Lagab Yanis
Reffay Thomas
Vialatte Sylvain

M1 IMIS TER
2022

Programmation par contraintes pour un courtier Cloud orienté sécurité

Rapport de Travaux d'études et de recherche et Technique de communication

1/ Introduction 3
2/ Organisation 4
3/ Fonctionnement d’un courtier cloud 5
4/ Modélisation du courtier cloud de Salwa Souaf 6
5/ État de l’art 8
6/ Définition d’un modèle 10
7/ Implémentation 17
8/ Correction 20
9/ Tests 22
10/ Comparaison avec les résultats de Souaf 23
11/ Conclusion 25
12/ Bibliographie 26

1/ Introduction

L'objectif de ce sujet de TER est de proposer un courtier cloud codé en programmation par contrainte et qui dispose des mêmes caractéristiques que celui proposé par Salwa Souaf dans sa thèse "*Formal methods meet security in a cost aware cloud brokerage solution*".

Il nous a fallu dans un premier temps bien comprendre la solution proposée par Souaf dans sa thèse, et donc lire attentivement cette dernière pour en extraire les caractéristiques du courtier cloud.

Nous avons ensuite fait un état de l'art sur les méthodes de programmation par contraintes appliquées au cas des courtiers cloud.

Enfin, grâce à ces informations, nous avons pu nous intéresser à l'implémentation de notre courtier.

2/ Organisation

Dans un premier temps nous avons organisé des réunions entre nous et nos professeurs encadrants une fois toutes les deux semaines.
Puis étant donné que nous bloquions sur l'implémentation du modèle, nous n'avons plus trop donné de nouvelles et une seule réunion a été organisée depuis la soutenance de mi-parcours, à la demande de notre professeur encadrant.

De notre côté nous avons utilisé un dossier sur google drive pour stocker tous les fichiers et pour rédiger le rapport, le diaporama ainsi que toutes nos prises de notes. De cette façon nous pouvons travailler tous ensemble sur les fichiers et avons accès aux notes de nos partenaires.

Nous avons aussi créé un serveur discord pour échanger entre nous de façon à ce que, de la même façon, chacun puisse voir tous nos messages. Au besoin on peut s'en servir pour se transmettre des fichiers et surtout comme salon de discussion vocal.

Nous avons surtout utilisé le dépôt git pour gérer les différents codes. Il y a également les papiers que nous avons lus et le diapo de la soutenance de mi-parcours.

L'implémentation s'est essentiellement passée en pair programming afin de réduire le nombre de conflits git étant donné que les fichiers sont très courts. Nous avons aussi décidé de travailler en présentiel à l'université, cela nous a permis de nous entraider et de se concerter même si tout le monde ne travaillait pas sur le TER, de pouvoir utiliser un tableau si un problème persistait et qu'on avait besoin de tous se pencher dessus.

3/ Fonctionnement d’un courtier cloud

Un courtier cloud est une application qui agit comme un intermédiaire entre les vendeurs de services cloud (Amazon, Microsoft,...) et les clients qui souhaitent louer des ressources cloud.

Un client se présente avec des demandes (il a besoin de faire fonctionner certaines applications, qui ont des exigences fonctionnelles, et il -le client- peut avoir en plus des exigences non fonctionnelles, souvent liées à la sécurité) et le courtier lui présente une combinaison de vms qui permet de répondre à ses exigences.

Cela ressemble à un problème de placement vu sous cet angle, mais pour les clients, il est aussi important de réduire les coûts au maximum et donc cela devient un problème d’optimisation.

C’est un problème NP-difficile et plusieurs approches sont étudiées pour le résoudre dans un temps le plus court possible. Par exemple la programmation linéaire ou encore comme dans notre cas la programmation par contraintes.

Le courtier cloud présenté par Salwa SOUAF a pour mission de résoudre un problème de placement de VMs sur des serveurs physiques en respectant un certain nombre d’exigences, en particulier en prenant en compte des exigences de sécurité.

Le courtier fonctionne par étape. Dans un premier temps, les fournisseurs cloud lui attribuent des ressources qu’il pourra par la suite utiliser pour répondre aux attentes des clients.

Ensuite, quand un client effectue une demande, le courtier commence par vérifier la cohérence des exigences fonctionnelles (la quantité et la description des ressources) et des exigences non fonctionnelles (propriétés de sécurité). En effet, rien n’indique au préalable que la demande du client est réalisable, même en considérant qu’on a accès à toutes les vms imaginables, sa demande pouvant être incohérente.

Si les exigences du client ne sont pas cohérentes, alors le courtier prévient le client et lui présente un contre-exemple afin de mettre en évidence les éventuelles incohérences.

Une fois que les exigences du client sont cohérentes, le courtier s’applique à trouver une disposition valide et fait ensuite une proposition de déploiement au client. Le client peut confirmer la proposition, auquel cas, il est par la suite guidé dans le déploiement.

4/ Modélisation du courtier cloud de Salwa Souaf

Le courtier cloud de Salwa Souaf repose sur trois principales parties différentes, la relation entre les courtières et les fournisseurs de cloud, la relation entre le courtier et le client et enfin la recherche de solution pour le problème de placement en lui-même.

Mais avant d'expliquer ces 3 points, nous allons d'abord décrire les différentes modélisations.

Le courtier doit préalablement réserver auprès des fournisseurs d'accès cloud un certain nombre de produits, de façon à pouvoir par la suite les proposer aux clients. Pour représenter ces services des "VMs" sont utilisées. Concrètement c'est une liste de caractéristiques qui définissent le produit du courtier.
Cette liste se compose

- d'un OS (ex: Windows)
- d'un nombre de coeur virtuel pour le processeur (une mesure de la puissance de calcul) (ex: 8)
- d'une quantité de mémoire RAM (ex: 16 Go)
- d'une localisation géographique (ex: France)
- d'un coût (ex: 200€/an)
- d'un fournisseur (ex: Amazon)

Le courtier dispose ainsi d'une liste de produits qu'il peut utiliser pour satisfaire les demandes du client.

De son côté, le client modélise ses exigences par une liste de "composants". Un composant est lui aussi une liste de caractéristiques qui sont :

- d'un OS (ex: Windows)
- d'un nombre de coeur virtuel pour le processeur (une mesure de la puissance de calcul) (ex: 8)
- d'une quantité de mémoire RAM (ex: 16 Go)
- d'une localisation géographique (ex: France)

Le but du courtier va être d'attribuer à chaque composant une vm sur laquelle il pourra s'exécuter.
Le client peut également préciser des relations entre ses composants qui seront des contraintes supplémentaires pour le courtier:

➔ flow(x,y) qui indique que les deux composants x et y doivent pouvoir disposer d'un canal de communication

- ➔ alone(x) qui indique que le composant doit être le seul composant placé sur la vm sur laquelle il sera placé
- ➔ concurrence(x,y) qui indique que les deux composants x et y ne doivent pas disposer d'un canal de communication direct ou indirect

À l'aide de cette modélisation du problème et de règles de programmation linéaire, on peut résoudre le problème de placement.

Le courtier réserve à l'avance une certaine quantité de ressources et avec des caractéristiques différentes, de différents fournisseurs. L'approvisionnement pourrait se faire en tenant compte du nombre de clients et de ressources utilisées ou non utilisées, sur une période donnée. Une période appropriée sera sélectionnée et une estimation des ressources sera effectuée avant le début de chaque période. Nous supposerons que le courtier disposera toujours des ressources nécessaires pour répondre à la demande du client.

Le client spécifie sa demande au courtier en précisant les exigences fonctionnelles et non fonctionnelles. Puis le courtier vérifie la cohérence de la requête et si celle-ci ne contient pas des exigences contradictoires.

Puis, le courtier recherche un placement qui répond aux exigences fonctionnelles du client tout en réduisant au minimum le coût d'installation. Une fois que le client a approuvé la stratégie de placement retournée, le courtier s'occupe alors du déploiement et des configurations de réseau appropriées pour satisfaire les exigences de sécurité du client.

5/ État de l'art

Différents courtiers cloud ont été créés auparavant, certains utilisent la programmation linéaire, d'autres la programmation par contrainte. Nous avons étudié surtout 3 courtiers cloud :

- Le courtier cloud BtrPlace, nous nous sommes basés sur une publication datant de 2013 et écrite par Fabien Hermenier, Julia Lawall et Gilles Muller.
- Le courtier OptiPlace qui est une amélioration de BtrPlace et est cité dans une publication de 2017 écrite par Hélène Coullon, Guillaume Le Louet et Jean-Marc Menaud
- Les courtiers décrits dans la thèse de Salwa Souaf, ceux-ci sont assez similaires et le fonctionnement de la dernière version de ces courtiers a été décrite dans la partie d'avant (Partie 3).

Papier de 2013 : "BtrPlace: A Flexible Consolidation Manager for Highly Available Applications"

BtrPlace est un gestionnaire de consolidation flexible qui peut être configuré dynamiquement avec des contraintes de placement, il s'appuie sur la programmation par contrainte pour modéliser et résoudre les reconfigurations problem, le core reconfiguration problème c'est de calculer une configuration viable en choisit d'abord pour chaque VM un serveur d'hébergement qui répond aux besoins des VM, puis planifier les actions qui convertiront la configuration actuelle en la configuration choisie, Le calcul d'une solution pour la reconfiguration problème peut prendre du temps de consommation pour les grands data center donc l'optimisation du processus de résolution BtrPlace utilise 3 stratégies la simplification du full Reconfiguration Problem, heuristics guiding the CP solver pour de la reconfiguration plans rapide et le Partitionnement.

Papier de 2017:

Dans "*Virtual Machine Placement for Hybrid Cloud using Constraint Programming*", OptiPlace est utilisé et sert de base de travail. L'objectif étant de proposer des améliorations à OptiPlace dans le cas spécifique des clouds hybrides. C'est-à-dire les situations où un client souhaite disposer d'un cloud privé qui lui appartient et dans le même temps, louer des ressources auprès de cloud providers.

De base, OptiPlace a été conçu pour répondre à un problème de reconfiguration capable de gérer un seul fournisseur de cloud à la fois, interdisant la possibilité d'avoir une fédération ou une solution de cloud hybride.

Les contributions de ce papier :

- un modèle générique d'infrastructure pour pouvoir gérer les clouds privés et publics en même temps
- de nouvelles contraintes pour le contexte des clouds hybrides
- une évaluation des nouvelles contraintes proposées

Un ensemble d'*external nodes* est noté ε. Un external node est équivalent à un internal node sauf qu'il est disponible en quantité infinie. Il n'est pas non plus associé à des composants hardware. Le courtier pourra privilégier les nœuds externes ou internes en fonction des coûts ou des limites d'alimentation.

Il y a également *Site* : un *site* est un sous-ensemble de serveurs, chaque *node* n'est associé qu'à un seul site. Le concept d'un *site* est de pouvoir regrouper des serveurs ensemble en fonction de caractéristiques qu'ils partagent. Que cela soit matériel ou géographique. L'ensemble de tous les *site* est noté S.

Et enfin *Tag* : chaque *node* peut être associé à un ou plusieurs *tag*, ce qui crée des sous-ensembles. Très proche de *site*, il y a tout de même deux différences majeures : un *node* peut avoir plusieurs *tags* et les *tags* ne sont pas utilisés dans le CSP.

La *SkyPlaceView* propose ensuite de nouvelles contraintes qui utilisent les nouveaux concepts d'infrastructure cités au-dessus :

- *Onsite* : Une certaine machine doit appartenir à un certain site. Formellement pour une machine $v_j \in$ V et un site s $\in$ S, la propriété *OnSite*(S,v_j) signifie : $h_{ij} = 1 \Rightarrow n_i \in S$
- *OffSite*: Exactement l'inverse la machine v_j ne doit pas être hébergée sur le site s. Plus formellement pour une machine $v_j \in$ V et un site s $\in$ S, la propriété *OffSite*(S,v_j) signifie : $h_{ij} = 1 \Rightarrow n_i \notin S$
- *Near :* indique qu'un ensemble de machines virtuelles doivent être hébergées sur le même *site*
- *Far* : indique qu'un ensemble de machines virtuelles doivent toutes être hébergées sur différents *sites*

6/ Définition d’un modèle

La programmation par contrainte se base essentiellement sur la définition du problème et sur la modélisation de celui-ci. Nous avons dû réaliser un CSP (problème de satisfaction de contraintes), il est défini par des variables, leurs domaines et des contraintes.

Nous avons décidé de reprendre une version simplifiée du Courtier de Salwa Souaf en ajoutant quelques modifications. Comme le courtier de Salwa Souaf, les composants et les VMs ont une capacité de mémoire, un nombre de processeurs, un système d’exploitation. Les VMs ont en plus un coût de location et un fournisseur. Nous avons effectué 1 changement dans le modèle pour les spécifications, un composant à plusieurs localisations possibles ce qui permet au client de pouvoir placer ses composants où il veut.

L’on a aussi gardé les 3 relations entre les composants qui sont Alone, Flow et Concurrence, un composant avec la contrainte Alone sera le seul composant sur la VM qui l’héberge. La relation Flow permet à un composant de communiquer avec un autre, par cela est unilatéral, mais cela peut être bilatéral permettant aux 2 machines de communiquer dans les 2 sens. Chaque lien créé inclut un coût supplémentaire à l’installation si les composants ne sont pas hébergés sur la même VM et ce coût varie si les VMs sont du même fournisseur ou bien si les fournisseurs des VMs sont différents.

Et enfin concurrence contrairement à flow interdit un lien entre 2 Composants que ce soit dans un sens ou dans l’autre. Cette relation est transitive, c'est-à-dire que si entre un lien est créé entre les 2 composants qui n’est pas direct et passe à travers différentes VMs cela crée un modèle incohérent et il ne peut pas être résolu. Cette notion de transitivité n’a pas pu être implémentée dans notre courtier. Malgré un échange avec nos encadrants, nous ne comprenions pas comment réaliser une partie du problème qui était un prétraitement afin d’avoir tous les ensembles de VM qui communique entre elles et nous n’avons pas réussi à l’implémenter de matière satisfaisante dans les délais qu’il nous restait.

Notre Courtier Cloud va donc décider quelle VM va héberger quels composants tout en minimisant le coût. Le coût total est la somme du coût des liens et du coût de location. Sauf que pour que cela soit pertinent, nous avons dû rajouter une autre valeur, le nombre de mois de location voulu par le client , afin que le coût total puisse être calculé correctement, en ajustant l’impact du coût de location dans le total.

Avant tout, nous avons besoins de poser des données qui nous servent à construire les tableaux en minizinc :

- NComp : le nombre de composants
- NMachine : le nombre de machines
- NLocalisation : le nombre de localisations différentes disponibles
- NOs : le nombre d'Os différents disponibles
- NAlone : le nombre de composants marqués alone
- NFlow : Le nombre de flow(), une relation unilatérale binaire
- NConcurrence : le nombre de concurrence(), une relation unilatérale binaire
- NFourn : le nombre de providers différents
- NMoisLocation : le nombre de mois estimé de location

Avec ceci nous avons construit différents tableaux afin de représenter les données utiles qui fixent le "contexte" :

- Localisations le tableau des localisations : tableau de string de taille NLocalisation contenant les localisations acceptées
- OS le tableau des systèmes d'exploitations : tableau de string de taille NOs contenant les OS acceptés
- Fournisseurs : tableau de string de taille NFourn contenant la liste de tous les providers

Dans l'écriture de nos contraintes nous ne pouvons pas utiliser des tableaux de string pour représenter les os des composants ou leur localisation ou les fournisseurs, les os et la localisation des VMs. Nous avons donc transformer les tableaux des composants et des VMs en tableau d'entier, et ce qui permet de retrouver la string associée à un entier sont ces tableaux décrits plus hauts. Pour l'entier 1, la string associée sera celle à l'indice 1 du tableau de string correspondant.

Dans le but de modéliser les composants, l'on a construit un tableau pour chaque caractéristique technique :

- CompOs : tableau d'entiers de taille NComp*NOS pour les systèmes d'exploitation des composants. Pour chaque composant, il y a NOS entiers à 0 ou à 1. Cela permet d'indiquer quels OS sont acceptés pour ce composant et lesquels ne le sont pas.
- CompMem: tableau d'entier de taille NComp pour les capacités de mémoires des composants
- CompProc: tableau d'entier de taille NComp pour le nombre de coeur de processeurs des composants
- CompLoc: tableau d'entiers de taille NComp*NLocalisations pour les localisations des composants. Il fonctionne exactement sur le même modèle que le tableau des OS.

C’est l’indice qui fait le lien entre les différents tableaux et qui représente l’identifiant d’un composant de façon générale. Le composant 1 aura ainsi son tableau de 0 ou 1 d’os à l’indice 1 de CompOs, sa taille de mémoire à l’indice 1 de CompMem, etc.

Pour CompOs et CompLoc, un exemple :
si nous avons 2 OS “Windows” et “Linux” dans le tableau d’os dans cet ordre là, et que le composant 1 fonctionne sur linux mais pas sur windows CompOs[1] = [0,1] car à l’indice 1 de CompOs[1] se trouve le booléen pour Windows car Windows est à l’indice 1 du tableau des Os. A l’indice 2 de CompOs[1] se trouve le booléen pour Linux car Linux est à l’indice 2 du tableau des os.

Il y a aussi des relations définies sur les composants comme nous en avons déjà parlé plus haut, voici comment elles sont représentées en tant que données :

- CompAlone : tableau d’entier de taille NAlone, il contient l’indice de tous les composants qui doivent être les seules de la configuration sur leur VM.
- CompFlow : tableau à deux dimensions de taille NFlow*2, il contient toutes les relations de flow() exprimées sous la forme d’un couple d’indice de composant. CompFlow[1] = [4,8] signifie qu’il faut qu’il y ait un canal de communication de 4 à 8. La relation est unilatérale, pour avoir de 4 à 8 il faut qu’il y ait un autre couple [8,4] dans CompFlow
- CompConcurrence : tableau à deux dimensions de taille NConcurrence*2, il contient toutes les relations de concurrence modélisées de la même façon que les flow, à la différence près que cette relation est bilatérale

Pour les VMs on a aussi un tableau de taille NMachine par caractéristique :

- VmOs : tableau d’entiers pour les systèmes d’exploitation des machines
- VmMem: tableau d’entiers pour les capacités de mémoires des machines
- VmProc : tableau d’entiers pour le nombre de processeurs des machines
- VmLoc : tableau d’entiers pour la localisation des machines
- VmCout : tableau d’entiers pour le coût de location des machines
- VmFourn : tableau d’entiers pour le fournisseur des machines

Pour VmOs, VmLoc et VmFourn ce sont des tableaux d’entiers simples car chaque VM réservée par le broker ne se trouve qu’à un seul endroit, n’a qu’un seul OS et n’a qu’un seul fournisseur. Ce qui associe l’entier à la string correspondante est encore une fois le fait que l’entier est l’indice de la string dans le tableau les rassemblant toutes.

Il y a une autre donnée un peu particulière : VMPrixLien. C’est un tableau NFourn*NFourn et il permet d’indiquer le coût de la création d’un lien entre chaque couple de fournisseurs, y compris le coût entre un fournisseur et lui-même. Nous nous en servons pour le calcul du coût total et il devra être rempli au préalable en même temps que s’effectue la réservation des machines.

En programmation par contraintes nous avons des données, c'est ce que nous avons décrit jusqu'ici. Il s'agit ensuite de décrire des variables. Ce sont les objets pour lesquels le solveur va chercher une affectation qui satisfasse les contraintes compte tenu des données :

- Placement : c'est un tableau de variables entières de taille NComp, à l'indice i de ce tableau se trouve stocké l'indice de la machine v sur laquelle il sera hébergé.
- NbCompParMachine : c'est un tableau de variables entières de taille NMachines, il représente à l'indice i le nombre de composants stockées sur la VM i.
- verificationModele : c'est un tableau de variables entières de taille NConcurrence de booléen. verificationModele[i] vaut 0 si la concurrence i est respectée, c-à-d qu'il n'y a pas de flow() entre les deux composants de la concurrence i.
- coutLien : une variable entière, elle sert à faire la somme du coût des flow. Le coût d'un flow dépendant des providers impliqués dans le lien. Les providers impliqués dépendent des providers des machines sur lesquelles sont hébergés les composants du flow.

On peut maintenant construire des contraintes afin de résoudre notre CSP. On a défini 9 contraintes et une fonction à minimiser dans le but de placer correctement les composants sur les VM disponibles. En-dessous la version Minizinc de ces contraintes.

La localisation de chaque composant doit correspondre à celui de la vm sur laquelle il est hébergé.

$$\forall i \; allant \; de \; 1 \; à \; NComp, \; Comp_{Loc}[i][VM_{Loc}[X[i]]] = 1$$

```
constraint forall(c in 1..NComp)(CompLoc[c,VMLoc[Placement[c]]] = 1);
```

La seconde est sensiblement la même que la première, mais pour l'OS.

L'os de chaque composant doit correspondre à celui de la vm sur laquelle il est hébergé.

$$\forall i \; allant \; de \; 1 \; à \; NComp, \; Comp_{OS}[i][VM_{OS}[X[i]]] = 1$$

```
constraint forall(c in 1..NComp)(CompOs[c,VMOs[Placement[c]]] = 1);
```

La troisième sert à faire en sorte que le nombre de processeurs d'une machine est supérieur ou égale à la somme des processeurs des composants hébergée sur cette machine. Même chose pour la quatrième, mais avec la capacité de mémoire.

La somme du nombre de cœurs de processeurs requise par les composants hébergés sur une machine v ne doit pas excéder le nombre de cœurs de processeur de la machine v.

$$\forall v \; allant \; de \; 1 \; à \; NMachine, \; \sum_{i=1}^{NComp} ((X[i] = v) * Comp_{PROC}[i]) \leq VM_{PROC}[v]$$

```
constraint forall(v in 1..NMachine)(sum( [(Placement[c]==v)*CompProc[c] | c in
1..NComp])<=VMProc[v]);
```

La somme de la mémoire requise par les composants hébergés sur une machine v ne doit pas excéder la mémoire disponible sur la machine v.

$$\forall v \; allant \; de \; 1 \; à \; NMachine, \; \sum_{i=1}^{NComp} ((X[i] = v) * Comp_{MEM}[i]) \leq VM_{MEM}[v]$$

```
constraint forall(v in 1..NMachine)(sum( [(Placement[c]==v)*CompMem[c] | c in
1..NComp])<=VMMem[v]);
```

Une contrainte fixe les valeurs de NbCompParMachine en fonction des valeurs affectées à Placement :

$$\forall v \; de \; 1 \; à \; NMachine, \; NbCompParMachine[v] = \sum_{i=1}^{NComp} (Placement[i] = v)$$

```
constraint forall(v in 1..NMachine)( count(Placement, v, NbCompParMachine[v]) );
```

L'ensemble $Comp_{Alone}$ est un ensemble qui rassemble des composants qui doivent être les seuls de la configuration à être hébergés sur leur machine.

```
constraint forall(c in 1..NAlone) ( NbCompParMachine[Placement[CompAlone[c]]] = 1);
```

$$Comp_{Alone} : \forall i, \; i \in Comp_{Alone}, \; \forall j, \; j \in [1, NComp], \; i \neq j \Rightarrow X[i] \neq X[j]$$

Nous vérifions la cohérence des liens et de la proposition du client grâce à deux contraintes pour plus de lisibilité. La première définit verificationModele, un tableau de booléen qui indique pour chaque concurrence s'il existe un flow entre les composants de la concurrence (0 pour non, 1 pour oui) :

$\forall i, j \; in \; NComp, \forall c \; in \; NConcurrence \; CompConcurrence[c] \; = \; (i, j) \Rightarrow verificationModele[c] \; = \; flow(i, j) \vee flow(j, i)$

```
constraint forall(conc in 1.. NConcurrence,flow in 1..NFlow)
(if (CompConcurrence[conc,1]==CompFlow[flow,1] /\
     CompConcurrence[conc,2]==CompFlow[flow,2])
     \/
     (CompConcurrence[conc,1]==CompFlow[flow,2] /\
     CompConcurrence[conc,2]==CompFlow[flow,1])
then verificationModele[conc] = 1
else verificationModele[conc] = 0
endif);
```

La seconde contrainte, beaucoup plus simple, impose à verificationModele d'être à 0. De cette façon on impose le fait que toutes les concurrence soient respectées dans une solution trouvée par le solveur.

$$\sum_{c=1}^{NConcurrence} verificationModele[c] \; = \; 0$$

```
constraint sum(verificationModele)==0;
```

Une contrainte sur la variable coutLien calcule le coût total des liens, En regardant les fournisseurs des VMs qui hébergent les composants x et y et s'en servant comme indice pour le tableau VMPrixLien qui stocke les coûts de lien entre les VMs des fournisseurs. On a juste alors à faire la somme de ces valeurs lorsqu'il y a un lien à créer entre x et y (testé grâce à flow (x,y)).

$$coutLien \; = \; \sum_{x,y=0}^{NComp} (flow(x, y) \; * \; VMPrixLien[VM_{Fourn}[X[x]]][VM_{Fourn}[X[y]]])$$

```
constraint coutLien = sum( [(Placement[CompFlow[x,1]] != Placement[CompFlow[x,2]])*
VMPrixLien[VMFourn[Placement[CompFlow[x,1]]],VMFourn[Placement[CompFlow[x,2]]]] | x in
1..NFlow]);
```

Il y a également une fonction de coût que l'on souhaite minimiser. Elle est définie comme suit:

Le coût total de la configuration correspond à la somme du coût de location de toutes les machines v utilisées pour héberger des composants multipliée par le nombre de mois + le coût d'installation des liens.

$$min(\sum_{v=1}^{m}((NbCompParMachine[v] \geq 0) * VM_{Cout}[V]) * NMoisLocation + coutLien)$$

```
solve minimize (NMoisLocation*sum([ (NbCompParMachine[v]>0)*VMCout[v]| v in 1..NMachine]))
+coutLien;
```

7/ Implémentation

Pour la partie implémentation nous avons tout d'abord décidé de commencer par faire un jeu de données afin de pouvoir tester notre futur programme et de mieux comprendre pourquoi le modèle ne fonctionnerait pas. l'architecture et le même que le modèle présenté dans la partie modélisation.

```
%Rapport localisation et os / entier
Localisations = {"Suisse","France","Etats-Unis","Italie"};
OS = {"Windows","Linux"};

%Spec Composant
%Array des OS array[1..NComp] of string
CompOs = [1,1,1,2,2];
%Array des memoires array[1..Ncomp] of int
CompMem = [4,8,4,4,8];
%Array des processeurs array[1..Ncomp] of int
CompProc = [2,2,2,4,2];
%Array des localisations array[1..NComp] of string
CompLoc = [2,2,2,1,1];
```

Jeu de données MiniZinc : Représentation des composants

Par exemple ici nous avons un jeu de données avec 5 composants à placer et 10 machines disponibles avec 4 localisations différentes et 2 systèmes d'exploitation différents. Ci-dessus vous pouvez voir les caractéristiques des différents composants disponibles. Pour les tableaux CompOS et CompLoc, les valeurs correspondent à un indice des tableaux OS et Localisation. Le Composant 1 est représenté par toutes les valeurs d'indice 0, le composant 2 par les valeurs d'indice 2,..

```
%Spec Machine
%Array des OS array[1..NMachine] of string
VMOs = [2,1,1,1,2,2,1,1,2,2];
%Array des memoires array[1..NMachine] of int
VMMem = [16,8,8,8,8,8,8,8,16,32];
%Array des processeurs array[1..NMachine] of int
VMProc = [8,4,4,4,8,2,2,8,8,4];
%Array des localisations array[1..NMachine] of string
VMLoc = [1,2,2,2,1,1,3,3,3,4];
%Array dess cout array[1..NMachine] of int
VMCout = [100,50,60,30,40,78,32,40,100,120];
%Array des fournisseurs array[1..NMachine] of string
VMFourn = ["Amazon","Microsoft","Amazon","Microsoft","Amazon","Microsoft","Amazon","Microsoft","Amazon","Microsoft"];
```

Jeu de données MiniZinc : Représentation des machines

Pour les machines cela fonctionne de la même manière que les composants. Le résultat qui est attendu sur ce jeu de données est que les composants 1, 2 et 3 soient répartis sur les composants 2, 3 et 4 (qu'importe qui est hébergé ou), les composant 4 et 5 doivent, quant à eux, être hébergés sur la machine 1.

Nous avons essayé deux implémentations différentes, une en minizinc et l'autre en python en utilisant la librairie cpmpy.

Nous avons commencé sur conseil de notre enseignante encadrant par essayer d'implémenter notre modèle en python à l'aide de la librairie cpmpy. Elle permet de faire de la programmation par contraintes sur python en utilisant notamment le calcul vectoriel pour raccourcir et éclaircir la syntaxe.

Malheureusement, nous avons bloqué sur des problèmes de syntaxe et d'implémentation.

Notre blocage nous a amené à essayer d'implémenter notre modèle sur minizinc. En effet, nous nous sommes déjà servis de minizinc pour notre cours sur la programmation par contraintes, donc nous disposons d'exemples et de cours et surtout d'un peu plus d'expérience que sur la librairie cpmpy.

Mais nous n'avons pas réussi dans un premier temps à exprimer toutes les contraintes sans erreur. La contrainte de count en particulier nous a posé des difficultés. En effet aucune des signatures de la fonction en minizinc ne correspondait à l'usage que nous cherchons à en faire apparement.

```
/home/etud/o2173830/Documents/TER/ter-broker-cloud-et-pcc/Modele_MiniZinc/modele1.mzn:
46.42-51:
MiniZinc: type error: no function or predicate with this signature found:
`count(array[int] of var int,int)'
Process finished with non-zero exit code 1
Finished in 53msec
Compiling modele1.mzn
/home/etud/o2173830/Documents/TER/ter-broker-cloud-et-pcc/Modele_MiniZinc/modele1.mzn:
46.37-52:
MiniZinc: type error: no function or predicate with this signature found:
`count(array[int] of var int,int,var int)'
Process finished with non-zero exit code 1
Finished in 52msec
```

Rapport d'erreur de MiniZinc

En réalité, l'erreur venait des ordinateurs sur lesquels nous étions en train de coder, même si ça reste surprenant. Une fois que nous sommes passés sur nos ordinateurs personnels avec des versions fraîchement installés de Minizinc, nous n'avons plus eu de gros problèmes d'implémentation.

Nous avons dû faire une petite modification sur les données de Localisation et d'OS. En effet, en minizinc, on ne peut pas utiliser une variable pour accéder à un certain indice des tableaux de chaînes de caractère.

Nous avons une liste de toutes les localisations possibles et une liste de tous les OS possibles et de tous les fournisseurs dont nous nous servons pour avoir une

équivalence entre chaque string et un entier (l'indice de la string dans le tableau correspondant).

Ensuite pour CompLoc et CompOs et VMLoc et VMOs au lieu d'être des tableaux de chaînes de caractère, ce sont des tableaux d'entiers, l'entier étant l'indice de la localisation (ou OS ou Fournisseurs pour les VMs) dans le tableau des localisations (ou OS ou Fournisseurs pour les VMs).

De cette façon si CompOs[i] = VMOs[i] alors ils correspondront automatiquement à la même localisation réelle puisque l'indice qu'ils stockent est identique.

8/ Correction

Après un débogage rigoureux et organisé, nous sommes arrivés à une implémentation qui fonctionne et nous renvoie le bon résultat en très exactement 0 changements. Mais alors d'où venait l'erreur qui nous avait bloqués si longtemps ?

Il a fallu un test de plus pour le comprendre : elle venait des pc de la fac (ceux de la salle E09 exactement) que nous utilisions pour avancer sur Minizinc. En effet sur ces machines (qui disposent de la version 2.2.1 de Minizinc), impossible de faire fonctionner notre modèle, la fonction count() n'étant pas valide d'après l'erreur renvoyée. Sur nos pc persos (sur lesquels minizinc a été récemment installé et donc correspond à la dernière version disponible : 2.6.3), le modèle tourne parfaitement sans erreur et en renvoyant le bon résultat.

Une fois cette erreur de count() derrière nous, nous avons enfin pu nous pencher sur des tests mais aussi l'implémentation de toutes les contraintes et également certaines améliorations auxquelles nous avions pensé entre-temps.

Pour ces améliorations, dans un premier temps nous avons changé le fonctionnement de CompOs et de CompLoc. En effet, dans notre première version, nous obligions chaque composant du client à avoir une et une seule localisation, ainsi qu'un seul OS. En y réfléchissant, ça ne paraissait pas réaliste du tout, un client pourrait très bien indiquer que son composant peut avoir n'importe quelle localisation, et le forcer à en choisir une fausse complètement la solution, car il n'a probablement pas choisi la localisation la plus rentable.

Nous avons donc transformé CompOs et CompLoc en deux tableaux à deux dimensions. La première dimension est comme d'habitude l'indice du composant et la seconde est l'indice de l'Os/la localisation dans la liste des Os/localisations tandis que la valeur est 0 ou 1.

Par exemple avec :

OS = [“windows”, “linux”];

CompOs[3][2]=1 signifie que pour le composant 3, l'OS linux est accepté (en minizinc on indexe les tableaux à partir de 1).

A l'inverse CompOs[3][1] = 0 signifie que pour le composant 3, l'OS windows n'est pas accepté.

De cette façon, le client peut définir une liste d'OS et de localisations valides pour chaque composant.

La seconde amélioration que nous avons ajoutée est une prise en compte des flow dans le coût total. En effet, en ré-examinant la thèse de Souaf, nous avons constaté que dans le coût était pris en compte la création de canaux de communication entre les machines. De plus, dans un souci de réalisme, les coûts varient selon les fournisseurs impliqués.

Nous avons estimé que c'était quelque chose d'important à implémenter de notre côté également car cela peut jouer sur la configuration la moins chère.

Nous avons donc une liste des Fournisseurs, chaque VM est associé à l'un d'eux, et nous avons une table (de taille NbFournisseurs*NbFournisseurs) qui indique pour chaque couple de fournisseurs le prix de l'installation du canal de communication. Cette table doit être remplie par les fournisseurs ou alors déterminée à partir d'une étude des prix moyens.

Nous avons aussi besoin des flow entre les composants. La contrainte a été ajoutée au modèle et les données sont sous la forme d'un tableau de couples, où chaque couple représente deux composants qui doivent être reliés par un canal de communication.

Cependant, les coûts pour les VMs dans notre modèle sont des coûts/mois de location. Or il nous semble que les coûts de ce genre de canaux de communication ne se posent qu'à l'installation. Ce n'est donc pas la même unité.

Nous avons décidé de demander une information supplémentaire au client : une estimation de la durée de la location de ses VMs. Le coût total calculé comprend ainsi le coût/mois de l'ensemble des VMs multiplié par le nombre de mois estimé + le coût d'installation des liens.

9/ Tests

Nous avons créé plusieurs jeux de données de taille variable. A chaque fois avec une technique similaire : on définit un certain nombre de types de VM. Par type de VM il faut comprendre une sorte de modèle de VM. Chaque type est différent de tous les autres types au moins sur une des caractéristiques des VM (os, mémoire, localisation, fournisseur, coût ou nombre de cœurs de processeur). Puis on définit le nombre d'exemplaires de chaque type de VM. Nous nous sommes inspirés de ce qu'a fait Salwa Souaf dans sa thèse pour construire les jeux de données.

Nous nous sommes servis de ces jeux pour tester nos modèles et les différentes évolutions. Sur des jeux de petite taille (5 composants, 11 types et 8 exemplaires de chaque type), cela prend déjà plusieurs secondes au solveur pour explorer tout l'espace de recherche.

Sur de plus grands jeux de données (7 composants, 130 types de VM et 20 exemplaires de chaque type), explorer l'intégralité de l'espace de recherche prend un temps tellement long que nous n'avons jamais pu l'atteindre, et c'est un minimum de plusieurs heures. On ne peut donc pas s'assurer d'avoir trouvé la solution optimale en un temps acceptable.

Cependant, le solveur trouve relativement rapidement (quelques minutes au maximum et quelques secondes la plupart du temps), une solution qui semble très proche de la solution optimale, si elle ne l'est pas.

Pour obtenir des valeurs plus fiables, nous avons effectué à chaque fois plusieurs tests (10) et avons fait la moyenne des résultats. Mais la variance des résultat était très faible

10/ Comparaison avec les résultats de Souaf

Nous n'avons pas pu utiliser exactement les mêmes jeux que Souaf dans sa thèse car on ne prend pas en compte les mêmes informations et on ne les prend pas toujours en compte de la même façon non plus.
Par exemple, nous prenons en compte un prix différent de connexion entre chaque provider au lieu de simplement un prix de connexion intra provider et un prix de connexion extra provider.
Nos localisations possibles sont plus nombreuses et sont sous la forme d'une liste.
Notre gestion de flow et concurrence n'est pas tout à fait la même non plus.
Nous avons aussi plus d'os possibles.

Mais nous avons reproduit un jeu de données d'ordre similaire au sien, avec les quelques modifications dont nous avions besoin. C'est-à-dire un jeu avec 161 types de VMs et 4 fournisseurs. En plusieurs versions selon le nombre d'exemplaires de chaque type d'instance.

Nous avons également reproduit exactement les scénarios du côté client (en faisant encore une fois les modifications obligatoires). Nous estimons que nos tests sont très proches de ceux de Salwa et que nos résultats sont comparables dans une certaine mesure.

Scénario Client	Nombre de Composants	Nombre de relations	Nombre de fournisseurs	Nombre de types de VM	Nombre de VM de chaque type	Temps pour résoudre (Thèse Salwa Souaf)	Temps pour résoudre programmation par contrainte (limite de temps 10 min)*
N°1	4	4	4	161	20	30,2s	3,5641 s
					10	1,9s	151,5 s
					5	0,5s	13,4646s
N°2	5	5	4	161	20	378,4s	20,037s
					10	62,3s	3,4529s
					5	2s	477,6s
N°3	5	6	4	161	10	5,6s	3,06s
					5	0,7s	448,33s
N°4	6	8	4	161	10	39,2s	143,33s
					5	5,5s	441,67s
N°5	7	9	4	161	10	253s	3,468s
					5	5,6s	88,67s
N°6	7	10	4	161	10	40,6s	3,316s
					5	3,8s	65s
N°7	8	11	4	161	10	764,2s	427,33s
					5	6,8s	107,33s
N°8	8	12	4	161	10	222,1s	427s
					5	6,3s	114,75s

Tableau de comparaison entre les résultat obtenu et ceux de la thèse de Salwa Souaf

*Comme dit plus haut, le temps pour explorer l'espace de recherche complet est immense et nous avons donc fixé une limite de temps de recherche. Le temps enregistré est le temps requis pour la meilleure solution trouvée dans cette intervalle de 10 minutes.

11/ Conclusion

Au départ, nous étions sur une bonne lancée, nous avons commencé l'état de l'art dès le début du semestre, en nous répartissant les tâches.

Mais au cours du mois de février nous n'avons pas bien compris ce qu'on attendait de nous à cette étape du travail. Nous n'avons pas été productifs et n'avons pas beaucoup avancé malgré des réunions régulières.

Ensuite, nous avons défini ensemble au cours d'une réunion un modèle à implémenter.

L'implémentation ne s'est pas bien déroulée, nous avons beaucoup buté sur certaines erreurs et nous aurions dû demander plus d'aide ou au moins donner plus de nouvelles sur l'avancement du travail.

Une fois les problèmes résolus (ou contournés), nous avons pu avancer plus efficacement, même s'il ne restait que la semaine supplémentaire qui nous a été accordée. Nous sommes arrivés à un broker dont les performances ne sont pas très intéressantes mais qui réponds au problème correctement.

Il y a tout de même des choses qui manquent comme par exemple, une interface pour le client.

En effet, nous trouvons que le format du fichier .dzn n'est vraiment pas user friendly. De plus les informations du broker cloud, sont incluses dans le .dzn alors que le client n'a absolument pas à renseigner ces informations.

C'est pourquoi nous avions en tête qu'il faudrait développer une application qui permette à l'utilisateur de rentrer ces informations, cette application ayant connaissance des données du broker cloud (liste des vms, prix, etc…), elle pourrait alors combiner les deux pour créer le fichier de données dont le solveur a besoin.

Il faudrait également prendre en compte les différents types de service. Actuellement on cherche juste à placer des vms sur des composants mais dans la réalité il y a énormément de services différents avec des caractéristiques différentes. Notre modèle se rapproche surtout du cas où on recherche de la puissance de calcul.

L'expérience qu'on retire de ce TER, est que nous avons manqué de communication même parfois entre membres du groupe. Cela a compliqué le débogage et ralenti la résolution des problèmes.

12/ Bibliographie

BtrPlace: A Flexible Consolidation Manager for Highly Available Applications, 2013, Fabien Hermenier, Julia Lawall, and Gilles Muller

Virtual Machine Placement for Hybrid Cloud using Constraint Programming, 2017, Hélène Coullon, Guillaume Le Louet, Jean-Marc Menaud

A Cloud Brokerage Solution: Formal Methods Meet Security in Cloud Federations, 2018, Salwa Souaf, Pascal Berthomé, Frédéric Loulergue