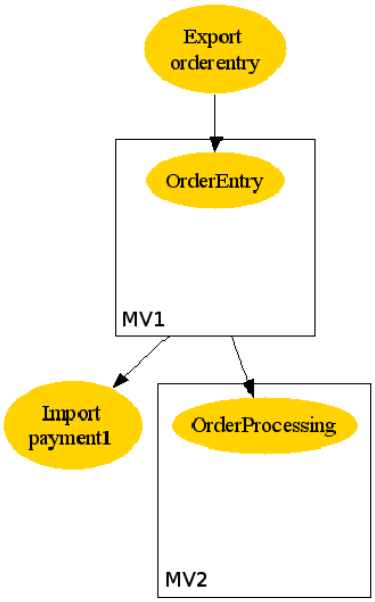

FIG. 4 – *Service de paiement distant*

2.4 Troisième possibilité : ajout de constructeurs

Lorsque les serveurs sont surchargés, il peut être utile d'avoir deux serveurs dédiés à la même tâche. Lorsqu'un client distant fait alors appel au service qu'ils implantent, deux possibilités se présentent :

- on donne cette tâche à effectuer à l'un des deux serveurs : c'est le constructeur Or ;
- on donne cette tâche à effectuer aux deux serveurs en parallèle : c'est le constructeur Par. Le résultat renvoyé sera celui du premier serveur ayant fini le calcul. Cela peut être utile par exemple si les deux serveurs fonctionnent à des périodes différentes de la journée.

Le constructeur Or est l'équivalent en *pi*-calcul du constructeur "+", et le constructeur Par est l'équivalent de la composition parallèle, à cette différence près qu'il renvoie le premier résultat calculé. On voit donc comment se sont introduites naturellement ces notions de concurrence, à partir du cas concret des serveurs surchargés.

Un exemple d'utilisation du constructeur Or dans l'exemple que nous étudions est décrit FIG. 5. Cela permet d'avoir deux serveurs dédiés au service `OrderProcessing`.

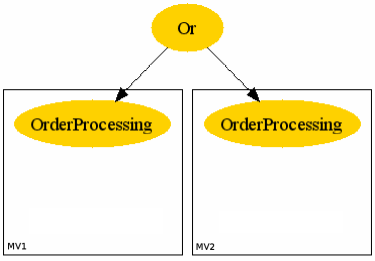

FIG. 5 – *Constructeur Or*

2.5 Quatrième possibilité : ajout de références

On peut également ajouter des références à notre langage, qui permettront de changer la topologie en cours de fonctionnement.

Prenons l'exemple de la FIG. 6, qui regroupe tout ce qu'on a introduit précédemment. Au lieu de pointer directement sur l'une des deux machines virtuelles implantant `OrderProcessing`, on passe par une référence. Ainsi, si cette machine tombe en panne ou est saturée, il est aisé de changer en machine en changeant simplement le pointeur de la référence.

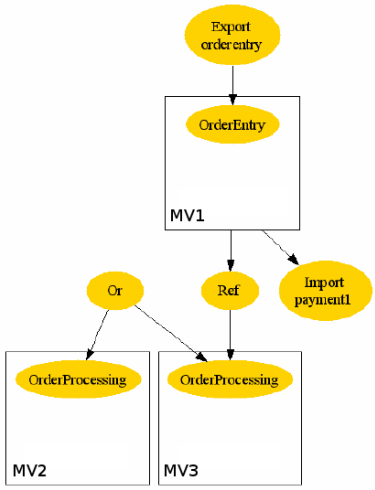

FIG. 6 – *Constructeur de références*

2.6 Conclusion

L'implantation qui est choisie pour que l'application soit robuste fait intervenir naturellement un système concurrent. Notamment, on comprend tout à fait l'intérêt dans ce cas précis des constructeurs Or et Par. Le parallèle avec les systèmes connus est évident, même si on peut remarquer quelques différences :

- on n'introduit *a priori* pas de constructeur de restriction ;
- on peut utiliser des références, qui permettent de changer rapidement la topologie alors que le système est en cours d'utilisation.

3 Architecture du système

Nous allons rapidement évoquer l'architecture du système que crée l'interface du projet Baltic lorsqu'on lui fournit une spécification. L'article présente nombre de détails techniques qui ne sont pas vraiment dans notre sujet et dont nous ne parlerons pas.

Comprendre l'architecture du système permet de bien identifier comment communiquent les différentes machines virtuelles, c'est-à-dire les différents processus, de notre implantation.

3.1 Description de l'architecture

On peut voir un schéma descriptif de l'architecture Fig. 7. Ce schéma se restreint à une seule machine physique, mais en pratique, on peut avoir plusieurs machines physiques reliées par un réseau.

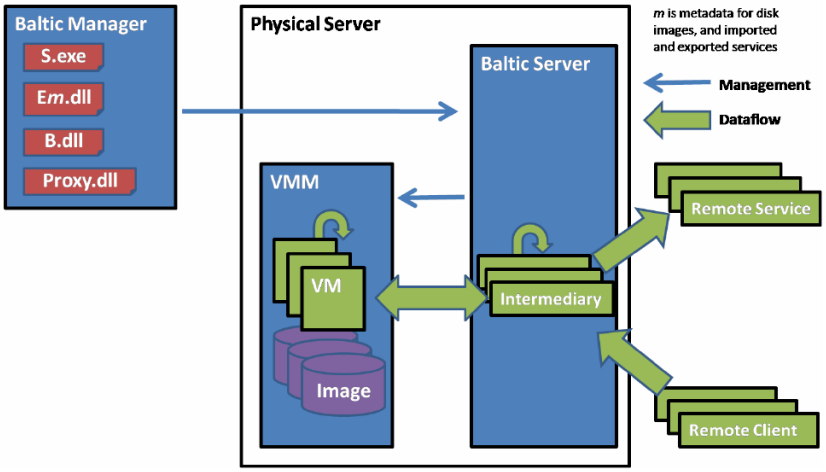

Fig. 7 – *Architecture du système*

Les développeurs du projet Baltic ont donc fourni une interface qui permet de créer les fichiers `dll` et `exe` qui vont contrôler le "Baltic server". Ce serveur est un superviseur des différentes machines virtuelles, et fait le lien entre elles et les exportations et importations. C'est lui qui va faire tourner le simulateur du langage concurrent que nous implantons, qui va gérer les machines virtuelles en parallèle.

Là encore, on retrouve l'idée que les machines virtuelles démarrent à partir d'images disques, qui implantent les rôles que l'ont va donner à chaque serveur.

3.2 Remarque

On peut maintenant cerner les liens qu'il y a entre virtualisation et systèmes concurrents. En effet, dans cette architecture, c'est la virtualisation qui permet de donner tout son sens aux machines virtuelles tournant en parallèle. Elle sont capables de communiquer entre elles et avec leur environnement. De plus, grâce au réseau virtuel, des processus même distants peuvent se synchroniser pour communiquer, et ce de manière totalement transparente.

4 Le λ-calcul partitionné

Pour pouvoir étudier leur système concurrent puis le mettre en pratique, les développeurs ont créé un modèle de programmation adapté à leur besoin, le "λ-calcul partitionné". Il intègre donc la concurrence, et possède également un système de type qui le rend robuste au niveau de l'utilisateur.

4.1 Les ingrédients du λ-calcul partitionné

Le λ-calcul partitionné tire son nom du fait qu'en pratique, chaque processus représentera une partition d'un serveur physique, qui fera tourner une machine virtuelle.

Il est créé à partir du λ-calcul simplement typé contenant le pattern-matching. On va petit à petit lui ajouter des éléments permettant de gérer des processus en parallèle (par un procédé similaire à celui de la partie 2).

4.1.1 Syntaxe

On utilise une syntaxe très similaire à celle du π-calcul. Une différence fondamentale que nous avons déjà évoquée concerne la composition parallèle, qui perd sa symétrie en λ-calcul partitionné : en effet, le premier processus qui donne la réponse fait s'arrêter tous les autres. On emploie donc le symbole "$\Rsh$" pour désigner la composition parallèle, symbole qui montre bien l'asymétrie entre les processus.

4.1.2 Sémantique

On lui donne une sémantique qui est la même qu'en λ-calcul et qui fonctionne donc également bien pour ce modèle : on introduit congruence structurelle, règles de réduction et contextes d'évaluation classiques.

4.1.3 Typage

On utilise un typage assez classique dans l'idée, puisqu'il comporte :
- des types du premier ordre, comme la chaîne de caractère ;
- des types pour désigner les canaux de communications.

Ces types permettent de contrôler que de "mauvaises actions" n'ont pas lieu. Pour cela, on crée des environnements de typage, et des jugements qui peuvent prendre plusieurs formes :
- $E \vdash \diamond$: l'environnement E est correct ;
- $E \vdash T$: le type T est correct ;
- $E \vdash A : T$: l'expression A retourne un objet de type T lorsqu'il est exécuté ;
- $E \vdash D \rightsquigarrow I$: le module D (c'est-à-dire la fonction D) implante l'interface I.

On peut utiliser cette syntaxe pour définir des règles de typage, qui sont très similaires à celles du π-calcul. La logique ainsi engendrée vérifie de nombreuses propriétés utiles, comme par exemple la réduction du sujet.

Le but de ce système de typage, outre de contrôler que les données passant sur un canal son de bon type, permet de prouver qu'une implantation réalisée par cette API à partir des spécifications est correcte, ce qui est très intéressant. L'utilisateur a donc la certitude que les fichiers créés lors de la compilation correspondent à la spécification demandée.

4.2 Simulation en F#

Ce modèle de langage est simulé en F# par les développeurs de l'article, et fonctionne donc en pratique. Nous allons voir succinctement les outils dont ils ont besoin pour créer un langage concurrent, et en particulier pour implanter les opérateurs Par et Or.

4.2.1 Opérateurs de communication, concurrence et partitionnement

On ajoute des opérateurs pour la communication, la concurrence et le partitionnement, dont les spécifications sont données Fig. 8.

ref: $\alpha \Rightarrow (\alpha)$ref	create fresh ref
get: (α)ref$\Rightarrow\alpha$	get contents of ref
set: $(\alpha \times (\alpha)$ref$) \Rightarrow$unit	set contents of ref
chan: string$\Rightarrow(\alpha)$chan	create fresh channel
send: $((\alpha)$chan $\times \alpha) \Rightarrow$unit	send message on channel
recv: (α)chan$\Rightarrow\alpha$	receive message off channel
fork: (unit $\rightarrow$ unit)$\Rightarrow$unit	start thread in parallel
name: string$\Rightarrow$name	create fresh name
box: (unit $\rightarrow$ unit)$\Rightarrow$part	create new partition
paste: (name $\times$ part)$\Rightarrow$unit	paste named partition
cut: name$\Rightarrow$part	cut named partition

Fig. 8 – *Nouveaux opérateurs*

4.2.2 Environnement et exécution d'un programme

On définit des environnements, qui correspondent à des états de la mémoires. On exécute alors des programmes (D) sur des données d'entrées et des environnements, pour obtenir les données de sorties et un nouvel environnement :

$$[\![D]\!](E, A) \rightarrow (E', A')$$

On peut maintenant fabriquer les opérateurs Par et Or.

4.2.3 Opérateur Par

Les auteurs ont choisi d'implanter l'opérateur Par dans ce F# enrichi comme décrit Fig. 9.

On écrit tout d'abord une fonction auxiliaire `par` qui crée un canal `c`, le divise en deux, et envoie sur chacun des sous-canaux `f1` ou `f2` appelée avec son argument. Elle écoute ensuite

```
let par f1 f2 x =
  let c = chan "resp" in
    fork (fun () → send c (f1 x));
    fork (fun () → send c (f2 x));
    recv c
let ePar (ep1) (ep2) =
  epFun (fun x → par (call ep1) (call ep2) x)
```

FIG. 9 – *Implantation de Par*

sur c pour récupérer le premier résultat envoyé. Il est alors aisé de créer `ePar`, qui appelle `par` sur les appels aux deux fonctions `cep1` et `ep2`.

4.2.4 Opérateur Or

Pour construire cet opérateur, on se sert de la fonction `par` qui nous avait déjà servi à construire `ePar`. L'implantation est décrite FIG. 10.

```
let pick x1 x2 = par (fun () → x1) (fun () → x2) ()
let eOr ep1 ep2 =
  epFun (fun x → call (pick ep1 ep2) x)
```

FIG. 10 – *Implantation de Or*

La fonction `pick` choisit aléatoirement l'un de ses arguments, grâce à `par` qui renvoie le premier résultat calculé. Les fonctions `fun () -> x1` et `fun () -> x2` ont effectivement *a priori* la même probabilité de retourner en premier. Il est alors simple de créer `eOr`.

4.3 Conclusion

Les auteurs ont su créer un modèle de langage qui répond à leurs besoins, et le simuler dans un langage préexistant. Bien que fait pour résoudre un problème spécifique, leur langage est très général et expressif.

5 Remarques

Cet article est très intéressant à de nombreux niveaux :
- il est très précis et bien détaillé ;
- on peut y trouver le niveau de détail que l'on souhaite, depuis apprendre à utiliser l'API qu'ils décrivent, à découvrir toute la théorie qui en découle ;
- il est très original, puisqu'il met en avant des liens entre virtualisation et systèmes concurrents, ce qui n'avait jamais été fait auparavant ;
- il traite d'un projet concret en train de voir le jour : on a un bon exemple de simulation d'un langage concurrent qui est utilisé en pratique ;
- il ouvre donc la porte à de nouveaux travaux dans les domaines des systèmes concurrents et de la virtualisation.

Les auteurs sont très précis et très clairs, ce qui est appréciable pour un article aussi riche. Le seul reproche qu'on pourrait lui faire et de ne pas chercher assez à généraliser à des systèmes déjà existants (tout tourne autour de l'implantation du projet Baltic), mais cela est tout à fait naturel compte tenu de la nature de cet article, qui est en réalité davantage un rapport technique.

Conclusion et perspectives

Cet article, tout à fait nouveau aussi bien dans le domaine de la virtualisation que dans celui des systèmes concurrents, décrit le travail réalisé par le projet Baltic. Ce dernier a créé une API permettant de gérer de manière automatique des fermes virtuelles, qui a l'originalité d'être de haut niveau, et de vérifier que l'implantation résultant des spécifications est correcte, grâce à un système de types.

Il laisse des problèmes ouverts, comme par exemple la preuve du bon fonctionnement du simulateur de λ-calcul partitionné en F#. Il précise également que certains points restent à améliorer, comme la sécurité et la résistance aux pannes.

Ces travaux tout récents ouvrent de nouveaux horizons dans les deux domaines dont il fait le lien : la concurrence est un moyen d'implanter la virtualisation, comme la virtualisation est un moyen de faire fonctionner aisément la concurrence.

Références

[1] Article de wikipedia sur la virtualisation. `http://fr.wikipedia.org/wiki/Virtualisation`.

[2] K. Bhargavan, A. D. Gordon et I. Narasamdya : Serive combinators for farming virtual machines. 2007.