Service combinators for farming virtual machines

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

Octobre 2007

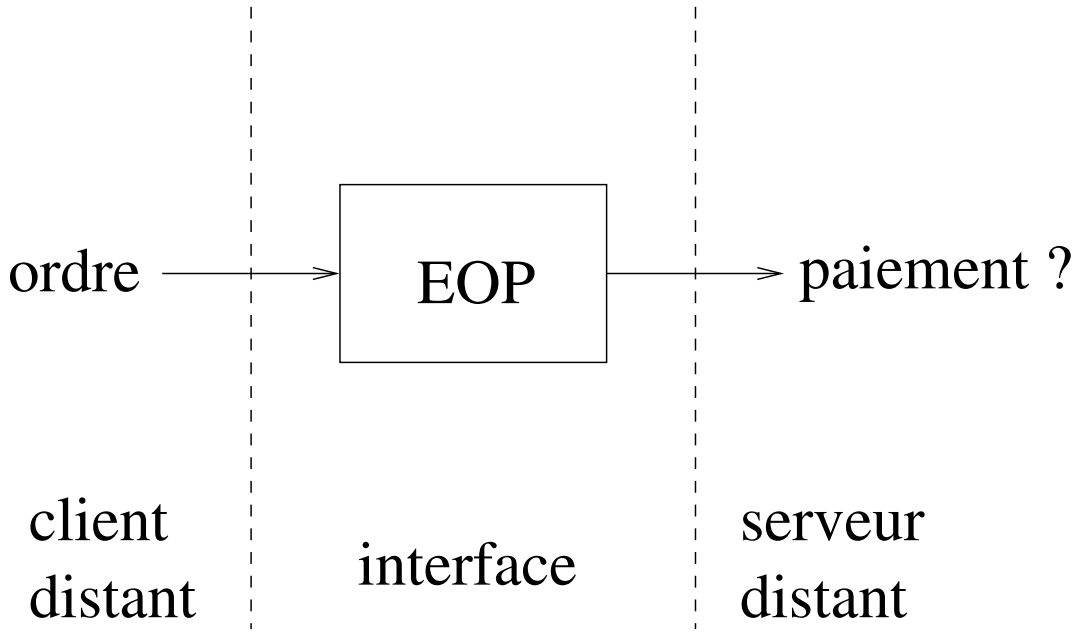
Introduction

Projet *Baltic* :

- gérer automatiquement une ferme de machines virtuelles
- introduction de systèmes concurrents

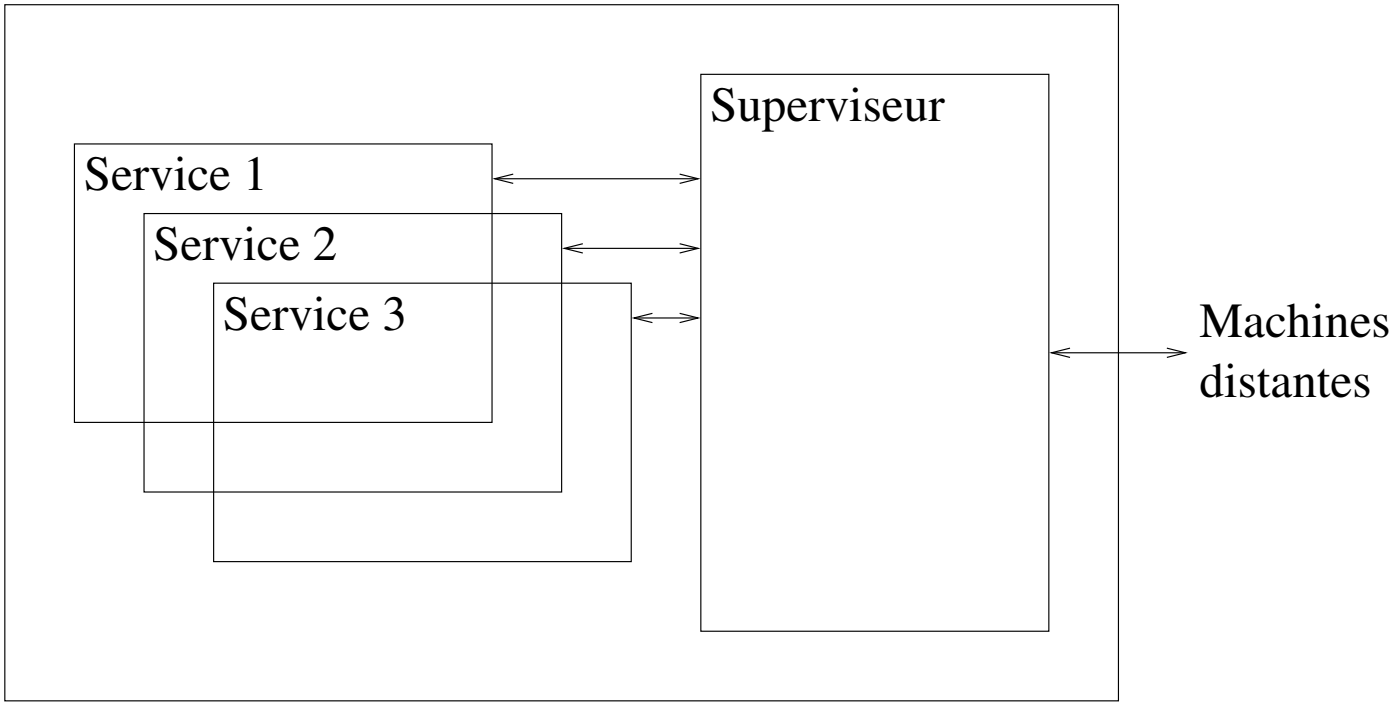
> *"La problématique de la virtualisation, c'est un champ d'application inespéré pour les résultats de la théorie de la concurrence."*
>
> A. Gordon

Cadre de l'étude : ferme orientée service

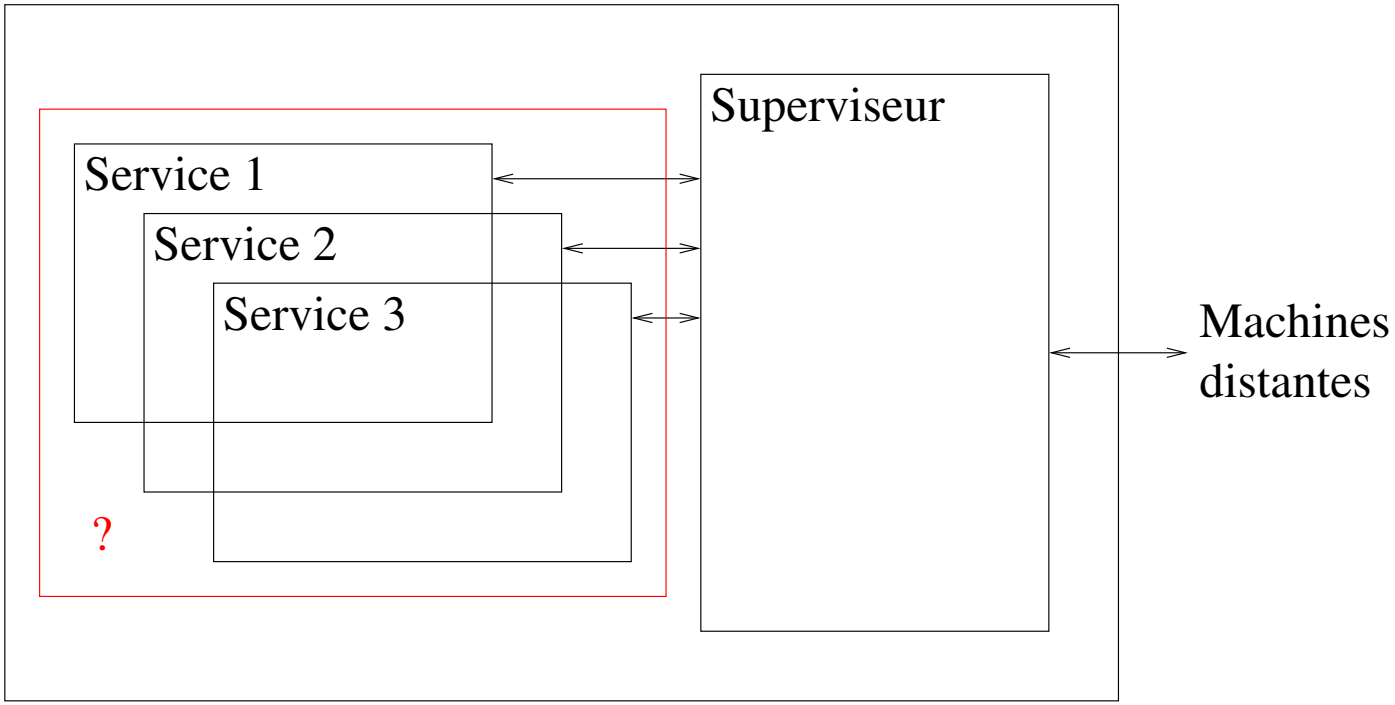


- services = importations/exportations
- eop = boîte noire qu'il faut mettre en place

Zoom


Superviseur
Service 1
Service 2
Service 3
Machines
distantes

Zoom


Superviseur
Service 1
Service 2
Service 3
?
Machines
distantes

L'article

L'article :

- rapport technique du projet Baltic
- plusieurs niveaux de lecture : de l'utilisateur au théoricien en passant par le développeur

Apports :

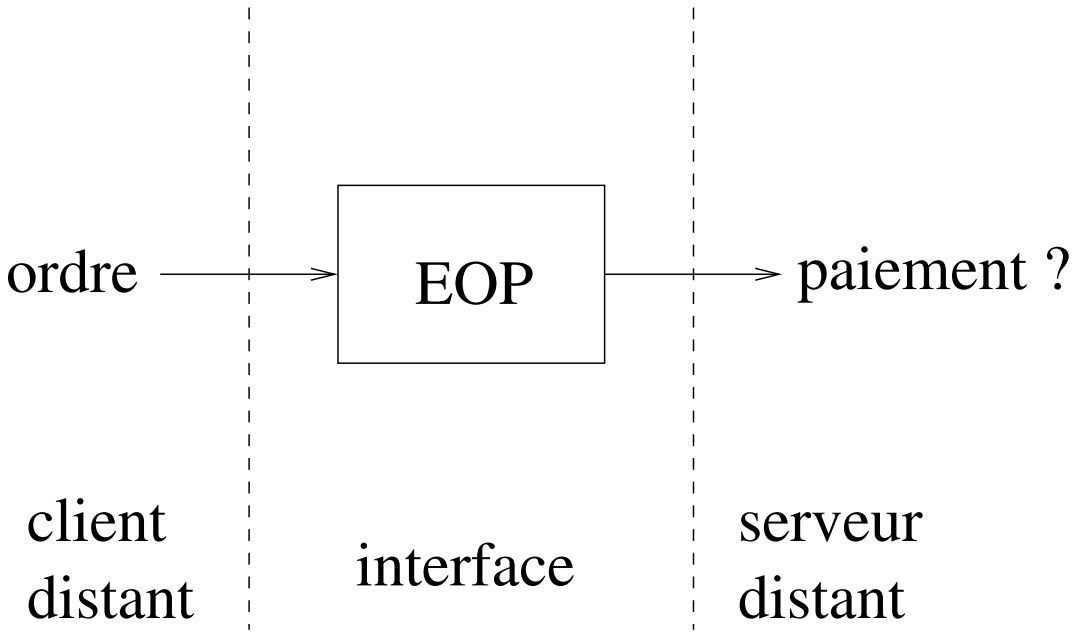
- interface de haut niveau pour gérer une ferme de services
- langage concurrent
- système de types garantissant la correction

Plan

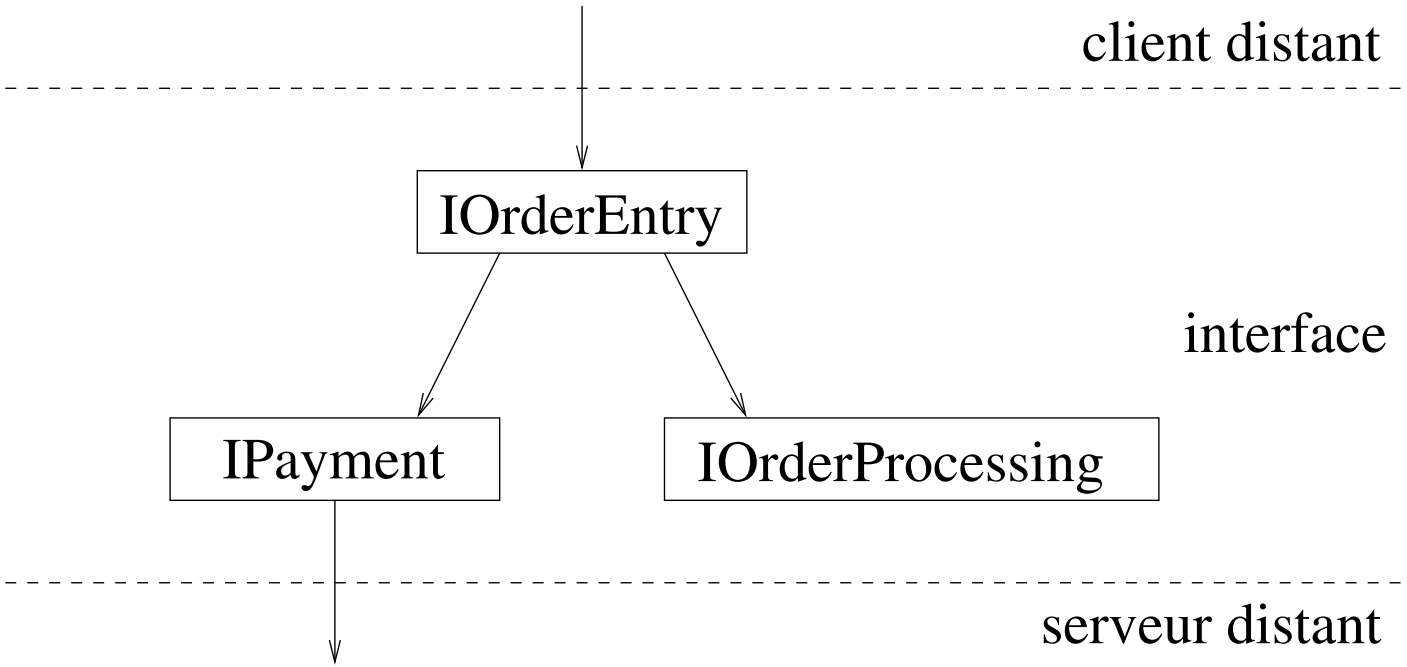
Première partie I

Baltic par l'exemple

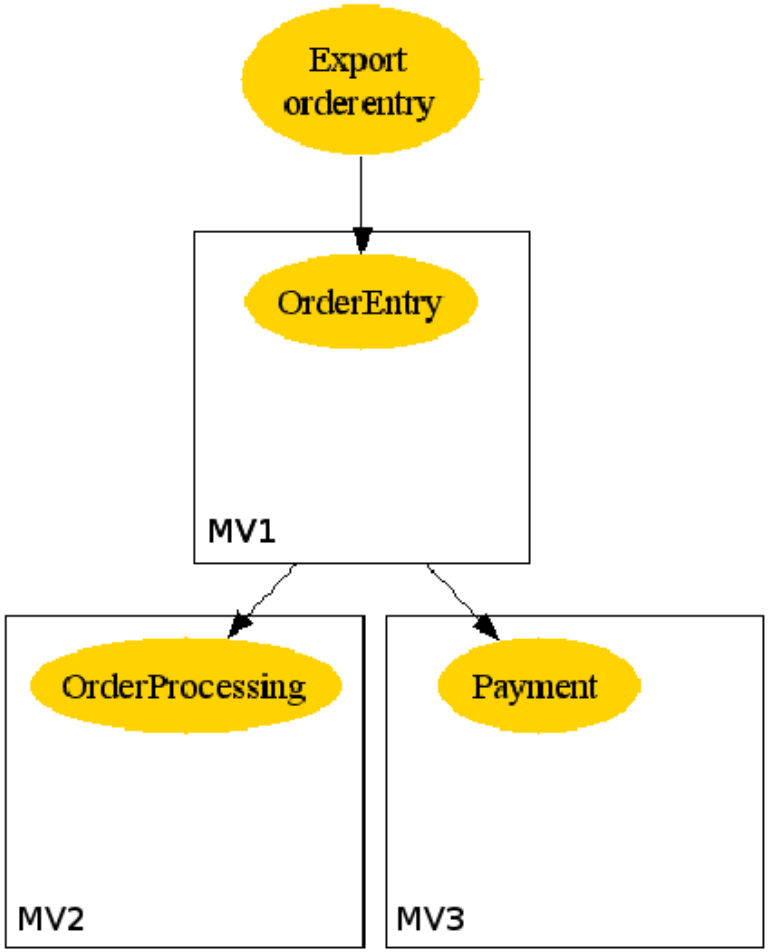
Point de départ



Découpage


client distant
IOrderEntry
interface
IPayment
IOrderProcessing
serveur distant

Ferme isolée



$MV3 \Rsh MV2 \Rsh MV1 \Rsh$ appel

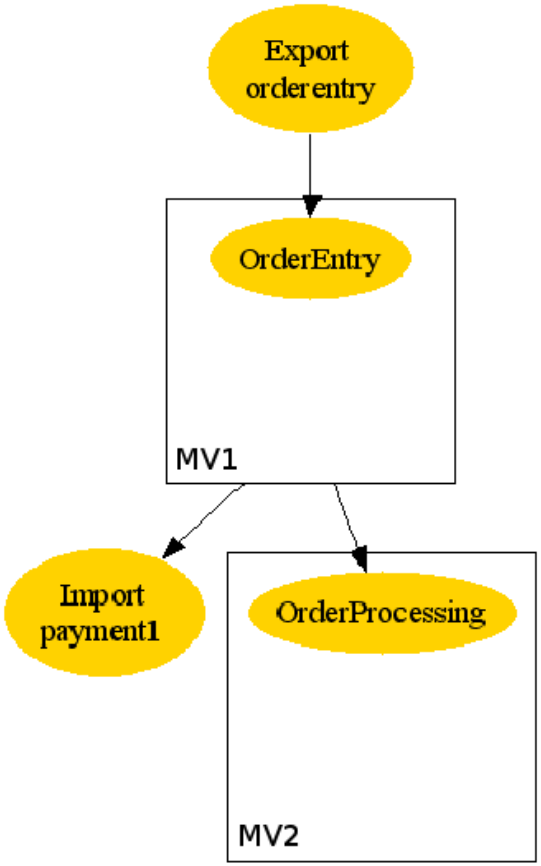
Détail

```
let callChan c m =
  let r = chan "réponse" in
  send c (m,r) ;
  recv r
```

```
let rec servChan c f :unit =
  let (m,r) = recv c in
  send r (f m) ;
  servChan c f
```

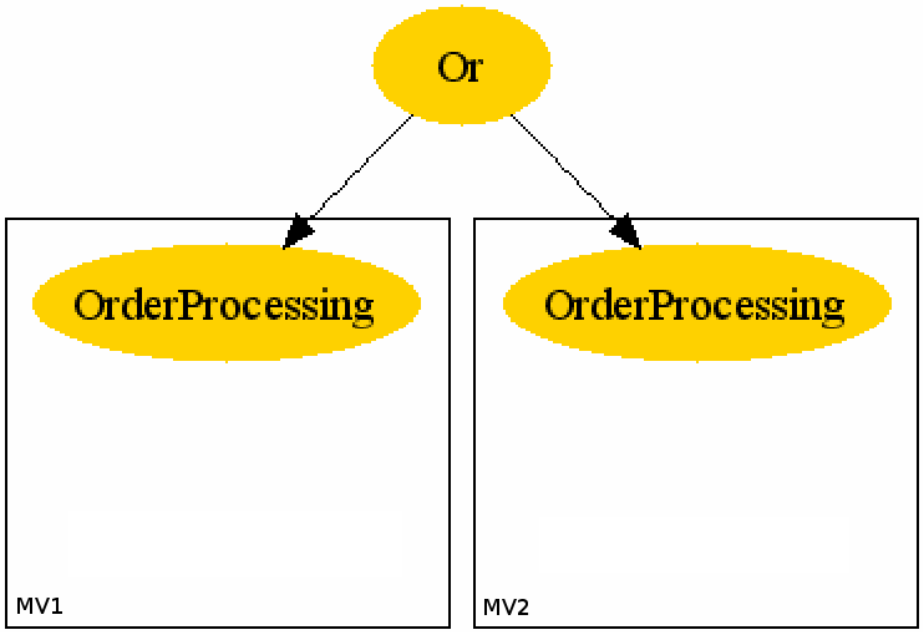
```
(new MV1, MV2, MV3)
(new payChan, procChan, orderChan)
  (vm3[servChan payChan paymentFunction] ↱
   vm2[servChan procChan processingFunction] ↱
   vm1[servChan orderChan
     (orderEntryFunction payChan procChan)] ↱
   let result = callChan orderChan o in ())
```

Serveur de paiement distant

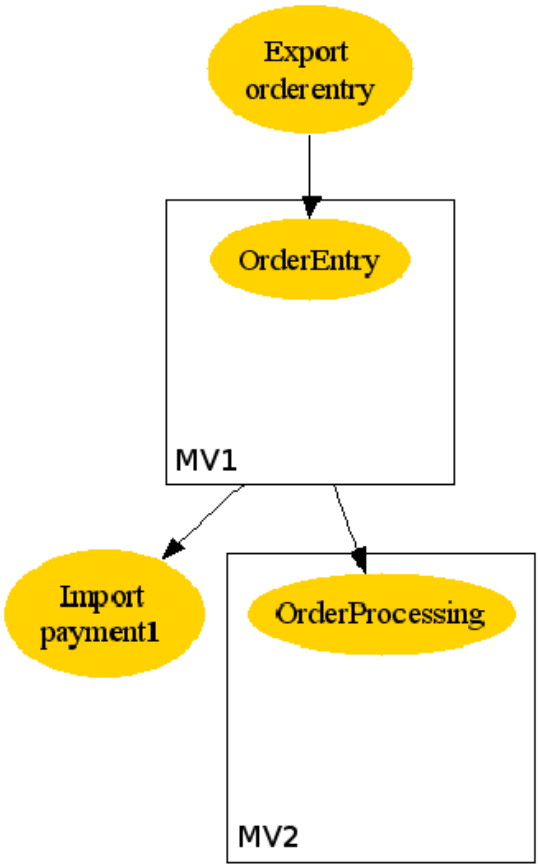


Constructeur Or

OrderPorcessing facilement surchargé $\Rightarrow$ on en a deux et lorsqu'on traite une donnée, on l'envoie sur un des deux.

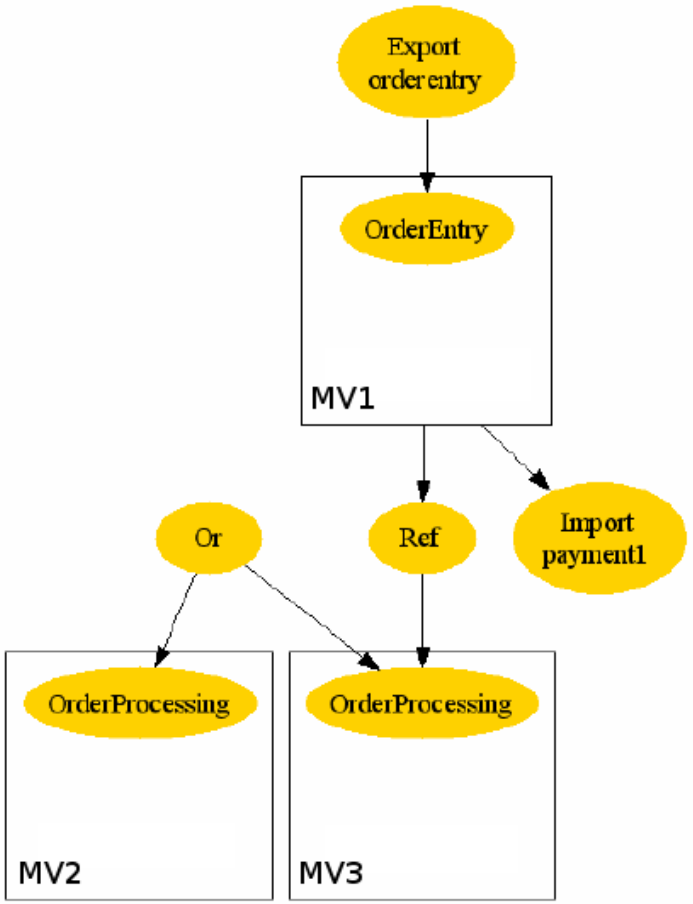


Constructeur Par



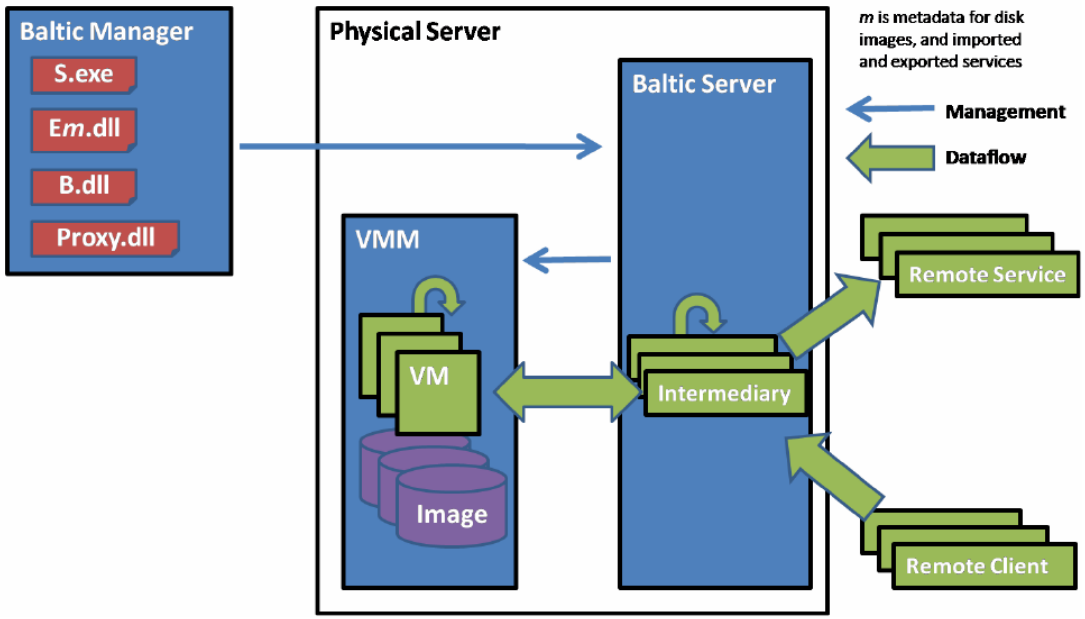
On a
deux serveurs distants dédiés au paiement.
On demande l'autorisation aux deux
et on attend une réponse de l'un des deux.

Constructeur Ref

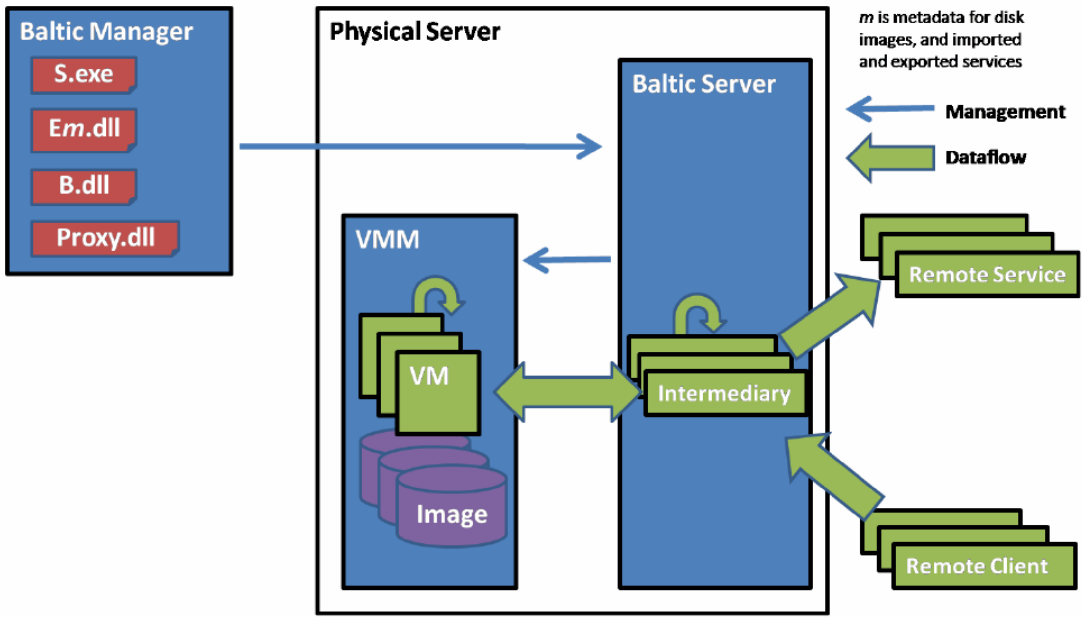


Permet de changer
la topologie en cours d'utilisation.

Schéma


Baltic Manager
S.exe
Em.dll
B.dll
Proxy.dll
Physical Server
Baltic Server
VMM
VM
Image
Intermediary
m is metadata for disk images, and imported and exported services
Management
Dataflow
Remote Service
Remote Client

Schéma



- machines virtuelles : serveurs
- machines distantes : contexte
- importations et exportations : communications

Deuxième partie II

Implantation en F#

Rappels

```
let callChan c m =
  let r = chan "réponse" in
  send c (m,r) ;
  recv r
```

```
let rec servChan c f :unit =
  let (m,r) = recv c in
  send r (f m) ;
  servChan c f
```

```
(new MV1, MV2, MV3)
(new payChan, procChan, orderChan)
  (vm3[servChan payChan paymentFunction] ↱
   vm2[servChan procChan processingFunction] ↱
   vm1[servChan orderChan
     (orderEntryFunction payChan procChan)] ↱
   let result = callChan orderChan o in ())
```

Fonctions F#

On se donne les fonctions :

Fonction	Type	Utilisation
`chan`	`string -> 'a chan`	création d'un canal
`send`	`'a chan -> 'a -> unit`	envoi sur un canal
`recv`	`'a chan -> 'a`	réception sur un canal
`fork`	`unit -> unit -> unit`	lancement d'un thread en parallèle
`call`	`('a,'b) proc -> 'a -> 'b`	procédure $\rightarrow$ fonction
`epFun`	`('a -> 'b) -> ('a,'b) proc`	fonction $\rightarrow$ procédure

Constructeur Par

Une fonction auxiliaire :

```
let par f1 f2 x =
```

Constructeur Par

Une fonction auxiliaire :

```
let par f1 f2 x =
  let c = chan "réponse" in
  fork (fun () -> send c (f1 x));
  fork (fun () -> send c (f2 x));
  recv c
```

Constructeur Par

Une fonction auxiliaire :

```
let par f1 f2 x =
  let c = chan "réponse" in
  fork (fun () -> send c (f1 x));
  fork (fun () -> send c (f2 x));
  recv c
```

La fonction principale :

```
let ePar ep1 ep2 =
```

Constructeur Par

Une fonction auxiliaire :

```
let par f1 f2 x =
  let c = chan "réponse" in
  fork (fun () -> send c (f1 x));
  fork (fun () -> send c (f2 x));
  recv c
```

La fonction principale :

```
let ePar ep1 ep2 =
  epFun (fun x -> par (call ep1) (call ep2) x)
```

Constructeur Or

Une fonction auxiliaire :

```
let pick x1 x2 =
```

Constructeur Or

Une fonction auxiliaire :

```
let pick x1 x2 =
  par (fun () -> x1) (fun () -> x2)
```

Constructeur Or

Une fonction auxiliaire :

```
let pick x1 x2 =
  par (fun () -> x1) (fun () -> x2)
```

La fonction principale :

```
let eOr ep1 ep2 =
```

Constructeur Or

Une fonction auxiliaire :

```
let pick x1 x2 =
  par (fun () -> x1) (fun () -> x2)
```

La fonction principale :

```
let eOr ep1 ep2 =
  epFun (fun x -> (pick ep1 ep2) x)
```

Continuons

On implante aussi :

- les références
- les machines virtuelles
- leurs rôles
- les exportations et importations
- ...

Troisième partie III

Le λ-calcul partitionné

Provenance

λ-calcul partitionné :

- *lambda*-calcul simplement typé
- $+$ concurrence (partitions)
- $+$ système de types pour la concurrence

Syntaxe et sémantique

Syntaxe :

- comme en *pi*-calcul, sauf ↱ pour la composition parallèle
- pas de somme

Sémantique :

- congruence structurelle
- règles de réduction

Typage

Deux sortes de types :

- des types du premier ordre, comme la chaîne de caractère
- des types pour désigner les canaux de communications

Règles de typage : règles habituelles.

Intérêts :

- assurer que passent sur les canaux des objets de bons types
- assurer que l'implantation correspond à la spécification

Conclusion

- virtualisation $\Rightarrow$ concurrence
- simulation d'un langage concurrent en ml
- création d'un modèle de langage très proche du λ-calcul

Remarques

- article très complet, original
- article précurseur
- la difficulté est que c'est un rapport technique
- manque un peu de généralité

Des questions ?