

$$\lim_{k \rightarrow \infty} R^k = R^*$$

Pour montrer que $\rho(\lambda \times \tilde{\Pi}) < 1$ nous allons nous appuyer sur le théorème de Gerschgorin [182] permettant de borner a priori les valeurs propres d'une matrice carrée. Pour montrer que $\rho(\lambda \times \tilde{\Pi}) < 1$ pour $\lambda \in (0, 1)$, il suffit de montrer que $\rho(\tilde{\Pi}) \leq 1$. En s'appuyant sur le théorème de Gerschgorin, il suffit donc de montrer que la somme des éléments sur chaque colonne c de la matrice $\tilde{\Pi}$ est inférieure ou égale à 1. Considérons la colonne $c = (k-1) \times n + i$ qui représente les poids des arcs du nœud n_{ki} vers tous les nœuds n_{jk} . La somme des poids de cette colonne est :

$$\sum_{s_j \in \text{Out}(t, s_k)} \omega_{jki}(\tau) \leq 1,$$

d'après la définition de la contribution indirecte (voir définition 4). Par conséquent le rayon spectral $\rho(\tilde{\Pi}) < 1$ et donc la suite $\{R^k(\tau)\}_{k \in \mathbb{N}}$ converge vers $R^*(\tau)$.

3.3.3 Vitesse de convergence

Nous étudions dans cette section la *vitesse de convergence* du calcul d'importance par les itérations de Jacobi. Nous allons pour cela estimer l'itération k à partir de laquelle l'erreur relative de l'estimation de l'importance $\frac{\|R^{k+1} - R^k\|_1}{\|R^{k+1}\|_1}$ devient inférieure à ε . Similairement au calcul de convergence de PAGERANK montré par [31], nous allons montrer que la vitesse de convergence du calcul d'importance est indépendante de la taille du graphe et qu'elle dépend des valeurs de λ et de la précision choisie, ε . Ce calcul est aussi confirmé expérimentalement (voir la figure 4.8 dans la section 4.3 du chapitre 4). Comme le vecteur initial R^0 n'influence pas l'importance des services, nous négligeons ce terme et nous calculons l'erreur relative au pas $k+1$ comme étant :

$$\begin{aligned} \frac{\|R^{k+1} - R^k\|_1}{\|R^{k+1}\|_1} &= \frac{\|\sum_{j=0}^k (\lambda \times \Pi)^{k-j} \times (\lambda \times Q) - \sum_{j=0}^{k-1} (\lambda \times \Pi)^{k-j-1} \times (\lambda \times Q)\|_1}{\|R^{k+1}\|_1} \\ &= \frac{\|\lambda^k \times \Pi^k \times (\lambda \times Q)\|_1}{\|R^{k+1}\|_1} \\ &\leq \frac{\lambda^{k+1} \times \|\Pi\|_1^k \times \|Q\|_1}{\|R^{k+1}\|_1} \end{aligned} \quad (3.9)$$

Cette dernière inégalité est due au résultat classique pour deux matrices M et N quelconques, $\|M \times N\|_1 = \|M\|_1 \times \|N\|_1$. Rappelons maintenant que la norme $\|Q\|_1 = \sum_{i,j \in [1,n]} \pi_{ji}$ est la somme de tous les scores de contribution directe entre tous les services du système. Comme $\|R^{k+1}\|_1 = \sum_{i=1}^n R_i^{k+1}$ comprend tous les chemins de contribution, y compris ceux de longueur 1, $\|R^{k+1}\|_1$ est supérieur à $\lambda \times \|Q\|_1$ (la valeur d'importance si le graphe $SI(\tau)$ n'avait que des chemins de longueur 1 entre tous les services). De plus, comme rappelé plus haut, nous avons avec les normes l'inégalité suivante : $\|\tilde{\Pi}^k\|_1 \leq \|\tilde{\Pi}\|_1^k$. Si l'on note par ω_{ij} la valeur dans la matrice $\tilde{\Pi}$ sur la ligne i et la colonne j , cette norme vaut $\|\tilde{\Pi}\|_1^k = (\max_j \sum_{i=1}^n |\omega_{ij}|)^k$ qui est inférieure ou égale à 1, puisque

la somme des contributions sur tous les arcs dans le graphe $SI(\tau)$ sortants de tous les nœuds n_{li} est inférieure ou égale à 1. Par conséquent, $\|\tilde{\Gamma}^k\|_1$ est aussi inférieur ou égal à 1 et l'erreur relative est alors inférieure à λ^k , autrement dit l'équation 3.9 conduit à l'inégalité suivante :

$$\frac{\|R^{k+1} - R^k\|_1}{\|R^{k+1}\|_1} \leq \lambda^k$$

Pour obtenir une erreur relative qui est inférieure à ε , il suffit de trouver l'itération k telle que $\lambda^k \leq \varepsilon$, ce qui implique que la valeur k est telle que $k \geq \frac{\log(\varepsilon)}{\log(\lambda)}$. Pour un λ donné, l'itération k est

$$k \geq \frac{\log(\varepsilon)}{\log(\lambda)} = c \times \log\left(\frac{1}{\varepsilon}\right)$$

où c est une constante ¹.

3.3.4 Exactitude du calcul

Nous allons montrer dans cette section que l'importance I_j définie dans l'équation 3.4 calcule effectivement la somme des contributions de s_j à tous les services du système sur tous les chemins d'utilisation dans le graphe $SC(\tau)$, d'après l'équation 3.3. Afin de simplifier les équations, nous ne prenons pas en considération la constante λ . Pour ce faire, nous allons exprimer l'importance d'un nœud n_{jk} dans le graphe d'importance représentant une arc $(s_k \rightarrow s_j)$ dans $SC(\tau)$ en fonction des valeurs d'importance reçues sur tous les chemins dans le graphe d'importance. Ensuite, nous allons exprimer l'ensemble des chemins entre deux nœuds dans $SC(\tau)$ en fonction de l'ensemble de chemins entre les nœuds dans $SI(\tau)$.

Le théorème suivant montre l'importance d'un nœud en fonction de l'importance de tous les nœuds n_{li} dans le graphe $SI(\tau)$.

Théorème 1. *L'importance $v_{jk}^*(\tau)$ d'un nœud n_{jk} dans le graphe $SI(\tau)$ est la somme des valeurs π_{li} reçues de la part de tous les nœuds n_{li} dans $SI(\tau)$ sur tous les chemins de n_{li} à n_{jk} dans le graphe $SI(\tau)$.*

Démonstration. La somme $\sum_{k=0}^{\infty} \tilde{\Gamma}^k$ calcule une matrice où un élément à la position $((j-1) \times n + k, (l-1) \times n + i)$ est la somme des poids de tous les chemins possibles de n_{li} à n_{jk} dans le graphe d'importance. $\mathbb{I}_{n^2 \times n^2} \times Q$ ajoute à n_{jk} la valeur $\pi_{jk}(\tau)$. La matrice $\tilde{\Gamma}$ contient sur la position $((j-1) \times n + k, (k-1) \times n + i)$ le poids de l'arc entre n_{ki} et n_{jk} et en conséquence $\tilde{\Gamma} \times Q$ ajoute à l'importance de n_{jk} les valeurs $\pi_{ki}(\tau)$ multipliées par le poids des chemins entre tous les nœuds n_{ki} et n_{jk} . En multipliant la matrice $\tilde{\Gamma}^k$ par Q , la valeur d'importance qui « arrive » à n_{jk} sur des chemins de longueur k qui partent de n_{li} est π_{li} multipliée par la somme des poids de ces chemins. L'importance totale reçue par n_{jk} sur des chemins de longueur k est la somme des importances reçues de la part de tous les nœuds n_{li} .

Par conséquent, après la multiplication de $\sum_{k=0}^{\infty} \tilde{\Gamma}^k$ avec Q , n_{jk} obtient de la part de n_{li} la valeur π_{li} multipliée par la somme des poids de tous les chemins possibles de n_{li} à n_{jk} . L'importance

¹ $\frac{1}{\log(\lambda)}$ est une constante négative ($\lambda < 1$)

totale de n_{jk} est calculée ensuite comme la somme des valeurs reçues de la part de tous les nœuds n_{li} . $\square$

Nous montrons maintenant le calcul de l'importance de services dans $SC(\tau)$ en fonction de l'importance des nœuds dans $SI(\tau)$. Chaque chemin $p = (s_i \rightarrow s_l \rightarrow s_m \rightarrow \dots \rightarrow s_n \rightarrow s_k \rightarrow s_j)$ dans $SC(\tau)$ correspond dans le graphe d'importance $SI(\tau)$ à un chemin $p' = (n_{li} \rightarrow n_{ml} \rightarrow \dots \rightarrow n_{kn} \rightarrow n_{jk})$, de poids $\omega_{p'}(\tau) = \omega_{mli}(\tau) \times \dots \times \omega_{jkn}(\tau)$. L'importance reçue par n_{jk} sur le chemin p' de la part du nœud n_{li} est $\pi_{li}(\tau) \times \omega_{mli}(\tau) \times \dots \times \omega_{jkn}(\tau)$, qui est le score de la contribution de s_j à s_i sur le chemin p (équation 3.2).

Le théorème suivant montre que la valeur d'importance de l'arc $(s_k \rightarrow s_j)$ dans $SC(\tau)$ correspond à la fraction d'importance de s_j qui est la contribution totale de s_j à tous les services du système par des chemins d'utilisation qui se finissent par $(s_k \rightarrow s_j)$. L'ensemble de tous les chemins possibles entre tous les services du système et le service s_j est noté par $\cup_{s_i \in S} \mathcal{P}_{ij}$. Pour un service $s_k \in In(t, s_j)$, l'ensemble de chemins p qui se terminent par $s_k \rightarrow s_j$ est noté par $\mathcal{A}_{kj}(t)$.

Théorème 2. L'importance $v_{jk}^*(\tau)$ d'un nœud n_{jk} est la somme des contributions du service s_j à tous les autres services du système S , sur les chemins dans $\mathcal{A}_{kj}(\tau)$.

Démonstration. L'ensemble $\mathcal{P}_{ij}$ de chemins de s_i à s_j dans le graphe d'appels de service $SC(\tau)$ peut être décomposé en $|Out(t, s_i)| \times |In(t, s_j)|$ sous-ensembles où chaque sous-ensemble contient les chemins débutant par $(s_i \rightarrow s_l)$ et finissant par $(s_k \rightarrow s_j)$ pour tous les $s_l \in Out(t, s_i)$ et les $s_k \in In(t, s_j)$. Par conséquent, cela signifie que tous les chemins possibles dans le graphe d'importance de n_{li} à n_{jk} correspondent à l'ensemble de tous les chemins possibles dans le graphe d'appels de service de s_i à s_j qui débutent par $(s_i \rightarrow s_l)$ et se terminent par $(s_k \rightarrow s_j)$. Les services n_{li} à partir desquels il existe un chemin vers le service n_{jk} représentent un sous-ensemble des arcs sortantes de s_i , plus précisément celles sur les chemins dans $SC(\tau)$ allant de s_i à s_j qui finissent par $(s_k \rightarrow s_j)$.

L'importance reçue de n_{jk} de n_{li} sur tous les chemins possibles de n_{li} à n_{jk} représente la contribution de s_j à s_i sur les chemins qui débutent par un sous-ensemble des arcs sortantes de s_i dans $SC(\tau)$ et qui finissent par $(s_k \rightarrow s_j)$. L'importance de l'arc $(s_k \rightarrow s_j)$ est la contribution de s_j à tous les services s_i sur les chemins qui débutent par un sous-ensemble d'arcs sortantes de s_i et qui se finissent par $(s_k \rightarrow s_j)$ (c'est à dire les chemins dans $\mathcal{A}_{kj}(\tau)$). $\square$

L'importance d'un service s_j calculée avec l'équation 3.4 comme la somme des valeurs d'importance des arcs $(s_k \rightarrow s_j)$ dans $SC(\tau)$ est par conséquent la contribution de s_j à tous les services s_i sur tous les chemins $\cup_{s_k \in In(\tau, s_j)} \mathcal{A}_{kj}(\tau)$, c'est à dire sur tous les chemins dans $SC(\tau)$. L'équation 3.4 calcule donc la même valeur d'importance pour chaque service s_j que l'équation 3.3.

Cas particulier : graphes sans cycle

Dans le cas où le graphe d'utilisation $SC(\tau)$ n'a pas de cycle, tous les chemins sont de longueur finie et l'ensemble des chemins $\mathcal{P}_{ij}$ est lui-aussi fini. Dans ce cas, le calcul d'importance s'arrête lorsque chaque service s_j a reçu sa contribution à chaque service s_i sur le(s) chemin(s) le plus long dans $\mathcal{P}_{ij}$. Le nombre d'itérations jusqu'à la convergence est par conséquent (au plus) le nombre

d'arcs sur le plus long chemin dans l'ensemble des chemins $\cup_{s_i, s_j \in \mathcal{S}} \mathcal{P}_{ij}(\tau)$.

Pour le cas encore plus particulier des applications où tous les chemins dans le graphe $SC(t)$ sont de longueur 1 (nous n'avons que de valeurs de contribution directe entre les services et pas de contribution indirecte) le calcul d'importance converge alors en une seule itération. Notons que dans ce cas, la matrice $\tilde{\Pi}(\tau)$ de l'équation 3.7 est la matrice nulle, et par conséquent l'équation 3.7 du calcul d'importance des arcs devient :

$$R(\tau) = \lambda \times Q(\tau)$$

3.4 Calcul d'importance

Cette section présente différents algorithmes pour calculer l'importance $I_k(\tau)$ de chaque service s_k à l'instant τ dans un système orienté services $\mathcal{S}$. Le calcul d'importance se fait avec des valeurs de contribution directe $\pi_{ji}(\tau)$ et indirecte $\omega_{ji}(\tau)$ calculées d'après l'utilisation des services jusqu'à τ . Ces valeurs ne changent pas pendant le calcul d'importance. Afin de simplifier les notations nous allons omettre dans la suite le temps τ dans la désignation des variables.

3.4.1 Calcul centralisé

Une première possibilité pour calculer l'importance des services à l'instant τ est de rassembler les valeurs de contribution directe et indirecte calculées par tous les services du système, de construire le graphe d'importance $SI(\tau)$ et de calculer la solution du système 3.7 en appliquant les itérations de Jacobi jusqu'à la convergence.

Une telle approche centralisée présente des limites, du fait que nous avons besoin que toutes les informations nécessaires au calcul soient rassemblées sur un service (ou un ensemble de services) autre que ceux ayant calculé ces informations. Ce service est alors chargé de calculer l'importance. Cependant, pour des raisons de confidentialité (lorsqu'un service ne veut pas publier ses valeurs de contribution), il est possible que les services du système refusent de donner accès aux informations concernant les valeurs de contributions qu'ils ont calculées. D'autre part, l'envoi de ces informations au service centralisé requiert des messages supplémentaires pouvant entraîner une saturation du réseau lors du calcul.

Cependant cette technique reste envisageable pour certaines applications. On notera par exemple que dans le cas du calcul d'importance des pages web avec la méthode PAGERANK tel qu'il est implanté et utilisé par *Google*, où le calcul est réalisé par un (cluster de) serveur(s) pour une dizaine de milliards de pages web.

Nous proposons par la suite que les services calculent eux-mêmes leurs valeurs d'importance de façon distribuée, en échangeant des valeurs d'importance en même temps que les appels de service.

3.4.2 Principes de la distribution

L'idée fondamentale de ce calcul est d'exploiter l'architecture orientée-services existante afin de distribuer le calcul sur les différents nœuds de service. Nous présentons deux algorithmes distribués qui diffèrent essentiellement par le choix du protocole pour échanger les valeurs d'importance. Pour le premier algorithme, les services calculent leur importance de manière synchrone. Le deuxième algorithme implante des itérations totalement asynchrones, décrites par la suite.

Les algorithmes présentés peuvent être déployés efficacement sur des infrastructures de services à large-échelle puisque chaque service calcule son importance en échangeant des messages seulement avec ses services voisins dans le graphe d'appels de service déjà connus.

Le problème du *link spam*

Le fait que les services calculent eux-mêmes leur importance permet aux services de garder leurs valeurs de contribution secrètes mais pose un autre problème de sécurité. En effet, un ensemble de services pourrait collaborer afin d'augmenter leur importance en échangeant des valeurs d'importance erronées. Ce problème, qui peut être comparé au problème de détection des *link spam* [92], est un problème orthogonal et n'a pas été envisagé dans nos travaux. Nous pensons également que les solutions existantes afin de sécuriser un calcul distribué (comme celle présentée dans [111]) peuvent être adaptées à nos algorithmes.

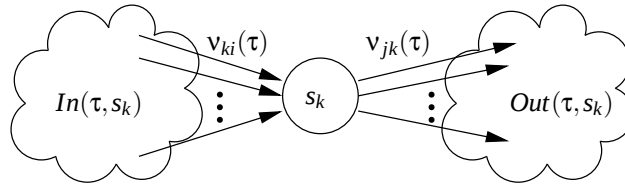


FIG. 3.3 – Envoi des mises à jour du vecteur R_k

Pour nos deux algorithmes, chaque service s_k calcule sa propre importance I_k à l'instant τ en échangeant des messages avec ses services voisins dans $In(\tau, s_k)$ et $Out(\tau, s_k)$. Chaque service s_k met à jour un vecteur N_k de valeurs d'importance reçues, avec les valeurs d'importance v_{ki} calculées et envoyées par les voisins s_i dans $In(\tau, s_k)$. Les valeurs d'importance dans N_k sont utilisées pour recalculer I_k ainsi que les valeurs d'importance v_{jk} qui vont être envoyées aux services $s_j \in Out(\tau, s_k)$.

Nous allons expliquer les algorithmes synchrone et asynchrone en fonction de la différence entre les instants quand les mises à jour d'importance v_{ki} sont reçues et les instants quand l'importance I_k est recalculée. Chaque service s_k a un compteur de mises à jour d'importance qui est incrementé à chaque fois que s_k recalcule I_k . A chaque mise à jour v_{ki} reçue de la part d'un service $s_i \in S$, nous enregistrons la valeur du compteur de mises à jour d'importance dans un vecteur Θ . Une valeur t dans Θ est désignée par *instant* dans la suite. Les valeurs d'importance v_{ki} reçues de la part des services s_i qui sont connues par s_k au dernier instant t dans Θ ($t = \max(\Theta)$) sont notées par :

$$N_k(t) = [v_{k1}(t'), \dots, v_{kn}(t'')]$$

Comme précédemment, $v_{ki} = 0$ si $s_i \notin In(\tau, s_k)$. La valeur $v_{k1}(t')$ est le dernier score d'importance que s_1 a envoyé à s_k . Ce score d'importance avait été reçu par s_k lorsque le compteur de mises à jour de l'importance avait la valeur t' . L'instant t est le dernier instant auquel s_k a recalculé son importance.

3.4.3 Calcul distribué synchrone

Dans le calcul *synchrone*, le service s_k attend d'avoir reçu toutes les valeurs v_{ki} de la part de tous les service dans $In(\tau, s_k)$ avant de recalculer son importance et l'importance à envoyer aux services s_j dans $Out(\tau, s_k)$. Dans ce cas toutes les valeurs t' du compteur d'importance sont égales à t puisque le compteur n'avance pas tant que s_k n'a pas reçu toutes les valeurs d'importance v_{ki} par ses voisins. Chaque service s_k doit également envoyer à t les valeurs v_{jk} à tous les services $s_j \in Out(\tau, s_k)$. Deux types de synchronisation peuvent être distingués :

- la *synchronisation globale*, si tous les services s_k du système ont la même valeur courante t du compteur d'importance ; c'est à dire que tous les services du système réalisent leur itération du calcul d'importance en même temps et ont donc réalisé le même nombre d'itérations ;
- la *synchronisation locale*, si on autorise que tous les services du système n'aient pas réalisé forcément le même nombre de mises à jour de leur importance ; il peut y avoir par exemple des ensembles de services du système qui communiquent entre eux et qui calculent leur importance plus vite que d'autres. Le calcul reste néanmoins synchrone, car chaque s_k attend d'avoir d'abord de nouvelles mises à jour de la part de tous les services $s_i \in In(\tau, s_k)$ avant de recalculer son importance.

L'algorithme synchrone COMPUTSYNC(τ) (voir algorithme 1) utilise la synchronisation locale pour calculer le vecteur de valeurs d'importance à un instant donné τ à l'aide d'un calcul de point fixe implanté sous forme d'itérations distribuées et synchronisées. A chaque itération, un service s_k collecte les valeurs d'importance $v_{ki}(\tau)$ de tous ses clients $s_i \in In(\tau, s_k)$, recalcule son importance et l'envoie aux services $s_j \in Out(\tau, s_k)$. L'échange des valeurs d'importance est synchronisé et le service s_k doit attendre les valeurs d'importance v_{ki} calculées à l'itération précédente avant de recalculer son importance à l'itération courante. Les itérations s'arrêtent lorsque toutes les valeurs d'importance des services sont proches du point fixe (pour une précision ε donnée). L'algorithme 1 est exécuté par chaque service $s_k \in \mathcal{S}$ pour calculer son importance à un instant τ .

La variable O_k mémorise l'ancienne valeur d'importance et la variable globale p désigne l'ensemble des services qui ont convergé pendant l'exécution de l'algorithme. Au début, la variable p est initialisée à l'ensemble vide et un instant τ pour lequel on veut réaliser le calcul est choisi. La valeur d'importance initiale est fixée à 0. L'importance reçue par un service s_j converge *localement* pendant une itération i lorsque la condition de convergence $|I_k - O_k|/I_k < \varepsilon$ est satisfaite

pour un $\varepsilon > 0$ donné. Néanmoins, la convergence locale ne garantit pas que la condition de convergence restera satisfaite pour tous $j \geq i$ (convergence globale) [108]. Chaque service arrête le calcul seulement lorsque tous les services $s_k \in \mathcal{S}$ ont convergé localement ($\rho = \mathcal{S}$).

Algorithme 1 : ComputSync(un instant τ , $\rho = \emptyset$)

```

1   $O_k = I_k = 0, N_k = 0$ 
2  répéter
3    // envoi des valeurs d'importance
4    pour chaque ( $s_j \in Out(\tau, s_k)$ ) faire
5       $v_{jk} = \lambda \times \pi_{jk} + \lambda \times \sum_{s_i \in In(s_k)} n_{ki} \times \omega_{jki}$ 
6      envoie  $v_{jk}$  à  $s_j$ 
7    //les services de la part desquels  $s_k$  n'a pas encore reçu l'importance
8     $Tmp = In(\tau, s_k)$ 
9    // calcule la nouvelle importance
10   tant que ( $Tmp \neq \emptyset$ ) faire
11     //attend qu'un service  $s_i$  envoie  $v_{ki}$ 
12      $n_{ki} = v_{ki}$ 
13      $Tmp = Tmp \setminus \{s_i\}$ 
14   //recalcule l'importance du service
15    $O_k = I_k$ 
16    $I_k = \sum_{s_i \in In(s_k)} n_{ki}$ 
17   si ( $|I_k - O_k|/I_k \geq \varepsilon$ ) alors
18      $\rho = \rho \setminus \{s_k\}$ 
19   sinon
20      $\rho = \rho \cup \{s_k\}$  // convergence locale
21   // stoppe quand tous les services ont convergé
22 jusqu'à ( $\rho \equiv \mathcal{S}$ ) ;
23 retourner (importance du service  $s_k$  à l'instant  $\tau$ )

```

3.4.4 Calcul distribué asynchrone

L'algorithme synchrone conduit à un coût de communication et à une charge du réseau supplémentaire au moment du calcul puisqu'il génère de nouveaux messages de calcul entre les services. L'algorithme que nous allons présenter maintenant utilise un protocole asynchrone pour le calcul et réduit la charge du réseau en intégrant les valeurs d'importance dans les appels de service « normaux » (applicatifs) sans générer de messages supplémentaires. Un autre avantage de l'algorithme asynchrone par rapport à l'algorithme synchrone est sa facilité de redémarrage [27]. Quand un nouveau service apparaît ou un service participant au calcul disparaît pendant le calcul synchrone, tous les services doivent en être informés et reprendre depuis le début leur calcul. Avec l'algorithme asynchrone, l'apparition ou la disparition d'un service n'implique pas le redémarrage

global du calcul, mais restreint l'effort nécessaire pour corriger l'erreur de calcul résultante uniquement aux services dont les scores sont affectés par le changement du graphe (voir la section 4.3 du chapitre 4 pour une estimation expérimentale de leur nombre).

Le calcul *asynchrone* évite l'envoi de messages de calcul supplémentaires ainsi que les retards dus à la synchronisation, en intégrant les valeurs d'importance dans des appels de services propres à l'application. Le service s_k n'attend plus d'avoir des mises à jour de la part de tous les services dans $In(\tau, s_k)$ pour recalculer son importance. Dès qu'il reçoit une valeur de la part d'un service s_i , il recalcule I_k en utilisant cette nouvelle valeur, alors que pour les autres services il utilise les anciennes valeurs qu'il avait stockées. Le service s_k ne doit plus envoyer les valeurs d'importance v_{jk} à chaque fois qu'il reçoit des mises à jour et qu'il recalcule son importance, mais seulement de temps en temps, lorsque l'application nécessite qu'il appelle le service s_j . Dans cette situation, les valeurs t' du compteur d'importance peuvent être différents entre eux. Supposons que la valeur du compteur d'importance de s_k lorsqu'il avait reçu la dernière mise à jour v_{ki} de s_i était t' . La différence entre $\max(\Theta)$ et t' exprime le nombre de mises à jour de I_k ayant eu lieu entre la version courante de I_k et la version de I_k à l'instant où le compteur était égal à t' . Remarquons qu'il peut aussi y avoir des services qui exécutent un plus grand nombre de calculs que d'autres, et qui communiquent plus fréquemment avec leurs voisins.

À chaque nouvel appel de service $s_k \rightarrow s_j$:

l'émetteur s_k

1. lit la valeur stockée v_{jk} qu'il avait calculée auparavant ;
2. ajoute cette valeur dans l'appel de service (comme un élément supplémentaire d'un message SOAP par exemple) ;
3. appelle le service s_j et lui transmet en même temps v_{jk} ;

le destinataire s_j

1. mémorise la valeur d'importance reçue dans n_{jk} ;
2. met à jour son importance $I_j = \sum_{s_k \in In(\tau, s_j)} n_{jk}$;
3. met à jour les valeurs $v_{lj} = \lambda \times \pi_{lj} + \lambda \times \sum_{s_k \in In(\tau, s_j)} n_{jk} \times \omega_{ljk}$

où n_{jk} est une valeur du vecteur N_k défini comme dans la section précédente. Les valeurs de n_{jk} sont initialisées à 0 au début du calcul. Chaque service s_k recalcule son importance locale à chaque appel de service entrant, sans attendre que tous les clients $s_i \in In(\tau, s_k)$ aient envoyé leur importance. D'une manière similaire, le service s_k communique ses nouvelles valeurs d'importance à un service $s_j \in Out(\tau, s_k)$ seulement lorsqu'il fait un nouvel appel à s_j . Nous obtenons ainsi un protocole asynchrone où chaque service calcule son importance à sa propre vitesse, en utilisant des valeurs d'importance qui peuvent être dépassées. Le seul point de synchronisation entre les services réside dans le choix de l'instant τ auquel le calcul est débuté. De façon similaire à l'algorithme précédent, un service considère qu'il a convergé localement lorsque la variation de l'importance $|I_k - O_k|/I_k$ ne dépasse pas un seuil ε donné. Le calcul est arrêté lorsque tous les services ont convergé.

3.4.5 Convergence du calcul distribué

Nous analysons dans cette section les conditions pour que les algorithmes synchrone et asynchrone convergent. En ce qui concerne le calcul synchrone, il implante de façon distribuée les itérations de Jacobi pour lesquelles la convergence dans notre cas a été démontrée à la section précédente. Pour le calcul asynchrone, dans le cas général, il n'y a pas convergence, même si le calcul synchrone qui lui correspond est de son côté convergent. Conformément à [27], afin que le calcul asynchrone converge vers la solution du système 3.7, il doit respecter les conditions suivantes :

1. le rayon spectral de $|\lambda \times \Pi|$, $\rho(|\lambda \times \Pi|)$ doit être inférieur à 1 ;
2. le calcul d'importance doit respecter la condition d'*asynchronisme total* ; cette condition implique que chaque valeur d'importance soit mise à jour infiniment souvent (jusqu'à la convergence) et que les anciennes valeurs soient finalement purgées du système.

Nous avons démontré dans la section précédente que la condition sur le rayon spectral est satisfaite pour notre système. Nous supposons que notre système satisfait aussi la condition d'asynchronisme total et que, pour un instant t donné, les mises à jour reçues avant t ne sont pas réutilisées infiniment pour calculer I_k , et que les services s_i envoient de temps en temps à s_k de nouvelles valeurs d'importance.

En ce qui concerne la vitesse de convergence, nous avons vérifié expérimentalement que la vitesse de convergence du calcul synchrone dépend du logarithme de ε et nous avons comparé le nombre d'itérations jusqu'à la convergence des calculs synchrone et asynchrone. Nous renvoyons à la section 4.3 pour les expériences portant sur une application orientée appels de service. Notons que nous considérons qu'avec le calcul synchrone un service réalise une itération chaque fois qu'il met à jour son importance. Pour l'algorithme asynchrone, nous estimons qu'un service fait une itération lorsqu'il reçoit un nombre de mises à jour égal au nombre de ses voisins dans $In(\tau, s_k)$.

3.4.6 Terminaison du calcul distribué

Pour les deux algorithmes, chaque service considère qu'il a une approximation suffisamment bonne de son importance lorsque l'erreur relative de la nouvelle importance qu'il vient de calculer ne varie pas de plus de ε de l'importance calculée précédemment, pour un ε (suffisamment petit) donné. Lorsque cette condition est satisfaite par I_k , nous pouvons dire que I_k a convergé *localement*. La condition de convergence locale n'implique pas forcément la convergence *globale*. Il peut ainsi y avoir des services qui calculent plus rapidement que d'autres et donc qui n'exécutent pas la même itération. Même si l'importance d'un service satisfait le critère de précision ε à un certain instant, il est possible qu'il ne le satisfasse plus après une nouvelle mise à jour des valeurs d'importance reçues de la part des voisins.

Lorsque le calcul a convergé, aucune mise à jour n'est envoyée entre les services. De façon similaire à [27] nous considérons que le calcul est distribué si les conditions suivantes sont satisfaites :

1. plus aucun message n'est en transit,
2. aucun recalcul de I_k ne fait plus changer sa valeur.

La deuxième propriété énoncée correspond à un ensemble de conditions de terminaison locales. Dans notre méthode pour détecter la terminaison du calcul distribué nous avons utilisé une méthode de détection que nous appelons *supervisée* [166], c'est à dire que nous détectons la terminaison des algorithmes distribués en utilisant une variable globale ρ qui mémorise l'identité des services ayant convergé localement et nous arrêtons le calcul d'importance lorsque $\rho \equiv S$.

En ce qui concerne le calcul distribué asynchrone, conformément à [27], pour que le calcul d'importance se termine, nous devons imposer les conditions suivantes sur la communication des valeurs d'importance entre les services :

1. lorsque s_k calcule des nouvelles valeurs d'importance v_{jk} , celles-ci doivent être finalement envoyées à tous les $s_j \in Out(\tau, s_k)$,
2. la valeur d'importance v_{jk} est envoyée si et seulement si l'erreur relative diffère de plus de ε de la dernière valeur envoyée par s_k à s_j ,
3. les mises à jour de l'importance sont reçues dans l'ordre dans lequel elles ont été transmises par les services dans $In(\tau, s_k)$,
4. le calcul d'importance doit être démarré, c'est à dire un service a envoyé au moins une mise à jour à ses voisins.

Généralement, un algorithme asynchrone qui est convergent n'est pas forcément garanti de se terminer dans un nombre d'itérations fini. Pour que le nombre d'itérations soit fini, conformément à [27] chaque service s_k qui réalise le calcul asynchrone recalcule une nouvelle valeur v_{jk} pour ses voisins $s_j \in Out(\tau, s_k)$ seulement si la nouvelle valeur de v_{jk} diffère de son ancienne valeur de plus de ε . Avec cette modification, d'après [27], le calcul asynchrone se termine en un nombre d'itérations fini si la fonction de mise à jour f des valeurs d'importance v_{ki} entre tous les couples de services s_i et s_k est une *contraction*² en utilisant la norme maximale pondérée. Dans notre calcul, la fonction f qui calcule les valeurs d'importance v_{ki} est, conformément à l'équation 3.7, $f(R) = \lambda \times \tilde{\Pi} \times R + \lambda \times Q$. [27] montre aussi que ce type d'itérations linéaires utilisant une matrice (dans notre cas $\lambda \times \tilde{\Pi}$) avec le rayon spectral inférieur à 1 est justement une telle contraction.

² $\|f(x) - f(y)\| \leq \alpha \|x - y\|, \forall x, y \in \mathbb{R}^n, \alpha \in [0, 1)$

3.5 Conclusion

Dans ce chapitre nous avons proposé un modèle générique de classement de services qui collaborent. Notre modèle d'importance repose sur la notion de *contribution effective* basée sur l'utilisation. Le graphe d'utilisation de services est construit à partir des instants enregistrés par les traces d'utilisation. Le calcul de contribution repose sur la notion de chemin d'utilisation et prend donc en compte la contribution directe d'un service à un autre service, mais également la contribution indirecte, via un ou plusieurs services intermédiaires. Nous définissons la contribution totale effective d'un service qui prend en compte sa contribution sur tous les chemins d'utilisation possibles à un service, l'importance du service étant sa contribution totale à tous les services du système. Nous avons fait une analyse du calcul basée sur un graphe d'importance équivalent au graphe d'utilisation de services. Nous avons proposé et analysé les propriétés des algorithmes de calcul synchrone et asynchrone répartis.

Ce modèle abstrait peut être instancié dans le cadre de différentes applications. Dans les chapitres suivants nous allons montrer comment ce modèle peut être déployé dans le cadre du classement de services basé sur les appels (chapitre 4), des applications de cache distribué (chapitre 5) et du classement de services par l'utilisation de leurs données (chapitre 6).

Chapitre 4

Classement de services basé sur les appels

Dans ce chapitre nous présentons l'application du modèle générique pour le classement de services web par leur utilisation, qui dépend des instants auxquels ils sont appelés. La notion de contribution de service est décomposée en une partie statique (indépendante des appels) qui reflète sa contribution relative à la qualité d'autres services, et une partie dynamique qui dépend de l'utilisation du service. Nous présentons des résultats expérimentaux qui comparent les performances des algorithmes de calcul d'importance synchrone et asynchrone.

Ce chapitre est organisé de la façon suivante : après une présentation du problème en section 4.1, nous présentons à la section 4.2 le modèle de contribution effective basé sur le modèle générique du chapitre 3. La section 4.3 fournit une validation expérimentale de notre modèle. La section 4.4 illustre rapidement l'application de notre modèle de classement au classement de services WEBCONTENT. La section 4.5 conclut.

Les travaux présentés dans ce chapitre ont été réalisés dans le cadre du projet ACI MD SEMWEB et ils ont été publiés dans la conférence internationale COOPIS [60], dans la conférence nationale BDA [61] ainsi que dans la revue nationale ISI [62]. Le modèle de classement présenté dans la section 4.4 a fait l'objet d'un livrable dans le cadre du projet RNTL WEBCONTENT.

4.1 Introduction

Aujourd'hui, de plus en plus d'applications s'appuient sur une architecture de services web. Ces services deviennent également plus complexes, faisant souvent appel à d'autres services pour réaliser leur tâche, eux-mêmes appelant généralement d'autres services. Cette croissance du nombre de services web disponibles ainsi que leur tendance à être de plus en plus composés soulèvent de nouveaux problèmes, comme par exemple celui du choix d'un service à appeler parmi plusieurs services du même type. Le classement des différents services afin de procéder au meilleur choix est alors une solution qui s'impose naturellement.

Dans ce chapitre nous proposons un modèle d'importance de services basé sur la contribution effective, où la contribution d'un service dépend uniquement des instants auxquels il est appelé, indépendamment de l'utilisation des réponses qu'il a générées. La contribution effective d'un ser-

vice est liée aux notions d'*utilisation* et de *qualité de service* et les liens d'utilisation entre les services sont déterminés par les appels de service. De nombreux exemples d'applications correspondent à notre définition d'un service et peuvent bénéficier de notre modèle de classement. Par exemple, dans un moteur de recherche distribué en pair-à-pair, chaque pair peut être considéré comme un service cherchant certains documents localement et interrogeant d'autres pairs (services). Les résultats des recherches peuvent être classés suivant des critères basés sur le contenu, comme la pertinence du document pour la requête. Pour trier des documents au contenu similaire, il peut également être utile de supposer qu'un pair important (par exemple ayant du renom) produit des résultats plus importants que les autres. Un autre exemple est la découverte et la sélection de services web.

4.1.1 Présentation de l'approche

Notre approche consiste à classer des services par leur importance basée sur la contribution d'un service aux autres services du système, comme nous l'avons présenté dans le chapitre 3. L'idée consiste à définir des scores de contribution qui expriment :

- (i) la contribution intrinsèque d'un service s_j à la *qualité* d'un service s_i , reflétant la contribution de s_j à s_i comparée à la contribution à s_i des autres services s_k du système et
- (ii) la manière dont s_j est *utilisé* réellement.

Notre définition de contribution effective d'un service est donc formée (i) d'une contribution *statique*, indépendante du temps des appels, à la qualité de service et qui définit une contribution maximale de s_j à s_i et (ii) d'une partie dynamique liée au temps des appels, qui dépend de l'*utilisation* de s_j par s_i pendant une période de temps. L'agrégation des valeurs de contribution et d'utilisation permet de comparer les services par leur importance globale.

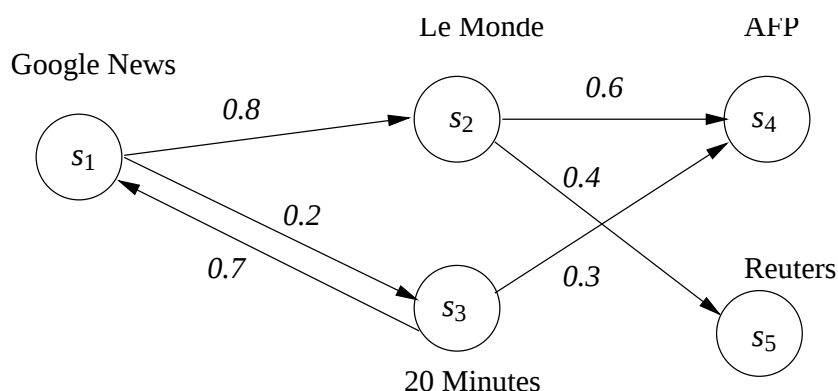


FIG. 4.1 – Graphe de contributions à la qualité

Nous illustrons notre approche à travers l'exemple d'un système de partage de nouvelles distribué (« news »), où chaque service peut jouer le rôle d'un fournisseur de nouvelles (serveur), d'un

consommateur de nouvelles (client) ou des deux à la fois (portail). La figure 4.1 montre un exemple avec cinq nœuds qui collaborent en échangeant des informations. Les services s_4 et s_5 sont fournis par les deux agences de presse *Agence France Presse (AFP)* et *Reuters*. Les services s_2 et s_3 sont publiés par les deux journaux *Le Monde* et *20 Minutes* et le service s_1 est proposé par le moteur de recherche *Google*.

Scores de contribution des services

Tous les services collaborent en échangeant des nouvelles via des appels de services. Dans notre approche nous supposons qu'une partie de la qualité de chaque service dépend des services qu'il appelle. Notre notion de « qualité de service » (QOS) est abstraite et chaque service pris individuellement peut utiliser des critères de qualité différents afin d'estimer la contribution des autres services.

Par exemple, un journal peut considérer que la qualité d'un article dépend de son intérêt pour ses lecteurs. En suivant ces critères, le journal français *Le Monde* estime alors qu'en général, l'agence de presse française *AFP* fournit des informations plus intéressantes pour ses lecteurs que l'agence américaine *Reuters*. D'un autre côté, *20 Minutes* essaie de réduire les coûts et préfère donc *Google News*, qui est gratuit, à *AFP*, qui propose un service payant. Ces mesures sont intégrées dans le graphe de la figure 4.1 où chaque arête $s_i \rightarrow^c s_j$ est étiquetée par la valeur $c \in [0, 1]$ représentant la *contribution* du service s_j à la qualité de son client s_i *relative aux autres services* utilisés par s_i .

Chaque service participe directement à l'amélioration de la qualité de ses clients, à la qualité des clients de ses clients, et ainsi de suite. Par exemple, *AFP* contribue indirectement à la qualité de *Google News* via *Le Monde*. Les cycles de contribution peuvent s'expliquer par le fait que chaque service peut contenir différentes données et implanter différentes fonctionnalités. Par exemple, *20 Minutes* utilise *Google News* parce qu'il est gratuit et *Google News* pourrait appeler *20 Minutes* afin de collecter des nouvelles de *AFP* (rappelons qu'il n'y a pas de lien direct entre *Google News* et *AFP*).

Scores d'utilisation de services et d'appels

Un service peut effectivement contribuer à la qualité d'autres services seulement s'il est appelé. Nous illustrons l'utilisation des services sur la figure 4.2, où chaque arc $s_i \hookrightarrow^t s_j$ correspond à un appel de service à un moment t . Cette figure montre que même si *Reuters* est supposé contribuer à la qualité de *Le Monde*, cette contribution est pour le moment « virtuelle » puisqu'il n'a pas encore été appelé. Nous appelons cette propriété qui dépend des appels effectivement reçus l'*utilisation du service*.

L'utilisation d'un service s_i est définie pour un certain instant t et dépend des appels de services reçus par s_i avant t . Plus formellement, cette utilisation est définie pour chaque couple de services (s_i, s_j) par une fonction qui calcule un *score d'utilisation de service* $u_{ji}(t)$ de s_j par s_i au moment t .

Par exemple, le service s_1 (*Google News*) est intéressé par toutes les nouvelles publiées par le service s_2 (*Le Monde*). Si *Le Monde* met à jour ses nouvelles tous les jours, *Google News*

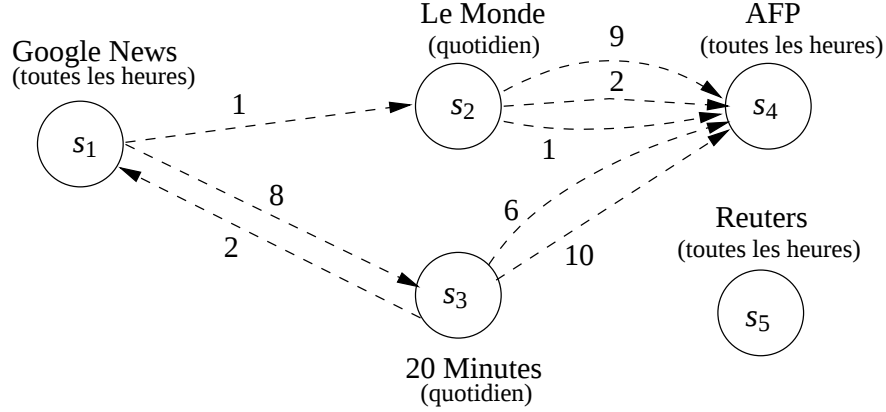


FIG. 4.2 – Appels de services jusqu'à l'instant 10

doit l'appeler quotidiennement pour maintenir un score d'utilisation maximal de $u_{21}(t) = 1$ et l'utilisation du service *Le Monde* par *Google News* à l'instant t peut être estimé par l'âge $|t - \tau|$ du dernier appel $s_1 \hookrightarrow^\tau s_2$ émis par le service s_1 avant t . Si cet âge est inférieur à une journée, *Le Monde* peut être considéré comme complètement utile pour *Google News* à l'instant t , c'est à dire $u_{21}(t) = 1$. D'un autre côté, si l'âge excède un certain nombre de jours, par exemple un mois, *Le Monde* peut être considéré comme parfaitement inutile ($u_{21}(t) = 0$) pour *Google News* à t . En effet, si *Google News* appelle *Le Monde* trop rarement, il manque une partie de ses nouvelles et *Le Monde* est considéré comme étant moins utile pour *Google News*.

La notion d'utilisation peut être étendue aux appels de services en estimant l'utilisation indirecte $u_{jki}(t)$ des résultats d'appels de service sortants $s_k \hookrightarrow^t s_j$ pour répondre aux appels de service entrants $s_i \hookrightarrow^{t'} s_k$.

De façon similaire à l'exemple précédent, ce score d'utilisation $u_{jki}(t)$ pourrait être estimé par la proximité temporelle entre les appels de service entrants $s_i \hookrightarrow^{t'} s_k$ et les appels de service sortants $s_k \hookrightarrow^t s_j$ avant t . En appliquant cette sémantique à la figure 4.2, l'appel $s_2 \hookrightarrow^1 s_4$ est plus utile pour l'appel $s_1 \hookrightarrow^1 s_2$ que les appels $s_2 \hookrightarrow^{2,9} s_4$ à l'instant 2 ou à l'instant 9. En général, $u_{jki}(t)$ exprime un taux d'utilisation des appels issus de s_i des résultats retournés par les appels de s_j et dépend de l'implantation de s_k . Par exemple, si le service *20 Minutes* stocke les résultats reçus par le service *AFP*, l'appel issu de *20 Minutes* vers *AFP* à l'instant 6 pourrait être utile pour l'appel ultérieur reçu par *20 Minutes* à l'instant 8.

Importance de service

L'idée principale de ce travail est d'estimer l'importance d'un service en combinant les scores de contribution et d'utilisation des services avec lesquels il collabore directement ou indirectement. La contribution effective d'un service s_j à un autre service s_i est définie comme le produit de la contribution à la qualité et les scores d'utilisation des services et des appels.

Nous illustrons ceci avec l'exemple ci-dessus. Tout d'abord, si nous considérons seulement le graphe de contribution à la qualité de la figure 4.1, il paraît sensé de dire que *AFP* devrait être plus

important que *Reuters* puisqu'il contribue globalement plus aux autres services. Cet argument est confirmé par la figure 4.2 qui montre que *Reuters* n'a encore été appelé par aucun service. Cependant, si nous essayons de comparer *Le Monde* et *20 Minutes* de la même façon, nous observons que les deux critères (contribution et utilisation) sont indépendants l'un de l'autre. Par exemple, *Le Monde* devrait sans doute être plus important que *20 Minutes* si l'on considère la valeur de sa contribution à *Google News*. Cependant, comme nous l'avons vu dans la figure 4.2, à $t = 10$ les deux services ont été appelés une fois seulement par *Google News* à différents moments. Si l'on estime que l'utilisation d'un service dépend de l'âge du dernier appel reçu, à l'instant 1 *Le Monde* est dans ce cas plus important que *20 Minutes* (qui n'a pas été encore utilisé), mais il perd la première place à l'instant 8 lorsque *20 Minutes* est appelé.

4.1.2 Travaux existants

Différentes méthodes pour classer les services web qui prennent en compte de nombreux critères ont été proposées dans la littérature. Par exemple, [46] propose d'utiliser des techniques d'échantillonnage avancées afin de comparer et classer des services web orientés données en fonction de leurs données locales. Les techniques s'appuyant sur les recommandations exploitent les appréciations des utilisateurs [127, 106] et les votes des services [72] pour la sélection dynamique de service. D'autres critères de classement s'appuient sur la conformité des critères de qualité de service durant des périodes données [183, 106]. La variation dans la conformité des niveaux de qualité projetés pendant une fenêtre de temps est aussi un paramètre [106]. La variation de la conformité à livrer les niveaux de qualité promis pendant une fenêtre temporelle est aussi prise en considération par [106]. Tous ces paramètres peuvent être combinés en utilisant des poids différents suivant les exigences spécifiques de l'utilisateur [183, 127].

Dans ce contexte, notre méthode peut être vue comme une approche basée sur des recommandations pour classer les services où chaque appel de service correspond à un vote implicite qui prend en compte la qualité de service et l'utilisation observées lors d'une période donnée. Notre notion de contribution peut aussi être interprétée comme des « votes » sur des périodes de temps données.

4.2 Modèle

Un service contribue directement ou indirectement à d'autres services du système. Plus précisément, nous considérons qu'un service s_j peut contribuer directement à un certain moment t à tous les services s_i , avec $s_i \in In(t, s_j)$, et par transitivité à tous les services s_k auxquels s_i contribue.

Nous faisons la distinction entre la contribution d'un service s_j à la *qualité* d'un service s_i et la manière dont s_i utilise s_j pendant une période de temps. Par conséquence, notre définition de contribution d'un service prend en compte une contribution à la qualité de service *statique* (indépendante des appels) et une contribution dynamique due à l'utilisation (dépendante des appels).

4.2.1 Contribution statique

Chaque service donné estime sa qualité et la contribution d'autres services à sa qualité en fonction de certains critères spécifiques à l'application. L'idée essentielle légitimant cette estimation est que la qualité d'un service dépend de la qualité des services qu'il appelle. Par exemple, un service de partage de nouvelles peut définir la contribution des autres services qu'il appelle en fonction de la pertinence des nouvelles reçues de la part de ces services vis à vis des intérêts de ses lecteurs. Dans un moteur de recherche pair-à-pair la contribution d'autres services peut être définie par des mesures comme la précision et le rappel. D'autres critères de qualité peuvent englober la fraîcheur moyenne de ses résultats (dans le cas de la réplication de données avec des mises à jour), le temps de réponse moyen, la confiance, ou des critères commerciaux s'appuyant sur le prix de chaque appel de service.

Contribution directe

Chaque service s_i du système définit et acquiert les mesures sur la contribution relative des services $s_j \in Out(t, s_i)$ qu'il utilise en fonction des critères qui lui sont propres. Nous ne sommes pas intéressés dans ce travail par la façon dont cette connaissance sur la contribution à la qualité d'un service est acquise et définie. Nous définissons le score de contribution d'un service s_j à la qualité d'un service s_i par une fonction Υ :

Définition 8 (contribution locale à la qualité). *Soit S un ensemble de services. La fonction de contribution $\Upsilon : S \times S \rightarrow [0, 1]$ définit pour chaque paire de services distincts (s_i, s_j) un score de contribution locale $\Upsilon(s_i, s_j) = \gamma_{ji}$ du service s_j à la qualité du service s_i tel que $\sum_{s_j \in S} \gamma_{ji} = 1$. Le score γ_{ji} est 0 si s_j ne contribue pas à la qualité de s_i .*

La fonction $\Upsilon(s_i, s_j)$ retourne un score γ_{ji} qui compare la contribution à la qualité de s_i de tous les services s_j directement utilisés par s_i . Nous supposons que ces scores sont statiques et statistiquement indépendants d'un appel particulier entre s_i et s_j (cette restriction peut être relâchée mais elle simplifie la présentation et la compréhension de notre modèle d'importance).

Contribution indirecte

Pour illustrer la notion de contribution indirecte, nous considérons le cas de trois services, le service s_i appelant le service s_k qui appelle à son tour s_j ($s_i \rightarrow s_k \rightarrow s_j$). Le service s_i définit la contribution locale γ_{ki} de tous les services s_k qu'il utilise, indépendamment des services s_j qui sont utilisés par s_k .

Définition 9 (contribution indirecte à la qualité). *Soit $s_i, s_k, s_j \in S$ tels que s_i utilise s_k et s_k utilise s_j . La contribution indirecte de s_j à la qualité de s_i à travers s_k est la partie de la contribution directe γ_{ki} de s_k à s_i qui est due à s_j .*

Dans notre modèle nous ne faisons aucune supposition sur le fonctionnement interne des services et sur la manière dont ils appellent d'autres services. Les services sont par conséquent considérés comme des « boîtes noires » et pour chaque service s_k nous ne connaissons pas la manière dont l'utilisation de s_j par s_k est influencée par l'utilisation de s_k par un autre service s_i .

Nous considérons que s_k définit la partie de la contribution γ_{ki} qui est due à s_j d'après ses propres critères de qualité, comme étant égale au score de contribution directe de s_j à s_k . Le score de *contribution indirecte* de s_j à s_i à travers s_k est par conséquent égal à γ_{jk} , et la contribution de s_k à s_i qui est due à s_j est calculée comme étant $\gamma_{ki} \times \gamma_{jk}$.

Exemple 6. Dans la figure 4.1, le service Google News estime que les nouvelles offertes par Le Monde contribuent localement avec un score de 0.8 à sa qualité. De même, Le Monde estime que la contribution de AFP à sa qualité ainsi qu'à celle de ses clients est de 0.6. La contribution de Le Monde à la qualité de Google News qui est due à AFP est de $0.8 \times 0.6 = 0.48$.

Contribution statique sur un chemin

Nous pouvons construire le graphe de contribution statique basé sur les valeurs de contribution statique locale définies précédemment :

Définition 10 (graphe de contribution statique). La fonction Υ définit un score de contribution entre des paires de services et génère un graphe étiqueté orienté appelé graphe de contribution $GC(\Upsilon) = (S, C, \Upsilon)$ où (i) S est l'ensemble de services du système (ii) C est l'ensemble des arcs tels que $(s_i \rightarrow s_j) \in C$ ssi $\Upsilon(s_i, s_j) \neq 0$ et (iii) chaque arc $(s_i \rightarrow s_j) \in C$ est étiqueté par la valeur γ_{ji} correspondante.

Exemple 7. Le graphe présenté dans la figure 4.1 illustre la contribution statique locale de chaque service de nouvelles.

Chaque chemin $p = s_i \xrightarrow{\gamma_{ki}} s_k \rightarrow \dots s_l \xrightarrow{\gamma_{jl}} s_j$ dans le graphe $GC(\Upsilon)$ représente une fraction de la contribution d'un service s_j à la qualité d'un service s_i . Le score de contribution à la qualité sur le chemin p est calculé comme étant :

$$\gamma_p = \gamma_{ki} \times \dots \times \gamma_{jl}$$

Le score de contribution à la qualité sur un chemin dépend donc des valeurs de contribution directe à la qualité entre les services sur ce chemin.

4.2.2 Utilisation de service

Dans le cas des services utilisés par appels les *traces d'utilisation* permettent d'enregistrer les instants des appels de service. Plus formellement, la définition de la trace d'utilisation (définition 1, page 48) est dans ce cas :

Définition 11 (fonction de trace d'appel). Soit S un ensemble fini de services identifiés qui s'appellent les uns les autres. Une fonction de trace d'appel $\Lambda : \mathcal{T} \times S \times S \rightarrow 2^{\mathcal{T}}$ retourne pour chaque instant $t \in \mathcal{T}$ et chaque paire de services distincts (s_i, s_j) , l'ensemble d'estampilles $\Lambda(t, s_i, s_j) = \{\tau | \tau \leq t\}$ qui représente la partie de $\mathcal{T}$ constituée de tous les instants $\tau \leq t$ auxquels il y a eu un message de type requête-réponse $s_i \xrightarrow{\tau} s_j$ où τ correspond à l'estampille du message représentant la requête envoyée par s_i à s_j .

Par requête on comprend un appel de service avec un ensemble de paramètres. Dans ce cas, le graphe d'utilisation construit à l'aide de Λ est désigné comme *graphe d'appels*, et les chemins d'utilisation correspondent aux chemins d'appel de service. Nous considérons qu'un service appelle seulement des services qui contribuent à sa qualité. Par conséquent le graphe d'appels de service correspond au graphe de contribution statique, un chemin d'appel de service étant en même temps un chemin de contribution à la qualité.

Les appels enregistrés dans $\Lambda(t, s_i, s_j)$ représentent l'utilisation effective de s_j par s_i à un instant t donné. Un service s_j peut contribuer effectivement à la qualité d'un service s_i seulement s'il est appelé et le nombre et les estampilles des appels reçus par s_j de la part de s_i ont une influence sur sa contribution effective. Pour chaque couple de services (s_i, s_j) nous introduisons un *score d'utilisation de service* qui exprime la façon dont s_j est utilisé par s_i , en fonction des appels reçus par s_j de s_i :

Définition 12 (utilisation locale de service). *La fonction d'utilisation de service $U_d : \mathcal{T} \times \mathcal{S} \times \mathcal{S} \rightarrow [0, 1]$ retourne pour chaque instant t et chaque paire de services (s_i, s_j) un score d'utilisation locale de service $U_d(t, s_i, s_j) = u_{ji}(t)$ du service s_j par le service s_i obtenu d'après les appels enregistrés dans $\Lambda(t, s_i, s_j)$.*

La définition concrète de la fonction d'utilisation dépend de la sémantique de l'application et n'importe quelle fonction agrégeant les estampilles d'appels de service dans $\Lambda(t, s_i, s_j)$ avec une sémantique appropriée peut être choisie. Des fonctions d'utilisation de service différentes peuvent être définies pour chaque couple (s_i, s_j) .

Dans la suite, nous supposons que le score d'utilisation $u_{ji}(t)$ est spécifique au service s_j et qu'il décroît avec l'ancienneté des derniers appels reçus de la part de ses clients s_i , avec un facteur de décroissance qui dépend de la fonctionnalité et du comportement de s_j . Par exemple, les clients d'un fournisseur de données qui met à jour ses informations très fréquemment peuvent considérer que son score d'utilisation décroît très rapidement, car ses données sont utiles seulement pendant une courte période.

Exemple 8. *Le service s_1 (Google News) est intéressé par toutes les nouvelles produites par le service s_2 (Le Monde). Puisque Le Monde met à jour ses nouvelles tous les jours, Google News doit l'appeler quotidiennement pour maintenir un score d'utilisation maximale de $u_{21}(t) = 1$. Si le seul appel $s_1 \hookrightarrow^1 s_2$ de Google News vers Le Monde a eu lieu plusieurs jours avant l'instant 10, l'utilisation de Le Monde pour Google News est inférieure à 1, reflétant le fait qu'il a probablement manqué plusieurs nouvelles.*

4.2.3 Utilisation d'appel

L'utilisation de service ne prend pas en compte les relations logiques ou temporelles possibles entre les appels entrants et les appels sortants d'un service s_i . Nous étendons cette notion d'utilisation aux appels de service en estimant l'utilisation des appels de service sortants $s_k \hookrightarrow^t s_j$ par les appels de service entrants $s_i \hookrightarrow^{t'} s_k$.

Exemple 9. Dans l'exemple de la figure 4.2, le service 20 Minutes a reçu un appel de la part de Google News à l'instant 8 et a émis deux appels à destination du service AFP avant (à l'instant 6) et après (à l'instant 10) cet appel. Le score d'utilisation de ces appels à destination de AFP pour les appels émis par Google News calculé par le service 20 Minutes peut alors prendre en considération le fait que les appels de service entrants utilisent uniquement des résultats obtenus avant ces appels. En particulier, l'appel de service à l'instant 8 utilise uniquement des résultats récupérés par 20 Minutes de la part de AFP à l'instant 6 et l'appel de service à l'instant 10 est inutile pour cet appel.

Pour chaque triplet de services (s_i, s_k, s_j) il existe une fonction qui calcule un score d'utilisation des appels $u_{jki}(t)$ de s_j pour les appels reçus de s_i à travers s_k à un certain instant t . Cette fonction combine les informations sur les appels de service entrants dans $\Lambda(t, s_i, s_k)$ avec les informations sur les appels de service sortants dans $\Lambda(t, s_k, s_j)$ d'un même service s_k :

Définition 13 (utilisation d'appel). Soit S un ensemble de services observés par une fonction de trace Λ . La fonction d'utilisation d'appel $U_i : \mathcal{T} \times S \times S \times S \rightarrow [0, 1]$ retourne pour chaque instant t et chaque triplet de services (s_i, s_k, s_j) le score d'utilisation d'appel $U_i(t, s_i, s_k, s_j) = u_{jki}(t)$ des appels de service $s_k \xrightarrow{t'} s_j \in \Lambda(t, s_k, s_j)$ au service s_j pour les appels de service $s_i \xrightarrow{t''} s_k \in \Lambda(t, s_i, s_k)$ reçus de la part du service s_i .

Le score $u_{jki}(t)$ est dynamique et montre le degré avec lequel les appels de s_k vers s_j sont liés aux appels reçus par s_k de la part de s_i . De manière analogue à l'utilisation de service, la définition des scores d'utilisation d'appel dépend de l'application et peut comparer de différentes façons les appels de services entrants et sortants d'un service donné. Par exemple un service s_i dont les appels vers un service s_k déclenchent régulièrement des appels de s_k vers un autre service s_j , conduit à un score d'utilisation d'appel $u_{jki}(t)$ élevé. Le score d'utilisation des appels $u_{jki}(t)$ pourrait être exprimé par la proximité temporelle entre les appels de service entrants $s_i \xrightarrow{t'} s_k$ et les appels de service sortants $s_k \xrightarrow{t} s_j$ avant t .

Exemple 10. Google News a appelé Le Monde à l'instant 1. Si nous considérons que cet appel exploite seulement les nouvelles générées avant cet instant, l'utilisation des appels vers AFP émis ultérieurement est égal à 0. Dans le cas où l'utilisation d'appel exprime la part des appels vers AFP qui ont été déclenchés par les appels issus de Google News, la valeur de l'utilisation d'appel à l'instant 1 est 1, et 1/3 à l'instant 10 (nous supposons que chaque appel de service entrant a déclenché tous les appels de service sortants avec la même estampille).

Utilisation sur un chemin d'appel

L'utilisation d'appel permet de généraliser la notion d'utilisation de service en définissant le score d'utilisation sur un chemin d'appel d'un service s_j par n'importe quel service $s_i \in S$.

Considérons un service s_i qui appelle un service s_k , le service s_k appelant à son tour s_j ($s_i \rightarrow s_k \rightarrow s_j$). Nous pouvons calculer l'utilisation du service s_j par le service s_i à travers s_k en prenant en considération l'utilisation de service $u_{ki}(t)$ de s_k par s_i combinée avec l'utilisation d'appel $u_{jki}(t)$ entre les appels dans $\Lambda(t, s_i, s_k)$ et les appels dans $\Lambda(t, s_k, s_j)$. L'utilisation de service de s_j par le

service s_i à travers s_k estime l'utilisation de s_k par s_i qui est due à s_j et est calculée comme étant $u_{ki}(t) \times u_{jki}(t)$.

Plus généralement, pour un chemin d'appel $p = s_i \rightarrow s_l \rightarrow s_m \rightarrow \dots \rightarrow s_n \rightarrow s_k \rightarrow s_j$, le service s_i utilise indirectement à travers s_l tous les autres services sur le chemin p . Le score d'utilisation de s_j par s_i sur ce chemin est défini par le produit des scores d'utilisation directe et indirecte sur ce chemin :

$$u_p(t) = u_{li}(t) \times u_{mli}(t) \times \dots \times u_{jkn}(t) \quad (4.1)$$

L'utilisation sur un chemin dépend donc des appels de s_i à s_l , et de la manière dont ces appels déclenchent indirectement les appels de s_k à s_j sur ce chemin.

4.2.4 Contribution effective et importance

Notre objectif est de comparer et de classer des services en fonction de leur activité observée à l'aide d'une fonction de trace d'appels Λ combinée avec les informations sur la façon dont chaque service contribue à la qualité de tous les autres services dans un système orienté services.

Contribution et importance maximales

Nous allons étudier d'abord le calcul d'importance dans le cas où l'on ne prend pas en compte l'utilisation de services et des appels (tous les scores d'utilisation sont 1). Dans cette situation la contribution effective et l'importance de service sont *maximales*.

Nous pouvons définir la *contribution globale* d'un service donné s_j à la qualité d'un service s_i comme étant la somme des contributions sur tous les chemins dans $GC(Y)$ de s_i à s_j (de manière similaire à la définition de la contribution effective globale dans l'équation 3.2, page 52).

Définition 14 (contribution globale à la qualité). Nous notons par $\mathcal{P}_{ij}$ l'ensemble de tous les chemins reliant s_i à s_j dans le graphe de contribution $GC(Y)$. La contribution γ_{ji}^* du service s_j à la qualité de s_i est la somme des contributions de s_j à s_i sur tous les chemins p allant de s_i à s_j dans $GC(Y)$.

$$\gamma_{ji}^* = \sum_{p \in \mathcal{P}_{ij}} \gamma_p \quad (4.2)$$

À cause de la possible présence de cycles dans le graphe de contribution l'ensemble des chemins de contribution entre deux services s_i et s_j peut être infini. Lorsque les valeurs de contribution directe sont inférieures à 1, la contribution sur un chemin p décroît avec sa longueur. Nous pouvons restreindre l'ensemble des chemins entre s_i et s_j à un ensemble fini, en négligeant tous les chemins p de s_i à s_j pour lesquels $\gamma_p < \varepsilon$ pour un $0 < \varepsilon < 1$ fixé.

Exemple 11. Il existe un ensemble infini de chemins de contribution allant de Google News à AFP qui passent par 20 Minutes. La contribution de AFP à la qualité de Google News peut alors être calculée (conformément à la figure 4.1) comme étant $(0.8 \times 0.6 + 0.2 \times 0.3) \times \sum_{i \geq 0} (0.7 \times 0.2)^i = 0.63$.

L'importance maximale d'un service s_j reflète sa contribution totale à la qualité d'autres services. Elle est dépendante de la sémantique de l'application et est calculée, conformément à l'équation de l'importance d'un service (équation 3.3, page 53), comme étant :

$$I_j^{max} = \sum_{s_i \in S} \gamma_{ji}^*$$

Dans notre modèle, l'importance de chaque service s_j est comprise entre 0 et l'importance maximale I_j^{max} , en fonction de son utilisation.

Contribution effective

Dans la suite nous allons montrer l'application des différentes notions de contribution effective présentées dans le chapitre 3 afin de définir la contribution et l'importance basées sur la contribution à la qualité ainsi que l'utilisation des services et des appels.

Contribution effective directe : Pour un chemin de longueur 1 de s_i à s_j , la contribution effective directe à l'instant t (décrite par la définition 3) est calculée en pondérant la contribution locale à la qualité γ_{ji} par l'utilisation de service, c'est à dire :

$$\pi_{ji}(t) = \gamma_{ji} \times u_{ji}(t)$$

Exemple 12. Les figures 4.1 et 4.2 montrent que le service s_2 =Le Monde contribue au service s_1 =Google News avec un score $\gamma_{21} = 0.8$ et qu'il a été appelé par Google News à l'instant 1. Si l'utilisation de service $u_{21}(t)$ à l'instant $t = 10$ est estimée par l'âge du dernier appel et si le temps écoulé entre les instants 10 et 1 est de seulement quelques heures, la contribution effective de Le Monde à Google News est élevée (proche du score de la contribution à la qualité). D'autre part, même si Le Monde contribue beaucoup à la qualité de Google News, s'il a été appelé par celui-ci plusieurs jours avant l'instant 10, son score de contribution effective est faible.

Contribution effective indirecte : Nous pouvons maintenant calculer la contribution indirecte de s_j à s_i à travers s_k (décrite par la définition 4) comme étant le produit entre la contribution indirecte de s_j à la qualité de s_i à travers s_k et l'utilisation des appels de s_k vers s_j par les appels de s_i vers s_k :

$$\omega_{jki}(t) = \gamma_{jk} \times u_{jki}(t)$$

Le score de contribution indirecte de s_j à s_i à travers s_k est donc compris entre 0 et la contribution à la qualité γ_{jk} en fonction de l'utilisation des appels de s_k à s_j par les appels de s_i à s_k .

Exemple 13. Nous pouvons voir sur les figures 4.1 et 4.2 que Le Monde estime que la contribution indirecte de AFP à la qualité de Google News est égale à la contribution directe à sa propre qualité, qui est $\gamma_{42} = 0,6$. Même si cette contribution est élevée, la contribution effective $\gamma_{42} \times u_{421}(10)$ est faible lorsque l'utilisation d'appel est calculée par la proximité temporelle entre les appels entrants et les appels sortants. Ceci est dû au faible score d'utilisation d'appel des appels vers AFP pour les appels émis par Google News (si l'intervalle de temps entre les instants 1 et 9 est important).

Contribution effective sur un chemin : Dans le cas de trois services qui s'appellent, $s_i \rightarrow s_k \rightarrow s_j$, la contribution effective de s_j à s_i à travers s_k est obtenue, comme nous l'avons décrit dans le chapitre 3, en multipliant la contribution effective $\pi_{ki}(t)$ de s_k à s_i par la contribution indirecte $\omega_{jki}(t)$: $\pi_{ki}(t) \times \omega_{jki}(t) = (\gamma_{ki} \times \gamma_{jk}) \times (u_{ki}(t) \times u_{jki}(t))$. La deuxième équation montre qu'afin d'obtenir la contribution effective de s_j à s_i nous prenons en compte la contribution à la qualité de s_j à s_i et l'utilisation estimée de s_j par s_i à travers s_k .

La contribution effective $\pi_p(t)$ de s_j à s_i sur un chemin d'appel $p = s_i \rightarrow s_l \rightarrow s_m \rightarrow \dots \rightarrow s_n \rightarrow s_k \rightarrow s_j$ est calculée, en appliquant la définition de la contribution effective sur un chemin d'utilisation (équation 3.1, page 52) comme étant :

$$\begin{aligned}\pi_p(t) &= \pi_{li}(t) \times \omega_{mli}(t) \times \dots \times \omega_{jkn}(t) \\ &= (\gamma_{li} \times u_{li}(t)) \times (\gamma_{ml} \times u_{mli}(t)) \dots \times (\gamma_{jk} \times u_{jkn}(t))\end{aligned}$$

Nous remarquons que cette équation peut être réécrite sous la forme :

$$\begin{aligned}\pi_p(t) &= \gamma_{li} \times \gamma_{ml} \dots \gamma_{kn} \times \gamma_{jk} \times \\ &\quad \times u_{li}(t) \times u_{mli}(t) \times \dots \times u_{jkn}(t) \\ &= \gamma_p \times u_p(t)\end{aligned}\tag{4.3}$$

L'idée principale du calcul de contribution effective dans le cas des services utilisés par appels est donc de pondérer pour chaque chemin d'appel p de s_i à s_j le score de contribution statique avec le score d'utilisation correspondant. Ainsi, pour un chemin d'appel de service donné p , la contribution effective de s_j à s_i à travers p , notée par $\pi_p(t)$, est le produit du score de contribution à la qualité γ_p et du score d'utilisation $u_p(t)$ sur ce chemin.

Notons que dans le cas où p est de longueur 1 nous obtenons la définition de la contribution effective directe. La valeur du score d'utilisation $u_p(t)$ est comprise entre 0 et 1. La contribution effective π_p de s_j à s_i sur le chemin p est donc maximale (égale à la contribution à la qualité) si nous ignorons l'utilisation que s_i fait de s_j . La contribution effective vaut 0 si s_j n'est pas (ou plus) utilisé par s_i .

Exemple 14. La contribution de AFP à Google News sur le chemin $s_1 \rightarrow s_3 \rightarrow s_4$ est calculée en pondérant la contribution à la qualité $\gamma_{43} \times \gamma_{31}$ par le score d'utilisation $u_{31}(10) \times u_{431}(10)$. La

contribution effective de AFP à Google News sur ce chemin est proche de la contribution maximale si l'utilisation de service $u_{31}(10)$ dépend de l'âge du dernier appel $s_1 \hookrightarrow^8 s_3$ (à l'instant 10 l'âge du dernier appel est de 2) et si l'utilisation d'appel $u_{431}(10)$ est élevée à cause de la proximité temporelle des deux derniers appels correspondants (8 et 10).

Importance d'un service

Un service est important si sa valeur de contribution effective à tous les autres services est élevée, plus précisément s'il contribue beaucoup à la qualité des autres services du système et si son utilisation par d'autres services est élevée.

Si nous notons par $\mathcal{P}_{ji}(t)$ les chemins d'appel de service de s_i à s_j à l'instant t , la contribution effective totale de s_j à s_i est dans ce cas :

$$\pi_{ji}^*(t) = \sum_{p \in \mathcal{P}_{ij}(t)} \pi_p(t)$$

Notons que la contribution effective totale est inférieure ou égale à la contribution totale à la qualité, en fonction de l'utilisation de service ($\pi_{ji}^*(t) \leq \gamma_{ji}^*$).

L'importance du service s_j est par conséquent comprise entre 0 et l'importance statique I_j^{max} et est calculée (conformément à l'équation 3.3, page 53) comme étant :

$$I_j(t) = \sum_{s_i \in \mathcal{S}} \pi_{ji}^*(t)$$

Nous notons que même si un service contribue beaucoup à la qualité de l'application, son importance n'est pas élevée si son utilisation est faible.

4.3 Évaluation expérimentale

Nous avons testé expérimentalement le calcul d'importance en utilisant les algorithmes synchrone et asynchrone présentés dans la section 3.4.2. Nous avons implanté les deux algorithmes en Java (JDK 1.5) sur un ordinateur AMD Turion 64 (1.6GHz, 2Gb RAM) sous l'environnement Linux SUSE 10.0. Dans les expériences suivantes, nous avons considéré un réseau de 1000 services collaborant qui ont été simulés sous la forme de fils d'exécution Java.

Nous avons généré différentes configurations de réseau (décrites plus tard) qui se distinguent par la façon dont chaque service s_i choisit les services voisins $s_j \in Out(t, s_i)$ qui contribuent à sa qualité à un instant t donné.

Les valeurs de contribution et d'utilisation ont été générées comme suit :

- (i) la contribution à la qualité est distribuée uniformément entre chaque service dans $Out(t, s_i)$:
 $\gamma_{ji} = 1/|Out(t, s_i)|$;

- (ii) afin de modéliser l'utilisation de service et d'appel de service, nous affectons à chaque arc de s_i à s_j une valeur aléatoire $\delta_{ji} \in [0, 10]$ qui représente l'ancienneté du dernier appel de s_i à s_j relativement à l'instant t . Nous considérons que tous les services utilisent les mêmes fonctions d'utilisation $u_{ji}(t) = 1 - \alpha \times \delta_{ji}$ (utilisation de service) et $u_{jki} = 1 - \alpha \times |\delta_{ki} - \delta_{jk}|$ (utilisation d'appel). Le facteur d'utilisation α contrôle l'influence de la fonction d'utilisation sur le calcul d'importance ($\alpha = 0$ signifie que tous les liens de contribution de service sont pris en compte lors du calcul en ignorant l'ancienneté des derniers appels de service).

Les valeurs d'importance des arêtes v_{ji} échangés entre les services s_i et s_j sont calculées d'après l'équation 3.5.

Dans la suite de la section, nous appelons les services dans $Out(t, s_i)$ et $In(t, s_i)$ les voisins de s_i . Chaque service s_i débute son calcul avec une valeur d'importance de 0 et stoppe son calcul après avoir atteint une *convergence locale*, c'est à dire quand l'erreur relative $|N_i - O_i|/N_i < \epsilon$ entre la nouvelle importance N_i et l'ancienne valeur O_i est inférieure à un seuil donné ϵ . L'algorithme synchrone 1 décrit à la section 3.4.2, est *synchronisé localement*, c'est à dire que chaque service se synchronise seulement avec ses voisins, par conséquent certains services peuvent converger plus vite que d'autres. Nous considérons qu'un service s_i réalise une *itération* lorsqu'il reçoit des mises à jour d'importance de tous les services dans $In(t, s_i)$. Nous avons aussi implanté un algorithme dans lequel les services sont synchronisés *globalement*, c'est à dire que tous les services réalisent les mêmes itérations en même temps.

Pour le calcul asynchrone, nous supposons qu'un service s_i réalise une itération quand il reçoit un nombre de mises à jour d'importance qui est égal au nombre de ses voisins dans $In(t, s_i)$. Dans la suite, sauf si cela est explicitement spécifié, toutes les expériences ont été effectuées avec un seuil $\epsilon = 10^{-4}$, un facteur d'utilisation $\alpha = 0$ et $\lambda = 0,85$.

4.3.1 Génération des graphes de services

Les voisins d'un service sont choisis à l'aide des quatre stratégies suivantes, conduisant à des graphes d'importance de services avec des topologies différentes :

Grappe-Max [MAX]. Ce graphe est semblable au modèle de graphe du web proposé par [8]. Chaque service choisit pour voisins avec une probabilité de 0,75 cinq services « populaires », c'est-à-dire des services qui ont déjà été choisis par beaucoup d'autres services. Les 5 services sont choisis avec une probabilité proportionnelle à leur popularité.

La figure 4.3 montre par exemple que lorsqu'on ajoute un nouveau service au système de service existant, ce dernier choisit d'appeler principalement des services populaires (ici a_1 correspond à l'appel du service le plus populaire, et a_2 d'un appel à un autre service très populaire). Même si cela est plus rare, il peut appeler des services moins populaires, comme ici l'appel a_3 .

Grappe Linear-Copying [LC]. Chaque service sélectionne aléatoirement, avec une équiprobabilité, un service « prototype » p parmi tous les services existants lors de sa création. Il choisit

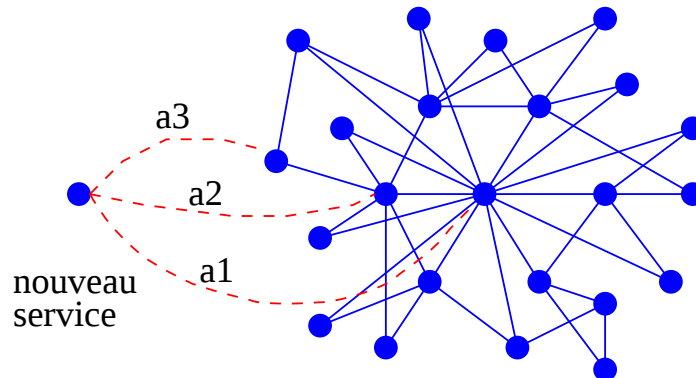


FIG. 4.3 – Exemple d’ajout d’un nouveau service pour [MAX]

ensuite avec une probabilité élevée certains des voisins du prototype, et avec une probabilité faible certains des autres services dans le graphe. Ceci est semblable au modèle de graphe du Web proposé par [118]. L’intuition est que les services choisissent avec une forte probabilité d’appeler des services « recommandés » par des services auxquels ils font confiance. Chaque service est connecté en moyenne à 5 services qui sont choisis avec une probabilité de 0,75 comme étant les services appelés par un prototype.

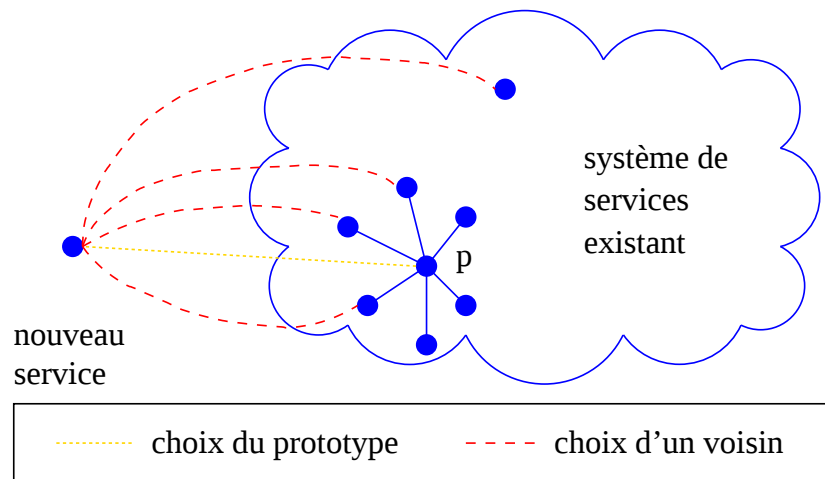


FIG. 4.4 – Exemple d’ajout d’un nouveau service pour [LC]

Nous pensons que ce modèle de graphe est approprié pour simuler des services qui intègrent des données produites par d’autres. Les services sont intéressés par les services qui sont fortement utilisés, parce qu’ils offrent probablement des données avec une qualité élevée.

Notons que la génération du graphe proposée par [118] conduit à un graphe acyclique (puisque’un service ne contacte qu’un service déjà existant). Afin de le transformer en graphe contenant des cycles, nous procédons à une deuxième étape durant laquelle des nœuds sont extraits de l’arbre

initial et réinsérés suivant le même algorithme. L'intuition derrière cette transformation de l'arbre en graphe est que quand les services sont créés ils appellent uniquement des services plus « vieux », mais comme le graphe de services évolue, ils peuvent découvrir plus tard des services nouveaux, créés après eux.

La figure 4.4 illustre l'ajout d'un nouveau service pour cette génération de graphe [LC]. Le nouveau service choisit son prototype p et ensuite il choisit aléatoirement 4 services à appeler. Comme la probabilité que ceux-ci soient choisis parmi les voisins de p est grande, on obtient ici 3 serveurs parmi les 4 voisins de p et un seul qui est un autre service du système.

Réseau Small-World [SW]. Cette configuration simule un *réseau small-world*, à savoir des communautés partageant un petit nombre de services qui s'appellent et contribuent fortement entre eux, et ont peu d'interactions avec les services d'autres communautés. Un service dans une communauté « small-world » se connecte avec une probabilité élevée à un petit nombre de voisins, qui sont choisis au hasard. La raison pour ceci est que nous supposons que les voisins sont choisis d'après les besoins du service, indépendamment d'une stratégie donnée. Les connexions entre les communautés sont aussi générés aléatoirement, avec une probabilité plus petite. Nous générons 20 communautés, chacune composées de 50 services. Chaque service dans une communauté appelle en moyenne 5 autres services. Chaque communauté interagit en moyenne avec 5 autres communautés choisies au hasard. Nous choisissons également au hasard le service au sein de la communauté qui appelle le service d'une autre communauté.

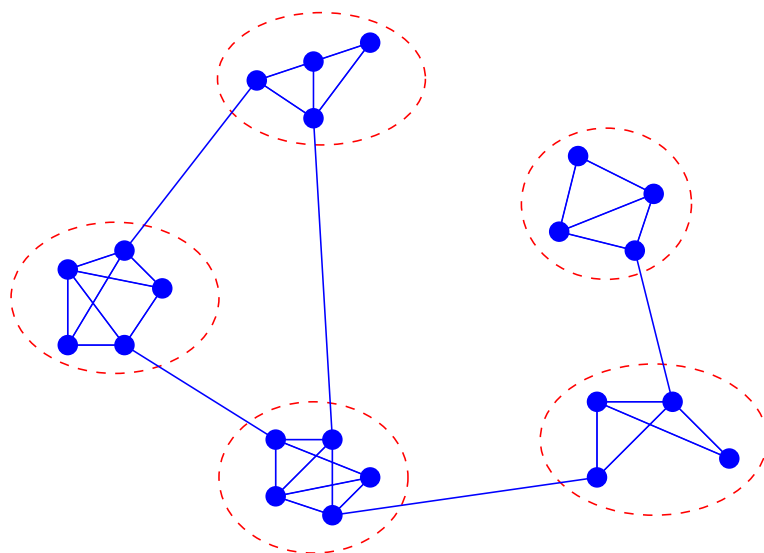


FIG. 4.5 – Exemple de graphe de services pour [SW]

La figure 4.5 montre un exemple de graphe SW avec 5 communautés de services.

Configuration Client-Serveur [CS]. Cette configuration combine les trois stratégies ci-avant afin de modéliser un environnement client-serveur dans lequel il existe beaucoup de communautés de

clients de petite taille qui appellent des services dans une communauté de serveurs, comme par exemple des communautés de services personnalisés et de serveurs avec un accès public aux données. Nous avons considéré 80 communautés de clients qui appellent les services d'une unique communauté de serveurs (SW). Chaque communauté de clients contient peu de services (ici 10) avec peu de liens entre services d'une même communauté et a un service « prototype » connecté à des services serveurs choisis aléatoirement. Les 9 autres services au sein de la même communauté sont connectés à un service de leur communauté choisi au hasard, et à au plus 5 services serveurs, chaque serveur étant avec une probabilité de 0,75 un voisin du prototype de la communauté (stratégie LC). Les serveurs sont choisis au hasard, afin de modéliser le fait que les services d'une communauté choisissent leurs serveurs en fonction de leurs besoins, indépendamment d'une stratégie quelconque de connexion. La communauté de serveurs est un graphe de type *MAX* composé de 200 services serveurs connectés en moyenne à 5 autres services serveurs.

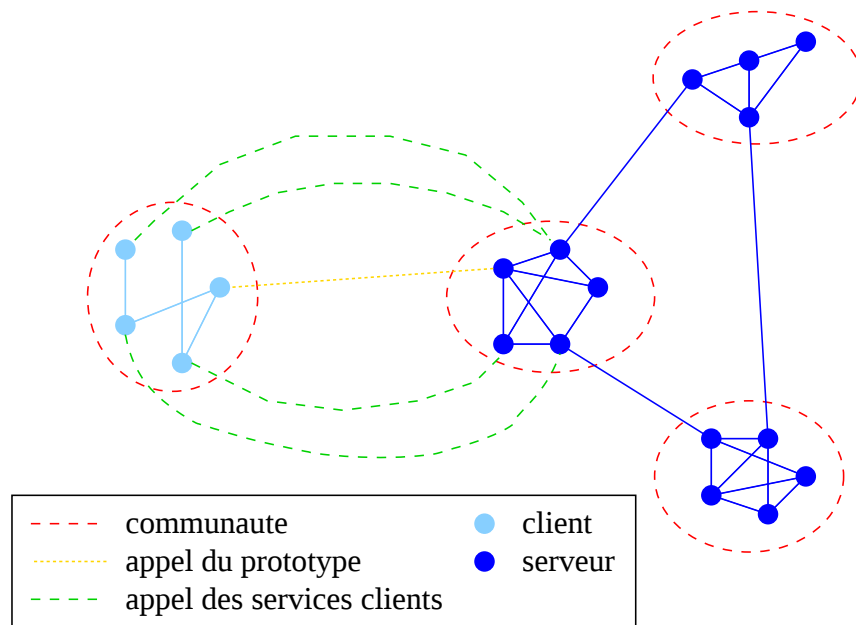


FIG. 4.6 – Exemple de graphe de services pour [CS]

La figure 4.6 montre un exemple de graphe obtenu avec cette stratégie de génération. Une communauté de clients (à gauche) choisit un prototype qui appelle un service serveur choisi aléatoirement. Les services serveurs ont également une topologie *small world* c'est à dire regroupés en communautés. Les services de la communauté du service client choisissent alors d'appeler un service appartenant à la communauté de serveur choisi par leur prototype.

4.3.2 Résultats expérimentaux

La figure 4.7 montre, pour chacune des quatre configurations, le nombre moyen de messages de calcul générés par service jusqu'à la convergence, en utilisant la synchronisation globale, la

synchronisation locale et aucune synchronisation. Cette figure montre que le nombre d'itérations pour la convergence diminue lorsque la synchronisation entre les services est relâchée.

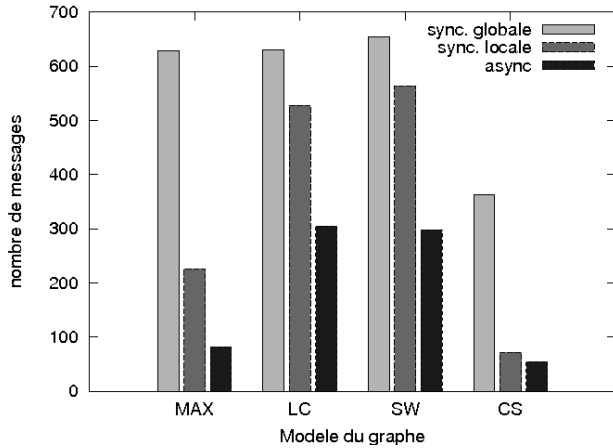


FIG. 4.7 – Nombre de messages pour différents modèles de graphe

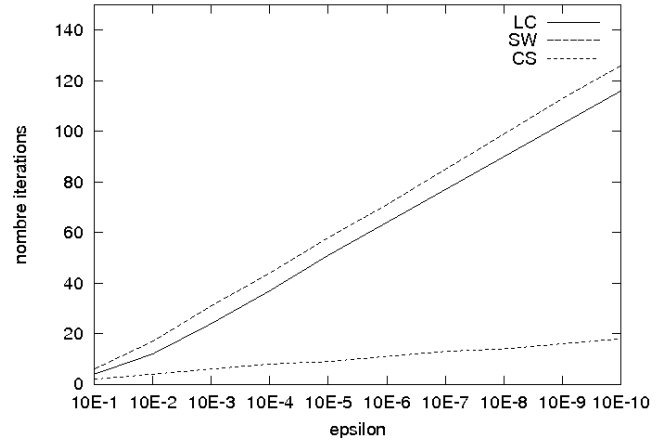


FIG. 4.8 – Nombre d'itérations pour la convergence en fonction de ϵ

Nous voyons que la synchronisation globale génère le plus grand nombre de messages puisque les services qui ont déjà convergé localement à un certain pas d'itération i continuent encore d'envoyer des messages jusqu'à ce que tous les services aient convergé.

Avec la synchronisation locale, il existe des services qui ont convergé localement et qui n'échangent plus des messages. En utilisant l'algorithme asynchrone les services convergent plus vite qu'avec l'algorithme localement synchrone (comme nous le montre la figure 4.7). La raison pour ce gain est qu'un service n'envoie pas systématiquement les valeurs d'importance qu'il vient de calculer à tous ses voisins, comme cela se fait pour l'algorithme synchrone, mais il « optimise » la communication en envoyant des nouvelles valeurs d'importance seulement de temps en temps.

Nous voyons que *SW* et *LC* génèrent plus de messages que les autres configurations. Les services dans *SW* sont connectés aléatoirement, ce qui peut conduire à la présence de nombreux cycles et des chemins de contribution longs pour beaucoup de services. Le même argument s'applique pour les services prototypes dans *LC* qui sont choisis au hasard. Au contraire, les chemins de contribution dans *MAX* sont plutôt courts puisque tous les services sont liés à des services populaires. Il y a beaucoup de services qui ne sont voisins d'aucun autre service et qui convergent très vite. Dans *CS* il y a beaucoup de petites communautés indépendantes avec peu de connexions.

La figure 4.8 montre l'influence de la valeur seuil ϵ utilisée pour la convergence sur le nombre d'itérations. Comme attendu, des valeurs faibles de ϵ conduisent à un nombre élevé d'itérations, puisque nous devons prendre en considération des chemins plus longs (avec des contributions plus faibles) avant d'atteindre la convergence. Néanmoins la croissance du nombre d'itérations est logarithmique, indépendamment du modèle de graphe. Ce résultat confirme le résultat théorique montré dans la section 3.3. Des résultats similaires ont été relevés par [31] sur la convergence de PAGERANK.

Les figures 4.9 et 4.10 montrent le taux de valeurs d'importance qui ont atteint leur classement final en fonction du nombre d'itérations.

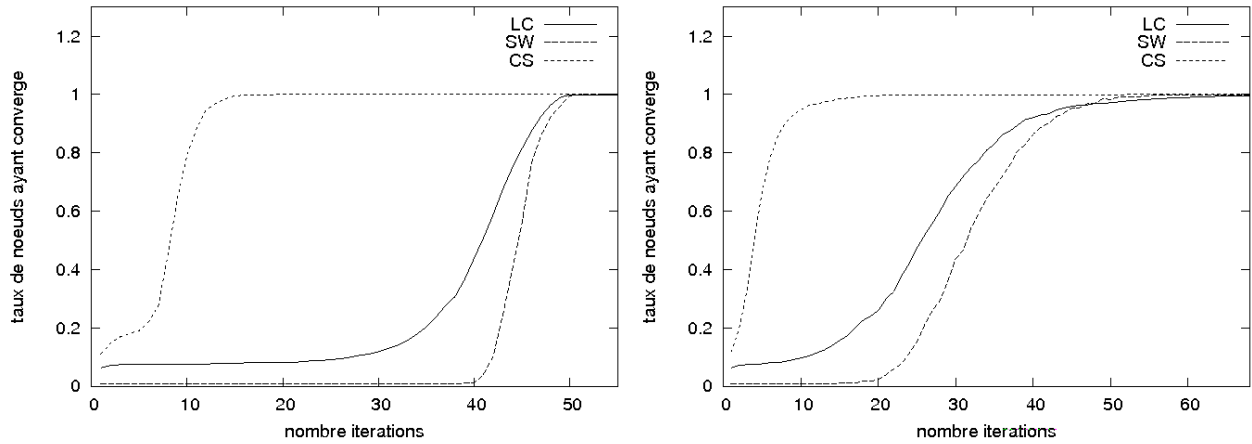


FIG. 4.9 – Taux de nœuds ayant convergé en fonction du nombre d'itérations (algorithme synchrone) FIG. 4.10 – Taux de nœuds ayant convergé en fonction du nombre d'itérations (algorithme asynchrone)

Avec l'algorithme synchrone (figure 4.9) pour *LC* (resp. *SW*) seulement 10 % (resp. 1 %) des services ont convergé après 30 itérations. La convergence globale est atteinte après environ 50 itérations. Cela peut s'expliquer par la connectivité élevée des communautés small-world qui conduit à un grand nombre de chemins possibles à explorer. Au contraire, les services dans *CS* convergent plus rapidement (après 10 itérations, presque tous les services ont convergé) puisque les services clients sont regroupés en beaucoup de petites communautés.

En comparant la figure 4.9 avec la figure 4.10 nous remarquons tout d'abord que la suppression de la synchronisation permet aux services de converger plus rapidement. La raison est qu'un service envoie dans chaque message d'importance plus d'informations sur les chemins de contribution que dans les algorithmes synchrones, ce qui accélère la convergence. En d'autres termes, avec l'algorithme asynchrone, les services « apprennent » plus rapidement quels sont tous leurs chemins de contribution.

Nous avons également réalisé plusieurs expériences pour illustrer l'impact de la fonction d'utilisation sur le calcul d'importance. Sur la figure 4.11 nous voyons que le nombre d'itérations décroît lorsque la valeur de α augmente, indépendamment du modèle du graphe. Ce résultat était attendu puisque des petites valeurs d'utilisation conduisent à une faible connectivité ainsi que des chemins plus courts puisque la contribution à la qualité d'un service sur un chemin est réduite par le facteur d'utilisation α (par exemple, pour $\alpha = 0,2$, tous les appels de services entre $t = 0$ et $t = 5$ ne sont plus considérés lors du calcul).

Sur la figure 4.12 nous étudions l'influence du facteur d'utilisation α sur le classement des services pour trois configurations de graphe. Nous considérons comme référence le classement

obtenu pour $\alpha = 0,0$ (classement obtenu en considérant seulement les liens de contribution de service sans l'utilisation de service).

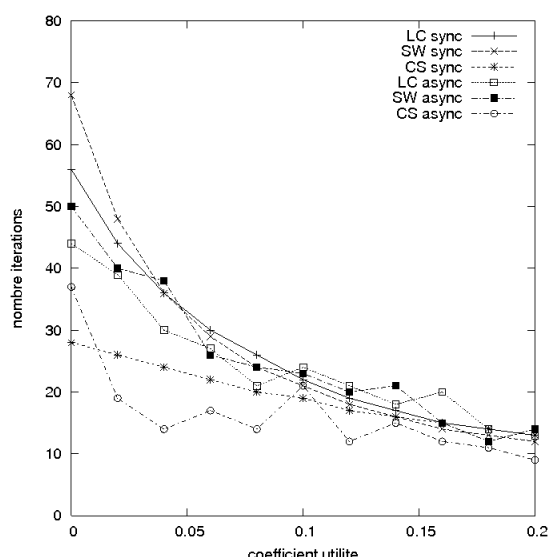


FIG. 4.11 – Nombre d'itérations pour différentes fonctions d'utilité

α	LC		SW		CS	
	10	50	10	50	10	50
0.0	10	50	10	50	10	50
0.04	9	46	7	39	10	47
0.08	8	45	6	30	10	44
0.12	7	42	4	26	10	42
0.16	6	39	2	17	8	38
0.2	4	25	2	10	4	27

FIG. 4.12 – Influence de l'utilisation sur l'ensemble des résultats

Pour différentes valeurs de α nous calculons la fraction des services qui sont encore parmi les 10 premiers et les 50 premiers. Nous voyons que l'utilisation de service a un impact fort sur le classement obtenu. Par exemple, dans la configuration SW et pour $\alpha = 0,2$ seulement 20 % des services contribuant le plus sont encore parmi les 10 (resp. 50) premiers. Pour une faible valeur de α de 0,04, l'impact sur le classement des services est faible puisque 46 services restent encore classés dans le top 50 avec la configuration LC et 47 avec CS.. Cela peut également signifier que l'importance ne doit pas nécessairement recalculée si l'utilisation d'un service ou si des appels changent.

Le tableau 4.1 illustre l'influence d'un service sur l'importance des autres services. Après un calcul d'importance initial, nous avons enlevé un nœud choisi au hasard et avons recalculé les valeurs d'importance de tous les services. Le tableau montre le nombre de services impliqués dans le recalcul de l'importance, le nombre de messages qui sont échangés pour ce recalcul et la longueur maximale des chemins qui sont pris en considération. Nous voyons qu'avec l'algorithme localement synchrone, presque tous les services sont impliqués dans le calcul d'importance, alors qu'avec l'algorithme asynchrone, seulement une partie des services dans le voisinage des nœuds retirés recalcule leur importance.

Par exemple avec $\alpha = 0,1$ et le graphe CS, seulement 94 nœuds participent au calcul avec l'algorithme asynchrone. Notons également que les longueurs des chemins et que le nombre de messages échangés sont plus faibles pour $\alpha = 0,1$. Les différences dans les longueurs des chemins et dans le nombre de messages échangés entre les algorithmes synchrone et asynchrone semblent dépendre de la topologie du graphe. Par exemple, avec le graphe LC, le nombre de messages de

	sans utilisation					
	sync			async		
	nœuds	chemin	mess	nœuds	chemin	mess
LC	938	18	16682	887	99	7670
SW	990	32	12577	450	210	16325
CS	896	200	230	101	31	648

	utilisation : $\alpha = 0,1$					
	sync			async		
	nœuds	chemin	mess	nœuds	chemin	mess
LC	938	14	12694	764	61	2979
SW	995	16	2223	256	85	2854
CS	898	126	200	94	20	245

TAB. 4.1 – Coût du recalcul lors du départ d'un pair

l'algorithme synchrone est plus important que celui de l'algorithme asynchrone pour les deux valeurs de α , alors que pour SW c'est le contraire.

4.4 Application WEBCONTENT

Dans cette section nous présentons un travail en cours sur l'application de notre modèle de classement de services à la définition d'un modèle de classement de sources de données ainsi qu'à des composants dans le cadre du projet WEBCONTENT.

Après une description du modèle de ressources WEBCONTENT, nous allons faire une présentation succincte du modèle de classement que nous avons proposé. Ce travail a fait objet d'un premier livrable qui a été fourni en juin 2007.

4.4.1 Présentation de WEBCONTENT

Le but du projet WEBCONTENT est de réaliser une plate-forme logicielle flexible et générique pour la gestion de contenu et l'intégration des technologies du Web Sémantique. La plate-forme doit être capable d'accueillir des outils nécessaires pour extraire et exploiter efficacement des informations sur le web. Les applications cibles sont les applications de type veille scientifique, technologique et économique.

Nous présentons ici le modèle WEBCONTENT et les différentes définitions s'y rattachant.

Ressources WEBCONTENT : L'information gérée dans une application WEBCONTENT est décrite dans un modèle de référence qui définit un format pivot XML unique pour représenter les différents types de ressources WEBCONTENT (documents et fragments XML, annotations RDF,

entités sémantiques, ...). Toutes les ressources sont identifiées par une URI unique et peuvent être annotées par une ou plusieurs annotations. Pour chaque document source (HTML, PDF, Word, ...) traité par la plate-forme il existe une unique ressource de type Document qui réunit les résultats de tous les traitements effectués sur le document.

Services WEBCONTENT : Un service est une ressource abstraite qui encapsule une fonctionnalité générique (par exemple, stockage et interrogation de ressources) ou une fonctionnalité métier (par exemple, service INDEXING, RELATIONSEXTRACTION). Chaque service est défini par une description de sa fonctionnalité, sa signature WSDL (types de paramètres entrées/sorties) et des pré- et post-conditions.

Processus métiers : Un processus métier est une composition de différents types de sources de données (par exemple, source HTML, source XML, source d'ontologies, ...) et de services qui est décrite sous forme d'un « graphe de tâches UML ». La composition doit respecter les contraintes liées à la signature et les pré-conditions des services composés.

Composants WEBCONTENT : Un composant est une source de données ou un logiciel qui implante un ou plusieurs services abstraits. Le même type de source ou de service peut être fourni par plusieurs composants. Chaque composant est défini par le logiciel et des paramètres de configuration (par exemple, filtre de surveillance d'une source, motif d'extraction pour l'extraction d'entités nommées, ...). Les composants sont identifiés par des URL uniques.

Workflow : Un workflow est une instance d'un processus métier avec des composants qui réalisent tous les services du processus sur les documents d'une source. Chaque service du processus est instancié par un seul composant dans un workflow. La composition peut être spécifiée « manuellement », par un langage de composition de services (BPEL) ou par d'autres techniques d'intégration de services (documents ActiveXML). Le modèle d'exécution peut être fondé sur des protocoles synchrones (RPC) ou asynchrones (messages).

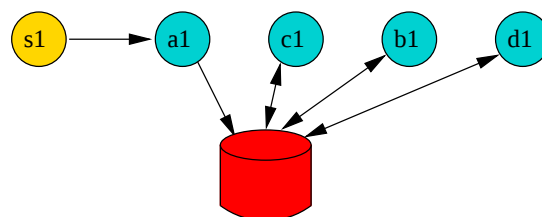


FIG. 4.13 – Workflow WEBCONTENT avec un entrepôt de données (asynchrone)

Exemple 15. La figure 4.13 montre un workflow pour un processus métier qui contient un type de sources *s* et quatre services *a*, *b*, *c*, et *d*. Le workflow utilise un entrepôt de données pour stocker les résultats intermédiaires et le résultat final. Chaque noeud du workflow représente un couple (type de sources, source) ou (service, composant). Le composant a_1 implante le service *a* et prend

en entrée des données provenant de la source s_1 de type s . Le composant c_1 implante le service c etc.

Exécution : Une exécution est une application d'un workflow à un document provenant d'une source source suivant le processus métier correspondant. Par exemple, l'exécution suivante (figure 4.14) prend comme argument un document D_1 et génère un résultat R_1 .

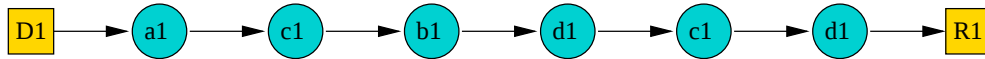


FIG. 4.14 – Exécution d'un workflow WEBCONTENT

4.4.2 Principe du modèle de classement

Un des problèmes fondamentaux dans la construction d'une application WEBCONTENT est d'une part le *choix des composants* qui implantent des services et d'autre part du *choix des sources d'information disponibles*. Nous proposons une méthode de classement des sources de données et des composants de services par leur contribution aux applications WEBCONTENT. La contribution peut être estimée pour un composant particulier ainsi que pour un ensemble de composants.

La *qualité des applications* est reflétée par la qualité des résultats produits par les exécutions de différents workflow. Nous supposons que la qualité du résultat d'une exécution de workflow dépend des sources de données ainsi que des composants de service utilisés pendant cette exécution, suivant le processus métier.

Exemple 16. Par exemple, un processus d'acquisition et d'extraction d'informations est composé d'un service WEBCRAWLING, d'un service de FORMATING, d'un service DOCUMENTSEGMENTATION et d'un service NAMEDENTITIESEXTRACTION. Chaque service de ce processus peut être implanté par un ou plusieurs composants différents et il est ainsi possible de définir différents workflow. La qualité du résultat de l'exécution d'un tel workflow dépend de la sources interrogée par le crawler et des composants avec leurs paramètres (règles de formatage, dictionnaire utilisé pour l'extraction des entités nommées etc.)

Étant donné un ensemble de composants C , il existe un ensemble $W(C)$ de workflows possibles qui contiennent tous les composants de C . Chaque exécution d'un workflow produit un résultat qui est noté par l'utilisateur. Nous estimons la *qualité* d'un ensemble de composants ainsi que son *importance* pour l'application en exploitant des notes attribués par des experts aux résultats des exécutions de workflow qui utilisent ces composants.

Qualité d'un ensemble de composants

Nous définissons pour chaque ensemble de composants C sa *qualité* $Q(C)$ qui est la moyenne des notes reçues par les résultats produits par les exécutions qui utilisent C . Un ensemble de composants C est ainsi considéré comme étant de bonne qualité si les exécutions basés sur C

produisent des bons résultats.

Nous pouvons ainsi classer des ressources WEBCONTENT en fonction de la qualité de leurs résultats :

- classement de workflows : si $Q(w) > Q(w')$, le workflow w génère en moyenne des meilleurs résultats que le workflow w' ;
- classement des composants a_j qui implantent le même service a : si $Q(a_i) > Q(a_j)$ les exécutions utilisant le composant a_i génèrent en moyenne des meilleurs résultats que les exécutions utilisant le composant a_j ;
- classement des sources de données s_j du même type s : $Q(s_i) > Q(s_j)$: les exécutions utilisant des données provenant de s_i génèrent en moyenne des meilleurs résultats que les exécutions utilisant les données provenant de la source s_j .

Importance

Un ensemble de composants C est important pour l'application s'il produit des bons résultats (reflété par un score $Q(C)$ élevé) et s'il est *utile*. La notion d'importance prend donc également en compte « l'utilité » d'un ensemble de composants dans l'ensemble total de composants. Nous associons à chaque ensemble C un score d'*utilité* P_C qui dépend du nombre relatif de workflows qui sont basés sur C . Le score P_C a des valeurs entre 0 (aucun workflow n'utilise C) et 1 (tous les workflows utilisent C).

Nous avons déduit l'importance $I(C)$ d'un ensemble de composants C en pondérant sa qualité de C par son utilité :

$$I(C) = P_C \times Q(C)$$

Nous présentons ci-dessus un exemple de classement de workflows basé sur leur qualité et leur importance.

Exemple 17. La figure 4.15 montre trois workflows w_1 , w_2 , et w_3 . Pour simplifier, nous supposons que le résultat de chaque exécution obtient une note de 0 ou de 1. Nous supposons que le workflow w_1 a produit 50% des résultats avec la note 1 et 50% des résultats avec la note 0. Le workflow w_2 n'a produit que des résultats avec la note 0 et le workflow w_3 n'a produit que des résultats avec la note 1. Nous considérons que l'utilité de chacun de ces workflows est de $1/3$, car il existe à chaque fois un seul processus métier sur les trois qui utilise un workflow donné. La qualité et l'importance de chaque workflow sont les suivantes :

$$\begin{array}{ll} Q(w_1) = 1/2 & I(w_1) = 1/2 \times 1/3 = 1/6 \\ Q(w_2) = 0 & I(w_2) = 0 \\ Q(w_3) = 1 & I(w_3) = 1 \times 1/3 = 1/3 \end{array}$$

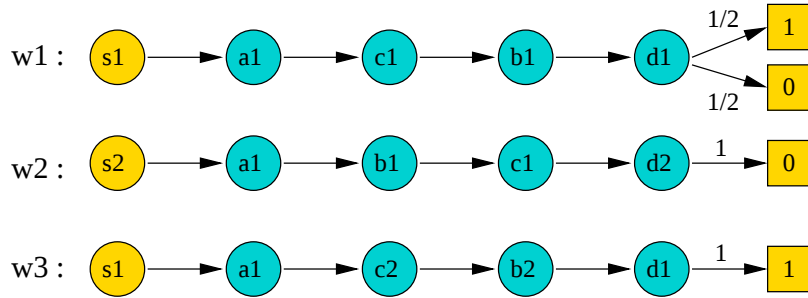


FIG. 4.15 – Plusieurs workflows WEBCONTENT

Le composant s_1 contribue aux deux workflows w_1 et w_3 alors que le composant a_1 contribue aux trois workflows w_1 , w_2 et w_3 . Leur qualité et leur importance peuvent alors être estimées de la manière suivante :

$$Q(s_1) = 3/4$$

$$P_{s_1} = 2/3$$

$$I(s_1) = 3/4 \times 2/3 = 1/2$$

$$Q(a_1) = 1/2$$

$$P_{a_1} = 1$$

$$I(a_1) = 1 \times 1/2 = 1/2$$

Le composant s_1 a la même importance que le composant a_1 , car il est moins utile même s'il produit en moyenne des meilleurs résultats que a_1 .

4.5 Conclusion

Dans ce chapitre nous avons présenté un modèle de classement de services basé sur les appels. La contribution effective entre les services est calculée à partir d'une contribution statique à la qualité de service qui est indépendante des appels. Cette contribution représente la contribution maximale d'un service et reflète la sémantique de l'application. Un service contribue effectivement à un autre service seulement s'il est appelé. Les instants d'appel de service reflètent son utilisation. Nous avons introduit deux notions d'utilisation : (i) l'utilisation d'un service, qui dépend des appels qu'il reçoit de la part de ses voisins, et (ii) l'utilisation d'appel, qui montre la dépendance entre les appels entrants et les appels sortants d'un service.

Nous avons comparé expérimentalement les performances des algorithmes synchrone et asynchrone de calcul d'importance de services. Les tests ont été réalisés sur quatre configurations de graphe de services inspirés des modèles de graphes de pages web existants dans la littérature.

Nous avons présenté également un travail qui est en cours sur l'application de notre modèle de classement de services afin de classer des sources de données et des composants dans le cadre du projet RNTL WEBCONTENT.

Chapitre 5

Stratégies de cache distribué

Les entrepôts de données distribués doivent faire face à des coûts de traitement de requêtes importants, principalement dus à la surcharge de communication élevée entre les pairs de données. La matérialisation (mise en cache) des résultats de requêtes peut réduire de manière drastique le nombre de messages et la charge du système.

Nous proposons une nouvelle stratégie de cache fondée sur la *contribution d'une mise en cache* des données à améliorer le coût global des requêtes du système. Cette contribution prend en compte le coût des requêtes ainsi que l'*utilité* de cache sur les chemins d'évaluation de requêtes, qui reflète la stratégie de cache d'autres nœuds du système. Nous estimons l'utilité sur les chemins d'évaluation à l'aide d'un algorithme distribué sans échange de messages supplémentaires.

Nous comparons les performances de notre stratégie avec celles d'une stratégie de cache existante pour des infrastructures client-serveur. Nos expériences montrent que la prise en compte des scores d'utilité réduit de manière drastique les coûts d'exécution des requêtes distribuées.

Ce chapitre est organisé comme suit. La section 5.1 introduit notre approche et présente des stratégies de cache existantes. La section 5.2 illustre un modèle abstrait de traitement de requêtes distribuées basé sur le cache. Le problème d'optimisation des requêtes distribuées basée sur le coût est introduit dans la section 5.3. La section 5.4 décrit une politique de cache existante pour les pages web que nous deployons dans un contexte distribué. La section 5.5 définit formellement notre stratégie de remplacement de cache basée sur la contribution. Nous faisons une parallèle entre notre stratégie globale et la stratégie locale, dans la section 5.6. L'implantation de notre politique de cache est expliquée dans la section 5.7. La section 5.8 montre les résultats de nos expériences sur des données et des requêtes synthétiques. La section 5.9 conclut.

Les travaux présentés dans ce chapitre ont été réalisés dans le cadre du projet ACI SEM-WEB. Ils ont fait l'objet d'une publication dans la conférence internationale WISE [11] ainsi que dans la conférence nationale BDA [10].

5.1 Introduction

Les stratégies de cache sont des technique largement adoptées dans les applications de traitement de données distribuées afin de réduire les coûts de calcul et de communication. Du côté du serveur, la mise en cache de données évite de recalculer les données générées dynamiquement (résultat de requête), alors que du côté du client elle réduit les coûts de communication et augmente la disponibilité des données.

Dans une configuration pair-à-pair il est possible de définir des stratégies de cache flexibles fédérant le stockage et les calculs de différents pairs. Pouvoir combiner des politiques de cache efficaces et le traitement de requêtes distribuées est intéressant pour deux raisons. Tout d'abord, certaines requêtes sont fréquemment soumises parce qu'elles produisent des données populaires qui contribuent à de nombreux pairs et requêtes du système. Ensuite, le fait de mettre en cache les résultats d'une requête évite l'évaluation de toutes les sous-requêtes appelées directement ou indirectement pour établir ce résultat, ce qui multiplie les bénéfices de coûts réalisés.

5.1.1 Approche et contribution

Nous considérons des systèmes pair-à-pair dans lesquels les pairs évaluent des requêtes sur des données locales et distantes, de façon similaire à [4]. L'évaluation de requêtes distribuées dans ce contexte génère des séquences d'appels de requête (chemins d'évaluation de requête), où chaque requête réalise un calcul/transformation particulier sur les données qu'elle manipule.

Une approche de cache possible dans un système distribué consiste à déployer sur chaque pair des stratégies utilisées dans les applications web. Leur performance est cependant limitée car la mise en cache des résultats d'une requête influence de façon récursive l'utilisation des données mises en cache par d'autres pairs participant à son évaluation.

Notre stratégie de remplacement de cache basée sur la *contribution* prend en compte cette influence et permet à chaque pair de prendre de meilleures décisions et éviter ainsi de mettre en cache des données inutiles. Ceci est particulièrement utile lorsque le système doit s'adapter à des changements soudains de la charge (nombre) des requêtes, tels que les *flash crowds* [179], pour lesquelles aucun cache n'a été réalisé.

Nous supposons que les sous-requêtes sont définies de façon statique et que l'évaluation de requêtes n'inclut pas de décisions de routage dynamique. Bien que ces restrictions réduisent l'impact de notre travail, ce dernier fournit un premier pas vers la compréhension du problème de la redondance de cache.

En résumé, les contribution de ce travail sont les suivantes :

- (i) un modèle d'évaluation et de coût basé sur les stratégies de cache pour des requêtes distribuées,
- (ii) une politique de remplacement de cache basée sur la contribution estimée à diminuer le coût des requêtes du système,
- (iii) un algorithme en-ligne distribué exploitant efficacement les messages d'exécution de requête existants afin d'estimer la redondance de cache globale et calculer l'importance
- (iv) des expériences et simulations évaluant les performances de notre approche.

5.1.2 Travaux existants

La matérialisation temporaire et la réplication de données ont été étudiées dans différents domaines d'applications et appliquées dans des architectures web clients-serveurs [150], bases de données distribuées [142], gestion de données mobiles [125], intégration de données [91] et entrepôts de données distribués [4]. Dans ces domaines d'application, les stratégies de cache sont confrontées aux problèmes dus à la taille limitée de la mémoire de cache et au manque de connaissances sur la sémantique de l'application. L'objectif est de définir des politiques de remplacement de cache efficaces qui peuvent être déployées facilement et qui n'ont pas besoin d'un travail d'administration complexe. Par conséquent, le placement en cache est généralement considéré comme étant bien adapté aux applications à large-échelle (web, P2P) pour lesquelles des solutions plus sophistiquées de réplication de données ou de matérialisation de vues deviennent non-applicables.

Politiques de remplacement de cache web :

Le remplacement de cache correspond à un problème d'allocation dynamique d'un espace mémoire limité pour la matérialisation d'un ensemble d'objets (pages web, documents) potentiellement grand. Toutes les techniques de remplacement de cache [150] s'appuient sur des heuristiques afin de choisir les pages web à évincer du cache, comme LRU (*least recently used* - utilisée le moins récemment), LFU (*least frequently used* - utilisée le moins fréquemment) ou LLF (*lowest download latency* - latence de téléchargement la plus basse). Parmi les heuristiques ayant rencontré le plus de succès, il y a les extensions de l'algorithme dual-glouton (*Greedy-Dual*, *GD*) [198] qui prennent leurs décisions de remplacement de cache en utilisant sur une fonction de coût. Par exemple *Greedy-Dual-Size* (*GDS*) [44] prend en compte la taille, le coût d'acquisition et l'*ancienneté du cache* des pages web à mettre dans le cache. Des extensions de *GD* sont proposées dans [51, 28, 105]. La méthode proposée par [28], appelée *GDSP*, est une extension de *GDS* et prend en compte la fréquence des accès récents afin de calculer l'utilité de placer en cache chaque objet. *GreedyDual** [105] est une extension plus complexe qui ajoute la notion de popularité d'un document pondérée par un facteur de corrélation pour les objets d'égales popularité. Nous déployons la méthode de *GDSP* dans un contexte distribué et la comparons à notre stratégie globale basée sur la contribution

Cache web distribué :

Les stratégies de cache collaboratif améliorent les performances d'une application en partageant l'espace de cache distribué et la bande passante entre les différents nœuds de l'application. Ceci génère des problèmes bien connus de réplication de données et requiert en général des décisions de routage de requêtes et de stockage efficaces. [117] introduit un proxy de cache hiérarchique coopératif qui minimise les coûts d'accès en plaçant les fichiers proche des clients. *Akamai* [67] distribue l'espace de cache sur un large ensemble de serveurs dédiés afin d'éviter les arrivées de requêtes massives soudaines (*flash crowds*, correspondant à une surcharge non-prévue du serveur). Il alloue plus de serveurs aux sites qui font face à de grandes charges. [202] montre expérimentalement que le proxy de cache collaboratif exploitant les caches des clients

dans un système P2P améliore les performances lorsque la taille des espaces de cache est limitée. BuddyWeb [185] est un système de cache web collaboratif implanté au-dessus d'un système P2P. Le réseau de pairs peut se reconfigurer dynamiquement afin de former des communautés d'intérêt. *Squirrel* [102] est un cache web P2P décentralisé qui met en commun l'espace de cache des navigateurs web locaux sans nécessiter un matériel dédié et le coût d'administration associé. De même, [194] présente une politique de cache web collaboratif distribué qui utilise l'espace de cache et la bande passante supplémentaires chez les clients.

Notre stratégie de cache globale peut-être vue comme une forme de stratégie de cache collaboratif puisque la stratégie de cache d'un pair prend en compte l'utilité sur des chemins d'évaluation de requêtes qui dépend des décisions faites par d'autres pairs. La collaboration entre les pairs dans notre stratégie ne génère aucun message supplémentaire entre les pairs.

Cache de requêtes :

De plus en plus de contenus sur le web sont générés dynamiquement par les des requêtes structurées sur des bases de données relationnelles ou XML. Ceci permet d'implanter des stratégies de cache plus puissantes en étendant et en adaptant des technologies de bases de données existantes. Par exemple, *Ace-XQ* [49] est un moteur de requêtes XML de type client-serveur, qui place en cache les résultats générés dynamiquement et qui intègre des décisions de cache dans le processus d'évaluation de requêtes. Les système de mises en cache sur les niveaux intermédiaires pour les bases de données comme *DBCACHE* [128, 34] et *DBProxy* [16] améliorent la latence d'accès et le passage à l'échelle, en utilisant des techniques avancées de matérialisation de vue et de détection de correspondance de requêtes afin de afin d'optimiser l'accès aux résultats de requêtes générés fréquemment. Les services pouvant s'adapter à un grand nombre de données [139] appliquent des techniques similaires dans une configuration distribuée afin d'éviter des charges imprévisibles et fluctuantes au niveau des serveurs de données. [162] porte sur l'évaluation des requêtes d'intervalle dans un réseau P2P multidimensionnel (CAN). Il considère une base de données dont le schéma est connu par la totalité des pairs. Le cache réalisé par un pair est déterminé en fonction de l'intervalle couvert par les données associées à ce pair. *PeerOLAP* [107] propose de placer des résultats de requêtes OLAP (On-Line Analytical Processing) sur un entrepôt de données centralisé, dans le cache sur un grand nombre de clients connectés via un réseau P2P arbitraire. Les pairs, de la même façon que dans [162], n'ont pas leurs propres données locales, mais agissent seulement comme un cache distribué pour un entrepôt de données centralisé.

Notre approche permet de réduire le coût d'évaluation des requêtes distribuées. Dans un soucis de simplification, nous ne considérons pas le problème de l'inclusion de requêtes dans notre modèle de cache. Ce problème a été étudié dans le contexte des langages de requête relationnels [135] et XML [169, 131]. Notre modèle de cache est défini indépendamment d'un modèle de données et d'un langage de requête particulier, et il est assez général pour intégrer les résultats existants concernant l'inclusion de requêtes afin de décider quels résultats de requête pourrait contribuer à une requête donnée qui doit être évaluée.

Traitement de requêtes en P2P :

Les techniques de cache ont été appliquées pour l'optimisation de requêtes dans un contexte distribué. L'idée de base est que beaucoup de clients de données sont intéressés par les mêmes données distribuées ou centralisées et peuvent fournir un certain espace de cache local afin de placer en cache les résultats des requêtes demandées fréquemment et ainsi d'améliorer le coût d'évaluation des requêtes du système. Dans *PeerDB* [138], les données et les requêtes sont distribuées et chaque pair peut mettre localement en cache les réponses retournées par des nœuds distants afin de réduire le temps de réponse pour les requêtes ultérieures. Le remplacement du cache est géré localement en appliquant la politique LRU. La redondance de cache est minimisée en mettant en cache sur chaque pair seulement une copie d'une relation pour chaque nœud source. Le problème de la redondance sur un chemin d'évaluation n'est pas considéré. [152] décrit une architecture de traitement de requête incluant un mécanisme de cache collaboratif dans lequel les résultats des requêtes sont éventuellement découpés et mises en cache par un groupe de pairs, en fonction d'une fonction de coût. DiCAS [184] propose un mécanisme de cache distribué dans lequel les valeurs d'index d'un pair sont mises en cache par un groupe de pairs « voisins ». Les requêtes sont transmises seulement aux pairs ayant la plus grande probabilité d'avoir en cache le résultat cherché. [179] propose une méthode de réplication pour un réseau P2P non-structuré, qui crée ou supprime à la demande des répliques sur les pairs de façon décentralisée. De manière similaire à [107], les pairs placent en cache les données au nom des autres pairs qui les demandent explicitement. Dans notre cas, les pairs ne demandent pas explicitement à d'autres pairs de mettre des données en cache pour eux, mais ils envoient au lieu de cela un score de matérialisation en même temps que les paramètres de la requête afin d'exprimer leur décisions de cache futures.

5.2 Évaluation de requêtes basée sur le cache

Système pair-à-pair

Nous considérons un système distribué de pairs évaluant des requêtes sur des données locales et distantes. Tous les pairs sont identifiés par un identifiant global unique p (par exemple une URL) et publient des ensembles mutuellement disjoints de requêtes $Q(p)$. Une requête $q \in Q(p)$ peut accéder aux données locales stockées sur p ainsi qu'aux résultats retournés par les requêtes $q' \in Q(p')$ définies par d'autres pairs p' du système. Dans la suite, nous écrirons $q@p$ pour désigner de manière explicite une requête q qui est publiée par le pair p , $q(t)$ pour désigner l'exécution de la requête q à l'instant t et $Out(q)$ pour désigner l'ensemble des requêtes dont les résultats sont nécessaires à l'évaluation de la requête q . La définition de $Out(q)$ dépend de la stratégie de routage de requêtes de p et est un problème que nous n'avons pas abordé dans ces travaux.

Nous voulons optimiser (diminuer) le coût total des requêtes posées sur ce système, qui sont appelés dans la suite la *charge (workload)* $\mathcal{W}$ du système. $\mathcal{W}$ est une séquence potentiellement infinie d'exécutions de requêtes $q(t)$ ordonnée en fonction du temps. Une stratégie de cache estime les requêtes du workload qui vont être posées en fonction des requêtes reçues dans le passé, la

charge du système à l'instant t étant définie comme :

$$\mathcal{W}(t) = \bigcup_{\tau \leq t} \{q(\tau)\}$$

5.2.1 Modèle de cache

Chaque pair p alloue un espace mémoire de taille fixe afin de stocker les résultats de requêtes reçus de la part des autres pairs. Nous faisons la distinction entre :

- (i) les données stockées localement par p ,
- (ii) les données matérialisées qui correspondent aux résultats des requêtes (temporairement) mémorisés, et
- (iii) les données virtuelles qui doivent être calculées en appelant des requêtes sur d'autres pairs.

Nous supposons que les données matérialisées et virtuelles sont identifiées par l'identifiant de la requête les ayant envoyées. Lorsqu'une requête q est évaluée à un certain moment t , l'ensemble $\mathcal{Mat}(q, t) \subseteq \text{Out}(q)$ désigne l'ensemble des résultats de requête matérialisés pour évaluer q et $\mathcal{Vir}(q, t) = \text{Out}(q) \setminus \mathcal{Mat}(q, t)$ désigne l'ensemble des données virtuelles qui doivent être obtenues de la part des pairs voisins pour pouvoir répondre à q . Le calcul de $\mathcal{Vir}(q, t)$ et $\mathcal{Mat}(q, t)$ peut être comparé au problème de l'identification des vues matérialisées dans un système de médiation de requêtes. La définition de ces ensembles dépend du modèle de données sous-jacent et du langage de requête, et pourrait prendre en compte l'inclusion de requêtes. Ce problème est aussi orthogonal au problème du cache considéré dans ce chapitre.

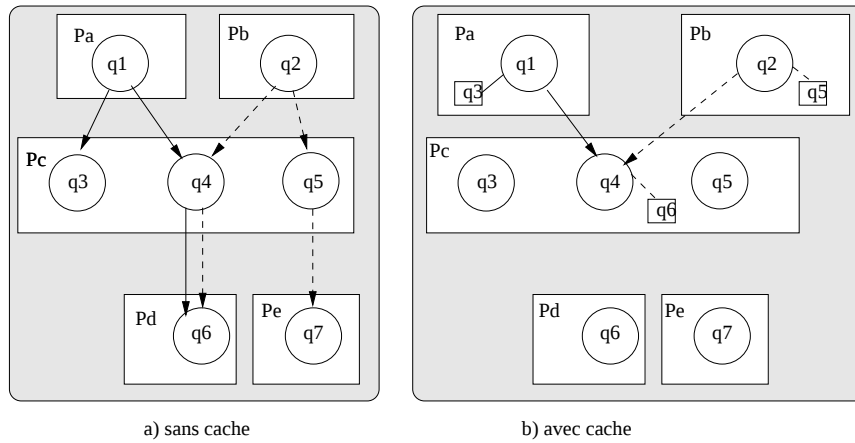


FIG. 5.1 – Traitement de requêtes distribué avec du cache

$\text{Cache}(p, t)$ désigne l'ensemble de tous les résultats de requête matérialisés au niveau d'un pair p et est appelé la *configuration locale de cache* de p à l'instant t :

$$\text{Cache}(p, t) = \bigcup_{q \in Q(p)} \mathcal{Mat}(q, t)$$

Exemple 18. La figure 5.1-a) montre cinq pairs p_a , p_b , p_c , p_d et p_e , qui publient des requêtes sur des données locales et distantes. Les données locales ne sont pas représentées sur la figure (la requête q_3 interroge seulement les données locales de p_c) et chaque arc $q_i \rightarrow q_j$ d'une requête q_i vers une requête q_j signifie que q_i déclenche l'évaluation de q_j (les estampilles temporelles ont été supprimées pour des raisons de clarté). Par exemple q_1 doit évaluer les requêtes q_3 et q_4 ($\mathcal{V}ir(q_1, t) = \{q_3, q_4\}$ à l'instant t lorsque q_1 est exécutée).

La figure 5.1-b) représente une configuration de cache différente pour laquelle le pair p_a a mis dans son cache les résultats de la requête q_3 et l'exécution de q_1 déclenche seulement celle de q_4 à l'instant t lorsqu'elle est évaluée (puisque $\mathcal{M}at(q_1, t) = \{q_3\}$ et $\mathcal{V}ir(q_1, t) = \{q_4\}$).

Dans la suite nous allons considérer pour notre exemple que les exécutions des requêtes q_1 et q_2 font partie du workload $\mathcal{W}$ que nous voulons optimiser.

5.2.2 Chemins d'évaluation de requête

L'évaluation d'une requête q à un certain moment t déclenche l'exécution des requêtes dans $\mathcal{V}ir(q, t)$ et, récursivement, l'exécution d'autres requêtes dont les résultats n'ont pas été matérialisés. Ce schéma d'exécution récursive génère pour chaque requête $q_0(t_0)$ du workload $\mathcal{W}$ posée à l'instant t_0 sur un pair un ensemble de chemins d'évaluation $\mathcal{P}(q_0, t_0)$. Chaque chemin $c \in \mathcal{P}(q_0, t_0)$ est de la forme $c = q_0(t_0) \rightarrow \dots q_i(t_i) \rightarrow \dots q_n(t_n)$ où chaque requête $q_i(t_i)$ exécutée à un instant $t_i \geq t_0$ donné est déclenchée par une requête $q_{i-1}(t_{i-1})$ exécutée à un instant t_{i-1} si et seulement si $q_i \in \mathcal{V}ir(q_{i-1}, t_{i-1})$. Par conséquent, l'ensemble des chemins d'évaluation $\mathcal{P}(q, t)$ dépend des configurations de cache locale des pairs participant à l'évaluation de q .

Exemple 19. Sur la figure 5.1-a), q_1 génère deux chemins d'exécution de requête $q_1 \rightarrow q_3$ and $q_1 \rightarrow q_4 \rightarrow q_6$ (les estampilles temporelles ont été retirées). De même, q_2 génère deux chemins $q_2 \rightarrow q_4 \rightarrow q_6$ and $q_2 \rightarrow q_5 \rightarrow q_7$. Utiliser la mémoire cache peut évidemment réduire le nombre de requêtes impliquées dans l'évaluation, comme cela est illustré sur la figure 5.1-b) où p_a matérialise le résultat de q_3 , p_b a mis en cache le résultat de q_5 et p_c a matérialisé le résultat de la requête q_6 . q_1 et q_2 génèrent alors l'un comme l'autre seulement un chemin d'exécution de requête, c'est à dire $q_1 \rightarrow q_4$ et respectivement $q_2 \rightarrow q_4$.

5.3 Optimisation du coût total des requêtes

Chaque pair p dans le système applique une politique de cache qui décide quelles sont les données qui doivent être stockées dans son cache à l'instant t , pour que le coût total des requêtes du workload $\mathcal{W}(t)$ soit minimisé.

Définition 15 (coût total du workload). Le coût total du workload $\mathcal{W}(t)$ est la somme des coûts d'évaluation des requêtes $q(\tau) \in \mathcal{W}(t)$:

$$\text{cost}(\mathcal{W}, t) = \sum_{q(\tau) \in \mathcal{W}(t)} \text{cost}_e(q, \tau) \quad (5.1)$$

où $cost_e(q, \tau)$ est le coût d'évaluation de q à l'instant $\tau \leq t$ de son exécution.

5.3.1 Coût d'une requêtes

Le coût d'évaluation d'une requête q exécutée à un instant t prend en compte : (i) le coût de calcul local de q sur le pair p qui l'évalue, (ii) le coût d'évaluation de toutes les sous-requêtes q' dans $\mathcal{V}ir(q@p, t)$, (iii) ainsi que des coûts de communication pour transmettre les résultats des requêtes q' au pair p .

Définition 16. (coût d'évaluation d'une requête) Soit $\mathcal{V}ir(q@p, t)$ l'ensemble des résultats virtuels de requête nécessaires pour évaluer $q@p$ à un instant t donné. Le coût d'évaluation de $q@p$ à t est noté $cost_e(q, t)$ et est défini récursivement comme suit :

$$\begin{aligned}
 cost_e(q, t) = & \overbrace{\alpha \times comp(q, t)}^{\text{coût d'évaluation locale}} \\
 & + \sum_{q'@p' \in \mathcal{V}ir(q@p, t)} \overbrace{cost_e(q', t')}^{\text{coût d'évaluation de } q'} + \overbrace{size(q') \times T(p, p')}^{\text{coût de communication}}
 \end{aligned} \tag{5.2}$$

où $size(q')$ désigne la taille du résultat de $q'@p'$ exécutée à l'instant t' et $T(p, p')$ désigne le taux de transfert entre p et p' . Le coût de calcul est exprimé en cycles CPU. Observons que chaque sous-requête $q'@p'$ est appelée à un instant $t' > t$.

Le poids α exprime l'importance relative du coût de calcul comparé au coût pour obtenir le résultat des sous-requêtes q' et peut être défini comme dans [4]. Dans un souci de simplification, nous considérons que le coût total de communication est constant (la taille des résultats et le taux de transferts sont considérés constants) et qu'il ne prend pas en compte le fait que les résultats des requêtes pourraient être obtenus plus rapidement en ouvrant plusieurs canaux de communication en parallèle. De même, le coût d'envoi des requêtes et l'établissement des canaux de communication ne sont pas pris en considération.

5.3.2 Influence du cache sur le coût

D'après la définition du coût d'évaluation d'une requête $cost_e(q, t)$ dans l'équation 5.2, nous pouvons remarquer que le coût de q à un instant t dépend récursivement du coût d'évaluation d'autres requêtes virtuelles q'' sur ses chemins d'évaluation. Par conséquent, le coût de q à t peut être différent à chaque exécution, en fonction de $cache(p, t)$ et des configurations de cache $cache(p'', t'')$ d'autres pairs.

Lorsque p matérialise à l'instant t les résultats d'une sous-requête $q' \in \mathcal{V}ir(q@p, t)$, le coût des requêtes locales q est réduit par le coût des résultats virtuels q' .

Définition 17 (coût des résultats virtuels). Le coût à l'instant t pour obtenir les résultats d'une requête $q' \in \mathcal{V}ir(q, t)$ est noté par $cost(q', t)$ et est calculé comme étant :

$$\text{cost}(q'@p', t) = \text{cost}_e(q'@p', t') + \text{size}(q'@p') * T(p, p') \quad (5.3)$$

Exemple 20. Nous considérons que le coût d'évaluation des requêtes sur la figure 5.1 ne prend pas en considération le coût de calcul sur les pairs correspondants ($\alpha = 0$). Sur la figure 5.1-a), $\mathcal{M}at(q_1, t)$ et $\mathcal{M}at(q_4, t)$ sont vides et le coût d'évaluation de q_1 à chaque instant t avec cette configuration est calculé comme étant le coût de communication : $\text{size}(q_3) + \text{size}(q_4) + \text{size}(q_6)$ (nous supposons un taux de transfert constant et égal à 1).

Sur la figure 5.1-b) le résultat de q_6 est matérialisé au niveau de p_c et le résultat de q_3 est matérialisé au niveau du pair p_a ce qui réduit le coût de q_1 à $\text{size}(q_4)$. Le coût de q_1 dépend donc de la configuration de cache sur son pair p_a ainsi que de celle sur le pair p_c car elle utilise la requête q_4 pendant son évaluation.

5.3.3 Politiques de remplacement de cache

Généralement, la taille du cache d'un pair est limitée comparé à la taille totale de tous les résultats des requêtes qu'il utilise. Chaque pair doit implanter une *politique de remplacement de cache* $\mathcal{R}$ qui modifie les configurations locales de cache afin de placer en cache les résultats de requête qui sont estimés comme étant utiles pour l'exécution de futures requêtes. L'évolution des configurations locales de cache d'un pair p donné dépend des requêtes qu'il exécute. Nous supposons que chaque pair peut seulement prendre des décisions de cache pour des résultats de requêtes intermédiaires à l'instant t sans avoir une connaissance explicite des requêtes exécutées après t .

Dans ce chapitre nous considérons des politiques de remplacement de cache basées sur le coût [44], dont l'objectif est de minimiser le coût total d'évaluation d'un ensemble de requêtes du workload $\mathcal{W}$. Les *scores de bénéfice de cache* $\beta(q', t)$ associés aux données de chaque requête q' , estiment l'amélioration du coût global des requêtes dans $\mathcal{W}$ lorsque les résultats de q' sont mis en cache sur un pair p . Pour un résultat d'une requête q' donnée, telle que $q' \notin \text{Cache}(p, t)$, la stratégie de remplacement $\mathcal{R}$ essaie d'identifier un ensemble $\text{Evicted} \subseteq \text{Cache}(p)$ de résultats déjà matérialisés tels que $\sum_{q'' \in \text{Evicted}} \beta(q'') < \beta(q')$ et $\text{size}(q') \leq \sum_{q'' \in \text{Evicted}} \text{size}(q'') + \text{FreeSpace}$, où FreeSpace est la taille de cache qui est déjà libre (voir l'algorithme 3 à la page 115). La stratégie met en cache les nouvelles données q' seulement si elle trouve un tel ensemble non vide de données déjà matérialisées.

Le bénéfice de cache d'un résultat q' prend en compte les paramètres suivants :

- (i) le coût pour obtenir ce résultat ; mettre en cache un résultat de requête avec un score plus élevé est plus utile que de mettre en cache des résultats avec un coût faible,

- (ii) la taille du résultat ; placer en cache un résultat de grande taille est moins avantageux que de placer un grand nombre de résultats de taille plus petite (nous considérons qu'un cache contenant plusieurs résultats différents est probablement plus utile qu'un cache ne contenant qu'un seul), et
- (iii) l'utilité de mettre en cache certaines données afin d'améliorer le coût global des requêtes du workload.

Nous utilisons la définition suivante pour le bénéfice de cache [28] :

Définition 18. (bénéfice de cache) Le score de bénéfice de cache $\beta(q', t)$ pour le résultat d'une requête $q' \in \mathcal{V}ir(q, t)$ reçu à un certain instant t est défini comme suit :

$$\beta(q', t) = u(q', t) \times \frac{\text{cost}(q', t)}{\text{size}(q')} \quad (5.4)$$

où la variable $u(q', t)$ est appelée le score d'utilité de matérialiser q' à l'instant t et $\text{cost}(q', t)$ est le coût pour calculer et obtenir les résultats de q' défini par l'équation 5.3.

Le score d'utilité de cache d'une requête q' donnée peut être obtenu suivant différentes heuristiques propres à l'application sous-jacente et à l'ensemble des requêtes du système. Dans ce chapitre nous présentons deux méthodes pour estimer le bénéfice de cache d'un résultat de requête. Dans la première méthode, le bénéfice de mise en cache prend en compte des informations locales à chaque pair sur la fréquence d'accès aux données. Une technique similaire a été proposée pour le cache de pages web dynamiques dans une configuration client-serveur [28]. La deuxième méthode calcule un bénéfice qui prend en compte des informations globales sur les décisions de cache d'autres pairs du système.

5.4 Stratégie de cache locale

Chaque pair p applique la politique de cache proposée par [28] qui estime la fréquence d'accès des résultats matérialisés et virtuels des requêtes q' qu'il appelle en utilisant un compteur qui décroît à l'aide d'une *fonction d'âge* afin de diminuer l'influence des accès anciens aux données. La fréquence et la récence d'accès aux données sont exprimées par un *score d'utilité locale* dépendant du temps, qui est défini récursivement comme suit :

Définition 19 (utilité locale). Soit $u_l(q', t^k)$ le score d'utilité locale calculé à l'accès k aux résultats de q' sur le pair p à l'instant t^k . Le score d'utilité locale $u_l(q', t^{k+1})$ calculé à l'accès $(k+1)$ aux résultats de q' est obtenu en utilisant le score précédent $u_l(q', t^k)$ comme étant :

$$u_l(q', t^{k+1}) = u_l(q', t^k) \times 2^{-(t^{k+1}-t^k)/T} + 1 \quad (5.5)$$

le score $u_l(q', t^0)$ est initialisé à 0 pour toutes les requêtes q' , l'utilité décroît avec la fonction d'âge $2^{-(t^{k+1}-t^k)/T}$ où T est la période après laquelle l'utilité est divisée par 2 si les résultats de q' ne sont plus accédés.

Le *bénéfice local* de mettre en cache les données d'une requête q' calculé par un pair p est obtenu à partir de l'équation 5.4 comme étant :

$$\beta_l(q', t) = u_l(q', t) \times \frac{\text{cost}(q', t)}{\text{size}(q')} \quad (5.6)$$

Le bénéfice de cache local ne prend pas en considération les configurations de cache d'autres pairs. Il est possible que la matérialisation des données q' par le pair p ne contribue pas à minimiser le coût des requêtes du workload $\mathcal{W}$ comme prévu, si d'autres pairs mettent aussi des données en cache pour minimiser le coût de ces requêtes. Ainsi, si un pair p_{i-1} sur le chemin d'évaluation $c = q_0@p_0 \rightarrow \dots \rightarrow q_{i-1}@p_{i-1} \rightarrow q_i@p_i \rightarrow q_{i+1}@p_{i+1} \rightarrow \dots \rightarrow q_n@p_n$ décide de mettre en cache les résultats de $q_i@p_i$, la matérialisation des résultats $q_{k+1}@p_{k+1}$ (pour $k \geq i$) par les pairs p_k est *redondante* car elle ne contribue pas à améliorer le coût de q_0 sur ce chemin.

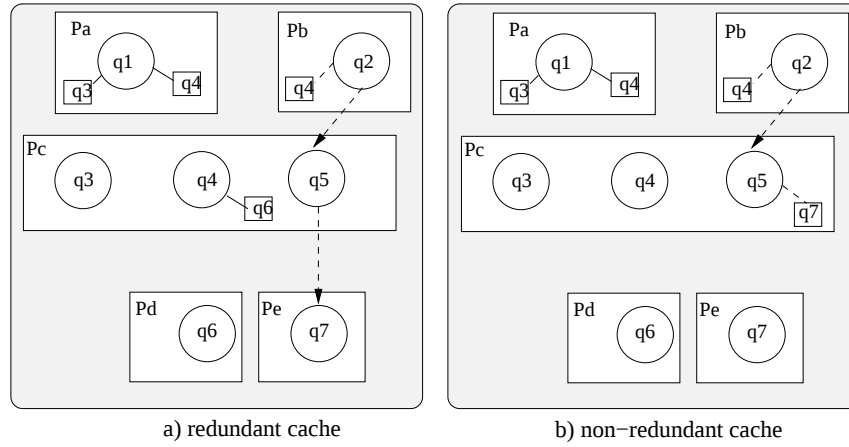


FIG. 5.2 – Cache redondant

Exemple 21. Considérons que tous les pairs appliquent un score de bénéfice local basé sur la fréquence afin de décider des résultats à mettre en cache. Supposons que p_a peut matérialiser tous les résultats, que p_b ne peut pas matérialiser les résultats de $q_5@p_c$ et que q_1 et q_2 ont été exécutées une fois. Alors l'utilité locale de la mise en cache des résultats de q_6 calculée par p_c est deux fois plus grande que l'utilité de q_7 . Par conséquent, p_c place en cache les résultats de q_6 .

Supposons que p_a et p_b mettent en cache q_4 (voir figure 5.2-a). L'évaluation suivante de $q_1@p_a$ et $q_2@p_b$ utilise les résultats mis en cache et génère des appels à q_5 et q_7 . Elle ne bénéficie donc pas de la matérialisation de q_6 par p_c . Après un certain temps, le bénéfice de q_6 qui n'a pas été utilisé devient plus petit que celui de q_7 qui continue à être utilisé, et par conséquent p_c place dans son cache le résultat de q_7 (voir figure 5.2-b)). Si le score d'utilité locale pour mettre en cache q_6 est grand, cela prend plus de temps à p_c de le supprimer du cache et de le remplacer par q_7 .

Dans la suite de ce chapitre nous allons appliquer le calcul d'importance d'un service basée sur la contribution effective, présenté dans le chapitre 3, afin de définir un *bénéfice de cache global* qui réduit les comportements de cache mutuellement redondants lors de l'évaluation de requêtes distribuées.

5.5 Stratégie de cache globale

Dans cette section nous présentons une nouvelle politique de cache distribuée pour minimiser le coût des requêtes du workload $\mathcal{W}$. Cette nouvelle politique définit pour chaque donnée q' à mettre en cache sur un pair p un *bénéfice de cache global* similaire au score de contribution totale sur les chemins d'utilisation que nous avons défini dans le chapitre 3. Le bénéfice de cache global de q' représente la *contribution totale effective* d'une matérialisation de q' à diminuer le coût de toutes les requêtes dans $\mathcal{W}$. Nous allons montrer dans la suite l'application du modèle générique de contribution et d'importance de services afin de définir des stratégies de cache pour l'évaluation des requêtes distribuées.

5.5.1 Application du modèle générique

Nous avons montré dans la section 3.3 du chapitre 3 que l'importance $v_{ji}(t)$ reçue par le service s_j de la part d'un service $s_i \in In(t, s_j)$ représente la *contribution totale* de s_j à tous les services du système sur tous les chemins d'utilisation de s_j qui passent par s_i . Ce principe est illustré dans la figure 5.3.

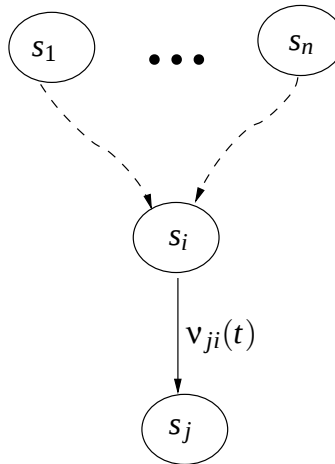


FIG. 5.3 – Graphe d'utilisation

Supposons sur cette figure que le service s_i correspond à un pair p , que les services $s_1, \dots, s_n$ correspondent aux requêtes du workload et que s_j correspond à une requête q' appelée pendant l'évaluation des requêtes de p . Dans ce cas l'importance $v_{ji}(t)$ donnée par p à q' correspond à la *contribution* de q' à toutes les requêtes du workload qui l'utilisent à travers p . Dans la suite nous définissons la *contribution* d'une requête q' comme étant la *contribution de sa mise en cache* par le pair p à réduire le coût total des requêtes du workload. Les chemins d'utilisation de services correspondent, dans notre application de cache, aux chemins d'évaluation de requêtes du workload qui utilisent les résultats de q' sur le pair p .

Trace d'accès aux résultats d'une requête

Nous définissons la *trace d'accès aux résultats d'une requête* de manière similaire à la trace d'utilisation de services que nous avons introduit par la définition 1 (page 48). Sur chaque pair p , la fonction de *trace d'accès aux résultats d'une requête* $\Lambda(t, p, q')$ enregistre les instants auxquels les requêtes de p ont utilisé les données $q'@p'$ pendant leurs évaluation jusqu'à t , indépendamment du fait qu'ils soient matérialisés ou virtuels.

Notons que la trace d'accès $\Lambda(t, p, q')$ enregistre les instants $\tau \leq t$ auxquels les chemins $c = q_0@p_0(t_0) \rightarrow \dots q_i(t_i) \rightarrow \dots \rightarrow q@p(\tau)$, d'évaluation des requêtes $q_0(t_0) \in \mathcal{W}(t)$ accèdent aux données de q' à travers des requêtes q du pair p .

Exemple 22. Par exemple, si nous considérons la figure 5.1, dans laquelle q_1 et q_2 sont des requêtes du workload, la trace d'accès $\Lambda(t, p_c, q_6)$ enregistre les instants auxquels les chemins d'évaluation de requête $q_1 \rightarrow q_4$ et $q_2 \rightarrow q_4$ ont accédé aux résultats de q_6 sur le pair p_c .

Contribution de la mise en cache d'une requête

La mise en cache des résultats de q' sur p diminue non seulement le coût d'évaluation de chaque requête locale q de p utilisant q' mais également le coût de chaque requête du workload qui utilise q' sur un chemin d'évaluation qui passe par q par le coût des résultats virtuels $\text{cost}(q', t)$ (voir l'équation 5.3).

Exemple 23. Dans la figure 5.1-b), la mise en cache des réponses de q_6 sur le pair p_c diminue le coût de q_4 , ainsi que le coût d'évaluation des requêtes du workload q_1 et q_2 sur les chemins d'évaluation qui utilisent q_6 à travers q_4 .

Nous pourrions alors estimer à l'instant t que la contribution de la mise en cache de q' sur p est grande s'il existe beaucoup de chemins d'évaluation de requêtes du workload dont les instants sont enregistrés dans $\Lambda(t, p, q')$ et si le coût $\text{cost}(q', t)$ est élevé. Nous allons montrer dans la suite que cette heuristique correspond au calcul du *bénéfice local* (équation 5.6) que nous avons montré précédemment.

En réalité, la contribution de la mise en cache de q' doit prendre en compte, pour chaque chemin d'évaluation, l'*utilité* d'améliorer le coût du workload sur ce chemin. Par exemple, si p_k décide de placer dans son cache le résultat d'une requête q_{k+1} , tous les chemins d'évaluation $q_0@p_0(t_0) \rightarrow q_1@p_1(t_1) \rightarrow \dots p_i@q_i(t_i) \rightarrow q_{i+1}@p_{i+1}(t_{i+1}) \rightarrow \dots q_n@p_n(t_n)$ s'arrêteront alors à q_k et la matérialisation de q_{i+1} sur p_i sera inutile sur ce chemin. Ceci est illustré sur la figure 5.2-a) qui montre que la mise en cache de q_6 sur p_c est inutile pour la diminution du coût sur les chemins $q_1 \rightarrow q_4$ and $q_2 \rightarrow q_4$.

Nous définissons dans la suite l'*utilité* et le *bénéfice global* de mise en cache des données d'une requête q' sur un pair p .

5.5.2 Scores d'utilité sur un chemin

Nous montrons dans cette section le calcul du score d'utilité de mettre en cache q' sur un pair p afin d'améliorer le coût du workload sur un chemin d'évaluation de requête. L'idée de base est de calculer et de propager des *scores de redondance* sur les chemins d'exécution de toutes les requêtes q_0 du système de requêtes $\mathcal{W}$.

Considérons un chemin d'évaluation $c = q_0@p_0(t_0) \rightarrow \dots q_i@p_i(t_i) \dots \rightarrow q_n@p_n(t_n)$. Chaque pair p_{i-1} sur c estime la probabilité qu'il mettra en cache le résultat de q_i envoyé par p_i . p_{i-1} combine ce score de *matérialisation locale* avec le score reçu de la part de p_{i-2} obtenant ainsi un *score de redondance sur le chemin* qui est propagé ensuite au pair p_i . Si p_i reçoit via c un score de redondance qui est proche de 0, il estime comme fortement probable qu'au moins un pair p_k « avant » p_i sur ce chemin ait mis en cache le résultat intermédiaire de $q_{k+1}@p_{k+1}$ et que l'évaluation future de $q_0@p_0$ n'atteindra plus p_i sur ce chemin. L'utilité de réaliser une mise en cache de q_{i+1} pour améliorer le coût de $q_i@p_i$ est 0 pour les évaluations futures de $q_0@p_0$ sur c .

Définition 20. (*score de redondance sur un chemin*) Soit $c = q_0@p_0(t_0) \rightarrow \dots q_n@p_i(t_n)$ un chemin d'évaluation dans $\mathcal{P}(q_0, t_0)$ pour une requête du workload q_0 donnée, et $m(q_i, t_i)$ le score de matérialisation locale reflétant la probabilité que p_{i-1} mette dans le cache le résultat de q_i (nous allons montrer dans la suite comment calculer cette probabilité). Le score de redondance $m^*(q_i, t_i)$ d'une requête q_i , $0 < i \leq n$, sur le chemin c exécutée à l'instant t_i , est défini par le produit des scores de matérialisation locale calculés par tous les pairs p_k évaluant des requêtes q_k avant q_i ($k < i$) :

$$m^*(q_i, t_i) = m(q_0, t_0) \times \dots \times m(q_i, t_i)$$

Nous considérons le score de redondance de la requête du workload $q_0@p_0$ comme étant de 1 ($q_0@p_0$ est posée par l'utilisateur).

Pour chaque chemin d'évaluation $c = q_0@p_0(t_0) \rightarrow \dots q_i@p_i(t_i) \dots \rightarrow q_n@p_n(t_n)$ le score d'utilité sur ce chemin de mettre en cache les résultats de q_{i+1} sur p_i est défini comme étant :

Définition 21. (*score d'utilité sur un chemin*) Le score d'utilité sur un chemin c de mettre en cache un résultat de requête $q_{i+1}@p_{i+1}$ est défini par le score $m^*(q_i, t_i)$ de la requête $q_i(t_i)$ utilisant le résultat de q_{i+1} à l'instant t_i : $\rho(q_{i+1}, t_i) = m^*(q_i, t_i)$.

Estimation du score de matérialisation locale

Dans la suite nous considérons un pair p_{i-1} qui appelle une requête $q_i@p_i$ lorsqu'il exécute une requête locale q_{i-1} ($q_{i-1}(t_{i-1}) \rightarrow q_i(t_i)$).

Définition 22 (score de matérialisation locale). Le score de matérialisation locale $m(q_i, t_i) \in [0, 1]$ calculé par le pair p_{i-1} pour une requête $q_i@p_i$ donnée qui est exécutée à t_i lorsqu'il évalue une requête locale q_{i-1} à l'instant t_{i-1} estime la probabilité que p_{i-1} ne matérialisera pas le résultat de q_i .

Si $m(q_i, t_i)$ vaut 0, alors p_{i-1} estime qu'il mettra certainement en cache le résultat de q_i et qu'il est inutile que p_i améliore le coût de q_i pour le chemin d'évaluation correspondant. Si $m(q_i, t_i) = 1$, alors p_{i-1} estime qu'il ne placera pas en cache le résultat de q_i . Alors, n'importe quelle mise en cache faite par p_i pour améliorer le coût de q_i est estimée comme utile pour le chemin d'évaluation de requête qui passe par p_{i-1} . Evidemment, ce score est uniquement une estimation de la décision réelle qui sera effectivement prise par p_{i-1} au moment où il reçoit q_i .

Le score de matérialisation $m(q_i, t_i) \in [0, 1]$ pourrait prendre en considération différentes propriétés (taille, fréquence d'accès) de q_i et d'autres requêtes appelées par p_{i-1} , et aussi la configuration courante et les configurations futures de cache de p_{i-1} . Néanmoins, la fonction d'estimation doit être calculée de manière efficace sans une surcharge de requêtes supplémentaire.

Dans nos expériences nous avons considéré pour le score de matérialisation une heuristique simple qui est basée sur la taille des données. Le pair p_{i-1} peut seulement mettre en cache les résultats des requêtes dont la taille est inférieure à la taille totale de son cache, qui est notée par $Total(p_{i-1})$. Par conséquent, $m(q_i, t_i) = 1$ pour toutes les requêtes q_i dont les résultats sont plus grands que la taille totale du cache. Si la taille du résultat de q_i permet de le mettre en cache, nous estimons sa probabilité de mise en cache en fonction de l'espace total occupé $Occup(p_{i-1}, t_{i-1})$ du cache lorsque q_{i-1} est évaluée et de la taille estimée du résultat de la requête q_i , comme suit :

$$m(q_i, t_i) = \begin{cases} 1 & \text{si } size(q_i) > Total(p_{i-1}) \\ 1 - \min(1, \frac{Total(p_{i-1})}{Occup(p_{i-1}, t_{i-1}) + size(q_i)}) & \text{sinon} \end{cases} \quad (5.7)$$

Chaque pair p_{i-1} estime qu'il est plus utile de mettre en cache plusieurs données de petite taille plutôt qu'une seule donnée dont la taille est plus grande. Ce score de matérialisation local considère que les probabilités de matérialisation de tous les résultats de requêtes sont indépendantes. Le score de matérialisation est calculé pour chaque requête $q_i \in \mathcal{V}ir(q_{i-1}, t_{i-1})$ appelée pendant l'évaluation de q_{i-1} , indépendamment des autres requêtes dans $\mathcal{V}ir(q_{i-1}, t_{i-1})$.

Une autre solution que nous avons étudié considère la taille de toutes les données virtuelles nécessaires à $q_{i-1}@p_{i-1}$, puisque tous ces résultats vont être candidats à être mis dans le cache de p_{i-1} . Nous notons par :

$$Vsize(q_{i-1}, t_{i-1}) = \sum_{\substack{q_k \in \mathcal{V}ir(q_{i-1}, t_{i-1}) \\ \wedge size(q_k) \leq Total(p_{i-1})}} size(q_k)$$

la taille de tous les résultats virtuels dont la taille estimée est inférieure à la taille totale de cache $Total(p_{i-1})$. Le score de matérialisation d'une requête $q_i \in \mathcal{V}ir(q_{i-1}, t_{i-1})$ exécutée à l'instant t_i est défini comme étant :

$$m(q_i, t_i) = \begin{cases} 1 & \text{si } size(q_i) > Total(p_{i-1}) \\ 1 - \min(1, \frac{Total(p_{i-1})}{Occup(p_{i-1}, t_{i-1}) + Vsize(q_{i-1}, t_{i-1})}) & \text{sinon} \end{cases} \quad (5.8)$$

Notons que le calcul ci-dessus suppose que l'ensemble $\mathcal{V}ir(q_{i-1}, t_{i-1})$ est connu avant l'évaluation de $q_{i-1}@p_{i-1}$. Nous supposons également que la taille $size(q_i)$ d'un résultat de q_i est connu par le pair p_{i-1} quand il appelle q_i . Lorsque q_i est appelée par p_{i-1} pour la première fois, $size(q_i)$ est estimée à l'aide de la taille moyenne des données placées en cache.

5.5.3 Bénéfice global

Nous estimons l'*utilité globale* de mettre q_{i+1} dans le cache de p_i en agrégeant les scores d'utilité sur tous les chemins d'évaluation dont les instants sont enregistrés dans $\Lambda(t, p_i, q_{i+1})$. Les scores d'utilité sur les chemins d'évaluation c qui utilisent q_{i+1} sont calculés à différents instants de temps. Afin de prendre en compte le caractère récent de l'utilisation des données q_{i+1} et pour donner moins de signification aux accès passés, les valeurs d'utilité sur les chemins d'évaluation sont pondérés par leur ancienneté. Nous utilisons une fonction décroissante d'ancienneté $age : \mathcal{T} \times \mathcal{T} \rightarrow [0, 1]$ telle que $age(\tau, t)$, avec $\tau \leq t$, est utilisée pour estimer l'ancienneté relative à l'instant t d'un accès réalisé à l'instant τ ($age(\tau, t)$ est non-définie pour $\tau > t$). La fonction age est décroissante avec l'intervalle de temps $t - \tau$, depuis la valeur 1 quand $\tau = t$ jusqu'à la valeur 0 lorsque $(t - \tau) \rightarrow \infty$.

Nous utilisons dans la suite la fonction d'ancienneté $age(\tau, t) = 2^{-(t-\tau)/T}$ utilisée aussi par [28] (T est la période après laquelle l'utilité est divisée par 2). Nous considérons donc qu'il est utile d'améliorer le coût du workload sur le chemin c si la probabilité que d'autres pairs sur ce chemins n'ont pas fait du cache est élevée et si cette probabilité a été calculée récemment. Le score d'utilité globale de mettre en cache q_{i+1} sur p_i dépend de son utilisation (fréquence d'accès et caractère récent) et des scores d'utilité sur les chemins d'évaluation reçus dans le passé :

Définition 23 (utilité globale). *L'utilité globale de mettre q_{i+1} dans le cache de p_i à l'instant t est définie comme étant :*

$$u_g(q_{i+1}, t) = \sum_{\tau \in \Lambda(t, p_i, q_{i+1})} age(\tau, t) \times \rho(q_{i+1}, \tau) \quad (5.9)$$

Le score d'utilité globale $u_g(q_{i+1}, t)$ des résultats d'une requête q_{i+1} donnée représente le nombre total *estimé* de chemins d'évaluation de requête qui utilisent le résultat de q_{i+1} à travers p_i jusqu'à l'instant t et qui n'ont pas mis en cache des résultats intermédiaires avant q_{i+1} .

Le *bénéfice global* de mettre en cache les données d'une requête q' calculé par un pair p est déduit à partir de l'équation 5.4 et de l'équation de l'utilité globale comme étant :

$$\beta_g(q', t) = u_g(q', t) \times \frac{\text{cost}(q', t)}{\text{size}(q')} \quad (5.10)$$

5.6 Comparaison des deux stratégies

Nous allons comparer dans cette section la méthode de cache basée sur le bénéfice local avec la méthode basée sur le bénéfice global. Pour ce faire, nous allons montrer que l'utilité globale peut être calculée récursivement de la même manière que l'utilité locale (voir l'équation 5.5) en s'appuyant sur la valeur d'utilité globale calculée lors du dernier accès au résultat de q_{i+1} par un chemin d'évaluation de requête.

Proposition 2. Soit $\text{age}(\tau, t)$ une fonction d'ancienneté définie comme auparavant, telle que $\text{age}(\tau, t) = \text{age}(\tau, t') \times \text{age}(t', t)$ pour tout $\tau \leq t' \leq t$. Soit $u_g(q_{i+1}, t^k)$ désignant le score d'utilité globale calculé lors du k -ème accès au résultat de q_{i+1} à l'instant t^k . Alors, le score d'utilité globale $u_g(q_{i+1}, t^{k+1})$ du $(k+1)$ -ème accès au résultat de q_{i+1} peut être calculé à partir du score d'utilité précédent $u_g(q_{i+1}, t^k)$ comme suit :

$$u_g(q_{i+1}, t^{k+1}) = u_g(q_{i+1}, t^k) \times \text{age}(t^k, t^{k+1}) + \rho(q_{i+1}, t^{k+1}) \quad (5.11)$$

où $u_g(q_{i+1}, t^0) = 0$ pour tout q_{i+1} .

Preuve: Supposons que la trace d'accès aux résultats q' à l'instant t est $\Lambda(t, p, q') = \{t^1, t^2, \dots, t^n\}$. A l'instant t^1 l'utilité globale est $u_g(q', t^1) = \rho(q', t^1)$ et à t^2 , l'utilité globale est $u_g(q', t^2) = \rho(q', t^1) \times \text{age}(t^1, t^2) + \rho(q', t^2)$. Ceci implique qu'à l'instant t^2 nous avons $u_g(q', t^2) = u_g(q', t^1) \times \text{age}(t^1, t^2) + \rho(q', t^2)$. A l'instant t^3 , nous pouvons calculer l'utilité globale $u_g(q', t^3) = \rho(q', t^1) \times \text{age}(t^1, t^3) + \rho(q', t^2) \times \text{age}(t^2, t^3) + \rho(q', t^3)$. L'utilité globale peut être réécrite : $u_g(q', t^3) = (\rho(q', t^1) \times \text{age}(t^1, t^2) + \rho(q', t^2)) \times \text{age}(t^2, t^3) + \rho(q', t^3)$ ce qui implique $u_g(q', t^3) = u_g(q', t^2) \times \text{age}(t^2, t^3) + \rho(q', t^3)$. Par induction sur k nous obtenons ainsi la formule dans l'équation 5.11. $\square$

Les deux manières de calculer l'utilité globale décrites par les équations 5.9 et 5.11 sont donc équivalentes. Ceci nous permet de calculer l'utilité globale de manière récursive et ainsi d'optimiser le calcul et d'économiser de la mémoire en évitant de stocker $\Lambda(t, p_i, q_{i+1})$.

Nous remarquons que l'utilité locale peut s'écrire aussi de manière équivalente comme étant :

$$u_l(q', t) = \sum_{\tau \in \Lambda(t, p, q')} 1 \times \text{age}(\tau, t)$$

Cette équation nous permet d'observer que l'utilité locale peut être déduite de l'utilité globale si nous considérons les scores d'utilité sur tous les chemins d'évaluation de requête enregistrés

dans $\Lambda(t, p, q')$ comme étant de 1. Nous déduisons donc que le score d'utilité locale $u_l(q_{i+1}, t)$ prend en compte seulement le nombre et l'ancienneté des chemins d'évaluation dans $\Lambda(t, p, q')$. Le calcul de l'utilité locale estime donc que la mise en cache de p_i des résultats de $q'@p'$ est utile pour améliorer le coût des requêtes du système sur *tous* les chemins qui utilisent q' à travers p .

5.7 Implantation

Les deux stratégies de remplacement de cache mentionnées précédemment ont été implantées par un algorithme distribué où chaque pair estime localement, en fonction de la stratégie appliquée, (i) le bénéfice local de mise en cache (équation 5.6) ou (ii) le bénéfice global de mise en cache (équation 5.10) des résultats r des requêtes q qu'il appelle. Les métadonnées nécessaires pour calculer le score de mise en cache d'un certain résultat de requête r sont désignées comme suit :

$T_a[r]$:	estampille temporelle du dernier accès à r
$U[r]$:	score d'utilité de mise en cache de r à l'instant $T_a[r]$
$S[r]$:	taille de r
$C[r]$:	coût d'obtenir r (équation 5.3)
$L_c[r]$:	ancienneté de mise en cache de r à $T_a[r]$
$B_c[r]$:	score de bénéfice cumulatif de mise en cache de r à $T_a[r]$

Ancienneté du cache et bénéfice cumulatif de mise en cache

Suivant la stratégie, $U[r]$ fait référence soit à l'utilité locale ou globale. $U[r]$ est mise à jour à chaque instant t où r est accédé de la façon suivante :

$$U[r] = U[r] \times age(T_a[r], t) + x_i$$

où $x_i = 1$ pour l'utilité locale et $x_i = \rho(r, t)$ pour l'utilité globale (voir les équations 5.11 et 5.5).

Afin d'évincer les données placées en cache qui ne sont plus accédées depuis une longue période de temps, nous utilisons une solution standard appliquée dans les politiques de remplacement de cache existantes [28]. L'idée de base est de stocker pour chaque donnée matérialisée r une *valeur d'horloge de cache* $L_c[r]$ et un *score de mise en cache cumulatif* $B_c[r] = L_c[r] + U[r] \times \frac{C[r]}{S[r]}$. $L_c[r] = B_c[r']$ correspond au bénéfice cumulatif du résultat r' évincé le plus récemment avant le dernier accès à r . Si $L_c[r] > L_c[r']$, alors r a été utilisé après r' et il y a eu au moins une modification du cache entre ces deux accès ($L_c[r]$ reflète l'ancienneté de l'accès à r en fonction de l'évolution du cache).

Gestion des métadonnées

Les métadonnées pour le calcul du bénéfice de mise en cache des résultats de requête sont stockées sur chaque pair p à l'aide de deux tables de hachage : la table *All* stocke des informations sur à la fois des données matérialisées et virtuelles, et la table *Cached* maintient des informations

seulement sur les données matérialisées. Les deux tables de hachage sont indexées par l'identifiant de résultat correspondant :

- $All[r] = (T_a[r], U[r], S[r])$ contient des informations utiles pour le calcul des scores de matérialisation ($S[r]$) et pour maintenir à jour les informations sur toutes les données accédées ($U[r]$ et $T[r]$).
- $Cached[r] = (C[r], L_c[r], B_c[r])$ retourne les métadonnées utilisées accompagnées des données dans $All[r]$ afin de calculer le bénéfice cumulatif d'un résultat de requête matérialisé r .

$All[r]$ est inconnu pour de nouveaux résultats de requête et $Cached[r]$ est inconnu pour tous les résultats de requête virtuels.

Afin d'obtenir des estimations précises, All doit stocker des métadonnées pour *tous* les résultats de requête (mis en cache ou virtuels) reçus dans le passé. La taille de cette table dépend du nombre de requêtes différentes générées par un pair donné et, en pratique, conserver cette information peut devenir vite très gourmand en espace et en temps. Différentes techniques d'approximation peuvent être utilisées pour borner la taille de All . Par exemple, les métadonnées des résultats peuvent être maintenues pour une fenêtre temporelle donnée $[T_a[r], T_a[r] + \Delta]$ où Δ est l'ancienneté maximale des résultats de requête considérés comme candidates à la matérialisation (par exemple, $\Delta = \max\{T|age(T_a[r], T_a[r] + T) > \varepsilon\}$, c'est à dire que la période maximale de temps où l'ancienneté n'est pas inférieure à un ε donné). Cette technique d'approximation nécessite un chemin d'accès optimisé à des résultats de requête obsolètes dans All . De façon similaire, la politique de cache doit disposer d'une manière efficace pour identifier les entrées avec des valeurs de bénéfice cumulatif faibles dans $Cached$, entrées qui seront considérées pour une éventuelle élimination du cache. Ceci peut être réalisé en maintenant des structures d'index appropriées comme les arbres $B+$ ou les listes triées définies sur les attributs à trier correspondants (ancienneté d'accès ou bénéfice cumulatif).

Stratégie de remplacement de cache

Dans ce qui suit, nous présentons l'algorithme distribué qui implante le remplacement du cache en s'appuyant sur la stratégie basée sur le bénéfice global qui utilise les scores d'utilité globale définis par l'équation 5.11 à la page 111. Remarquons que le même algorithme peut être appliqué pour implanter la stratégie basée sur l'utilité locale en fixant tous les scores de matérialisation $m^*[q_i]$ des requêtes q_i à la valeur 1.

Par soucis de simplicité, nous présentons l'algorithme de remplacement de cache pour des protocoles requête-réponse où chaque appel de requête $q_i@p \rightarrow q_j@p'$ correspond à un appel de procédure distant (RPC) « étendu » qui est implanté à l'aide de la procédure $CachedRPC$ (voir Alg. 2). $Clock$ désigne l'horloge temps-réel et $CacheClock$ désigne l'horloge du cache contenant le bénéfice cumulatif des données les plus récemment évincées. Cet algorithme est exécuté à chaque fois que $q_i@p$ est appelé. Le résultat r est le résultat d'une requête $q_j \in Out(q_i)$ nécessaire à

l'évaluation de q_i . A chaque accès à un résultat r placé en cache, nous mettons à jour $L[r]$ et $U[r]$, et déplaçons r à la position correspondante à sa nouvelle valeur $B_c[r]$. La liste *Cached* est déjà triée et cette opération à la complexité temporelle $O(\log(Total))$, où *Total* est la taille totale du cache. Si r n'est pas en cache (ligne 8) nous calculons m_r , le score de matérialisation locale estimé de q_j , conformément à l'équation 5.7 (page 109).

Algorithme 2 : CachedRPC($m^*[q_i]$)

```

1   $r$  = identifiant du résultat de  $q_j$ 
2   $T_a[r] = Clock$ 
3  si (Cached[ $r$ ] est défini) alors
4       $L[r] = CacheClock$ 
5       $U[r] := U[r] \times age(T_a[r], t) + m^*[q_i]$ 
6       $B_c[r] = L[r] + U[r] \times \frac{C[r]}{S[r]}$ 
7      Trier Cached par ordre croissant de  $B_c$ 
8  sinon
9      si ( $S[r] > CacheSize$ ) alors  $m_r = 0$ 
10     sinon  $m_r = 1 - \min(1, CacheSize / (OccSpace + S[r]))$ 
11      $r = call\ q_j @ p'$  avec le score  $m^*[q_i] \times m_r$ 
12      $L[r] = CacheClock$ 
13      $C[r] = \text{coût d'évaluation de } q_j @ p'$ 
14      $S[r] = \text{taille de la réponse}$ 
15     si (All[ $r$ ] est indéfini) alors
16          $B_a[r] := 0.33 \times m^*[q_i]$ 
17     sinon
18          $B_a[r] :=$ 
19          $B_a[r] \times age(T_a[r], Clock) + m^*[q_i]$ 
20      $B_c[r] := L[r] + B_a[r] \times \frac{C[r]}{S[r]}$ 
21     AppliqueStratégie( $r$ )

```

A chaque fois que le résultat de q_j est reçu, nous mettons à jour les informations locales sur sa taille et son coût. Nous donnons moins d'importance à l'accès aux données non-populaires en multipliant le premier score d'utilité par 0,33 (de façon similaire à [28]). Le score de matérialisation $m^*[q_i] \times m_r$ (conformément à la définition 20) est envoyée à q_j en même temps que les paramètres de l'appel (ligne 11 de l'algorithme), ainsi nous n'ajoutons pas de coûts additionnels.

La stratégie de cache (voir algorithme 3) choisit un sous-ensemble des résultats matérialisés *candidats* tel que le bénéfice global de *candidats* est plus petit que $B[r]$ et que la taille totale de *candidats* est plus grande que $S[r] - FreeSpace$. Il peut y avoir plusieurs sous-ensembles satisfaisant ces deux conditions suivant le bénéfice et la taille des résultats. Le bénéfice et la taille ne sont pas forcément corrélés : un résultat de petite taille pourrait avoir un bénéfice élevé et vice-versa.

Par conséquent, l'algorithme de remplacement doit explorer la totalité de l'ensemble des résultats matérialisables possibles qui pourrait être remplacé par r . Le nombre de sous-ensemble qui devrait être testé est 2^{Total} , où $Total$ est la taille du cache. Afin d'accélérer le calcul, nous utilisons la même heuristique que [107] qui consiste à tester seulement la taille et le bénéfice des données mises en cache avec les valeurs de bénéfice de mise en cache les plus faibles (en début de la liste).

Algorithme 3 : AppliqueStratégie(r)

```

1   $pos = 0$ 
2   $candidates := \emptyset$                                 /* candidats à l'éviction */
3   $B_t := 0$                                               /* bénéfice total des données évincées */
4   $B_m := 0$                                               /* bénéfice maximal des données évincées */
5   $total\_size := 0$ 
6  pour (  $(FreeSpace + total\_size) < S[r]$  ) faire
7     $r_k = Cached[pos]$                                 /* on obtient le résultat mis en cache sur  $pos$  */
8     $candidates := candidates \cup \{r_k\}$ 
9     $B_t := B_t + B_c[r_k]$ 
10    $B_m := B_c[r_k]$ 
11    $total\_size := total\_size + S[r_k]$ 
12    $pos := pos + 1$ 
13 si ( $B_t \leq B[r]$ ) alors
14    $Cached = Cached \setminus candidates \cup \{r\}$ 
15    $CacheClock := B_m$  /* CacheClock= bénéfice maximal des données évincées
    */

```

Notre politique d'admission en cache est proche de celle proposée par [28]. Alors que leur politique met toujours en cache les données accédées récemment, notre méthode place seulement en cache les données avec le bénéfice le plus important, puisque nous souhaitons améliorer le coût global des requêtes du système. Le sur-coût de la stratégie de cache globale comparée à la stratégie locale est seulement généré par les informations et calculs additionnels nécessaires à l'estimation des scores de matérialisation, c'est à dire le stockage de $S[r]$ et le calcul de m_r (ligne 9 et 10 de l'algorithme 2). Il faut également envoyer le score de matérialisation comme paramètre additionnel à l'appel de q_j . Calculer le score de matérialisation m_r en utilisant l'équation 5.8 présente un coût plus élevé, puisqu'il faut calculer la somme des tailles de tous les résultats virtuels nécessaires à q_i . Par conséquent, nous devons garder pour chaque requête q_i de chaque pair, la liste des résultats nécessaires pour son évaluation. Ceci pourrait être réalisé à l'aide d'une table de hachage associant à chaque identifiant de requête d'un pair, l'identifiant d'une requête nécessaire pour l'évaluer.

5.8 Évaluation expérimentale

Cette section présente plusieurs expériences conduites sur un environnement de simulation pour des systèmes de traitement de requêtes distribués à grande échelle. Toutes les simulations ont

été implantées en Java (JDK 1.5) et exécutées sur un ordinateur AMD Turion 64 (1.6 GHz, 2Go RAM) sous Linux SUSE 10.0.

5.8.1 Configuration expérimentale

Nous avons considéré un réseau de $P = 1000$ pairs et un ensemble de requêtes du workload $\mathcal{W} = \{q_i(t_i)\}$ qui sont uniformément distribuées sur tous les pairs du système. Chaque requête du workload génère un graphe acyclique de sous-requêtes (qui ne sont pas nécessairement des requêtes du workload). Les requêtes du workload $\mathcal{W}$ sont générées en choisissant les paramètres suivants :

N : le nombre de requêtes du système,

M : le nombre maximal de sous-requêtes voisines pour chaque requête (largeur d'une requête),

H : le nombre maximal de sauts (hops) générés lors de l'évaluation d'une requête du système (hauteur d'une requête),

F : la fréquence minimale d'exécution d'une requête (en heures),

C : la taille du cache : pour faciliter l'interprétation des résultats de la simulation, nous supposons que tous les résultats des requêtes ont la même taille (fixée à 1), et nous exprimons la taille du cache d'un pair p par un ratio du nombre total de sous-requêtes appelées par p (pourcentage).

La largeur et la hauteur maximale de chaque requête du système sont des variables générées aléatoirement, dont les valeurs sont distribuées uniformément sur les intervalles $[1, M]$ et $[1, H]$, respectivement, et chaque requête dans $\mathcal{W}$ déclenche alors en moyenne l'évaluation d'approximativement $(M/2)^{H/2}$ sous-requêtes. Toutes les sous-requêtes d'une requête $q@p$ sont des requêtes distantes fournies par un pair p' différent qui est choisi aléatoirement parmi tous les pairs du système. La probabilité de choisir une requête existante $q'@p'$ ou de créer une nouvelle requête $q_{new}@p'$ est fixée à 50%.

Sauf si c'est précisé autrement, toutes les expériences utilisent un système de $N = 10000$ requêtes avec une largeur maximale $M = 3$ et une hauteur maximale $H = 10$. Chaque requête w du workload est exécutée avec une fréquence $f(w) \in [1, F]$ mesurée en heures (chaque requête $w \in \mathcal{W}$ a donc une probabilité de $\frac{1}{f(w)}$ d'être exécutée à chaque heure). Nous fixons F à 8 pour nos expériences. Les valeurs de fréquence sont distribuées uniformément sur toutes les requêtes du workload $\mathcal{W}$ (le nombre de requêtes w pour une fréquence $f(w)$ donnée est donc $\frac{1}{F} \times N$ et le nombre total de requêtes du workload exécutées à chaque heure dans le système est donc en moyenne $\sum_{k=1}^F (1/k) \times (N/F) \sim N/F \times \ln(F)$). Le cache a par défaut une taille correspondant à un ratio de 25% des résultats de requêtes.

Les performances de chaque stratégie de remplacement de cache basée sur le coût pour un workload donné $\mathcal{W}$ sont estimées par la somme des coûts d'exécution de toutes les requêtes du système $\mathcal{W}$ jusqu'à un instant D donné (voir l'équation 5.1). Nous avons simulé l'exécution des requêtes du workload durant deux jours ($D = 48$ heures). Le taux de transfert entre chaque couple de pairs (p, p') est choisi aléatoirement dans l'intervalle $[0.01, 1.00]$. Le coût de calcul est le même

pour toutes les requêtes et est fixé à 0.001, autrement dit nous considérons que le coût d'une requête est principalement influencé par le coût de communication de résultats de ses sous-requêtes.

Politique de cache

Nous avons implanté et comparé trois mesures pour estimer le bénéfice de mise en cache du résultat r d'une requête $q(t)$ donnée :

- LOCALE [107] : le score de mise en cache est uniquement basé sur le coût et la taille des résultats de requête. Ce score est calculé de manière similaire au bénéfice local dans l'équation 5.6 (page 105), en considérant le score d'utilité comme étant constant $u_l(q, t) = 1$. Cette méthode utilise également une horloge de cache qui évince les données mises en cache qui n'ont pas été accédées depuis longtemps.
- LOCALE+ [28] : les données sont mises en cache en utilisant le bénéfice local calculé d'après l'équation 5.6. Le bénéfice local prend en compte le score d'utilité locale $u_l(q, t)$ qui estime la fréquence d'accès aux données (voir équation 5.5, page 104).
- GLOBALE (notre contribution) : le score utilisé pour la mise en cache d'une donnée est son bénéfice global, calculé avec l'équation 5.10 (page 111) qui est basé sur l'utilité globale des données $u_g(q, t)$ (équation 5.11, page 111).

5.8.2 Expériences

Comme nous l'avons montré dans l'exemple 21, les décisions de remplacement de cache prises localement à chaque pair p lors du traitement de requêtes distribuées pourraient introduire de la redondance dans l'ensemble des résultats de requêtes matérialisés. Les scores d'utilité *globale* estiment pour chaque chemin d'évaluation de requête atteignant un pair p donné, la probabilité qu'un certain pair avant p matérialise un résultat de requête intermédiaire. Une stratégie de remplacement de cache basée sur GLOBALE peut ainsi éviter plus rapidement les stratégies de cache redondantes que les stratégies LOCALE et LOCALE+ qui ne prennent pas en compte les informations sur des chemins d'évaluation. Ceci est illustré sur les figures 5.4, 5.5 et 5.7 qui montrent que les décisions de remplacement de cache de la stratégie GLOBALE sont meilleures que celles de LOCALE+ lorsque le nombre de sous-requêtes et le nombre de requêtes du workload augmentent. Cette plus grande réactivité permet également à la stratégie GLOBALE de s'adapter plus rapidement que LOCALE+ à une augmentation soudaine du nombre de requêtes dans $\mathcal{W}$ (voir figure 5.10).

Nos expériences montrent que GLOBALE offre de meilleures performances pour toutes les configurations d'expériences, à l'exception de trois situations particulières pour lesquelles la redondance est bénéfique :

- (i) quand les périodes d'expiration des données dans le cache sont courtes ; dans ce cas les données mises en cache de façon redondante pourraient contribuer à rafraîchir les résultats des requêtes (voir figure 5.9) ;
- (ii) à l'apparition de nouveaux chemins de requêtes qui peuvent profiter des données matérialisées par les stratégies de mise en cache redondantes (voir figure 5.10) ;
- (iii) dans le cas de requêtes impopulaires qui pourraient bénéficier des stratégies de cache redondantes (voir figure 5.8).

Largeur et hauteur de requêtes moyennes

Les figures 5.4 et 5.5 comparent le coût d'évaluation du système de requêtes pour les stratégies LOCALE, LOCALE+ et GLOBALE, en fonction de la largeur de requête maximale M (voir figure 5.5) et du nombre maximal de sauts H pour l'évaluation d'une requête du système (voir figure 5.4).

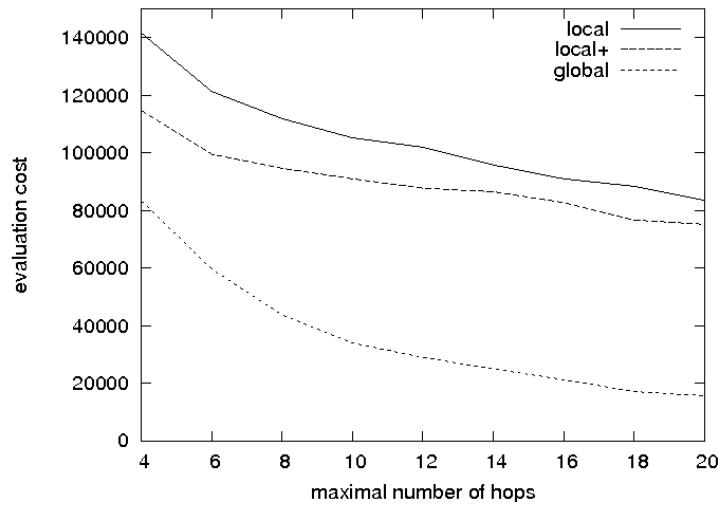


FIG. 5.4 – Nombre de sauts H variable

Le fait d'augmenter la hauteur et/ou largeur moyennes de toutes les requêtes du workload augmente le nombre total de sous-requêtes déclenchées par une requête dans $\mathcal{W}$ et, sans mise en cache, le coût d'évaluation augmenterait. Cependant, comme les deux figures le prouvent, toutes les politiques de mises en cache *diminuent* le coût d'exécution de requête dans cette situation. Ceci peut être expliqué comme suit : avec un nombre fixe de pairs et de requêtes dans $\mathcal{W}$, une augmentation de la hauteur et/ou largeur moyennes augmente la fréquence moyenne à laquelle chaque pair est interrogé et également la probabilité que les données qu'il a matérialisées soient utiles pour ses pairs voisins (les mises en cache redondantes sont réduites). Les pairs peuvent profiter de la mise en cache de leurs voisins et matérialiseront des données dont le coût n'a pas

été amélioré par ceux-ci. Ce « comportement collaboratif » est équivalent à une croissance de la capacité de cache de chaque pair et réduit donc le coût *global* du workload. Cette affirmation est également confirmée par la figure 5.6 qui montre que le coût d'évaluation *global* des requêtes du workload décroît lorsque le nombre moyen de pairs voisins augmente (la mise en cache est intéressante pour les réseaux « dense »).

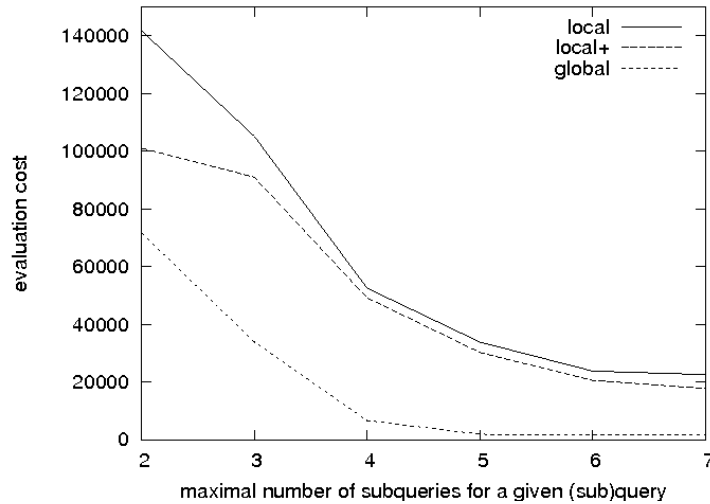


FIG. 5.5 – Nombre de voisins M variable

Notre politique de cache basée sur les scores de matérialisation de chemin suppose que l'influence d'une décision de mise en cache au niveau d'un pair p sur la décision prise par un autre pair p' décroît avec la distance entre p et p' : autrement dit placer dans le cache des données au niveau d'un pair a une plus grande probabilité d'être utile pour les pairs « proches » que pour les pairs distants. En effet, si le pair est plus éloigné il est possible qu'un autre pair situé entre eux ait mis aussi en cache des données. Ceci est aussi confirmé par les figures 5.4 et 5.5 qui montrent que la largeur de requête moyenne (voir figure 5.5) a un impact plus fort sur l'économie de coût des requêtes du système que la hauteur de requête moyenne (voir figure 5.4). Par exemple, les requêtes du système générées avec la configuration c_1 : ($M = 3, H = 20$) appellent le même nombre total de sous-requêtes qu'avec la configuration c_2 : ($M = 4.5, H = 10$) ($(3/2)^{20/2} = (4.5/2)^{10/2} \approx 58$). Néanmoins, les deux figures montrent que le coût atteint par la stratégie LOCAL+ est deux fois plus élevé avec c_1 qu'avec c_2 (ceci est également vrai pour LOCAL). Le coût du système de requêtes avec la stratégie GLOBALE est amélioré quant à lui de près de 75% dans la configuration c_2 par rapport à la configuration c_1 .

Nombre de voisins

Sur la figure 5.6, le coût d'exécution des requêtes dans $\mathcal{W}$ après $D = 48$ heures pour les trois stratégies dépend du nombre de voisins par pair autorisé ($P = 1000$).

Le coût total décroît avec le nombre croissant de voisins jusqu'à ce que ce nombre atteigne la valeur de 100 voisins par pair (c'est à dire que chaque pair connaît 0,1% de tous les pairs), puis il

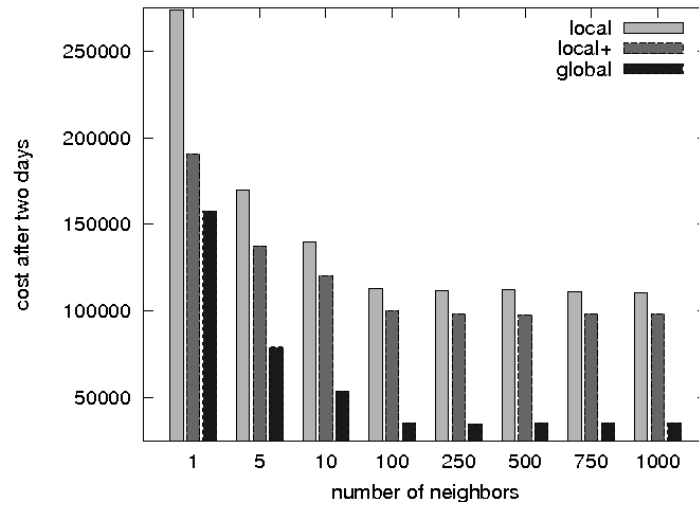


FIG. 5.6 – Nombre de voisins autorisés par pair

se stabilise. Lorsque le nombre de voisin est limité à 1, chaque pair reçoit, en moyenne, des appels d'un unique autre pair du système, et tout les chemins de requête pour lesquels un pair contribue au système de requête passent principalement par les mêmes pairs. Dans ce cas, l'amélioration obtenue par la stratégie GLOBALE comparée à celle obtenue par la stratégie LOCALE+ est d'environ 20%. En augmentant le nombre moyen de pairs voisins, l'amélioration relative entre les politiques GLOBALE et LOCALE+ atteint 65% pour 10 voisins par pair.

Passage à l'échelle (nombre de requêtes)

La figure 5.7 montre le coût total des requêtes du workload pour un nombre variable de requêtes. Pour chaque stratégie, le coût total augmente avec le nombre de requêtes du système, mais cette augmentation est beaucoup plus lente avec la stratégie GLOBALE qu'avec les deux autres. La mise en cache réalisée par GLOBALE donne donc de meilleurs résultats lorsqu'il y a beaucoup de requêtes dans le système. La stratégie LOCALE+ est meilleure que la stratégie LOCALE puisqu'elle prend en considération l'utilisation des données dans le cache. Mais elle crée également plus de mises en cache redondantes que la stratégie GLOBALE puisque les scores des données redondantes augmentent, donc LOCALE+ évince ces données du cache après une période plus longue.

Un nombre plus élevé de requêtes dans le workload augmente également la collaboration mais, en même temps, il y a plus de requêtes dont le coût doit être amélioré. Par conséquent le coût du workload augmente, contrairement aux cas précédents.

Requêtes populaires

La figure 5.8 présente le *Ratio Détaillé d'Economie de Coût (RDEC)* [107] à la fin de chaque journée pendant 7 jours pour deux configurations d'arbre de requête. Le RDEC est calculé comme

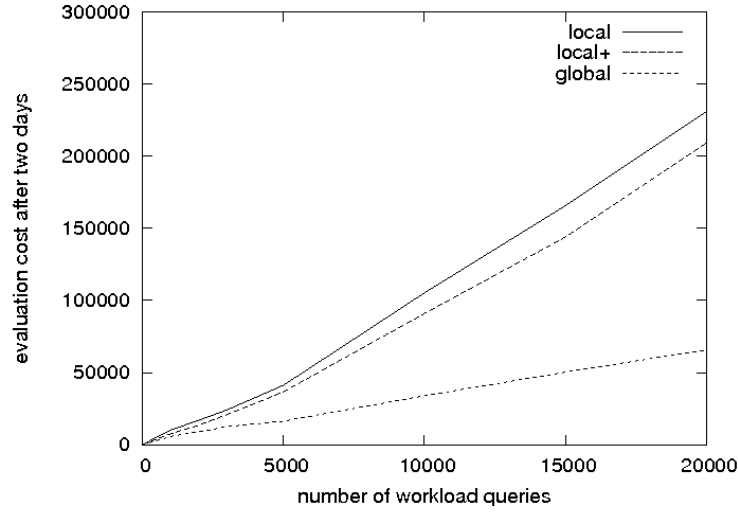


FIG. 5.7 – Nombre (N) de requêtes du système variable

suit :

$$RDEC(D) = \frac{Cost_l(\mathcal{W}, D) - Cost_u(\mathcal{W}, D)}{Cost_l(\mathcal{W}, D)} \quad (5.12)$$

où $Cost_u(\mathcal{W}, D)$ est le coût total du workload avec les stratégies LOCALE+ ou GLOBALE et $Cost_l(\mathcal{W}, D)$ est le coût avec la stratégie LOCALE. La définition de ces coûts est donnée par l'équation 5.1, pour $D = d * 24h$ ($d = 1 \dots 7$).

La première configuration, c_1 , est la configuration générée aléatoirement qui a été également utilisée ci-avant. Dans la deuxième configuration, c_2 , un sous-ensemble de requêtes (les requêtes populaires) sur chaque pair a beaucoup plus de clients que les autres requêtes du pair. Chaque pair a un sous-ensemble de résultats populaires (produits par des requêtes populaires des voisins) qui sont plus utilisés que les autres résultats pendant l'évaluation des requêtes locales de ce pair. Puisque de nombreuses requêtes de chaque pair partagent les mêmes résultats, le pair appelle peu de requêtes différentes sur d'autres pairs.

Avec la première configuration (configuration c_1 dans la figure 5.8), la différence entre les coûts des deux stratégies diminue à la fin du septième jour mais reste significative (22% contre 60%). La stratégie GLOBALE améliore plus rapidement le coût des requête du workload que LOCAL et LOCAL+ mais cette amélioration diminue car les deux stratégies locales évincent également les données inutiles du cache après une période de temps.

Dans la deuxième configuration (c_2), nous pouvons voir que l'amélioration du coût avec la stratégie GLOBALE, de même qu'avec la stratégie LOCALE+, ne change pas avec le temps. Les deux stratégies placent dans le cache dès le début les données pour les services les plus populaires. Cette situation ne change pas dans les heures suivantes lorsque d'autres requêtes du système sont

également exécutées.

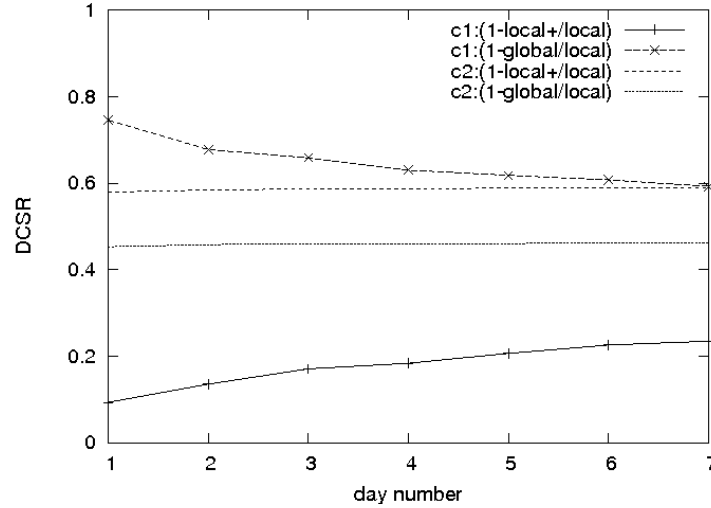


FIG. 5.8 – RDEC durant 7 jours

En d'autres mots, les requêtes populaires sont plus utilisées que les autres requêtes depuis le tout début, par conséquent la décision de mise en cache des pairs ne changent pas dans les heures suivantes. Cette configuration montre également que la stratégie GLOBALE est moins performante que la stratégie LOCALE+ (45% contre 60%). Les deux stratégies mettent en cache les données populaires, mais avec la stratégie GLOBALE il y a des données (non-populaires) qui ne sont mises en cache par aucun pair, parce que leur utilité estimée est très petite comparée à l'utilité des données populaires. Avec la stratégie LOCALE+ cette situation ne se produit pas, puisque chaque appel a une utilité estimée de 1, donc certaines données non populaires sont également mises en cache ce qui permet un gain de performance globale.

Expiration du cache

La figure 5.9 montre l'impact de l'expiration des données mises en cache sur le coût des requêtes du workload après deux jours. La période d'expiration permet de rafraîchir les données placées en cache. Nous remarquons tout d'abord que la stratégie GLOBALE donne de meilleurs résultats lorsque la période d'expiration est plus grande. Dans notre implantation, le temps d'expiration est mesuré depuis l'instant où les données ont été mises en cache, et toutes les données en cache expirent après la même période. Les données ayant expiré sont retirées du cache même si elles sont encore utilisées, et les requêtes les ayant produites sont alors appelées à nouveau si ces données sont encore nécessaires, augmentant par là-même le coût des requêtes du workload avec leur coût de ré-exécution.

Comme prévu, lorsque la période d'expiration est plus petite, le coût du workload augmente. Le coût de ré-exécution des requêtes dont les données sont expirées est à nouveau plus élevé avec la stratégie GLOBALE, à cause de l'absence de redondance dans la mémoire de cache des différents

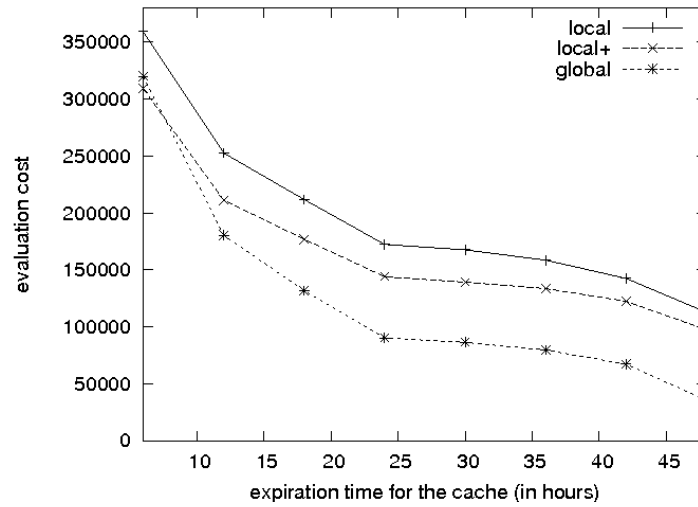


FIG. 5.9 – Expiration des données mises en cache

pairs. Par conséquent, lorsqu'une requête q est ré-exécutée, la plupart des pairs qui sont appelés n'ont pas les données nécessaires dans leur cache pour l'évaluation de q . On en déduit logiquement que lorsque la période d'expiration est très petite, l'amélioration obtenue par la stratégie GLOBALE est « masquée » par le coût de ré-exécution des requêtes dont le résultat a expiré.

Augmentation inattendue du nombre de requêtes

La figure 5.10 montre le comportement du système après une augmentation soudaine du nombre de requêtes dans le système (*flash crowd*). Cette figure illustre comment les deux stratégies de cache s'adaptent à un nombre de requêtes inattendues pour lesquelles aucune données n'a été placée en cache. Dans chaque expérience, nous exécutons $(100 - p)\%$ des requêtes du workload pendant deux jours, aux fréquences décrites ci-avant. Au bout des deux jours, nous envoyons le pourcentage $p\%$ de requêtes restantes chaque heure, durant 10 jours. Les $(100 - p)\%$ de requêtes continuent d'être envoyées avec leur fréquence respective attribuée.

Nous mesurons la somme des coûts des *nouvelles* requêtes chaque heure durant 10 jours, pour chacune des deux stratégies. Durant les premières heures du troisième jour, le coût total des nouvelles requêtes est légèrement plus élevé avec la stratégie GLOBALE. La raison est que la stratégie LOCALE+ fait des mises en cache redondantes sur les pairs sur lesquels les nouvelles requêtes sont exécutées, et une partie des nouvelles requêtes tirent un avantage de ces données placées en cache. Néanmoins durant les heures suivantes, la stratégie GLOBALE donne de meilleurs résultats que la stratégie LOCALE+.

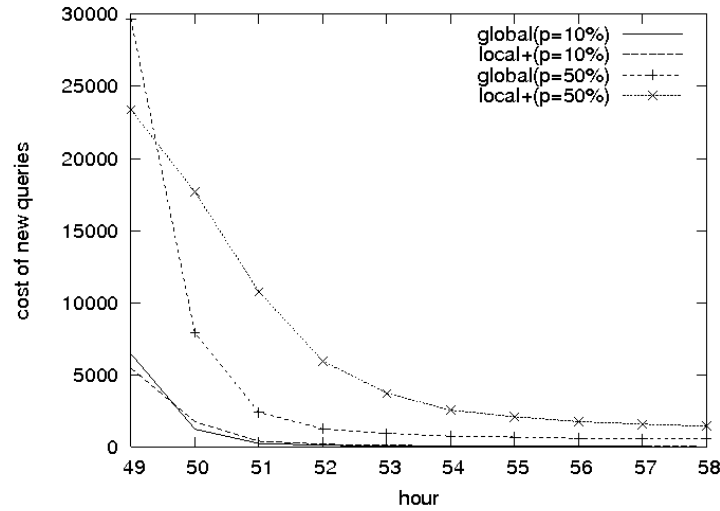


FIG. 5.10 – Coût des 10% et 50% de nouvelles requêtes

5.9 Conclusion

Dans ce chapitre nous avons présenté une stratégie de remplacement de cache afin d'optimiser l'évaluation de requêtes distribuées. Notre solution peut être considérée comme complémentaire à d'autres techniques d'optimisation de requêtes basées sur la réplication de données, la réécriture de requêtes et l'indexation de données. L'idée de base de notre approche est d'étendre le processus d'évaluation de requête à l'aide d'une politique de remplacement de cache distribuée et efficace, qui décide quels sont les résultats de requêtes qui doivent être matérialisés sur chaque pair basé sur un *bénéfice global*. Ce bénéfice représente la *contribution* estimée d'une mise en cache des données à diminuer le coût total des requêtes du système. Il prend en compte des informations locales sur la fréquence d'accès et le coût des requêtes ainsi que des estimations globales sur l'utilité d'une mise en cache.

Nous avons formellement défini la notion de score de matérialisation sur un chemin, qui estime pour chaque chemin d'exécution de requête c déclenchant l'évaluation d'une certaine requête q' , la probabilité qu'un p va mettre en cache le résultat d'une autre requête « avant » q' . Nous avons montré la manière dont ces scores peuvent être utilisés pour implanter une stratégie de remplacement de cache collaborative sans générer aucun message supplémentaire entre les pairs. Les performances quantitatives de notre stratégie ont été évaluées par des expérimentations sur des requêtes simulées.

Chapitre 6

Classement de services basé sur les données

Dans ce chapitre nous étudions le problème du classement de données et de sources de données dans un entrepôt de données XML distribué. Nous montrons la manière dont notre modèle abstrait de classement de services peut s'appliquer à des systèmes orientés-données, afin de fournir un classement pour les données et les sources de données en fonction de leur popularité. Le modèle de services et de données que nous considérons est le modèle ACTIVEXML [19]. Chaque pair publie un ensemble de services implantés comme des requêtes paramétrées sur un entrepôt de données (A)XML local intégrant des données générées localement ainsi que des résultats de requêtes reçus de la part d'autres services. L'importance d'un service ACTIVEXML basée sur sa contribution reflète *l'utilisation de ses réponses* pour l'évaluation d'autres requêtes dans une application utilisant des entrepôts de données XML distribués.

Ce chapitre est organisé comme suit : la section 6.1 introduit notre approche, la section 6.2 présente le modèle de données et la sémantique des requêtes. La notion de contribution est définie formellement dans la section 6.3. La section 6.4 décrit l'utilisation de la contribution pour le classement des données et des services. La section 6.5 présente une méthode pour estimer les contributions des sources de données dans une configuration préservant l'anonymité. La section 6.6 présente une implantation de notre modèle de classement au-dessus du système ACTIVEXML. Un travail en cours sur l'application de notre modèle aux services d'intégration et d'interrogation de flux RSS est présenté dans la section 6.7. La section 6.8 conclut.

Une partie de ces travaux a été publiée à SAC [59].

6.1 Introduction

Dans ce chapitre nous considérons des applications web intégrant des données et des services distribués à une large échelle. Les services auxquels nous nous intéressons sont des services *orientés données*, qui utilisent des données éventuellement produites par d'autres services afin de répondre à leurs clients.

6.1.1 Présentation de l'approche

Le présentation de notre modèle de classement est basée sur le modèle de données ACTIVEXML [19] qui réalise la distribution de données via la notion de *données intentionnelles* définies par des requêtes (appels de service) sur des documents XML distants. Un document sur un pair ACTIVEXML contient des appels à des services publiés sur d'autres pairs dont les réponses, qui sont des fragments (A)XML, sont insérés dans ce document. Les services publiés par chaque pair sont des requêtes paramétrées dont les résultats contiennent des fragments XML locaux et/ou des fragments provenant d'autres pairs. Par conséquent, les données d'un pair peuvent être répliquées sur d'autres pairs qui les utilisent pour l'évaluation de leurs propres requêtes. Nous illustrons l'utilisation des données dans ACTIVEXML par l'exemple dans la figure 6.1.

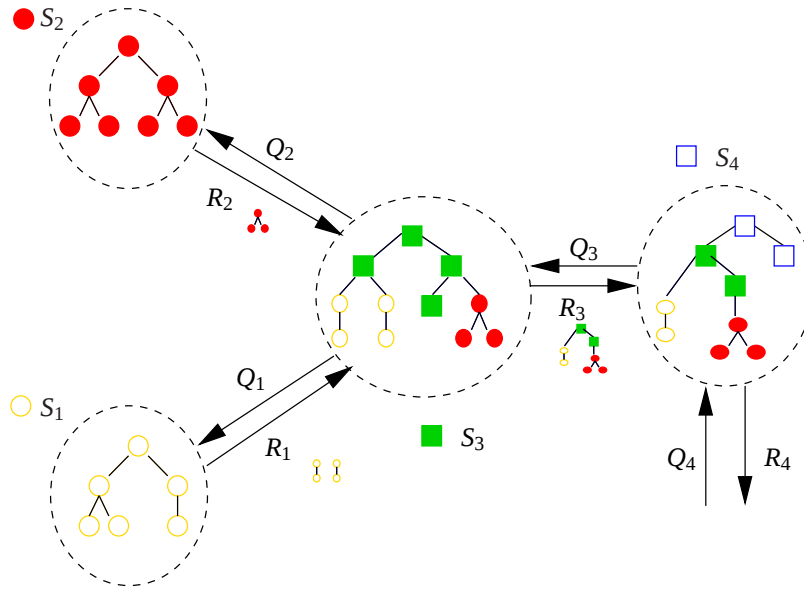


FIG. 6.1 – Exemple d'utilisation de services par leurs données

Les services S_1 jusqu'à S_4 sont implantés comme des requêtes sur les documents des pairs qui les publient. Lorsqu'un service S_i est appelé avec un ensemble de paramètres, la requête Q_i qui lui correspond est exécutée. La réponse R_i de la requête Q_i est un ensemble de fragments XML qui sont intégrés ensuite dans les documents des pairs clients. La réponse d'un service S_i peut contenir des données reçues de la part d'autres services. Ainsi, par exemple, la réponse R_3 obtenue après l'exécution de Q_3 est intégrée dans les documents de S_4 . Une partie des données dans R_3 provient de S_3 , tandis que d'autres parties ont pour source les services S_1 et S_2 .

Les données produites par un service peuvent par conséquent être utilisées dans l'évaluation d'autres requêtes du système (par exemple, les données de S_1 sont utilisées dans l'évaluation du service S_3 et de S_4). L'utilisation d'un service dans ce contexte est définie par l'utilisation de ses réponses. Nous remarquons aussi que les données d'un service peuvent être stockées par d'autres services et par conséquent elles peuvent continuer à être utilisées sans que le service soit rappelé. Par exemple, le service S_3 peut stocker la réponse R_2 et ne plus rappeler S_2 . Les liens d'utilisation

entre les services ne correspondent plus forcément à des appels de service, comme dans le cas des services présentés dans le chapitre 4. L'idée de base de notre approche est de définir des scores appropriés pour estimer l'importance ou la popularité d'une source de données par la *contribution* de ses données à l'évaluation de toutes les requêtes dans le système.

Nous présentons dans ce chapitre un modèle de classement des données et des sources XML basé sur leur contribution à l'évaluation des requêtes du système. La contribution des sources de données est définie dans le cas où les sources de toutes les données dans le système sont connues, ainsi que dans le cas anonyme, lorsque nous ne connaissons pas la source originelle d'une donnée.

6.1.2 Travaux existants

Récemment plusieurs travaux ont proposé des solutions de recherches avec classement pour des données semi-structurées (XML). Une première approche a consisté à étendre les méthodes de recherche traditionnelles par mots-clés pour les appliquer à des documents XML [58, 90, 99, 15, 12]. XRANK [90] classe des documents XML en utilisant une stratégie adaptée de PAGERANK basée sur la structure des liens dans la base de données. [58] présente XSEARCH qui inclut à l'évaluation d'une requête par mots-clé la notion de proximité sémantique afin de classer les résultats. XKEYWORD [99] donne un classement plus élevé les fragments résultats qui ont une taille plus petite. [15] décrit FLEXPATH qui combine la recherche par mots-clés et une recherche souple portant sur la structure basée sur XPATH.

La plupart des modèles proposés afin de classer des résultats de requêtes XML reposent sur des techniques de recherche d'information (*information retrieval* ou *IR*) [174, 167, 134, 7, 75, 73, 86, 14, 9], comme le score très largement utilisé $tf*idf$. Par exemple [75] présente un algorithme basé sur une approche probabiliste pour calculer des scores pour les arêtes entre deux nœuds, qui sont combinés lors du calcul du résultat. [73] étend XQUERY pour supporter des *matchings* proches de ceux dans IR avec de la pertinence. [7] présente TIX, une algèbre où les opérateurs manipulent des *arbres annotés de scores*, permettant de donner des scores pertinents et de trier les résultats. XXL [174] introduit une mesure de similarité basée sur les ontologies ainsi qu'un score $tf*idf$ afin de fournir des classements pertinents. [134] présente le moteur de recherche XIRCUS qui définit un score basé sur quatre mesures de similarité sur les termes, la structure XML, les éléments et les valeurs d'attributs, et la structure de liens. [14] propose un score inspiré de $tf*idf$ qui prend en compte à la fois la structure et le contenu. [86] applique des techniques XML-IR pour à la fois des données XML et web, et considère les liens entre documents. Cependant toutes ces approches estiment seulement l'importance locale d'un document en se basant seulement sur les requêtes et données locales. Aucune de ces propositions ne considère la contribution des sources nécessaires pour construire le document, à l'évaluation des requêtes locales, comme nous proposons de le faire.

6.2 Documents et services

Notre approche pour estimer l'importance basée sur l'usage des sources de données dans un entrepôt de données XML distribué s'appuie sur le paradigme ACTIVEXML [3] pour modéliser les

données et le traitement des requêtes distribuées. Dans cette section nous introduisons le modèle de données et de services que nous considérons dans notre système.

6.2.1 Modèle de documents

Un système ActiveXML est composé d'un ensemble de pairs où chaque pair stocke une collection de documents ACTIVEXML qui sont des documents XML standard enrichis par des appels à des services web. L'idée fondamentale est de considérer simultanément et indistinctement, au sein d'un même document ACTIVEXML, des données explicites, issues de fragments XML, et des données implicites, fournies en tant que réponses d'appels de service. Lorsqu'un appel de service est *matérialisé*, il est invoqué et ses résultats, qui sont aussi des fragments (A)XML, sont insérés dans le document à la place où se trouve cet appel.

Considérons l'exemple dans la figure 6.2 qui montre (avec une syntaxe volontairement simplifiée) un fragment d'un document *culture.xml* encapsulant deux appels de service.

```
1: <culture>
2:   <ville>Paris</ville>
3:   <événements>
4:     <appel_service
5:       service="culture.fr/getEvents(<ville>../../../../ville/text())</ville>")/>

6:   <appel_service
7:     service="villedeparis.com/getAgenda(<artiste>U2</artiste>")/>
8:   </événements>
9: </culture>
```

FIG. 6.2 – Interrogation de données par appels de services embarqués

La liste des événements culturels <événements> de la ville de *Paris* est ici implicitement définie par un appel au service *culture.fr/getEvents*, qui prend comme paramètre le nom de la ville dans laquelle on veut connaître les événements culturels. Le paramètre d'appel est la ville de *Paris*, qui est obtenue par une requête XPATH sur le document. La liste des événements culturels à Paris est enrichie avec les concerts de l'artiste *U2* en appelant *villedeparis.com/getAgenda*, qui renvoie les concerts d'un artiste dont le nom est reçu comme paramètre d'appel. L'appel de service est spécifié à l'aide de l'attribut *service* de l'élément *appel_service*, qui peut aussi avoir d'autres attributs précisant la fréquence, le mode d'appel, la validité ou l'expiration de l'appel (pour plus de détails voir [3]). Les paramètres spécifiés par des requêtes XPATH doivent être obtenus en évaluant la requête avant de faire l'appel.

La réponse d'un appel de service peut ainsi elle-même être composée de données implicites, qui seront obtenues (*matérialisées*) lors d'appels ultérieurs. De la même manière, les paramètres d'un appel de service peuvent être définis implicitement comme étant des appels à d'autres services qui doivent être faits par le service qui est appelé. Ceci est une caractéristique importante de ACTIVEXML, qui permet de réaliser facilement des compositions d'appels, mais soulève de

nouvelles problématiques (en termes de terminaison de calcul, d'optimisation de l'évaluation et de sécurité) qui ne seront pas abordées ici.

```
1: <culture>
2:   <ville>Paris</ville>
3:   <événements>
4:     <appel_service
5:       service="culture.fr/getEvents(<ville>../../ville/text()</ville>)"'/>
6:     <concert>
7:       <artiste>Manu Chao</artiste>
8:       <date>2007</date>
9:       <ville>Paris</ville>
10:    </concert>
11:    <exposition>
12:      <artiste>Braque</artiste>
13:      <date>2002</date>
14:      <ville>Paris</ville>
15:    </exposition>
16:    <appel_service
17:      service="villedeparis.com/getAgenda(<artiste>U2</artiste>)"'/>
18:  </événements>
19: </culture>
```

FIG. 6.3 – Matérialisation d'un appel de service dans le document *culture.xml*

Une fois l'appel de service exécuté, les réponses (implicites et/ou explicites) sont insérées dans le document source soit à la place de l'appel de service, soit en tant que frères du nœud `<appel_service>`. Le document *culture.xml* dans la figure 6.3, est le même que celui présenté à la figure 6.2 après la matérialisation de `culture.fr/getEvents` dont les réponses ont été insérées comme frères de l'appel de service. Les fragments `<concert>` (ligne 5) et `<exposition>` (ligne 10), résultats de l'appel à `getEvents`, sont explicitement intégrés au document source.

Représentation sous forme d'arbre

Les documents ACTIVEXML peuvent être modélisés comme des arbres non-ordonnés et étiquetés, composés de *nœuds de données* (données XML statiques) et de *nœuds de fonction* correspondant aux appels de services. Sous certaines contraintes [3], la sémantique formelle d'un document ACTIVEXML peut être définie par le document XML (fini ou infini) où tous les nœuds de fonction ont été remplacés par leurs résultats. Nous considérons qu'un document ACTIVEXML donné évolue dans le temps et correspond à une séquence d'*instances de document* qui contiennent des données locales, des appels de service et les résultats d'appels de service réalisés par le système. Cette sémantique « procédurale » implique que l'ensemble des nœuds de données contenus

dans un document d donné dépend de la stratégie de matérialisation appliquée par chaque pair pour l'évaluation des appels de service [2, 136]. Notre approche de classement ne dépend pas d'une stratégie de matérialisation particulière mais nous supposons que chaque requête est évaluée sur un ensemble cohérent (en ce qui concerne la validité et la complétude des données) de documents ACTIVEXML.

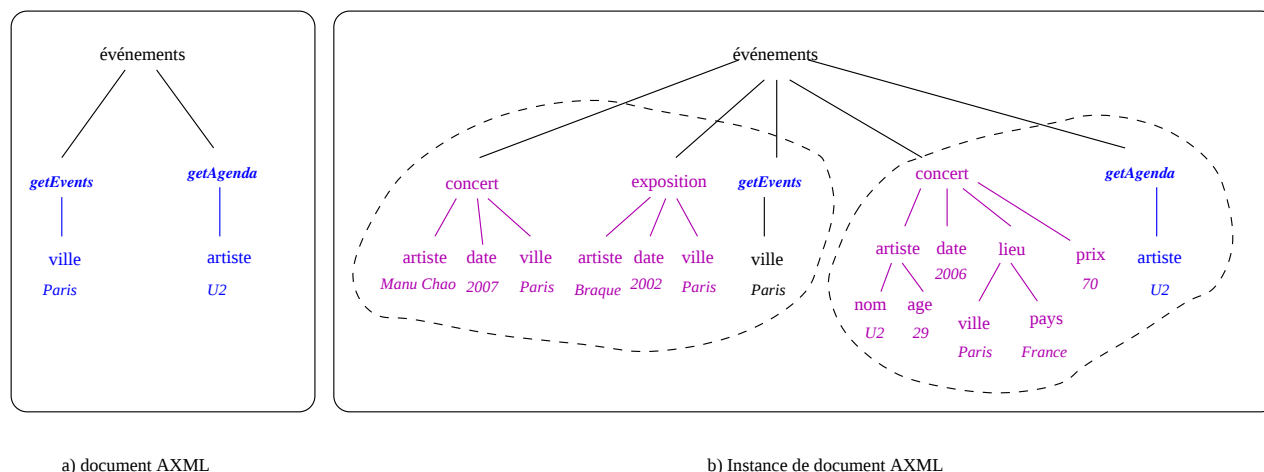


FIG. 6.4 – Deux instances d'un document ACTIVEXML

La figure 6.4 montre les deux instances du document ACTIVEXML qui stocke des événements culturels à Paris, présentés dans les figures 6.2 et 6.3. Sur la figure 6.4-(a) les sous-arbres ayant pour racine un nœud de fonction correspondent à l'ensemble des paramètres de l'appel qui sont transmis aux services correspondants lorsqu'ils sont appelés (par exemple *ville* = "Paris" obtenu après l'évaluation de la requête XPATH est transmis comme paramètre à chaque appel de *getEvents*). Les nœuds *événements*, *ville* et *artiste* sont des éléments XML *statiques* (nœuds de données) du document, alors que les nœuds *getEvents* et *getAgenda* correspondent à des nœuds de fonction (appels de service). La figure 6.4-(b) illustre l'instance du document de la figure (a) dans lequel les fragments retournés par les deux appels de service sont placés comme des frères des nœuds de fonction dans le document.

6.2.2 Services et requêtes

Un pair ACTIVEXML est composé d'une collection de services locaux et d'une base de documents XML, interrogés par ces services. Les services ACTIVEXML peuvent être définis comme étant des requêtes paramétrées en utilisant n'importe quel langage de requêtes XML, suivant le pouvoir d'expression et la complexité désirés. Lors de l'appel du service, la requête qui correspond au service instancié avec les paramètres d'appel est exécutée sur les documents locaux. Ainsi, on différenciera le **service**, qui peut être appelé avec un ensemble de paramètres, d'une **requête**, qui est un appel du service à un instant donné avec un ensemble de paramètres et qui fournit une réponse donnée.

Pour illustrer notre approche de classement, nous considérons un langage de requêtes simple utilisant des motifs (« patterns ») d'arbre paramétrés qui filtrent les informations basé à la fois sur la structure du document et sur son contenu.

Définition 24. (motif d'arbre) Un motif d'arbre est un arbre étiqueté non-ordonné $T = (r, N, E, P, \Theta, \lambda, \sigma, \pi)$ où :

- N est un ensemble de nœuds correspondants aux éléments XML ;
- r est un nœud racine distinct correspondant au document ou à la collection de documents que l'on interroge ;
- E est un ensemble de noms d'éléments XML (nœuds statiques) ;
- P est un ensemble de noms de paramètres appelés les paramètres de sélection de T ;
- $\Theta : N \rightarrow N \cup \{r\} \times \{ |, || \}$ définit la structure de l'arbre, où $\Theta(n) = (n', a)$ retourne le type de l'axe a (au sens d'axe dans XPATH) connectant un nœud $n \in N$ à son parent (ancêtre) n' ;
- $\lambda : N \cup \{r\} \rightarrow E \cup \{*, /\}$ est une fonction d'étiquetage des nœuds définissant pour chaque nœud $n \in N$ une étiquette $\lambda(n) \in E \cup \{*\}$ (* signifie n'importe quel nœud). L'étiquette / est réservée au nœud racine r ;
- $\sigma : N \rightarrow \text{Cond}$ est appelée la fonction de sélection de T . Si $n \in N$ est un nœud feuille de T , $\sigma(n)$ est soit non-définie soit une condition " ϕx " où $\phi \in \{<, >, =, <>\}$ et $x \in \text{String} \cup P$ est une valeur de chaîne de caractères ou un paramètre de sélection¹. Les conditions $\sigma(n)$ sont indéfinies pour tous les nœuds non-feuille n ;
- $\pi \subseteq N$ est un sous-ensemble de nœuds appelé l'ensemble projection de T .

Exemple 24. Par exemple, la figure 6.5-(a) montre la spécification sous forme de motif d'arbre d'une requête `getConcerts(p)` sur une instance du document `culture.xml` dans la figure 6.4. Cette requête retourne tous les artistes et la date de leur concerts dans une ville définie par un paramètre p . Les concerts peuvent se trouver à n'importe quel niveau dans les documents interrogés. L'ensemble projection de ce motif d'arbre contient les nœuds `artiste` et `date` qui sont soulignés sur la figure 6.5-(a).

Le service `getConcerts(p)` peut être spécifié, par exemple, dans le langage de requêtes X-QL [196], comme :

```

1: let service getConcerts($p) be
2:   select <concert>, v//artiste, v//date, </concert>
3:   from culture in load('culture.xml'), v in culture//concert
4:   where v//ville = $p

```

Lorsque l'on instancie les paramètres de la requête (service), on obtient un appel de ce service, qui est spécifié par une instance d'un motif d'arbre :

Définition 25. (instance d'un motif d'arbre) L'ensemble des instances $I(T)$ d'un motif d'arbre T est défini par l'ensemble de tous les motifs d'arbre (sans paramètre) qui peuvent être obtenus en associant tous les paramètres de sélection des conditions $\sigma(n)$ sur tous les nœuds $n \in N$ à une valeur sous forme de chaîne de caractères.

¹Par soucis de simplicité, nous considérons seulement les valeurs et paramètres de type chaîne de caractères

Exemple 25. Le service `getConcerts` spécifié par le motif d'arbre paramétré dans la figure 6.5-(a) peut être appelé avec le paramètre « Paris » conduisant à l'instance de motif d'arbre `getConcerts('Paris')`.

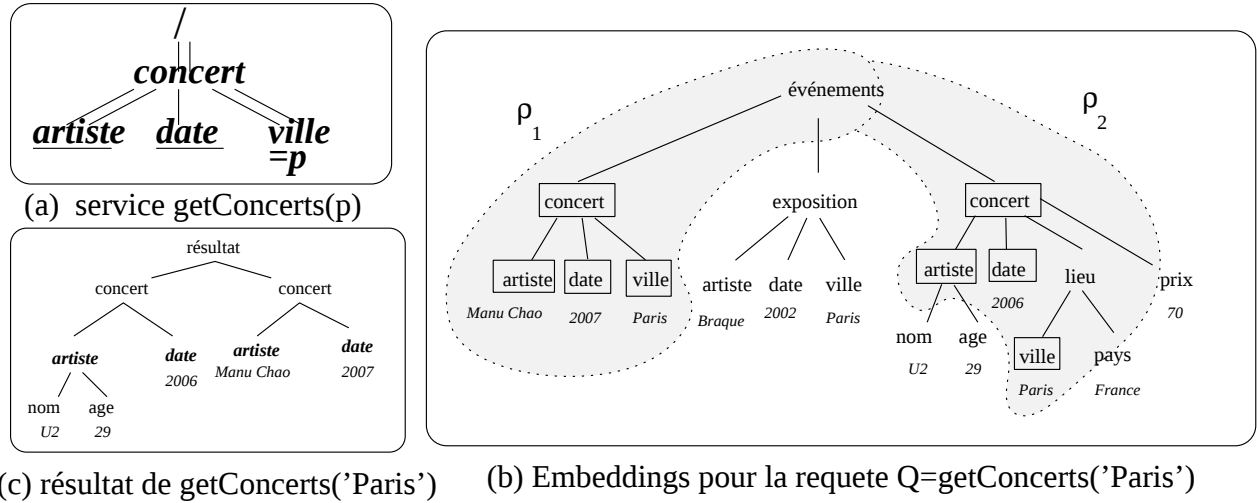


FIG. 6.5 – Service, embedding, résultat

Nous considérons que chaque service `ACTIVEXML` est défini comme un motif d'arbre T sur une collection d'instances de documents `ACTIVEXML`. Les paramètres d'entrée d'un service correspondent aux paramètres de sélection du motif d'arbre et la sortie du service dépend de l'ensemble projection. Un appel de service fournit les valeurs pour tous les paramètres de sélection afin d'instancier le motif d'arbre correspondant. Dans la suite nous utiliserons le terme de **service** (s) pour les motifs d'arbre définissant des services et le terme de **requête** (q) pour les instances de motifs d'arbre définies par un appel de service.

6.2.3 Sémantique des requêtes

Une requête q est une instance de motif d'arbre appliquée à une collection d'instances de documents `ACTIVEXML` D . La sémantique de q en fonction de D est définie de manière similaire à celle proposée dans [2] :

Définition 26. (embedding de requête) Soit une instance de motif d'arbre q et un document $d \in D$, un embedding de q dans d est un homomorphisme d'arbre ρ des nœuds de l'arbre q vers les nœuds de données des documents d représentés sous forme d'arbre, associant la racine de q à l'élément racine de d et préservant les relations structurelles (père-fils et ancêtre-descendant) et les contraintes de sélection (les nœuds étiquetés « $*$ » pouvant être associés à n'importe quel type d'élément). L'ensemble de tous les embeddings de q sur une collection de documents D est noté $\mathcal{E}(q, D)$.

Nous observons qu'un embedding ρ d'une requête q dans un document d est un sous-arbre de d . L'arbre ρ respecte la structure de l'instance de motif d'arbre q et chaque nœud r dans ρ satisfait une des conditions suivantes :

- (i) $r = \rho(n)$ est l'image dans d d'un nœud n de q .
- (ii) r est sur le chemin qui connecte deux nœuds r' et r'' dans d . Les nœuds r' et r'' sont les images dans d de deux nœuds n' et n'' dans q qui sont connectés par un arc *ascendant-descendant* dans l'instance de motif d'arbre q .

Exemple 26. La figure 6.5-(b) montre deux arbres ρ_1 et ρ_2 qui sont deux arbres embedding de la requête q getConcerts('Paris') dans le document culture.xml dans la figure 6.4-(b). Le nœud événements dans le embedding ρ_2 se trouve sur le chemin entre la racine du document et le nœud concert dans ρ_2 . Le nœud lieu fait partie de ρ_2 car il est sur le chemin entre les nœuds concert et ville qui sont des images de nœuds avec le même nom dans q . Observons que les nœuds nom et age ne font pas partie d'un embedding.

Étant donné un embedding (arbre) ρ , nous allons définir deux arbres supplémentaires appelés le *témoin* et le *résultat* de ρ :

- Le *témoin* de ρ pour q est noté ρ_q et est obtenu en éliminant de ρ les nœuds qui ne sont pas des images de nœuds de q : $\rho_q = \{\rho(n) \mid n \in q\}$.
- Le *résultat* de ρ pour q est noté ρ_π et correspond à un arbre non-ordonné qui contient les nœuds suivants :
 - (i) tous les nœuds $\rho(n)$ qui sont des images des nœuds n dans la projection π de q ($n \in \pi$).
 - (ii) les *descendants* des nœuds $\rho(n)$ dans d . Ces descendants ne sont pas nécessairement dans l'embedding ρ .
 - (iii) le *plus proche ancêtre commun* dans d des nœuds $\rho(n)$. Ce nœud devient la racine de ρ_π .

Exemple 27. Dans la figure 6.5-(b) les nœuds témoins de chaque embedding ρ_1 et ρ_2 sont signalés par un rectangle. Observons que le nœud lieu ne fait pas partie des témoins. Les résultats qui correspondent au embeddings ρ_1 et ρ_2 sont les deux nœuds concert dans la figure 6.5-(c). Observons que les nœuds projetés pour chacun des deux arbres d'embeddings ont été gardés ensemble, reliés par leur premier nœud ancêtre commun (concert).

Le résultat final d'une requête q est obtenu en reliant ensemble tous les résultats des ses embeddings par un nœud *résultat* :

Définition 27. (*résultat d'une requête*) Le résultat d'une requête q sur une collection de documents D est noté $\mathcal{R}(q, D)$ et correspond à un document ACTIVEXML non-ordonné avec un élément racine résultat contenant un sous-élément ρ_π pour chaque embedding ρ de q dans un document de la collection.

Exemple 28. Le résultat de la requête `getConcerts('Paris')` est montré sur la figure 6.5-(c). Le résultat final a été construit en créant un nœud résultat additionnel qui relie ensemble les deux arbres ρ_π correspondant à ρ_1 et ρ_2 .

Dans la suite, nous supposons que les requêtes et les collections de documents sont estampillées par des estampilles temporelles. Alors, le résultat $\mathcal{R}(q(t), D(t))$ d'une requête $q(t)$ à l'instant t dépend de « l'état d'instanciation » de la collection de documents $D(t)$ à l'instant t . Par soucis de simplicité, dans la suite $\mathcal{E}(q) = \mathcal{E}(q(t), D(t))$ et $\mathcal{R}(q) = \mathcal{R}(q(t), D(t))$ feront référence à l'ensemble des embeddings et des résultats d'une requête q donnée, évaluée à un instant t choisi.

Nous distinguons pour chaque nœud n d'un document `ACTIVEXML` donné le service $source(n)$ qui a créé n et le service $sentby(n)$ qui a retourné le nœud dans son résultat de requête. Lorsqu'une copie n' d'un nœud n est envoyée dans le résultat d'un service s , la copie n' conserve la même valeur de l'attribut $source(n') = source(n)$ que le nœud n . La valeur de l'attribut $sentby$ de la copie n' est changée à s .

Exemple 29. Le nœud `concert` dans la figure 6.4-(b) reçu après la matérialisation de `getEvents` a comme attributs (qui ne sont pas représentés sur la figure) $source(concert) = getEvents$ et $sentby(concert) = getEvents$. La copie de ce nœud dans le résultat de `getConcerts('Paris')` (figure 6.5-(c)) aura comme attributs $source(concert) = getEvents$ et $sentby(concert) = getConcerts$.

Dans la suite de ce chapitre nous allons montrer comment estimer l'importance d'un service à l'aide de l'utilisation des nœuds des données $produced(s) = \{n \mid source(n) = s\}$ qu'il a générées par d'autres requêtes du système. Par exemple, si nous considérons le document de la figure 6.5-(b) et le service `getConcerts(p)`, nous voyons que seuls les services ayant produit de l'information contenue dans les éléments de type `concert` peuvent contribuer au service `getConcerts(p)`. Évidemment, cette contribution dépend des valeurs du paramètre p et de la définition de `getConcerts(p)`. Par exemple, une source ne peut pas contribuer en générant des éléments de type `exposition`. Cette discussion sera formalisée dans la section suivante.

6.3 Modèle de contribution

Nous souhaitons estimer l'importance d'un service s par la contribution de ses données $produced(s)$ à tous les appels de service (requêtes) durant une période donnée. Nous considérons dans la suite que chaque service connaît la source $source(n)$ originelle de chaque nœud XML n qu'il manipule, indépendamment du fait que ce nœud ait été reçu *directement* depuis la source ou *indirectement* de la part d'un service voisin qui n'est pas la source du nœud. Une configuration *préservant l'anonymité* où l'information sur la source originelle des nœuds est cachée aux services qui ne sont pas ses voisins directs est présentée dans la section 6.5.

6.3.1 Application du modèle générique

Nous montrons ici comment notre modèle générique de classement de services présenté dans le chapitre 3 peut être appliqué au classement de services et aux données qu'ils ont produites. La notion d'*utilisation de service* désigne ici l'accès aux données produites par le service pendant l'évaluation d'une requête afin de construire le résultat de la requête. Comme nous l'avons vu à la section 3.1.1, notre modèle s'appuie sur une fonction de trace. Dans le cadre d'un entrepôt de documents ACTIVEXML, la fonction de trace enregistre, pour chaque couple de services (s_i, s_j) les instants où s_i a accédé aux données produites par s_j (c'est à dire aux nœuds XML n tels que $source(n) = s_j$). Formellement, la définition de la fonction de trace d'utilisation de services devient :

Définition 28 (fonction de trace d'accès aux données). Soit S un ensemble fini de services identifiés, tels que chaque service s_i peut utiliser pendant ses appels les données produites par d'autres services s_j . Une fonction de trace d'accès aux données $\Lambda_d : \mathcal{T} \times S \times S \rightarrow 2^{\mathcal{T}}$ retourne pour chaque instant $t \in \mathcal{T}$ et chaque paire de services distincts (s_i, s_j) , l'ensemble d'estampilles $\Lambda_d(t, s_i, s_j) = \{\tau | \tau \leq t\}$ qui représente la partie de $\mathcal{T}$ constituée de tous les instants auxquels s_i a utilisé des données (nœuds) provenant de s_j pendant ses appels avant t .

A noter qu'au moment t où s_i utilise les données produites par s_j ces données peuvent être déjà matérialisées ou non dans les documents sur lesquels la requête correspondant à l'appel de s_i est évaluée. L'utilisation par un service s_i d'une donnée provenant d'un service s_j se traduira par un nouvel enregistrement dans $\Lambda_d(t, s_i, s_j)$, que la donnée se trouve sur le pair de s_i (donnée matérialisée) ou pas (donnée alors obtenue par un appel lors de son utilisation). Par conséquent pour chaque couple (s_i, s_j) , les instants enregistrés dans Λ_d sont les instants d'appel de service si à chaque instant t quand s_i a eu besoin des données de s_j , s_i n'a pas utilisé des réponses obtenus précédemment de la part de s_j , mais il les a obtenues en faisant des appels de service.

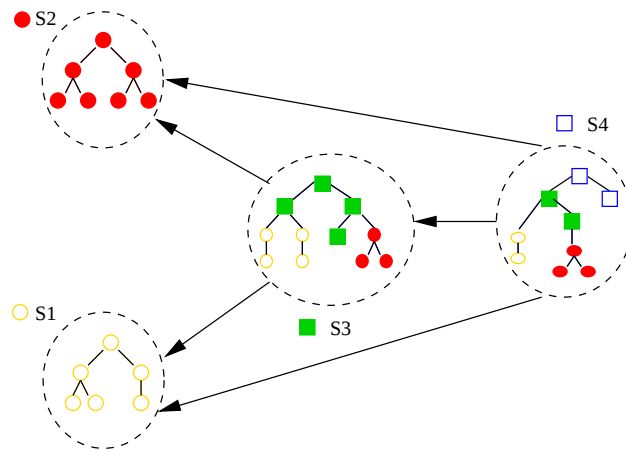


FIG. 6.6 – Exemple de graphe d'accès aux données

Puisque dans cette configuration nous connaissons la source originelle de chaque nœud n , la fonction de trace enregistre les liens d'utilisation entre des couples de services (s_i, s_j) qui ne sont pas des voisins directs dans le graphe d'appels de service.

Exemple 30. *Considérons à nouveau l'exemple présenté dans la figure 6.1. Le service S_4 utilise dans son évaluation les données produites par S_1 , S_2 et S_3 qui sont intégrées dans ses documents locaux. A chaque instant quand un appel à S_4 est exécuté et que la requête Q_4 a utilisé dans son évaluation les données de S_1 , cet instant est enregistré dans $\Lambda_d(t, S_4, S_1)$. Il existe en conséquence des liens d'utilisation de S_4 à S_1 , même si S_4 n'appelle pas directement S_1 , mais S_3 .*

Le graphe d'utilisation, que nous appelons pour ce type d'applications le *graphe d'accès aux données*, reliera donc chaque source de données à tous les services les ayant utilisées.

Exemple 31. *Supposons qu'à l'instant t les traces d'utilisation $\Lambda_d(t, S_3, S_1)$ et $\Lambda_d(t, S_3, S_2)$ contiennent les instants jusqu'à t quand S_3 a utilisé les données produites par S_1 et S_2 , et $\Lambda_d(t, S_4, S_1)$, $\Lambda_d(t, S_4, S_2)$ et $\Lambda_d(t, S_4, S_3)$ les instants où S_4 a utilisé les données de S_1 , S_2 et S_3 . Si ces traces ne sont pas vides, la figure 6.6 montre le graphe d'accès aux données qui correspond à l'utilisation des données des services dans la figure 6.1.*

Il est important de souligner que le modèle de contribution basée sur l'utilisation des données quand nous connaissons les sources originelles des données est un cas particulier, plus simple, du modèle générique présenté au chapitre 3. En effet, puisque chaque source originelle des données est connue par tous les autres services du système, il n'y a pas de notion de contribution indirecte due à l'utilisation indirecte d'un service à travers d'autres services du système. Nous n'avons ici que des contributions de service directes, autrement dit tous les chemins d'utilisation sont de longueur 1. Dans ce cas, l'importance d'un service est définie en fonction de sa contribution directe à tous les services du système.

Nous introduisons maintenant différentes définitions de contribution de données suivant les niveaux d'agrégation des données considéré, afin d'arriver à la définition de la contribution effective directe π_{ji} d'un service s_j à un service s_i définie dans notre modèle générique. La contribution effective de s_j à s_i est définie en fonction de la contribution des données de s_j à s_i , que nous allons présenter par la suite.

6.3.2 Contribution des données

Nous estimons tout d'abord la contribution d'un nœud de données à une seule requête. La contribution d'un nœud XML n à une requête q donnée dépend de son utilisation durant l'évaluation de q qui se traduit par sa présence dans les embeddings $\mathcal{E}(q)$ de q . Prenons par exemple la requête de la figure 6.5. Les différents nœuds du document jouent différents rôles durant l'évaluation de la requête. Certains nœuds sont inutiles (*exposition*), certains sont seulement nécessaires pour accéder à l'information (*événements*, *lieu*) et certains nœuds sont utilisés à la fois pour accéder et pour sélectionner l'information (*concert*, *artiste*, *date*, *ville*).

Contribution d'un nœud à un embedding

La contribution d'un nœud n à une requête q donnée est définie par la somme de ses contributions $C(n, \rho)$ à tous les embeddings $\rho \in \mathcal{E}(q)$. Cette contribution reflète sa contribution pour accéder (score d'accès) et pour sélectionner (score de sélection) les « informations d'intérêt » pour q :

- (i) tout nœud de donnée $n \in \rho$ dans le embedding de q est nécessaire pour accéder à l'information pertinente pour la requête et son score d'accès dépend de ses descendants qui sont des nœuds témoins pertinents dans ρ_q auxquels ce nœud permet d'accéder. Tous les nœuds de document qui ne sont pas dans ρ ont un score d'accès de 0.
- (ii) tout nœud n dans le témoin ρ_q d'un embedding donné ρ correspond à une donnée qui a été explicitement mentionnée dans la requête afin d'extraire de l'information (étiquette de nœud, prédicat de sélection) et son score de sélection reflète son « degré d'intérêt » pour cette information.

Ces deux heuristiques sont illustrées par la fonction de score de contribution normalisée $C(n, \rho)$ dans l'équation suivante :

$$C(n, \rho) = \alpha \times \frac{\overbrace{|n//\rho_q|}^{\text{score d'accès}}}{\sum_{n' \in \rho} \overbrace{|n'//\rho_q|}^{\text{score d'accès}}} + (1 - \alpha) \times \frac{\overbrace{is_witness(n)}^{\text{score de sélection}}}{\overbrace{|\rho_q|}^{\text{score de sélection}}} \quad (6.1)$$

où $\alpha \in [0, 1]$, $is_witness(n) = 1$ si n est un nœud témoin *pertinent* avec pour étiquette $\lambda(n) \neq *$, et vaut 0 sinon, $|\rho_q|$ est le nombre de nœuds témoins pertinents dans ρ_q et $n//\rho_q$ est l'ensemble des nœuds témoins pertinents n' qui sont descendants de n (y compris n) dans le embedding ρ .

Le facteur $\alpha \in [0, 1]$ permet de donner plus ou moins d'importance au score de sélection vis à vis du score d'accès. Le score d'accès traduit l'« importance d'accès » de n en considérant le nombre de nœuds témoins pertinents qui sont descendants du nœud n .

Tous les nœuds témoins pertinents ont le même score de sélection normalisé $1/|\rho_q|$, ce qui signifie que le score de sélection d'un nœud témoin n diminue lorsque le nombre de nœuds d'intérêt dans q augmente. Une justification derrière ceci est que le score de sélection d'un nœud n doit refléter son importance relative par rapport aux autres nœuds dans l'arbre témoin ρ_q afin de pouvoir sélectionner dans l'ensemble de documents sur lesquels q est évaluée l'embedding ρ et les informations qu'il contient.

Nous remarquons que la contribution d'un nœud à une requête donnée dépend de la façon dont le service a été défini. Par exemple, si le nœud *événements* avait été mentionné explicitement dans *getConcerts(p)* la contribution du nœud *événements* dans le document de la figure 6.5-(b) augmenterait d'un score de contribution de sélection positif (actuellement son score de sélection est 0). Des modèles plus sophistiqués associant des poids aux nœuds des requêtes comme dans le modèle TIX [7] pourraient également être utilisés pour pondérer les nœuds des requêtes d'après certains besoins de l'utilisateur ou de l'application.

Contribution d'un nœud à une requête

La contribution d'un nœud à une requête q est définie par sa contribution normalisée à tous les embeddings de q :

$$c(n, q) = \frac{1}{|\mathcal{E}(q)|} \times \sum_{\rho \in \mathcal{E}(q)} c(n, \rho) \quad (6.2)$$

Exemple 32. Prenons notre exemple courant et supposons $\alpha = 1/3$. Le nœud racine événements contribue avec un score d'accès positif aux deux embeddings ρ_1 et ρ_2 : $c(\text{événements}, \rho_1) = \alpha \times 4/11 + (1 - \alpha) \times 0 = 4/33$ ($|\text{événements}/\rho_1| = |\text{concert}/\rho_1| = 4$, $|\text{artiste}/\rho_1| = |\text{date}/\rho_1| = |\text{ville}/\rho_1| = 1$) et $c(\text{événements}, \rho_2) = 1/3 \times 4/12 = 4/36$. Observons que la contribution de événements à ρ_2 est moins importante à cause du nœud lieu qui est aussi nécessaire afin d'accéder au nœud ville dans l'embedding ρ_2 . Le nœud exposition a évidemment deux scores de contributions valant zéro $c(\text{exposition}, \rho_1) = c(\text{exposition}, \rho_2) = 0$ et le nœud lieu contribue seulement à l'embedding ρ_2 , $c(\text{lieu}, \rho_2) = 1/3 \times 1/12 = 1/36$. Son fils ville obtient la même contribution d'accès augmentée par une contribution de sélection positive : $1/3 \times 1/12 + 2/3 \times 1/4 = 7/36$. Observons aussi que les deux nœuds nom et age ne sont pas considérés comme contribuant à la requête bien qu'ils fassent partie de son résultat. Ceci est raisonnable puisqu'ils contribuent seulement à l'évaluation d'autres requêtes qui les reçoivent comme résultats.

Contribution d'un nœud à un service

Connaissant la contribution d'un nœud à une requête unique, nous pouvons maintenant agréger ces contributions sur une séquence de requêtes qui sont des appels d'un service donné. Nous considérons que la contribution d'un nœud n à un service s donné est une agrégation de la contribution de n à tous les appels de services q reçus par s qui sont dans l'« historique des appels » de s . Afin de prendre cet historique en considération, nous désignons par $c(n, s, \tau)$ la contribution du nœud n à s à l'instant τ .

Nous supposons d'abord que la contribution d'un nœud n à un service s décroît avec l'ancienneté du dernier appel q à s qui a utilisé n . Plus précisément, si τ_i correspond à l'instant du dernier accès à n par le service s , la contribution $c(n, s, \tau)$ de n à s à l'instant $\tau \geq \tau_i$ est définie comme suit :

$$c(n, s, \tau) = c(n, s, \tau_i) \times \phi(\tau_i, \tau) \quad (6.3)$$

où ϕ est une fonction d'ancienneté qui décroît avec la longueur de l'intervalle de temps $(\tau - \tau_i)$ afin de représenter l'ancienneté du dernier appel de s . Un exemple d'une telle fonction est $\exp(-(\tau - \tau_{i-1})/T)$, où T permet de fixer la période après laquelle la contribution de n est divisée par 2 si il n'est plus utilisé lors de l'évaluation de s .

A chaque appel de service q de s à l'instant τ les scores de contribution $C(n, s, \tau)$ de *tous les nœuds n qui contribuent à s* (c'est à dire les nœuds n tels que leur contribution à s est positive) sont mis à jour avec leur contribution nœud-à-requête $C(n, q)$ actuelle (cf. équation 6.2) comme suit :

$$C(n, s, \tau) := (1 - \beta) \times C(n, s, \tau) + \beta \times C(n, q) \quad (6.4)$$

La valeur $C(n, s, t_0)$ est initialisée à 0 pour tous les nœuds n . Le facteur de *réminiscence* $\beta \in (0, 1]$ contrôle l'impact de la nouvelle contribution de n calculée à l'instant d'accès τ sur la contribution de n à s (si $\beta = 1$, la contribution d'un nœud dépend seulement de sa contribution au dernier appel à s).

Maintenir ces scores de contribution pour tous les nœuds n et tous les services s est évidemment trop coûteux. Nous montrerons dans la suite qu'il est possible d'agréger toutes les valeurs de contribution des nœuds d'une source et de maintenir seulement le score de contribution agrégé d'une source s pour un service s' donné.

6.3.3 Contribution des sources

Nous pouvons maintenant définir la contribution effective d'un service s_j à d'autres services s_i du système en fonction de la contribution des données dont il est la source. La contribution d'une source s_j à s_i et calculée comme une agrégation sur les valeurs de contribution effective de s_j aux requêtes q_i qui sont des appels du service s_i .

Contribution d'une source à une requête

Nous définissons la contribution d'une source s_j à une requête q_i donnée comme étant la contribution totale à q_i de tous les nœuds n que s_j produit :

$$C(s_j, q_i) = \sum_{n \in \text{produced}(s_j)} C(n, q_i) \quad (6.5)$$

Contribution d'une source à un service

La contribution effective direct du service s_j au service s_i , que nous avons défini et notée π_{ji} dans notre modèle générique (voir la définition 3, page 51), peut être calculée de deux façons différentes.

Tout d'abord, la contribution d'une source s_j à un service s_i à un instant τ donné est la contribution totale à s_i à l'instant τ de tous les nœuds de données produits par s_j :

$$\pi_{ji}(\tau) = \sum_{n \in \text{produced}(s_j)} C(n, s_i, \tau) \quad (6.6)$$

Cette équation est basée sur la contribution à l'instant τ de tous les nœuds n produits de s_j , qui est calculée avec l'équation 6.3.

D'autre part, la contribution de s_j à s_i peut également être définie de manière équivalente par sa contribution à toutes les requêtes q_i du service s_i (cf. équation 6.5). Ceci est montré dans la suite.

La contribution effective d'une source s_j à un service s_i donné décroît avec l'ancienneté du dernier instant quand s_i a utilisé les données de s_j . Si τ_i correspond au dernier instant où s_i a utilisé les données de s_j , la contribution de s_j à s_i à l'instant $\tau \geq \tau_i$, (définie dans l'équation 6.6), peut être réécrite en utilisant l'équation de la contribution d'un nœud à un service (équation 6.3) comme étant :

$$\begin{aligned}\pi_{ji}(\tau) &= \sum_{n \in \text{produced}(s_j)} C(n, s_i, \tau_i) \times \phi(\tau_i, \tau) \\ &= \pi_{ji}(\tau_i) \times \phi(\tau_i, \tau)\end{aligned}\tag{6.7}$$

où ϕ est la fonction d'ancienneté définie précédemment.

Il est alors possible de généraliser l'équation 6.4 pour calculer la contribution effective entre deux services :

$$\pi_{ji}(\tau) = (1 - \beta) \times \pi_{ji}(\tau) + \beta \times C(s_j, q_i)\tag{6.8}$$

où $\pi_{ji}(t_0)$ est initialisée à 0, $C(s_j, q_i)$ est la contribution d'un service à une requête (équation 6.5) et β est le facteur de réminiscence comme présenté auparavant. Cette équation exprime le fait que pour estimer la contribution d'une source s_j à un service s_i donné, il est suffisant de mettre à jour (lors de chaque appel de service q_i de s_i) la contribution de toutes les sources s_j dont les données ont été utilisées par s_i (au lieu de mettre à jour la contribution de tous les nœuds comme présenté dans l'équation 6.4).

Résumé des notations utilisées

Les différentes définitions ont été résumées dans le tableau 6.1.

6.4 Calcul d'importance

Dans la section précédente nous avons montré comment estimer la contribution d'un nœud de données à une requête ou à un service (ensemble de requêtes) dans le système. Les scores de contribution effective sont utilisés pour définir l'importance des nœuds et des sources par leur contribution à tous les services du système.

Notation	Description	Définition
$C(n, q)$	contribution d'un nœud n à un appel de service (requête) q	équation 6.2
$C(n, s, \tau)$	contribution d'un nœud n à un service s à τ	équation 6.3
$C(s, q)$	contribution d'un service s à un appel de service q	équation 6.5
$\pi_{ij}(\tau)$	contribution d'une source s_i à un service s_j à τ	équation 6.6
$I(n, \tau)$	importance d'un nœud n à l'instant τ	équation 6.9
$I_i(\tau)$	importance d'un service s_i à l'instant τ	équation 6.10

TAB. 6.1 – Valeurs de contribution et d'importance

6.4.1 Importance des nœuds

Les scores de contribution peuvent être utilisés de différentes manières pour le classement de données, suivant la façon dont ils sont agrégés.

Importance locale

Tout d'abord nous pouvons calculer pour un certain document d placé sur un pair une *importance locale au document* $I(n, d, \tau)$ de chacun de ses nœuds XML n , en agrégeant les valeurs de contribution de n à toutes les requêtes évaluées sur ce document. L'importance locale au document estime l'utilisation de chaque nœud n dans un document donné. Elle peut être utile afin de décider quelles parties d'un document devraient être matérialisées (les fragments avec une importance élevée) et quelle parties devraient être au contraire calculées dynamiquement [2].

Exemple 33. Par exemple, si beaucoup de requêtes évaluées sur le document de la figure 6.4-(b) sont intéressées par les informations concernant les concerts de U_2 , le nœud *concert* retourné par *getAgenda* est plus important que le nœud *exposition* et *concert* retournés par *getEvents*. Par conséquent, si nous devons décider quel service matérialiser, nous pouvons choisir en priorité *getAgenda* puisque ses données sont utiles à plus de clients, alors que *getEvents* peut être matérialisé dynamiquement, lorsqu'un client a besoin d'accéder à ses données.

Importance globale

L'importance locale au document peut être agrégée sur toutes les répliques d'un nœud n donné afin d'obtenir l'*importance globale d'un nœud* $I(n, \tau)$ comme défini dans l'équation 6.9. L'importance $I(n, \tau)$ d'un nœud n à l'instant τ peut être définie comme étant sa contribution à tous les services du système $\mathcal{S}$ qui utilisent ce nœud lors de leur évaluation jusqu'à l'instant τ :

$$I(n, \tau) = \sum_{s \in \mathcal{S}} C(n, s, \tau) \quad (6.9)$$

Cette importance agrège la contribution basée sur l'utilisation de n et de ses répliques dans le système. L'importance des données pourrait être utilisée pour estimer leur « prix ». Par exemple,

considérons une source qui fournit une liste détaillée des centres d'intérêt touristiques en Europe. Si cette liste est répliquée et interrogée sur plusieurs pairs, alors les informations contenues dans cette liste peuvent être considérées comme étant d'intérêt pour beaucoup d'utilisateurs, et avoir par conséquent un prix élevé. Une autre application serait le rafraîchissement des données dans un système distribué. On pourrait imaginer que si une certaine donnée est très répliquée et utilisée, elle devrait alors être rafraîchie en priorité.

6.4.2 Importance des services

L'importance $I_j(\tau)$ d'une source s_j à τ peut être définie par :

- (i) la somme de ses contributions à *tous* les services du système $\mathcal{S}$ (conformément au modèle présenté au chapitre 3) et, de manière équivalente pour cette application, par
- (ii) l'importance totale de tous ses nœuds.

L'importance d'un service $s_j \in \mathcal{S}$ est par conséquent calculée comme étant :

$$I_j(\tau) = \sum_{s_i \in \mathcal{S}} \pi_{ji}(\tau) = \sum_{n \in \text{produced}(s_j)} I(n, \tau) \quad (6.10)$$

où $\pi_{ji}(\tau)$ est calculé comme présenté dans l'équation 6.7.

Dans ce cas, un service est important s'il contribue avec beaucoup de nœuds de données à l'évaluation de requêtes, et l'importance de service peut être considérée comme une mesure de popularité basée sur l'utilisation.

L'importance d'une source peut être utilisée comme heuristique pour trier les résultats des requêtes de recherche. Par exemple, si un utilisateur s'intéresse aux dernières informations (« news »), alors les informations produites par une source dont les données sont répliquées et fortement interrogées sont plus importantes. En effet, si cette source est importante, alors nous pouvons considérer qu'en général elle produit des informations qui intéressent un grand nombre d'utilisateurs.

Les valeurs d'importance dépendent du temps et combinent des scores de contribution de nœuds à des requêtes distribuées. Le principal problème est alors de collecter et de rafraîchir efficacement les estimations de contribution et d'importance distribuées. Le calcul d'importance doit avoir accès aux valeurs de contribution effective entre chaque couple de services. Une solution est de définir un catalogue de services centralisé qui stocke les valeurs de contribution entre les différentes paires de services. Chaque service rafraîchit alors régulièrement ce catalogue avec les nouvelles valeurs de contribution qu'il a calculées pour d'autres services. Cela signifie en particulier que chaque service connaît la source de chaque nœud, ce qui pourrait être restrictif dans certaines applications. Une solution différente est présentée dans la section suivante, où la contribution des sources peut être calculée de manière distribuée sans révéler l'identité de la source.

6.5 Contribution indirecte

Dans la section précédente nous avons supposé que chaque service connaît la source s de chaque nœud donnée qu'il utilise. Dans cette section nous utilisons la notion de *contribution indirecte* d'un service s_j à un service s_i à travers d'autres services du système, pour estimer l'importance d'un service sans propager son identité dans le réseau.

Considérons trois services $s_i \rightarrow s_k \rightarrow s_j$, les arcs d'un service vers un autre signifiant que le premier utilise les données du deuxième. Dans le cas anonyme, nous ne connaissons pas les sources originelles de données manipulées par chaque service, mais seulement les services voisins qui lui ont envoyé ces données. Ainsi, par exemple, tous les nœuds n reçus par s_i de la part de s_k ont comme attribut $\text{sentby}(n) = s_k$, indépendamment de leur source ($\text{source}(n)$ est inconnu).

Puisque les sources originelles de données sont inconnues, le calcul de contribution et d'importance précédent ne peut pas s'appliquer dans le contexte anonyme. Dans cette situation, s_i calcule la *contribution directe du service* s_k en fonction de la contribution de tous les nœuds n tels que $\text{sentby}(n) = s_k$. La définition de l'importance de services est basée sur la contribution totale sur des chemins d'utilisation, de manière similaire à la définition générique de l'importance que nous avons donnée dans le chapitre 3.

Exemple 34. *Considérons par exemple la figure 6.1, sur laquelle le service S_4 utilise le résultat R_3 reçu par l'appel de service Q_3 au service S_3 . Ensuite, puisqu'il connaît l'origine de chaque nœud dans le résultat, S_4 peut précisément calculer la contribution de S_1 , S_2 et S_3 à un appel de service Q_4 .*

Supposons maintenant que les services S_1 et S_2 veulent préserver leur anonymité et seul leur client (direct) S_3 sait qu'ils sont les sources des données qu'ils ont envoyées à S_3 . Tous les nœuds n dans les réponses envoyées par S_3 à S_4 ont comme attribut $\text{sentby}(n) = S_3$ ($\text{source}(n)$ est inconnu). La stratégie précédente ne peut alors plus être appliquée puisque dans ce cas S_4 ignore la source des nœuds de données reçus dans le fragment R_3 .

Chemins d'utilisation : Observons que dans ce cas la fonction de trace enregistre seulement des liens d'utilisation entre des couples de services (s_i, s_j) qui sont tels que s_i appelle s_j pour obtenir ses données. Le graphe d'accès aux données défini à partir des traces a dans ce cas les mêmes arrêtes que le graphe d'appels de services défini dans le chapitre 4.

Exemple 35. *Dans l'exemple précédent, les instants où le service S_4 utilise les nœuds envoyés par S_3 pour évaluer ses requêtes Q_4 jusqu'à t sont enregistrés dans $\Lambda_d(t, S_4, S_3)$.*

Les traces d'accès des données jusqu'à l'instant t sont $\Lambda_d(t, S_4, S_3)$, $\Lambda_d(t, S_3, S_2)$ et $\Lambda_d(t, S_3, S_1)$. Si ces traces ne sont pas vides, le graphe d'accès aux données est celui illustré sur la figure 6.7.

Nous allons expliquer le calcul de la contribution directe et rappeler le calcul d'importance sur un chemin introduit dans le chapitre 3.

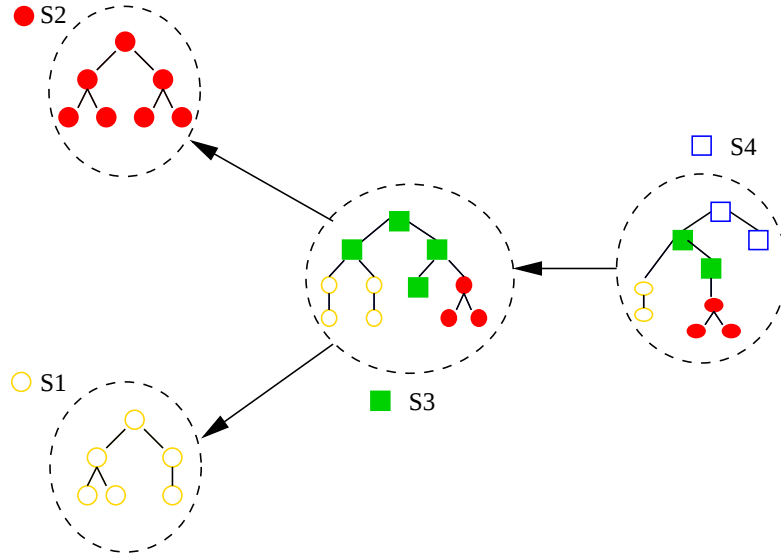


FIG. 6.7 – Graphe d'accès aux données pour des sources anonymes

Contribution directe dans le cas anonyme : Le score de contribution effective directe $\pi_{jk}^a(\tau)$ à l'instant τ d'un service s_j à un service client s_k est la somme des scores de contribution de tous les nœuds n envoyés par s_j à s_k ($\text{sentby}(n) = s_j$). Ce score est calculé de façon similaire à celui défini dans l'équation 6.6 (page 140), en remplaçant *produced* par *sentby* :

$$\pi_{jk}^a(\tau) = \sum_{\text{sentby}(n)=s_j} C(n, s_k, \tau) \quad (6.11)$$

Observons que le score de contribution directe $\pi_{jk}^a(\tau)$ prend à la fois en compte la contribution des nœuds produits par s_j ($n \in \text{produced}(s_j)$) et celle des nœuds envoyés par s_j dont il n'est pas la source.

Exemple 36. S_4 calcule la contribution directe de S_3 en fonction de tous les nœuds reçus dans R_3 . La contribution directe $\pi_{34}^a(\tau)$ prend en compte la contribution des nœuds dont la source originelle peut être S_1 , S_2 ou S_3 .

Calcul d'importance sur un chemin d'utilisation : Nous allons expliquer ce calcul sur la figure 6.8. Le service s_k envoie à s_j non seulement sa contribution directe $\pi_{jk}^a(t)$ (équation 6.11), mais aussi une partie du score de contribution total $v_{ki}(t)$ qu'il reçoit à son tour de la part de son client $s_i \in \text{In}(t, s_k)$. Le nœud s_k , qui est conscient de la façon dont il a construit les réponses qu'il a envoyées à s_i , distribue une partie du score reçu de la part de s_i au service s_j , conformément à sa contribution indirecte. La contribution indirecte ω_{jki} de s_j à s_i à travers s_k estime la partie de la

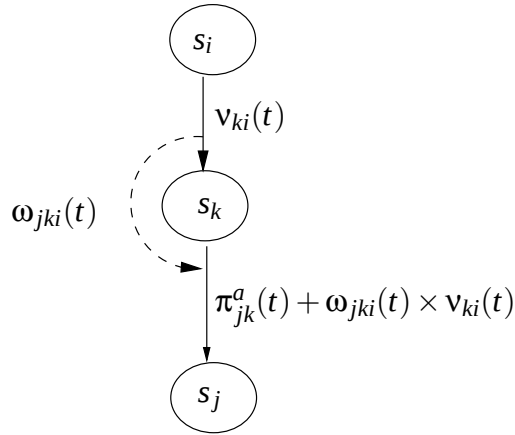


FIG. 6.8 – Calcul d'importance sur un chemin

contribution $v_{ki}(t)$ qui est due aux nœuds produits par s_j qui ont été envoyés à s_i à travers s_k .

Le score d'importance transmis sur un chemin d'utilisation a été introduit dans l'équation 3.5 (page 53) dans le chapitre 3. Nous allons rappeler ici son calcul :

$$v_{jk}(\tau) = \pi_{jk}^a(\tau) + \sum_{s_i \in In(\tau, s_k)} \omega_{jki}(\tau) \times v_{ki}(\tau) \quad (6.12)$$

Le *score de contribution totale* $v_{jk}(\tau)$ transmis à s_j par un service s_k donné inclut le score de contribution directe de s_j à s_k et une *fraction* du score de contribution totale $v_{ki}(\tau)$ reçu par s_k de la part de tous ses clients $s_i \in In(\tau, s_k)$.

Exemple 37. Par exemple, le score reçu par S_1 de la part de S_3 dépend du score de contribution directe de S_1 à S_3 , ainsi que d'une partie du score de contribution de S_3 à S_4 $v_{34}(t)$, conformément à la contribution indirecte $\omega_{134}(t)$:

$$v_{13}(\tau) = \pi_{13}(\tau) + \omega_{134}(\tau) \times v_{34}(\tau)$$

Chaque service s_k doit distribuer les scores reçus de la part d'un service client $s_i \in In(t, s_k)$ aux services $s_j \in Out(t, s_k)$ dont les données participent aux *résultats* envoyés par s_k à s_i . Le facteur $\omega_{jki}(\tau)$, correspondant à la contribution générique indirecte introduite dans le chapitre 3 (définition 4, page 51), estime la contribution relative des nœuds envoyés par s_j à s_i à travers les résultats produits par s_k . En particulier, si le service s_k n'envoie aucun des nœuds qu'il a reçus de s_j à destination de s_i , alors ce facteur sera égal à 0. Estimer $\omega_{jki}(\tau)$ est une tâche difficile puisque s_k n'a aucune information sur la façon dont ses résultats sont utilisés par ses clients s_i et par les autres services du système.

Heuristique pour le calcul de contribution indirecte

Nous proposons dans la suite une heuristique simple qui estime la contribution indirecte $\omega_{jki}(\tau)$ comme étant proportionnelle au nombre de nœuds de s_j qui ont été envoyés par s_k à s_i . La contribution indirecte $\omega_{jki}(\tau)$ est calculée, de manière similaire à la contribution directe, en agrégeant les scores de contribution de s_j à s_i à travers chaque requête q_k .

Pour chaque requête q_k invoquée par s_i , la contribution indirecte de s_j au service s_i via q_k peut être définie par le nombre relatif de nœuds de s_j qui sont envoyés à s_i :

$$\omega(s_j, q_k, s_i) = \frac{|\mathcal{R}(s_j) \cap \mathcal{R}(q_k)|}{|\mathcal{R}(q_k)|}$$

où $\mathcal{R}(s_j)$ représente l'ensemble de nœuds n envoyés par s_j à s_k ($\text{sentby}(n) = s_j$) et $|\mathcal{R}(q_k)|$ est le nombre total de nœuds envoyés par q_k à s_i . Un service s_j contribue donc plus si le résultat de q_k contient plus de nœuds provenant de s_j .

Exemple 38. Dans notre exemple, la contribution indirecte de S_2 à S_4 via la requête Q_3 est plus élevée que celle de S_1 , puisque R_3 contient plus de nœuds qui viennent de S_2 que de S_1 .

La contribution indirecte $\omega_{jki}(\tau)$ à un instant donné τ doit être mise à jour à chaque requête q_k reçue par s_k de la part de s_i . Si τ_{i-1} correspond à l'estampille temporelle de la dernière requête q_k reçue de la part de s_i , le nouveau facteur de contribution relative pour une requête donnée à un certain instant τ_i peut être estimé par l'équation suivante :

$$\omega_{jki}(\tau_i) = (1 - \delta) \times \omega_{jki}(\tau_{i-1}) + \delta \times \omega(s_j, q_k, s_i) \quad (6.13)$$

où le facteur δ est utilisé comme précédemment pour ajuster l'influence des nouvelles requêtes sur la contribution indirecte basée sur les requêtes précédentes.

Notons qu'afin de maintenir normalisée la contribution relative de tous les services s_j à s_i , les valeurs de contribution indirecte $\omega_{jki}(\tau_i)$ de tous les services s_j qui contribuent via s_k à s_i doivent être mises à jour, qu'ils aient été envoyés dans le résultat de q_k ou non. Si les nœuds de s_j ne sont pas utilisés dans q_k , alors $\omega(s_j, q_k, s_i)$ à τ_i est 0.

Remarquons également que cette formule ne fait pas décroître le facteur de contribution à l'aide d'une fonction d'ancienneté. En effet, le caractère récent de l'accès aux données de s_j par le service s_i et ses clients est déjà pris en considération dans le calcul de la contribution totale $v_{ki}(\tau)$ envoyée par s_i à s_k , dont une partie ω_{jki} est donnée à s_j .

Quelques heuristiques plus complexes

Nous considérons trois services $s_i \rightarrow s_k \rightarrow s_j$. Afin d'estimer la contribution indirecte $\omega_{jki}(t)$ de s_j à s_i , s_k doit estimer l'utilisation des nœuds de s_j qu'il a envoyés à s_i par d'autres requêtes du système.

Nous savons que la contribution d'un nœud n à une requête q est calculée en fonction de l'utilisation de n dans les arbres embeddings et témoins de q . Par conséquent, afin d'estimer l'utilisation des nœuds n de s_j par le service s_i , le service s_k doit estimer la contribution de n aux arbres embedding et témoins des requêtes q_i . Nous pouvons prendre en compte les heuristiques suivantes pour chaque nœud n dans le résultat $\mathcal{R}(q_k)$ envoyé par q_k à s_i :

- les nœuds dans un résultat $\mathcal{R}(q_k)$ qui sont plus proches de la racine ont, de par la définition d'un embedding, statistiquement plus de chance d'être accédés lors de l'évaluation d'une requête q_i que des nœuds plus bas dans les arbres $\mathcal{R}(q_k)$. La contribution indirecte des nœuds plus près de la racine doit être plus élevée que celle des nœuds plus proches des feuilles.
- le score de sélection de ces mêmes nœuds dépend de la définition de s_i , et en particulier la probabilité que certains éléments ou données soient sélectionnés par une requête q_i . Par exemple si nous savons que s_i est un portail français, alors nous pouvons estimer qu'il est plus probable qu'il répondra aux requêtes qui sélectionnent l'élément ville avec la valeur *Paris* plutôt que *Londres*.

Afin d'estimer l'utilisation des nœuds n de s_j par d'autres services du système, à part s_i , nous devons savoir quels sont les nœuds qui vont être envoyés comme réponse par s_i à ses clients. Nous pouvons considérer que plus un nœud est proche du niveau des feuilles dans le résultat $\mathcal{R}(q_k)$ envoyé à s_i , plus il a de chance d'appartenir au résultat $\mathcal{R}(q_i)$ envoyé par s_i à ses clients et par conséquent d'être répliqué dans le système. En effet, rappelons que tout le sous-arbre ayant pour racine le plus petit ancêtre commun aux nœuds de la projection est retourné dans le résultat de la requête. Donc, plus un nœud est bas, plus il appartient à un grand nombre de sous-arbres, et par conséquent a des chances d'être transmis dans le résultat. Autrement dit les nœuds proches des feuilles ont une plus grande probabilité d'être utilisés par les services clients de s_i .

De ces heuristiques introduites ci-dessus, nous voyons que les nœuds situés proche de la racine des résultats envoyés par s_k à s_i ont un score estimé de contribution à des arbres embeddings et témoins important. En même temps, leurs score de contribution à des clients de s_i est plus faible que celui des nœuds proches des feuilles. Le fait que ces deux scores aient des valeurs opposées pourrait justifier notre première heuristique pour calculer la contribution indirecte, qui considère la même contribution pour tous les nœuds dans $\mathcal{R}(q_k)$, indépendamment de leur hauteur.

6.6 Implantation

Cette section décrit l'implantation du calcul d'importance de services ACTIVEXML dans le cas anonyme réalisée afin de tester la faisabilité de notre proposition. Après une présentation du calcul d'importance nous décrivons l'architecture du prototype implanté.

Calcul d'importance

Dans notre implantation actuelle, pour calculer la contribution directe d'un service s_j à un service s_k nous nous sommes basés sur les données produites par s_j qui sont renvoyées dans la réponse des requêtes q_k du service s_k . Nous considérons, contrairement au modèle de contribution que nous avons proposé, que seulement les nœuds qui se trouvent dans la réponse d'une requête ont contribué à cette requête. Nous considérons aussi que la contribution à une requête q_k est la même pour tous les nœuds dans son résultat. Plus précisément, l'équation 6.2 qui donne la contribution d'un nœud à une requête est remplacée par l'équation suivante :

$$C(n, q_k) = \frac{1}{|\mathcal{R}(q_k)|}$$

Le calcul des autres valeurs de contribution reste inchangé.

Le calcul d'importance des services se fait par l'algorithme distribué asynchrone présenté dans la section 3.4.3 (page 62) qui calcule l'importance $I_j(\tau)$ de chaque service s_j dans le système à un instant τ basée sur les valeurs de contribution directe et indirecte calculées jusqu'à τ . Les valeurs de contribution sont calculées localement par chaque service et, au moment du calcul, des valeurs d'importance v_{jk} définies d'après l'équation 6.12 (page 145) sont échangées entre chaque service s_k et ses voisins dans $In(\tau, s_k)$ et $Out(\tau, s_k)$.

Architecture

Le calcul de l'importance de services a été implanté au dessus de la plate-forme ACTIVEXML [19]. La figure 6.9 montre notre extension d'ACTIVEXML pour le calcul de valeurs de contribution et d'importance. Les pairs communiquent en utilisant l'implantation SOAP de la plate-forme Apache AXIS.

Le calcul de contribution est fait à l'aide de *handlers* AXIS. Le handler *RequestHandler* intercepte les appels à tous les services qui participent au calcul d'importance et extrait l'identité de leur client à partir du message SOAP encapsulé dans le *MessageContext*. Le handler *ResponseHandler* intercepte quant à lui toutes les réponses à tous les appels de service (requêtes). Il calcule les valeurs de contribution, et étiquette les nœuds de la réponse en ajoutant/modifiant les valeurs des attributs *sentby*. Ceci est fait en changeant le message SOAP encapsulé dans le *MessageContext* avant de le transmettre à la chaîne de traitements du moteur AXIS.

L'étape d'étiquetage des nœuds consiste à fixer l'attribut *sentby* de tous les nœuds à l'identité du service dont les réponses sont interceptées. Nous utilisons des transformations XSLT de la représentation DOM du résultat afin de réaliser cet étiquetage. Les valeurs de contribution sont mémorisées et mises à jour chaque fois que *RequestHandler* intercepte un message sortant.

A cause du très grand nombre possible de mises à jour des valeurs de contribution et à cause des opérations d'agrégation sur ces valeurs nécessaires pour le calcul d'importance, notre

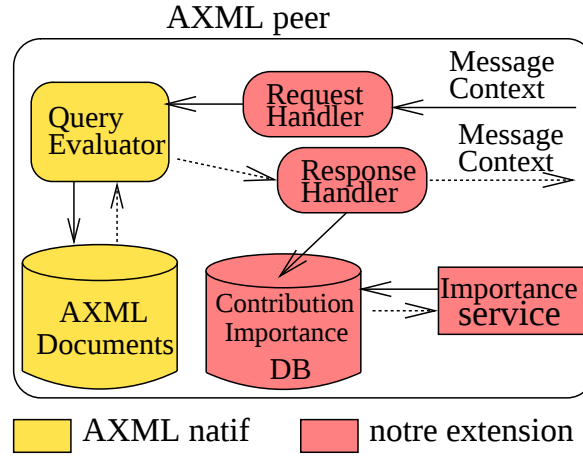


FIG. 6.9 – Architecture de l'implantation

implantation se base sur une base de données relationnelle (le SGBD HSQL [100]) pour mémoriser les valeurs de contribution. Nous utilisons deux tables, appelées Π et Ω , définies comme suit :

- $\Pi(s_k, s_j, \pi_{jk}, t)$ stockant pour chaque service s_k local et chaque service s_j distant les valeurs de contribution π_{jk} de chaque source s_j à l'évaluation du service local s_k , ainsi que l'estampille temporelle t du dernier accès de s_k aux données de s_j . Les tuples de Π sont mis à jour chaque fois que s_k utilise les données de s_j ;
- $\Omega(s_i, s_k, s_j, \pi_{jki}, \omega_{jki})$ stockant les valeurs de contribution indirecte ω_{jki} de s_j aux services s_i via le service local s_k . La valeur de π_{jki} est utilisée pendant le calcul d'importance pour stocker la partie de l'importance v_{ki} reçue par s_k de la part de s_i qui doit être envoyée au service s_j . Cette valeur est calculée donc comme étant $\pi_{jki} = \omega_{jki} \times v_{ki}$. Les valeurs de contribution indirecte dans Ω sont mises à jour à chaque fois que s_k envoie une réponse à s_i .

Chaque pair p_j offre un service spécial, appelé le *Service d'Importance (SI)* qui a deux opérations, $set(s_j, s_k, val)$ et $get(s_j)$ permettant respectivement de fixer et d'obtenir l'importance du service s_j , où s_k est le nom du service qui donne la valeur d'importance val à s_j .

Nous avons réalisé des premiers tests sur des données réelles afin de tester la faisabilité de notre approche. Nous étudions l'augmentation de la taille des fichiers après le processus d'étiquetage, ainsi que le temps nécessaire pour l'étiquetage. Les fichiers que nous avons utilisés pour nos tests sont des fichiers provenant de DBLP, SWISSPROT, SIGMODRECORD, IMDB (Internet Movie Database) et NASA. Quelques-uns de ces fichiers (SIGMODRECORD et IMDB) ont été transformés en XML à l'aide du générateur *W4F*. Le tableau 6.2 présente les résultats obtenus pour des fichiers de différentes tailles. La lettre *s* désigne un seul document XML pour un article, les mesures dans ce cas représentant la moyenne des valeurs sur la collection de documents respective. La lettre *c* signifie fichiers « collection » (comme par exemple le fichier qui correspond à SIGMODRECORD). *size+* représente la taille du fichier après que tous les nœuds aient été marqués. Nous remarquons

data source	size	size ⁺	incr.	processing
SigmodRecord(s)	2.8 Ko	3.1 Ko	9%	5ms
Nasa(s)	49.5 Ko	56.2 Ko	13%	42ms
IMDB(s)	2.3 Ko	2.6 Ko	11%	5ms
SigmodRecord(c)	497 Ko	584 Ko	17%	354ms
Nasa(c)	25.1 Mo	29.3 Mo	16%	1.9s
SwissProt(c)	115 Mo	142 Mo	23%	8.2s
DBLP(c)	419 Mo	506 Mo	20%	24.7s

TAB. 6.2 – Coût du processus d’étiquetage

que le marquage des nœuds a un impact limité sur la taille du fichier (une augmentation comprise dans [9%, 23%]). Cette valeur dépend du nombre de nœuds et de la proportion de données “brutes”(« raw ») dans le fichier. Notre test montre également la vitesse à laquelle nous pouvons réaliser le marquage.

6.7 Classement de services RSS

Dans le cadre d’un stage de Master 2 Recherche nous avons implanté un prototype qui est une variante de notre méthode de classement de services et de données distribuées dans le contexte des flux RSS. Nous présentons ici l’idée générale ainsi que les principes de l’implantation.

Modèle de données et de services

Flux RSS : RSS (REALY SIMPLE SYNDICATION) désigne une famille de formats XML utilisés pour la syndication de contenu web. Ce système est habituellement utilisé pour diffuser les mises à jour de sites dont le contenu change fréquemment, typiquement les sites d’information ou les blogs. L’utilisateur peut s’abonner aux flux, qui l’informent des dernières mises à jour de l’information qui l’intéresse. L’abonnement se fait à l’aide de lecteurs de flux (comme ceux qui sont incorporés dans des navigateurs web existants) ou des portails d’agrégation personnalisables (tels que IGOOGLE et NETVIBES).

La structure minimale d’un flux RSS est la suivante :

Services RSS : Les services que nous avons considérés offrent la possibilité d’interroger et de filtrer des flux RSS stockés localement. Chaque service recherche des items contenus dans des flux RSS contenant un ensemble de mots clé donné. Un service est implanté sous forme de requête XPATH de la forme `//item[contains(., « mots-clés »)]` où « mots-clés » sont les mots ou parties de mots que l’on souhaite voir apparaître dans les items de la réponse. La réponse de chaque service est une liste d’items donnée sous forme de flux RSS.

```
1 : <rss>
2 :   <item>
3 :     <title>Titre de la nouvelle</title>
4 :     <description>Description de la nouvelle</description>
5 :     <link>Lien vers une page web qui concerne la nouvelle</link>
6 :   </item>
7 :   <item>
8 :     ...
9 :   </item>
10: </rss>
```

Contribution et importance de données et de services : Le modèle de classement est basé sur les notes données par un expert sur les résultats de requêtes (services). Ce modèle dérive du modèle de classement présenté dans la section 6.3. Nous considérons que les nœuds qui contribuent à une requête sont ceux qui sont dans sa réponse et la contribution d'un nœud est estimée en fonction d'une note qui lui est attribuée par l'utilisateur. La contribution d'un service est calculée comme étant la moyenne des notes attribuées par l'utilisateur aux items dans ses résultats.

Implantation

Le but de cette implantation est de calculer l'importance d'un ensemble de services d'intégration et d'interrogation de flux RSS en fonction des notes reçues par leurs résultats pendant une période de la part de l'utilisateur. La notation des résultats des services se fait à travers une interface web.

Nous avons implanté plusieurs services RSS de même type. Chaque service est relié directement à différents flux RSS d'actualités présents sur le web comme par exemple ceux des journaux français LE FIGARO, LIBÉRATION, LE NOUVEL OBSERVATEUR, METRO, VINGT MINUTES, LE MONDE Les fragments de flux RSS présents sur les sites des différents journaux (auxquels sont reliés nos services web) contiennent en moyenne un nombre d'item très différent, allant de 12 seulement pour METRO jusqu'à 109 pour Libération. Chaque service peut aussi s'abonner aux autres services RSS de manière à récupérer une plus grande quantité d'informations.

Chaque service interroge périodiquement (toutes les 5 minutes) les services auxquels il est abonné et stocke les résultats reçus localement. En parallèle, l'utilisateur note à travers l'interface web les résultats des différents services. Les valeurs de contribution de chaque service dont les résultats sont notés sont stockées et envoyées régulièrement aux services concernés qui recalculent ainsi leur importance. La figure 6.10 illustre le fonctionnement de notre prototype.

Interrogation et Abonnements : Chaque service a une méthode *getItems(String XpathExpr)* qui prend comme paramètre une chaîne de caractères représentant la requête XPATH ainsi formulée.

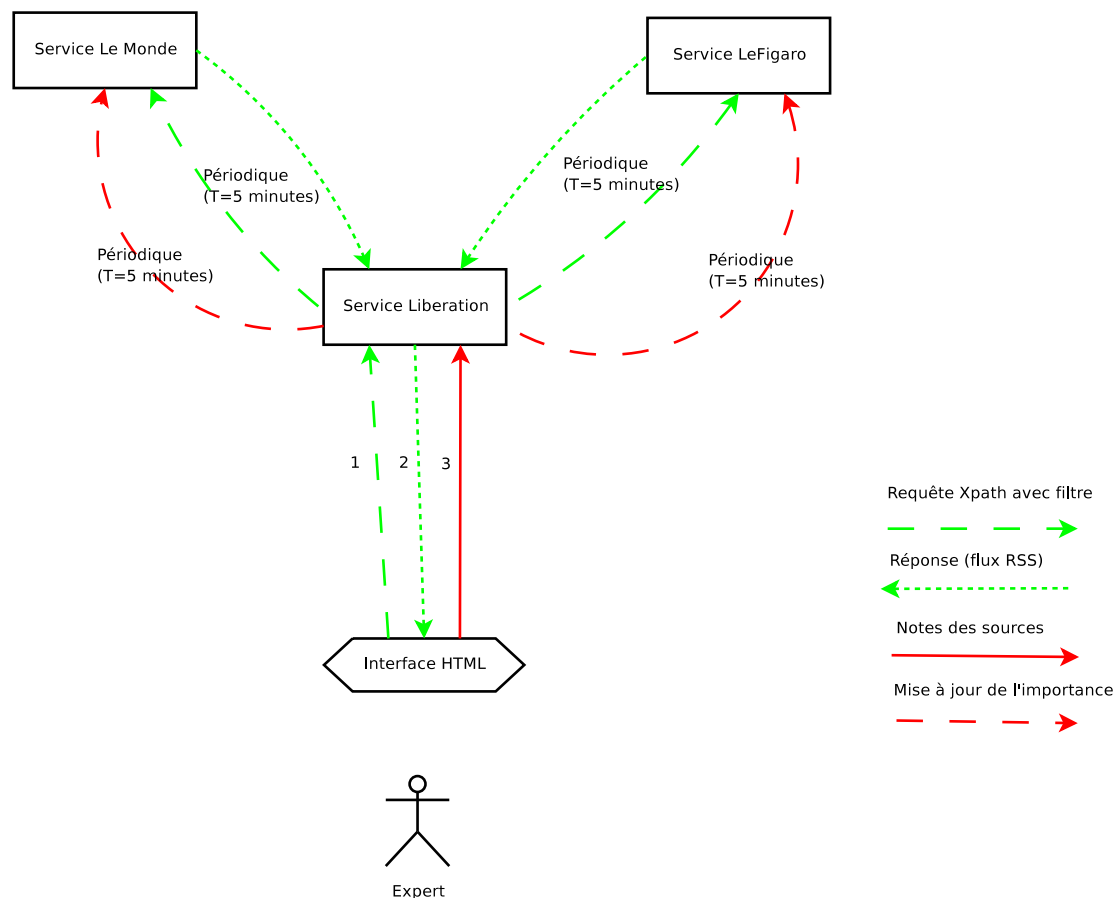


FIG. 6.10 – Scénario de fonctionnement de 3 services notés par une interface

Les abonnements de chaque service sont spécifiés dans un fichier de configuration XML, dont la structure est présentée dans la figure 6.7.

Chaque service interroge périodiquement (toutes les 5 minutes) le service cible, en appelant sa méthode *getItems* et stocke sa réponse dans un fichier local. Ainsi donc, pour créer un nouvel abonnement, il suffit d'ajouter un nouveau noeud XML dans ce fichier, en spécifiant l'adresse web du serveur auquel on souhaite s'abonner dans le noeud *epr* et les mots-clés dans un noeud *filter*.

Interface de notation : L'importance des services est calculée à partir d'une notation fournie par un « expert ». Nous avons développé une interface HTML/JAVASCRIPT, illustrée dans la figure 6.11 qui permet à l'utilisateur de réaliser les actions suivantes :

- Sélection du service à appeler
- Saisie du (des) mots clé
- Interrogation du service

```
1: <subscriptions>
2:   <subscription>
3:     <epr> adresse du services web auquel on souhaite s'abonner </epr>
4:     <filter>
5:       mots clé insérés dans l'expression XPath pour l'interrogation
6:     </filter>
7:   </subscription>
8:   ...
9: </subscriptions>
```

- Affichage du résultat (des items)
- Notation item par item

L'utilisateur (l'expert) peut attribuer à l'aide de l'interface les notes à chacun des items reçus en réponse : 0,1 (item peu pertinent), 0,5 (item acceptable) ou 1 (item très pertinent).

Calcul d'importance : Les valeurs de contribution de service sont calculées à partir des notes reçues à l'aide de l'interface. L'importance d'un service est la somme des contributions données par d'autres service. Nous avons implanté le calcul d'importance dans le cas où l'on connaît toutes les sources de données ainsi que dans le cas anonyme.

Lorsque l'expert clique sur le bouton *Noter*, l'interface calcule la contribution de chaque source de données comme étant la moyenne des notes. Ces valeurs de contribution sont renvoyées au service que l'on a choisi d'interroger, en appelant la méthode *setImportance()*, du service où elles sont stockées localement. Chaque service possède un fichier *importance.xml* dans lequel sont stockées toutes les données relatives à l'importance. Périodiquement, chaque service envoie les valeurs de contribution stockées localement aux services les ayant fournies (qui sont les sources originelles de données ou pas, en fonction du scénario choisi), en appelant leur méthode *setImportance()*.

Bilan : Ce premier travail nous a permis d'étudier la faisabilité de notre modèle à un contexte de publication de données à l'aide de flux RSS. Les résultats très encourageants obtenus sur un nombre restreints de sources de flux RSS nous permettent d'envisager un passage à l'échelle prochainement. De plus nous pensons intégrer ces travaux et ce prototype dans le projet RNTL WEB-CONTENT.

6.8 Conclusion

Ce chapitre présente un modèle de classement de données XML et de services dans un entrepôt de données distribué [19] en s'appuyant sur le modèle générique présenté dans le chapitre 3.

Selectionnez le service que vous voulez interroger

Liberation ▼

Entrez des mots cles pour chercher dans les flux RSS

req

Resultats

Zoom : Mappemonde des requêtes Google

Pour sonder « les préférences et intérêts de ses prochains », un étudiant allemand a choisi d'analyser leurs recherches

Mauvais ▼

Les requins peaux bleues chassés sur leur mer

Des écologistes protestent contre un concours de pêche amateur en Atlantique.

Moyen ▼

FIG. 6.11 – Interface de notation

Nos scores d'importance sont complémentaires des méthodes existantes qui classent les fragments réponses XML fondées sur des techniques de recherche d'information.

Nous considérons des services implantés comme des requêtes exprimées par des motifs d'arbre paramétrés et une liste de projection. La contribution a été définie à plusieurs niveaux de granularité : nœud XML, requête, service. La contribution des données est basée sur leur score de sélection et d'accès aux données d'intérêt pour les requêtes. La contribution d'une source est une agrégation des valeurs de contribution des données qu'elle a produites. Nous avons étudié le calcul de contribution et d'importance des sources dans un contexte où toutes les sources des données sont connues, ainsi que dans un contexte anonyme. Dans ce dernier cas, l'importance des sources ne peut plus être calculée directement sur la base de leur contribution directe, mais elle est calculée en distribuant des valeurs d'importance sur des chemins d'utilisation à l'aide des scores de contribution indirecte.

Nous avons réalisé une première implantation du calcul d'importance dans un contexte anonyme, au-dessus du système ACTIVEXML. Nous avons également décrit un prototype de classement de services d'intégration et d'interrogation des flux RSS qui utilise notre modèle d'importance.

Chapitre 7

Conclusion et perspectives

Cette thèse présente un modèle générique d'importance de services basé sur les liens de collaboration. Nous avons étudié trois applications de ce modèle pour : (i) le classement de services basé sur les appels, (ii) la définition de stratégies de cache distribuées et (iii) le classement de sources et de données basé sur l'utilisation des données.

Modèle générique d'importance de services

Nous proposons un modèle de classement générique basé sur la collaboration entre les services. Notre première intention a été de définir des scores d'importance qui peuvent être appliqués à différents types de « services ». Ainsi, notre modèle d'importance est générique et ne dépend pas d'un modèle de services particulier. La seule hypothèse faite est que le « service » à classer est un nœud dans un système, qui *communique* avec d'autres nœuds en échangeant des messages. Nous considérons ainsi qu'un *service* peut être par exemple un pair, un service de calcul, une requête, une exécution de requête, etc.

Notre modèle de classement est basé sur la notion de contribution d'un service, qui reflète les liens d'utilisation entre les services. Ainsi si un service *A* utilise un service *B*, on considère que *B contribue* au service *A*. Un premier problème auquel nous nous sommes confrontés a été de caractériser la notion de contribution de service. Nous distinguons entre l'utilisation/contribution d'un service par les appels qui reflète sa contribution aux autres services par les calculs qu'il fait pendant son exécution et l'utilisation/contribution d'un service par ses données. Dans notre modèle de classement de services, nous généralisons le calcul de contribution pour prendre en considération non seulement la contribution directe entre deux services, mais également la contribution *indirecte* d'un service à d'autres services du système, à travers ses clients directs. En effet l'information obtenue par *B* après l'utilisation de *C* a pu être nécessaire par la suite pour répondre au service *A* et récursivement à d'autres services du système. En utilisant ces deux notions de contribution (directe et indirecte) nous définissons la contribution d'un service sur des *chemins d'utilisation* à n'importe quel autre service. L'importance d'un service est sa contribution totale aux autres services, un service étant *important* s'il contribue beaucoup par son utilisation (ou celle de ses données) à d'autres services du système.

Le calcul d'importance de services est distribué sur les différents services du système, chaque service calculant ainsi sa propre importance. Nous avons proposé un algorithme de calcul *asynchrone* qui évite la génération des messages supplémentaires de calcul d'importance, en encapsulant les informations nécessaires au calcul d'importance dans les messages applicatifs.

Classement de services basé sur les appels

Nous avons appliqué dans un premier temps le modèle générique de classement de services au classement de services basé sur les appels et nous avons étudié la définition des scores de contribution entre les services dans ce contexte. Un service qui a besoin de plusieurs autres services pour ses exécutions doit définir d'abord *comment* chacun de ces services contribue par rapport aux autres. Il s'agit donc de donner aux différents services utilisés un score relatif qui estime la contribution locale *statique* des différents services à sa qualité, en fonction de la sémantique de l'application. Ce score de contribution est indépendant des appels effectués entre les services. Les instants des appels de services est pris en compte par des scores d'*usage de service*. Afin de caractériser la liaison entre les appels reçus de la part des clients et les appels qu'il fait à d'autres services, nous avons introduit le score d'*utilité d'appel*. Ainsi, un service *C* ne contribue pas à un service *A* lorsqu'il est utilisé par *B* si les appels de *B* vers *C* ne sont pas utiles pour les appels de *A* vers *B*. Les scores d'usage de service et d'utilité d'appel pondèrent la contribution statique, de telle manière qu'un service qui a une contribution statique élevée n'est pas important s'il n'est pas utilisé.

Nous avons implanté le calcul d'importance de services avec les algorithmes synchrone et asynchrone. Nos expériences ont montré qu'indépendamment de la topologie du graphe de services, l'algorithme asynchrone représente un meilleur choix que l'algorithme synchrone, puisqu'il permet de trouver les scores d'importance des services plus rapidement avec moins de messages échangés entre les services.

Stratégies de cache distribuées

Nous nous sommes posés ensuite le problème de l'utilisation des scores de contribution afin d'améliorer une stratégie de cache distribuée. Nous sommes partis du constat que le calcul d'importance de services permet de connaître quelle est sa contribution à tous les autres services du système sur tous les chemins d'utilisation.

Nous appliquons le calcul sur les chemins introduit dans le modèle de classement générique pour estimer l'*utilité* (le bénéfice) de mettre en cache une donnée pour optimiser les coût des requêtes du système. Nous avons analysé expérimentalement cette nouvelle heuristique de cache et nous l'avons comparée à une méthode de cache client-serveur qui ne prend pas en compte les chemins. Notre nouvelle politique de cache basée sur l'utilité sur les chemins s'est avérée plus adaptée dans le contexte distribué en réduisant les données matérialisés inutilement. Les expériences ont montré que la redondance de cache peut également être utile dans certains cas, notamment avec des fréquences de rafraîchissement élevées et pour améliorer le coût des requêtes moins populaires.

Classement de services basé sur les données

Le problème auquel nous nous sommes intéressés ensuite est l'application du modèle générique au classement des services par leurs données. Nous considérons qu'un service produisant des données est utilisé non seulement par les appels qu'il reçoit (appels qui demandent aussi des données) mais aussi par l'utilisation des données transmises dans ses réponses. Les services de données sont décrits par des requêtes (motifs d'arbre) sur des entrepôts locaux. La contribution d'un service dans ce cas dépend de la manière dont ses données contribuent à l'évaluation d'autres requêtes. Cette contribution est définie en analysant les différentes façons dont une donnée peut être utile pour évaluer une requête. Nous faisons la distinction entre les données qui sont d'intérêt pour la requête (c'est à dire les données mentionnées explicitement dans la définition de la requête) et les données dont on a besoin pour *accéder* à ces données. Une source (service) de données est alors importante si elle produit des données qui contribuent beaucoup à l'évaluation des requêtes dans le système. L'importance d'une source peut être estimée par la somme des contributions de ses données à toutes les requêtes. Dans un système où l'on ne connaît pas la source originelle des données, l'importance d'une source dépend de l'importance obtenue à travers les services par lesquels sont passées ses données. Il est important alors de connaître la manière dont chaque service doit distribuer l'importance qu'il reçoit parmi les sources qui ont participé à obtenir ces données. Nous avons implanté ce modèle de classement sur un système réel, ACTIVEXML.

Perspectives

Enrichissement du modèle de cache

Notre modèle de cache définit un score de cache pour chaque donnée qui est composé de deux parties : (i) le coût enregistré par le pair lorsqu'il reçoit la donnée et (ii) l'utilité de mettre cette donnée en cache sur chaque chemine d'évaluation de requête qui a accédé à cette donnée.

Estimation de coût : En ce qui concerne le coût, nous utilisons le dernier coût observé de la requête pour estimer le bénéfice de sa matérialisation. Il est possible que les serveurs qui ont fourni cette donnée aient aussi mis/enlevé des données dans leur cache ce qui implique que le coût d'évaluation de la requête ait pu changer. Une première extension de notre modèle de cache consiste à définir une estimation plus précise du coût d'une donnée qui est mise dans la mémoire cache d'un pair, en prenant en compte les changements possibles dans la mémoire de cache d'autres pairs sur son chemin d'évaluation.

Estimation des requêtes futures : Une mise en cache d'une donnée est effectivement utile s'il existe des requêtes futures qui vont l'utiliser. Dans notre calcul d'utilité nous estimons la probabilité qu'une requête sera exécutée dans le futur sur la base de sa fréquence dans le passé. L'estimation de l'utilisation future d'une donnée est en réalité plus compliquée et peut dépendre de trois facteurs : (i) la fréquence à laquelle les requêtes sont posées (ii) le changement dans la

topologie du graphe (les décisions de routage des pairs) et (iii) le rafraîchissement des données.

L'estimation de la fréquence des requêtes n'est pas un problème nouveau, et il existe en effet de nombreux travaux dans le contexte des stratégies de cache web [39] et des réseaux pair-à-pair [113]. Nous pensons que de telles techniques pourraient servir comme point de départ afin d'améliorer notre application de cache. Même si la fréquence des requêtes à optimiser est connue, la réception des chemins de ces requêtes par un certain pair est conditionnée par la dynamique du réseau déterminée par la connexion/déconnexion des pairs (le « churn ») et par le routage des requêtes distribuées. Nous pensons appliquer des techniques d'analyse et de caractérisation du « churn » [172] et de la *durée de vie* des pairs dans le réseau pair-à-pair ainsi que des estimation des décisions de routage des pairs afin de déduire les chemins d'évaluation futurs.

Nous avons constaté expérimentalement que le coût de rafraîchissement des données avec notre méthode de cache est plus élevé si les données sont rafraîchies souvent (lié au manque de redondance). Nous pensons que notre modèle de cache peut être combiné avec des techniques plus avancées de synchronisation de cache [45, 140].

Extension du modèle de classement de services et de données

Une application de notre modèle de classement de services est basée sur le calcul de la contribution des données d'un service (d'une source) à l'évaluation des requêtes exprimées sous forme de motifs d'arbre avec des sélections et des projections sur les documents XML. Il a été montré que des requêtes exprimées en XQUERY avec des opérations plus complexes comme comme la jointure, la différence où l'agrégation peuvent être décomposées en plusieurs motifs d'arbre [103]. Une première extension de notre modèle est de définir la contribution des données à des requêtes XQUERY avec des opérations plus complexes.

La contribution des données dépend de la définition des requêtes qui parfois n'est pas optimisée. Par exemple, pour la requête : *trouver les artistes et la date de leur concerts à Paris* que nous avons considéré dans le chapitre 6, si nous savons que tous les concerts dans les documents XML sur lesquels elle est évaluée sont à Paris, nous pouvons omettre le critère de sélection sur les nœuds de type ville dans la définition de l'arbre de la requête et considérer que les nœuds de ce type ne contribuent pas. En appliquant des techniques de minimisation de requêtes [13] nous pouvons ainsi raffiner le calcul de contribution pour prendre en considération seulement les données qui contribuent « réellement ».

La définition de la contribution des nœuds se base sur un score *de sélection* qui est combiné avec un *score d'accès* aux données. La pertinence de ces mesures n'a pas été testée en pratique, et nous pensons qu'en fonction de l'application les utilisateurs peuvent être intéressés plus par un certain score que par un autre. Par exemple, lorsqu'il s'agit du rafraîchissement des données, il pourrait être intéressant de rafraîchir en priorité les données qui ont un score d'accès élevé, car ce sont ces données qui permettent aux requêtes d'accéder à leurs informations d'intérêt. Au contraire, lorsqu'on veut trier les données pour les présenter à l'utilisateur, il serait indiqué de donner plus

de poids à leur score de sélection, une donnée avec un score de sélection élevé est probablement considéré comme étant une donnée d'intérêt pour plusieurs utilisateurs. Des méthodes existantes pour annoter les nœuds des arbres des requêtes avec des scores exprimant l'importance de chaque nœud pour l'utilisateur [7] pourraient nous permettre de raffiner l'estimation de la contribution pour chaque nœud.

Application au classement de flux RSS

La quantité d'information disponible sur l'Internet est immense et chaque utilisateur est intéressé seulement par une partie de cette information correspondant à ses intérêts et ses préoccupations personnelles. L'évolution du web a permis aux utilisateurs de pouvoir créer facilement leur propre espace d'information personnalisée en s'abonnant aux sources qui les intéressent qui leur transmettent les informations sous forme de flux d'annotation, les flux RSS. Ces flux sont spécifiés dans un format XML simplifié et représentent une séquence de messages appelés *item*. Actuellement les systèmes d'agrégation de flux sont seulement des systèmes client-serveur qui agrègent sur un seul portail centralisé les flux produits par plusieurs sources, comme par exemple *Google News* [85] qui agrège les informations de plusieurs sources de nouvelles. Nous pouvons imaginer que ces systèmes vont évoluer de plus en plus vers des systèmes distribués avec des sources d'informations qui collaborent, et où chaque source peut agréger les flux produits par d'autres sources et les intégrer avec des informations locales avant de produire ses propres flux. Nous avons ici à nouveau le problème des données produites par une source qui sont répliquées et utilisées par d'autres sources.

Dans le cadre d'un stage de recherche M2 nous avons déjà implanté un premier prototype d'un réseau de services RSS dans lequel nous avons considéré des services simples d'intégration de nouvelles RSS définis sous forme de requêtes XPATH qui filtrent les flux entrants. Nous avons proposé un modèle d'importance de services de nouvelles similaire au modèle de classement des sources basé sur leurs données, qui calcule l'importance de chaque source en fonction des notes des utilisateurs sur les données qu'elles ont produites. Dans le cadre du projet ANR-MDCO ROSES (qui débute en janvier 2008), nous allons continuer ce travail et enrichir le modèle d'importance de flux et de leurs sources. Actuellement le modèle que nous avons défini se base sur les notes données par l'utilisateur qui reflètent leurs préférences. Ce premier modèle de notation n'est pourtant pas applicable en pratique, car les utilisateurs ne sont pas toujours disposés à donner leurs avis et indiquer leur préférences personnelles [154]. Une première modification possible serait de remplacer la notation des données par des techniques d'apprentissage des préférences de l'utilisateur.

Bibliographie

- [1] K. Aberer and J. Wu. A Framework for Decentralized Ranking in Web Information Retrieval. In *Proc. Intl. Asian-Pacific Web Conference (APWeb)*, pages 213–226, 2003.
- [2] S. Abiteboul, O. Benjelloun, B. Cautis, I. Manolescu, T. Milo, and N. Preda. Lazy Query Evaluation for Active XML. In *Proc. ACM Symp. on the Management of Data (SIGMOD)*, pages 227–238, 2004.
- [3] S. Abiteboul, O. Benjelloun, and T. Milo. Positive Active XML. In *Proc. ACM Symp. on Principles of Database Systems (PODS)*, pages 35–45, 2004.
- [4] S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, and T. Milo. Dynamic XML Documents with Distribution and Replication. In *Proc. ACM Symp. on the Management of Data (SIGMOD)*, pages 527–538, 2003.
- [5] S. Abiteboul, M. Preda, and G. Cobena. Adaptive on-line page importance computation. In *Proc. Intl. World Wide Web Conference (WWW)*, pages 280–290, 2003.
- [6] A. Aitken. On bernoulli’s numerical solution of algebraic equations. *Proc. Roy. Soc. Edinburgh*, 46 :289–305, 1926.
- [7] S. Al-Khalifa, C. Yu, and H. V. Jagadish. Querying Structured Text in an XML Database. In *Proc. ACM Symp. on the Management of Data (SIGMOD)*, pages 4–15, 2003.
- [8] R. Albert, H. Jeong, and A.-L. Barabási. The Diameter of the World Wide Web. *Science*, 286 :509–512, 1999.
- [9] M. Ali, M. Consens, X. Gu, Y. Kanza, F. Rizzolo, and R. Stasiu. Efficient, Effective and Flexible XML Retrieval Using Summaries. In *INEX Workshop*, pages 6–20, 2006.
- [10] B. Amann and C. Constantin. A Collaborative Caching Policy Based on Path Materialization Scores. In *23èmes Journées Bases de Données Avancées, Actes (Informal Proceedings)*, 2007. á paraître.
- [11] B. Amann and C. Constantin. Collaborative Cache Based on Path Scores. In *Proc. Intl. Conf. on Web Information Systems Engineering (WISE)*, 2007. á paraître.
- [12] S. Amer-Yahia, C. Botev, and J. Shanmugasundaram. Texquery : a full-text search extension to xquery. In *Proc. Intl. World Wide Web Conference (WWW)*, pages 583–594, 2004.
- [13] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. Minimization of Tree Pattern Queries. In *Proc. ACM Symp. on the Management of Data (SIGMOD)*, pages 497–508, 2001.

- [14] S. Amer-Yahia, N. Koudas, A. Marian, D. Srivastava, and D. Toman. Structure and Content Scoring for XML. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 361–372, 2005.
- [15] S. Amer-Yahia, L. V. S. Lakshmanan, and S. Pandit. FleXPath : Flexible Structure and Full-Text Querying for XML. In *Proc. ACM Symp. on the Management of Data (SIGMOD)*, pages 83–94, 2004.
- [16] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. DBProxy : A Dynamic Data Cache for Web Applications. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, pages 821–831, 2003.
- [17] A. Arasu, J. Cho, H. Garcia-Molina, A. Paepcke, and S. Raghavan. Searching the Web. *ACM Trans. Inter. Tech.*, 1(1) :2–43, 2001.
- [18] A. Arasu, J. Novak, A. Tomkins, and J. Tomlin. PageRank Computation and The Structure Of The Web : Experiments and Algorithms. In *Proc. Intl. World Wide Web Conference (WWW), Poster Track*, 2002.
- [19] Active XML. url : <http://www.activexml.net/>.
- [20] Badrank. url : <http://pr.efactory.de/e-pr0.shtml>.
- [21] R. Baeza-Yates, C. Castillo, and V. López. Pagerank Increase under Different Collusion Topologies. In *First International Workshop on Adversarial Information Retrieval on the Web (AIRWeb)*, 2005.
- [22] R. A. Baeza-Yates, F. Saint-Jean, and C. Castillo. Web Structure, Dynamics and Page Quality. In *Proc. of Intl. Symposium on String Processing and Information Retrieval (SPIRE)*, pages 117–130, 2002.
- [23] A. Balmin, V. Hristidis, and Y. Papakonstantinou. ObjectRank : Authority-Based Keyword Search in Databases. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 564–575, 2004.
- [24] Z. Bar-Yossef, A. Z. Broder, R. Kumar, and A. Tomkins. Sic Transit Gloria Telae : Towards an Understanding of the Web’s Decay. In *Proc. Intl. World Wide Web Conference (WWW)*, pages 328–337, 2004.
- [25] A. Benczur, K. Csalogany, T. Sarlos, and M. Uher. Spamrank - Fully Automatic Link Spam Detection, 2005.
- [26] K. Berberich, M. Vazirgiannis, and G. Weikum. Time-Aware Authority Ranking. *Internet Mathematics*, 2(3) :309–340, 2005.
- [27] D. P. Bertsekas and J. N. Tsitsiklis. Some Aspects of Parallel and Distributed Iterative Algorithms-A Survey. *Automatica*, 27(1) :3–21, 1991.
- [28] A. Bestavros and S. Jin. Popularity-Aware Greedy Dual-Size Web Proxy Caching Algorithms. In *Proc. Intl. Conf. on Distributed Computing Systems (ICDCS)*, pages 254–261, 2000.
- [29] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword Searching and Browsing in Databases using BANKS. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, 2002.

- [30] K. Bharat and M. R. Henzinger. Improved Algorithms for Topic Distillation in a Hyperlinked Environment. In *Proc. Intl. Conference on Research and Development in Information Retrieval (SIGIR)*, pages 104–111, 1998.
- [31] M. Bianchini, M. Gori, and F. Scarselli. Inside PageRank. *ACM Trans. Inter. Tech.*, 5(1) :92–128, 2005.
- [32] P. Boldi, M. Santini, and S. Vigna. PageRank as a Function of the Damping Factor. In *Proc. Intl. World Wide Web Conference (WWW)*, pages 557–566, 2005.
- [33] I. M. Bomze and W. Gutjahr. The Dynamics of Self-Evaluation. *Appl. Math. Comput.*, 64(1) :47–63, 1994.
- [34] C. Bornhövd, M. Altinel, S. Krishnamurthy, C. Mohan, H. Pirahesh, and B. Reinwald. DB-Cache : Middle-tier Database Caching for Highly Scalable e-Business Architectures. In *Proc. ACM Symp. on the Management of Data (SIGMOD)*, page 662, 2003.
- [35] A. Borodin, G. O. Roberts, J. S. Rosenthal, and P. Tsaparas. Finding Authorities and Hubs from Link Structures on the World Wide Web. In *Proc. Intl. World Wide Web Conference (WWW)*, pages 415–429, 2001.
- [36] A. Borodin, G. O. Roberts, J. S. Rosenthal, and P. Tsaparas. Link Analysis Ranking : Algorithms, Theory, and Experiments. *ACM Trans. Inter. Tech.*, 5(1) :231–297, 2005.
- [37] Business process execution language for web services. url : <http://www.ibm.com/developerworks/library/specification/ws-bpel/>.
- [38] J. T. Bradley, D. V. de Jager, W. J. Knottenbelt, and A. Trifunovic. Hypergraph Partitioning for Faster Parallel PageRank Computation. In *Proc. European Performance Engineering Workshop (EPEW)*, pages 155–171, 2005.
- [39] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions : Evidence and Implications. In *Proc. Intl. Conference on Computer Communications (INFOCOM)*, pages 126–134, 1999.
- [40] S. Brin, R. Motwani, L. Page, and T. Winograd. What Can You Do With a Web in Your Pocket ? *Data Engineering Bulletin*, 21(2) :37–47, 1998.
- [41] S. Brin and L. Page. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks*, 30(1-7) :107–117, 1998.
- [42] A. Z. Broder, R. Lempel, F. Maghoul, and J. Pedersen. Efficient PageRank Approximation via Graph Aggregation. *Information Retrieval*, 9(2) :123–138, 2006.
- [43] A. Z. Broder, R. Lempel, F. Maghoul, and J. O. Pedersen. Efficient Pagerank Approximation via Graph Aggregation. In *Proc. Intl. World Wide Web Conference (WWW)*, pages 484–485, 2004.
- [44] P. Cao and S. Irani. Cost-Aware WWW Proxy Caching Algorithms. In *USENIX Symposium on Internet Technologies and Systems*, 1997.
- [45] P. Cao and C. Liu. Maintaining Strong Cache Consistency in the World Wide Web. *IEEE Trans. Computers*, 47(4) :445–457, 1998.

- [46] J. Caverlee, L. Liu, and D. Rocco. Discovering and Ranking Web Services with BASIL : a Personalized Approach with Biased Focus. In *Proc. Intl. Conf. on Service-Oriented Computing (ICSOC)*, pages 153–162, 2004.
- [47] S. Chakrabarti. Dynamic Personalized PageRank in Entity-Relation Graphs. In *Proc. Intl. World Wide Web Conference (WWW)*, pages 571–580, 2007.
- [48] S. Chakrabarti, B. E. Dom, S. R. Kumar, P. Raghavan, S. Rajagopalan, A. Tomkins, D. Gibson, and J. Kleinberg. Mining the Web’s Link Structure. *Computer*, 32(8) :60–67, 1999.
- [49] L. Chen and E. A. Rundensteiner. ACE-XQ : A Cache-aware XQuery Answering System. In *Proc. Intl. Workshop on the Web and Databases (WebDB)*, pages 31–36, June 2002.
- [50] Y.-Y. Chen, Q. Gan, and T. Suel. Local Methods for Estimating Pagerank Values. In *Proc. Intl. Conf. on Information and Knowledge Management (CIKM)*, pages 381–389, 2004.
- [51] L. Cherkasova and G. Ciardo. Role of Aging, Frequency, and Size in Web Cache Replacement Policies. In *Proc. Intl. Conf. of High-Performance Computing and Networking (HPCN)*, pages 114–123, 2001.
- [52] S. Chien, C. Dwork, R. Kumar, D. Simon, and D. Sivakumar. Link Evolution : Analysis and Algorithms. *Internet Mathematics*, 1(3) :277–304, 2004.
- [53] P. A. Chirita, W. Nejdl, R. Paiu, and C. Kohlschütter. Using ODP Metadata to Personalize Search. In *Proc. Intl. Conference on Research and Development in Information Retrieval (SIGIR)*, pages 178–185, 2005.
- [54] P.-A. Chirita, W. Nejdl, M. T. Schlosser, and O. Scurtu. Personalized Reputation Management in P2P Networks. In *Proc. Intl. ISWC Workshop on Trust, Security, and Reputation on the Semantic Web*, 2004.
- [55] J. Cho and H. Garcia-Molina. Parallel crawlers. In *Proc. Intl. World Wide Web Conference (WWW)*, pages 124–135, 2002.
- [56] J. Cho and S. Roy. Impact of Search Engines on Page Popularity. In *Proc. Intl. World Wide Web Conference (WWW)*, pages 20–29, 2004.
- [57] J. Cho, S. Roy, and R. E. Adams. Page Quality : in Search of an Unbiased Web Ranking. In *Proc. ACM Symp. on the Management of Data (SIGMOD)*, pages 551–562, 2005.
- [58] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSEarch : A Semantic Search Engine for XML. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 45–56, 2003.
- [59] C. Constantin and B. Amann. Usage-based Ranking of Distributed XML Data. In *SAC*, 2008. à paraître.
- [60] C. Constantin, B. Amann, and D. Gross-Amblard. A Link-Based Ranking Model for Services. In *Proc. Intl. Conf. on Cooperative Information Systems (CoopIS)*, 2006.
- [61] C. Constantin, B. Amann, and D. Gross-Amblard. A Link-Based Ranking Model for Services. In *22èmes Journées Bases de Données Avancées, Actes (Informal Proceedings)*, 2006.
- [62] C. Constantin, B. Amann, and D. Gross-Amblard. Un Modèle de Classement de Services par Contribution et Utilité. *Ingénierie des Systèmes d’Information*, pages 33–60, 2007.

- [63] G. M. D. Corso, A. Gullí, and F. Romani. Fast PageRank Computation via a Sparse Linear System. *Internet Mathematics*, 2(3) :251–273, 2005.
- [64] G. M. D. Corso, A. Gullí, and F. Romani. Comparison of Krylov Subspace Methods on the PageRank Problem. *Journal of Computational and Applied Mathematics*, pages 1–12, 2006.
- [65] P. K. Desikan, N. Pathak, J. Srivastava, and V. Kumar. Divide and Conquer Approach for Efficient Pagerank Computation. In *Proc. Intl. Conf. on Web Engineering (ICWE)*, pages 233–240, 2006.
- [66] M. Diligenti, M. Gori, and M. Maggini. A Unified Probabilistic Framework for Web Page Scoring Systems. *IEEE Trans. on Knowledge and Data Engineering (TKDE)*, 16(1) :4–16, 2004.
- [67] J. Dilley, B. M. Maggs, J. Parikh, H. Prokop, R. K. Sitaraman, and W. E. Weihl. Globally Distributed Content Delivery. *IEEE Internet Computing*, 6(5) :50–58, 2002.
- [68] L. Ding, T. Finin, A. Joshi, R. Pan, R. S. Cost, Y. Peng, P. Reddivari, V. Doshi, and J. Sachs. Swoogle : a Search and Metadata Engine for the Semantic Web. In *Proc. Intl. Conf. on Information and Knowledge Management (CIKM)*, pages 652–659, 2004.
- [69] The open directory project. url : <http://www.dmoz.org/>.
- [70] M. Eirinaki and M. Vazirgiannis. Usage-Based PageRank for Web Personalization. In *Proc. of the IEEE International Conference on Data Mining (ICDM)*, pages 130–137, 2005.
- [71] N. Eiron, K. S. McCurley, and J. A. Tomlin. Ranking the Web Frontier. In *Proc. Intl. World Wide Web Conference (WWW)*, pages 309–318, 2004.
- [72] F. Emekçi, O. D. Sahin, D. Agrawal, and A. E. Abbadi. A Peer-to-Peer Framework for Web Service Discovery with Ranking. In *Proc. Intl. Conf. on Web Services (ICWS)*, pages 192–199, 2004.
- [73] L. Fegaras. XQuery Processing with Relevance Ranking. In *Proc. Intl XML Database Symposium (XSym)*, pages 51–65, 2004.
- [74] G. Feng, T.-Y. Liu, Y. Wang, Y. Bao, Z. Ma, X.-D. Zhang, and W.-Y. Ma. AggregateRank : Bringing Order to Web Sites. In *Proc. Intl. Conference on Research and Development in Information Retrieval (SIGIR)*, pages 75–82, 2006.
- [75] N. Fuhr and K. Großjohann. XIRQL : An XML query language based on information retrieval concepts. *ACM Trans. Database Syst.*, 22(2) :313–356, 2004.
- [76] J. S. R. Gareth O. Roberts. Downweighting tightly knit communities in World Wide Web rankings. *Advances and Applications in Statistics (ADAS)*, 3 :199–216, 2003.
- [77] E. Garfield. *Citation Indexing : Its Theory and Application in Science, Technology, and Humanities*. John Wiley & Sons Inc, New York, 1979.
- [78] F. Geerts, H. Mannila, and E. Terzi. Relational Link-Based Ranking. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 552–563, 2004.

- [79] D. Gibson, J. Kleinberg, and P. Raghavan. Inferring Web Communities from Link Topology. In *Proc. ACM Conf. on Hypertext and Hypermedia : links, objects, time and space—structure in hypermedia systems*, pages 225–234, 1998.
- [80] C. L. Giles, K. Bollacker, and S. Lawrence. CiteSeer : An Automatic Citation Indexing System. In *Digital Libraries 98 - The Third ACM Conference on Digital Libraries*, pages 89–98, 1998.
- [81] D. Gleich, L. Zhukov, and P. Berkhin. Fast Parallel PageRank : a Linear System Approach. Technical report, Stanford University, 2004.
- [82] G. Golub and C. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD, third edition, 1996.
- [83] N. S. Good and A. Krekelberg. Usability and Privacy : a Study of KaZaA P2P File-Sharing. In *Proc. of the SIGCHI Conf. on Human Factors in Computing Systems*, pages 137–144, 2003.
- [84] Google. url : <http://www.google.com/>.
- [85] Google News. url : <http://news.google.fr/>.
- [86] J. Graupmann, R. Schenkel, and G. Weikum. The SphereSearch Engine for Unified Ranked Retrieval of Heterogeneous XML and Web Documents. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 529–540, 2005.
- [87] S. D. Gribble, A. Y. Halevy, Z. G. Ives, M. Rodrig, and D. Suciu. What Can Database Do for Peer-to-Peer ? In *Proc. Intl. Workshop on the Web and Databases (WebDB)*, pages 31–36, 2001.
- [88] A. Gulli and A. Signorini. The Indexable Web is More than 11.5 Billion Pages. In *Proc. Intl. World Wide Web Conference (WWW)*, pages 902–903, 2005.
- [89] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK : Ranked Keyword Search over XML Documents. In *Proc. ACM Symp. on the Management of Data (SIGMOD)*, pages 16–27, 2003.
- [90] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK : Ranked Keyword Search over XML Documents. In *Proc. ACM Symp. on the Management of Data (SIGMOD)*, pages 16–27, 2003.
- [91] A. Gupta and I. S. Mumick. Maintenance of Materialized Views : Problems, Techniques, and Applications. *IEEE Data Eng. Bull.*, 18(2) :3–18, 1995.
- [92] Z. Gyöngyi, P. Berkhin, H. Garcia-Molina, and J. Pedersen. Link Spam Detection Based on Mass Estimation. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 439–450, 2006.
- [93] Z. Gyöngyi and H. Garcia-Molina. Web Spam Taxonomy, 2005.
- [94] Z. Gyöngyi, H. Garcia-Molina, and J. Pedersen. Combating Web Spam with TrustRank. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 576–587, 2004.
- [95] T. H. Haveliwala. Efficient Computation of PageRank. Technical Report 1999-31, 1999.

- [96] T. H. Haveliwala. Topic-Sensitive PageRank. In *Proc. Intl. World Wide Web Conference (WWW)*, pages 517–526, 2002.
- [97] T. H. Haveliwala and S. Kamvar. The Second Eigenvalue of the Google Matrix. Technical report, 2003.
- [98] T. H. Haveliwala, S. Kamvar, D. Klein, and chris Maning anf Gene Golub. Computing PageRank Using Power Extrapolation. Technical report, 2003.
- [99] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword Proximity Search on XML Graphs. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, pages 367–378, 2003.
- [100] Hyperion SQL Database. <http://hsqldb.org/>.
- [101] C. F. Ipsen and S. Kirkland. Convergence Analysis of an Improved PageRank Algorithm. Technical report, North Carolina State University, 2004.
- [102] S. Iyer, A. I. T. Rowstron, and P. Druschel. Squirrel : a Decentralized Peer-to-Peer Web Cache. In *ACM Intl. Symp. on Principles of Distributed Computing (PODC)*, pages 213–222, 2002.
- [103] H. V. Jagadish, L. V. S. Lakshmanan, D. Srivastava, and K. Thompson. TAX : A Tree Algebra for XML. In *DBPL*, pages 149–164, 2001.
- [104] G. Jeh and J. Widom. Scaling Personalized Web Search. In *Proc. Intl. World Wide Web Conference (WWW)*, pages 271–279, 2003.
- [105] S. Jin and A. Bestavros. GreedyDual* Web Caching Algorithm : Exploiting the Two Sources of Temporal Locality in Web Request Streams. *Computer Communications*, 24(2) :174–183, 2001.
- [106] S. Kalepu, S. Krishnaswamy, and S. W. Loke. Reputation = f(User Ranking, Compliance, Verity). In *Proc. Intl. Conf. on Web Services (ICWS)*, pages 200–207, 2004.
- [107] P. Kalnis, W. S. Ng, B. C. Ooi, D. Papadias, and K.-L. Tan. An Adaptive Peer-to-Peer Network for Distributed Caching of OLAP Results. In *Proc. ACM Symp. on the Management of Data (SIGMOD)*, pages 480–497, 2002.
- [108] S. Kamvar, T. Haveliwala, and G. Golub. Adaptive Methods for the Computation of PageRank, 2003.
- [109] S. D. Kamvar, T. H. Haveliwala, C. D. Manning, and G. H. Golub. Exploiting the Block Structure of the Web for Computing PageRank. Technical report, Stanford University, 2003.
- [110] S. D. Kamvar, T. H. Haveliwala, C. D. Manning, and G. H. Golub. Extrapolation Methods for Accelerating PageRank Computations. In *Proc. Intl. World Wide Web Conference (WWW)*, 2003.
- [111] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The Eigentrust Algorithm for Reputation Management in P2P Networks. In *Proc. Intl. World Wide Web Conference (WWW)*, pages 640–651, 2003.
- [112] J. M. Kleinberg. Authoritative Sources in a Hyperlinked Environment. *J. ACM*, 46(5) :604–632, 1999.

- [113] A. Klemm, C. Lindemann, M. K. Vernon, and O. P. Waldhorst. Characterizing the Query Behavior in Peer-to-Peer File Sharing Systems. In *Proc. of the ACM Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 55–67, 2004.
- [114] C. Kohlschütter, P.-A. Chirita, and W. Nejdl. Efficient Parallel Computation of PageRank. In *Proc. Europ. Conf. on IR Research (ECIR)*, pages 241–252, 2006.
- [115] G. Kollias and E. Gallopoulos. Asynchronous PageRank Computation in an Interactive Multithreading Environment. In *Proc. Dagstuhl Seminar*, pages ? ?–? ?, 2007.
- [116] G. Kollias, E. Gallopoulos, and D. B. Szyld. Asynchronous Iterative Computations with Web Information Retrieval Structures : the PageRank Case. In *Proc. Intl. Conf. on Parallel Computing (PARCO)*, 2005.
- [117] M. R. Korupolu and M. Dahlin. Coordinated Placement and Replacement for Large-Scale Distributed Caches. *IEEE Trans. on Knowledge and Data Engineering (TKDE)*, 14(6) :1317–1329, 2002.
- [118] R. Kumar, P. Raghavan, S. Rajagopalan, D. Sivakumar, A. Tomkins, and E. Upfal. Random Graph Models for the Web Graph. In *Proc. Intl. Symp. on Foundations of Computer Science (FOCS)*, pages 57–65, 2000.
- [119] A. N. Langville and C. D. Meyer. Deeper Inside PageRank. *Internet Mathematics*, 1(3) :335–380, 2004.
- [120] A. N. Langville and C. D. Meyer. Updating PageRank with Iterative Aggregation. In *Proc. Intl. World Wide Web Conference (WWW) - Alternate Track Papers & Posters*, pages 392–393, 2004.
- [121] C. P.-C. Lee, G. Golub, and S. A. Zenios. A Fast Two-Stage Algorithm for Computing PageRank. Technical report, Stanford University, 2003.
- [122] H. C. Lee and A. Borodin. Perturbation of the Hyper-Linked Environment. In *Proc. Intl. Conference on Computing and Combinatorics (COCOON)*, pages 272–283, 2003.
- [123] R. Lempel and S. Moran. SALSA : the Stochastic Approach for Link-Structure Analysis. *ACM Trans. Inf. Syst.*, 19(2) :131–160, 2001.
- [124] R. Lempel and S. Moran. Rank-Stability and Rank-Similarity of Link-Based Web Ranking Algorithms in Authority-Connected Graphs. *Information Retrieval*, 8(2) :245–264, 2005.
- [125] E. Leontiadis, V. V. Dimakopoulos, and E. Pitoura. Cache Updates in a Peer-to-Peer Network of Mobile Agents. In *Proc. Intl. Conf. on Peer-to-Peer Computing (P2P)*, pages 10–17, 2004.
- [126] Y. Li. Toward a Qualitative Search Engine. *IEEE Internet Computing*, 2(4) :24–29, 1998.
- [127] Y. Liu, A. H. H. Ngu, and L. Zeng. QoS Computation and Policing in Dynamic Web Service Selection. In *Proc. Intl. World Wide Web Conference (WWW)*, pages 66–73, 2004.
- [128] Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. G. Lindsay, and J. F. Naughton. Middle-tier Database Caching for E-business. In *Proc. ACM Symp. on the Management of Data (SIGMOD)*, pages 600–611, 2002.

- [129] B. Manaskasemsak and A. Rungsawang. Parallel PageRank Computation on a Gigabit PC Cluster. In *Proc. Intl. Conf. on Advanced Information Networking and Applications (AINA)*, pages 273–277, 2004.
- [130] B. Manaskasemsak and A. Rungsawang. An Efficient Partition-Based Parallel PageRank Algorithm. In *Proc. Intl. Conf. on Parallel and Distributed Systems (ICPADS)*, pages 257–263, 2005.
- [131] B. Mandhani and D. Suciu. Query Caching and View Selection for XML Databases. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 469–480, 2005.
- [132] M. Marchiori. The Quest for Correct Information on the Web : Hyper Search Engines. *Computer Networks and ISDN Systems*, 29(8-13) :1225–1236, 1997.
- [133] Matteo Dell’Amico. Neighbourhood Maps : Decentralised Ranking in Small-World P2P Networks. In *Proc. Intl. Parallel and Distributed Processing Symposium (IPDPS)*, 2006.
- [134] H. Meyer, I. Bruder, A. Heuer, and G. Weber. The Xircus Search Engine. In *INEX Workshop*, pages 119–124, 2002.
- [135] T. Millstein, A. Halevy, and M. Friedman. Query Containment for Data Integration Systems. *J. Comput. Syst. Sci.*, 66(1) :20–39, 2003.
- [136] T. Milo, S. Abiteboul, B. Amann, O. Benjelloun, and F. D. Ngoc. Exchanging Intensional XML Data. In *Proc. ACM Symp. on the Management of Data (SIGMOD)*, pages 289–300, 2003.
- [137] A. Y. Ng, A. X. Zheng, and M. I. Jordan. Stable algorithms for link analysis. In *Proc. Intl. Conference on Research and Development in Information Retrieval (SIGIR)*, pages 258–266, 2001.
- [138] W. S. Ng, B. C. Ooi, K.-L. Tan, and A. Zhou. PeerDB : A P2P-based System for Distributed Data Sharing. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, pages 633–644, 2003.
- [139] C. Olston, A. Manjhi, C. Garrod, A. Ailamaki, B. M. Maggs, and T. C. Mowry. A Scalability Service for Dynamic Web Applications. In *Proc. Intl. Conf. on Innovative Data Systems Research (CIDR)*, pages 56–69, 2005.
- [140] C. Olston and J. Widom. Best-Effort Cache Synchronization with Source Cooperation. In *Proc. ACM Symp. on the Management of Data (SIGMOD)*, pages 73–84, 2002.
- [141] OWL-S : Semantic Markup for Web Services. url : <http://www.daml.org/services/owl-s/>.
- [142] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999.
- [143] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking : Bringing Order to the Web. Technical report, Stanford Digital Library Technologies Project, 1998.
- [144] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking : Bringing Order to the Web. Technical Report 1999-66, Stanford Digital Library Technologies Project, 1999.

- [145] J. X. Parreira, D. Donato, C. Castillo, and G. Weikum. Computing Trusted Authority Scores in Peer-to-Peer Web Search Networks. In *Proc. Intl. Workshop on Adversarial Information Retrieval on the Web (AIRWeb)*, pages 73–80, 2007.
- [146] J. X. Parreira, D. Donato, S. Michel, and G. Weikum. Efficient and Decentralized PageRank Approximation in a Peer-to-Peer Web Search Network. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 415–426, 2006.
- [147] J. X. Parreira, S. Michel, and M. Bender. Size Doesn’t Always Matter : Exploiting PageRank for Query Routing in Distributed IR. In *Proc. Intl. Workshop on Information Retrieval in Peer-to-Peer Networks (P2PIR)*, pages 25–32, 2006.
- [148] J. X. Parreira and G. Weikum. JXP : Global Authority Scores in a P2P Network. In *Proc. Intl. Workshop on the Web and Databases (WebDB)*, pages 31–36, 2005.
- [149] C. Patel, K. Supekar, Y. Lee, and E. K. Park. OntoKhoj : a Semantic Web Portal for Ontology Searching, Ranking and Classification. In *Proceedings Intl. Workshop on Web Information and Data Management (WIDM)*, pages 58–61, 2003.
- [150] S. Podlipnig and L. Böszörményi. A survey of web cache replacement strategies. *ACM Comput. Surv.*, 35(4) :374–398, 2003.
- [151] G. Pringle, L. Allison, and D. L. Dowe. What is a tall poppy among Web pages ? In *Proc. Intl. World Wide Web Conference (WWW)*, pages 369–377, 1998.
- [152] W. Qian, L. Xu, S. Zhou, and A. Zhou. COCACHE : Query Processing Based on Collaborative Caching in P2P Systems. In *Proc. Intl. Conf. on Database Systems for Advanced Applications (DASFAA)*, pages 498–510, 2005.
- [153] J. Qiao, B. Jones, and S. Thrall. An Efficient Algorithm and Its Parallelization for Computing PageRank. In *Proc. Intl. Conf. on Computational Science (ICCS)*, pages 237–244, 2007.
- [154] F. Qiu and J. Cho. Automatic Identification of User Interest for Personalized Search. In *Proc. Intl. World Wide Web Conference (WWW)*, pages 727–736, 2006.
- [155] L. Raschid, Y. Wu, W.-J. Lee, M. E. Vidal, P. Tsaparas, P. Srinivasan, and A. K. Sehgal. Ranking Target Objects of Navigational Queries. In *Proceedings Intl. Workshop on Web Information and Data Management (WIDM)*, pages 27–34, 2006.
- [156] M. Richardson, R. Agrawal, and P. Domingos. Trust Management for the Semantic Web. In *Proc. Intl. Semantic Web Conference (ISWC)*, pages 351–368, 2003.
- [157] M. Richardson and P. Domingos. The Intelligent Surfer : Probabilistic Combination of Link and Content Information in PageRank. In *Advances in Neural Information Processing Systems (NIPS)*, pages 1441–1448, 2001.
- [158] M. Richardson, A. Prakash, and E. Brill. Beyond PageRank : Machine Learning for Static Ranking. In *Proc. Intl. World Wide Web Conference (WWW)*, pages 707–715, 2006.
- [159] M. Ripeanu, A. Iamnitchi, and I. T. Foster. Mapping the Gnutella Network. *IEEE Internet Computing*, 6(1) :50–57, 2002.

- [160] A. Rungsawang and B. Manaskasemsak. Partition-Based Parallel PageRank Algorithm. In *Proc. Intl. Conf. on Information Technology and Applications (ICITA)*, pages 57–62, 2005.
- [161] A. Rungsawang and B. Manaskasemsak. Parallel Adaptive Technique for Computing PageRank. In *Proc. Intl. Conf. on Parallel, Distributed and Network-Based Processing (PDP)*, pages 15–50, 2006.
- [162] O. D. Sahin, A. Gupta, D. Agrawal, and A. E. Abbadi. A Peer-to-peer Framework for Caching Range Queries. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, pages 165–176, 2004.
- [163] K. Sankaralingam, S. Sethumadhavan, and J. C. Browne. Distributed Pagerank for P2P Systems. In *Proc. Intl. Symp. on High Performance Distributed Computing (HPDC)*, pages 58–69, 2003.
- [164] K. Sankaralingam, M. Yalamanchi, S. Sethumadhavan, and J. C. Browne. Pagerank Computation and Keyword Search on Distributed Systems and P2P Networks. *Journal of Grid Computing*, 1(3) :291–307, 2003.
- [165] T. Sarlós, A. A. Benczúr, K. Csalogány, D. Fogaras, and B. Rácz. To Randomize or not to Randomize : Space Optimal Summaries for Hyperlink Analysis. In *Proc. Intl. World Wide Web Conference (WWW)*, pages 297–306, 2006.
- [166] S. A. Savari and D. P. Bertsekas. Finite Termination of Asynchronous Iterative Algorithms. *Parallel Comput.*, 22(1) :39–56, 1996.
- [167] T. Schlieder and H. Meuss. Querying and ranking XML documents. *Journal of the American Society for Information Science and Technology (JASIST)*, 53(6) :489–503, 2002.
- [168] F. Schneider, N. Blachman, and E. Fredricksen. *How to Do Everything with Google*. McGraw-Hill Osborne Media, 2003.
- [169] T. Schwentick. XPath Query Containment. *SIGMOD Record*, 33(1) :101–109, 2004.
- [170] S. Shi, J. Yu, G. Yang, and D. Wang. Distributed Page Ranking in Structured P2P Networks. In *Proc. Intl. Conf. on Parallel Processing (ICPP)*, pages 179–186, 2003.
- [171] Simple object access protocol. url : <http://www.w3.org/TR/soap/>.
- [172] D. Stutzbach and R. Rejaie. "understanding churn in peer-to-peer networks". In *Proc. of the ACM Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 189–202, 2006.
- [173] Swoogle semantic web search engine. url : <http://swoogle.umbc.edu/>.
- [174] A. Theobald and G. Weikum. The Index-Based XXL Search Engine for Querying XML Data with Relevance Ranking. In *EDBT*, pages 477–495, 2002.
- [175] J. A. Tomlin. A New Paradigm for Ranking Pages on the World Wide Web. In *Proc. Intl. World Wide Web Conference (WWW)*, pages 350–355, 2003.
- [176] P. Tsaparas. Using Non-Linear Dynamical Systems for Web Searching and Ranking. In *Proc. ACM Symp. on Principles of Database Systems (PODS)*, pages 59–70, 2004.
- [177] A. C. Tsoi, M. Hagenbuchner, and F. Scarselli. Computing Customized Page Ranks. *ACM Trans. Inter. Tech.*, 6(4) :381–414, 2006.

- [178] A. C. Tsoi, G. Morini, F. Scarselli, M. Hagenbuchner, and M. Maggini. Adaptive Ranking of Web Pages. In *Proc. Intl. World Wide Web Conference (WWW)*, pages 356–365, 2003.
- [179] D. Tsoumakos and N. Roussopoulos. An Adaptive Probabilistic Replication Method for Unstructured P2P Networks. In *Proc. Intl. Conf. on Cooperative Information Systems (CoopIS)*, pages 480–497, 2006.
- [180] UDDI : Universal Description Discovery and Integration. url : <http://www.uddi.org/>.
- [181] T. Upstill, N. Craswell, and D. Hawking. Predicting Fame and Fortune : PageRank or Indegree ? In *Proceedings of the Australasian Document Computing Symposium (ADCS)*, pages 31–40, 2003.
- [182] R. S. Varga. *Matrix Iterative Analysis*. Englewood Cliffs, NJ : Prentice-Hall, 1962.
- [183] L.-H. Vu, M. Hauswirth, and K. Aberer. QoS-Based Service Selection and Ranking with Trust and Reputation Management. In *Proc. Intl. Conf. on Cooperative Information Systems (CoopIS)*, pages 466–483, 2005.
- [184] C. Wang, L. Xiao, Y. Liu, and P. Zheng. Distributed Caching and Adaptive Search in Multi-layer P2P Networks. In *Proc. Intl. Conf. on Distributed Computing Systems (ICDCS)*, pages 219–226, 2004.
- [185] X. Wang, W. S. Ng, B. C. Ooi, K.-L. Tan, and A. Zhou. BuddyWeb : A P2P-Based Collaborative Web Caching System. In *Proc. Intl. NETWORKING Workshops*, pages 247–251, 2002.
- [186] Y. Wang and D. J. DeWitt. Computing PageRank in a Distributed Internet Search Engine System. In *Proc. Intl. Conf. on Very Large Data Bases (VLDB)*, pages 420–431, 2004.
- [187] J. Wicks and A. R. Greenwald. More Efficient Parallel Computation of Pagerank. In *Proc. Intl. Conference on Research and Development in Information Retrieval (SIGIR)*, pages 861–862, 2007.
- [188] WSDL : Web Services Description Language. url : <http://www.w3.org/TR/wsdl>.
- [189] B. Wu and B. D. Davison. Identifying Link Farm Spam Pages. In *Proc. Intl. World Wide Web Conference (WWW) - (Special interest tracks and posters)*, pages 820–829, 2005.
- [190] B. Wu, V. Goel, and B. D. Davison. Topical TrustRank : Using Topicality to Combat Web Spam. In *Proc. Intl. World Wide Web Conference (WWW)*, pages 63–72, 2006.
- [191] J. Wu and K. Aberer. Using SiteRank for Decentralized Computation of Web Document Ranking. In *Proc. Intl. Conf. on Adaptive Hypermedia and Adaptive Web-Based Systems (AH)*, pages 265–274, 2004.
- [192] P. Wynn. On the Convergence and Stability of the Epsilon Algorithm. *SIAM Journal on Numerical Analysis*, 3 :91–122, 1966.
- [193] W. Xi, B. Zhang, Z. Chen, Y. Lu, S. Yan, W.-Y. Ma, and E. A. Fox. Link fusion : A Unified Link Analysis Framework for Multi-Type Interrelated Data Objects. In *Proc. Intl. World Wide Web Conference (WWW)*, pages 319–327, 2004.

- [194] L. Xiao, X. Zhang, A. Andrzejak, and S. Chen. Building a Large and Efficient Hybrid Peer-to-Peer Internet Caching System. *IEEE Trans. on Knowledge and Data Engineering (TKDE)*, 16(6) :754–769, 2004.
- [195] XML : Extensible Markup Language. url : <http://www.w3.org/XML/>.
- [196] X-OQL Homepage. url : <http://www.activexml.net/xoql/toc.html>.
- [197] A. Yamamoto, D. Asahara, T. Itao, S. Tanaka, and T. Suda. Distributed Pagerank : A Distributed Reputation Model for Open Peer-to-Peer Network. In *Proc. Intl. Symposium on Applications and the Internet Workshops (SAINTW)*, pages 389–394, 2004.
- [198] N. E. Young. On-line caching as cache size varies. In *Proc. of the Second Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 241–250, 1991.
- [199] H. Zhang, A. Goel, R. Govindan, K. Mason, and B. V. Roy. Making Eigenvector-Based Reputation Systems Robust to Collusion. In *Proc. Intl. Workshop on Algorithms and Models for the Web-Graph (WAW)*, pages 92–104, 2004.
- [200] X. Zhang, G. Cheng, and Y. Qu. Ontology Summarization Based on RDF Sentence Graph. In *Proc. Intl. World Wide Web Conference (WWW)*, pages 707–716, 2007.
- [201] Y. Zhang, W. Vasconcelos, and D. Sleeman. OntoSearch : An Ontology Search Engine. In *Proc. of the Twenty-fourth SGAI Intl. Conf. on Innovative Techniques and Applications of Artificial Intelligence (AI-2004)*, 2004.
- [202] Y. Zhu and Y. Hu. Exploiting client caches : An approach to building large web caches. In *Proc. Intl. Conf. on Parallel Processing (ICPP)*, pages 419–426, 2003.
- [203] Y. Zhu, S. Ye, and X. Li. Distributed PageRank Computation Based on Iterative Aggregation-Disaggregation Methods. In *Proc. Intl. Conf. on Information and Knowledge Management (CIKM)*, pages 578–585, 2005.

RÉSUMÉ : L'émergence des systèmes pair-à-pair et la possibilité de réaliser des calculs et d'échanger des données par des services web conduit à des systèmes d'intégration de données à large échelle où l'évaluation de requêtes et d'autres traitements complexes sont réalisés par composition de services. Un problème important dans ce type de systèmes est l'absence de connaissances globales. Il est difficile par exemple de choisir le meilleur pair pour le routage des requêtes, le meilleur service lors de la composition de services ou de décider parmi les données locales à un pair celles à rafraîchir, à mettre en cache, etc. La notion de choix implique celle de classement. Bien qu'il soit possible de comparer et classer des entités d'après leur contenu ou d'autres métadonnées associées, ces techniques sont généralement basées sur des descriptions homogènes et sémantiquement riches. Une alternative intéressante dans le contexte d'un système à large échelle est le classement basé sur les liens qui exploite les relations entre les différentes entités et permet de faire des choix fondés sur des informations globales.

Cette thèse présente un nouveau modèle générique de classement de services fondé sur leurs liens de collaboration. Nous définissons une importance globale de service en exploitant des connaissances spécifiques sur sa *contribution* aux autres services à travers les appels reçus et les données échangées. L'importance peut être calculée efficacement par un algorithme *asynchrone* sans génération de messages supplémentaires. La notion de contribution est abstraite et nous avons étudié son instanciation dans le cadre de trois applications : (i) le **classement de services basé sur les appels** où la contribution reflète la sémantique des services ainsi que leur utilisation avec le temps ; (ii) le **classement de services par l'utilisation des données** où la contribution des services est fondée sur l'utilisation de leurs données pendant l'évaluation des requêtes dans un entrepôt distribué ; (iii) la définition des **stratégies de cache distribuées** qui sont basées sur la *contribution* d'une mise en cache des données à réduire la charge du système.

MOTS-CLÉS : Services, données, contribution, utilisation, cache, algorithmes distribués.

ABSTRACT : The emergence of peer-to-peer systems and the possibility to use web services to perform computations and to exchange data lead to large-scale integration systems where query evaluation and other complex tasks are performed through service composition. A crucial problem in such systems is the lack of global knowledge. Therefore it is difficult to find the best peer for query routing, the best service for composition or to decide which local data of a peer must be refreshed or cached. Making a choice implies to perform a ranking. Although it is possible to rank entities according to their content or to other associated metadata, these techniques are generally based on homogeneous and semantically rich descriptions. An interesting alternative in the context of large-scale systems is a link-based ranking that exploits relations between the different entities and allows to make choices according to global information.

This thesis presents a new generic service ranking model based on their collaboration links. We define a global service importance by exploiting specific knowledge about its *contribution* to other services through received calls and exchanged data. The importance may be computed efficiently by an *asynchronous* algorithm without additional messages. Our notion of contribution is abstract and we study its instantiation in the context of three applications : (i) **service ranking based on calls** where the contribution reflects the service semantics and usage ; (ii) **service ranking based on data usage** where the service contribution is based on the usage of its data during the query evaluations in a distributed warehouse ; (iii) **distributed cache strategies** based on the contribution of a data cache on a peer to reduce the cost the system workload.

KEYWORDS : services, data, contribution, usage, cache, distributed algorithms.

