Construction de services distribués : une approche à base d'agents mobiles

Siegfried Rouvrais

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Construction de services distribués : une approche à base d'agents mobiles

Siegfried Rouvrais

N° 4315

Novembre 2001

THÈMES 1 et 2

ISSN 0249-6399 ISRN INRIA/RR--4315--FR

INRIA
RENNES

Construction de services distribués : une approche à base d'agents mobiles

Siegfried Rouvrais

Thèmes 1 et 2 — Réseaux et systèmes — Génie logiciel
et calcul symbolique
Projets Lande et Solidor

Rapport de recherche n° 4315 — Novembre 2001 — 28 pages

Résumé : Ce rapport présente une approche relative à la construction de systèmes répartis, restreinte aux applications de type client-service. Nous y proposons un modèle pour la spécification de services distribués complexes qui traite les interactions de type appel de procédure à distance, évaluation distante et agents mobiles. Cette formalisation ouvre la voie à des analyses de performance, de sécurité ou encore de fiabilité. Ces analyses permettent de guider au mieux l'exploitation effective d'agents mobiles dans le domaine d'application considéré. L'intégration de l'approche dans un environnement de développement permet un déploiement systématique de services distribués en fonction de propriétés non-fonctionnelles.

Mots-clés : agents mobiles, services distribués, qualité de service, architecture de logiciels.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex (France)
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Building of Distributed Services: an Approach using Mobile Agents

Abstract: This report presents an engineering approach for building distributed systems, restricted to client-service applications. We propose a model for the specification of complex distributed services which addresses remote procedure call, remote evaluation and mobile agent interactions. This formalism permits to analyze performance, security and dependability properties. These analyses provide guidelines for using mobile agent technology in the considered application domain. The integration of the model in a development environment permits a systematic deployment of distributed services taking into account non-functionnal properties.

Key-words: mobile agents, distributed services, quality of service, software architecture.

1 Introduction

Le déploiement des nouvelles applications distribuées repose de plus en plus sur les réseaux à grande échelle tel que l'Internet. De nombreuses machines sur ces réseaux sont de véritables îlots de ressources. Il devient possible d'y exploiter les technologies de code mobile [16] (c.-à-d. code à la demande, exécution à distance et agents mobiles). L'objectif de ce travail est de fournir un cadre pour la spécification, l'analyse et le déploiement de composants et de leurs interactions dans une architecture de services distribués. Nous considérons les services distribués comme une classe particulière d'architectures réparties, essentiellement basées jusqu'ici sur le style client-serveur. Dans un réseau de services distribués, des prestataires mettent à disposition de leurs clients potentiels certains services de base (p.ex. [24]), appelés primitifs par la suite. Ces services sont des applications modulaires, décrites et publiées par les prestataires, puis localisées et invoquées à travers le réseau. Les clients émettent des requêtes qui sont alors traitées pour rendre des résultats (invocation distante de procédure ou de méthode). Les résultats de certains de ces services servent souvent de données d'entrée pour d'autres. L'architecte, voire même le client, sont de plus en plus amenés à vouloir composer ces services primitifs afin de pouvoir rendre des services plus complexes.

Les technologies à base de code mobile fournissent une abstraction qui offre de nombreux atouts par rapport aux modèles de programmation classiques qui se restreignent à la seule mobilité des données. Le terme code mobile réfère à du code qui peut être envoyé d'une machine pour être exécuté sur une autre. Il devient possible de déplacer les codes et les unités d'exécutions à travers le réseau. Les différents paradigmes à base de code mobile peuvent être vu comme des variations du modèle client-serveur [3] en terme de schéma d'interaction. Plus particulièrement, pour les agents mobiles, le savoir-faire appartient au client. Il dispose de la logique requise pour réaliser la tache. Un agent mobile permet de réaliser successivement plusieurs services. De plus, de par leur capacité à effectuer des traitements déportés, les agents mobiles permettent d'étendre les fonctionnalités des services primitifs proposés.

Comparés à des modèles plus classiques largement utilisés tels que le client-serveur (basés sur l'appel de procédure à distance, RPC, ou de l'invocation distante de méthode, RMI) [1], les avantages et inconvénients [4, 12] des agents mobiles sont principalement de nature non-fonctionnelle. La notion de non-fonctionnel englobe les propriétés qui ne changent pas la nature des services fournis par l'application. Les propriétés non-fonctionnelles concernées nous semble être principalement :

- la performance : les débits à travers un réseau sont parfois faibles. Les agents mobiles peuvent réduire la consommation en bande passante en déportant les calculs vers les données plutôt que de rapatrier ces données pour les traiter localement à la manière d'une invocation distante [9].

1. Nous utiliserons par la suite le terme invocation distante, RI, qui englobe RPC et RMI.

- la sécurité : les agents mobiles posent de nouveaux problèmes de sécurité [15], notamment la protection de tels agents (p.ex. code, données et état) envers des hôtes malintentionnés (et *vice-versa*).
- la fiabilité : les agents mobiles peuvent permettre des interactions distantes plus robustes sur des réseaux non sûrs (p.ex. défaillances de canaux de communication ou de serveurs). Ils permettent d'envisager de nouveaux mécanismes de tolérance aux fautes [19], afin d'assurer une fiabilité accrue des exécutions.

« Quel est le meilleur protocole (p.ex. invocation distante, migration d'agent, etc.) à mettre en place pour couvrir les besoins en qualité ? ». La réponse à cette question est jusqu'ici laissée aux seules expériences et expertises du concepteur d'application. Il doit lui-même choisir entre la migration vers un site distant et l'invocation distante. Afin de guider le concepteur dans ce choix, nous modélisons les services et leur combinaison afin de spécifier des services complexes. Cette modélisation nous sert de base pour étudier les propriétés non-fonctionnelles de services complexes. Elle permet des analyses qui guident la mise en œuvre et garantit des contraintes de qualité, au plus tôt dans la phase de conception. Les services complexes sont exprimés à deux niveaux d'abstraction :

1. au niveau abstrait, un service complexe est décrit par une expression fonctionnelle simple. Elle spécifie la sémantique d'une requête et fait apparaître clairement les dépendances (flots de contrôle et flots de données), à la manière d'un script *workflow* [7]. L'approche est mono-requête.
2. au niveau concret, une expression peut être vue comme une collection d'invocations distantes, évaluation distante ou migration d'agents. De telles expressions concrètes peuvent être facilement analysées et comparées [5], en terme de propriétés non fonctionnelles. Les propriétés de qualité traitées concernent la performance *via* l'évaluation du trafic total engendré par un service complexe dans le réseau, la sécurité qui traite de la confidentialité (non-divulgation d'informations) et de l'intégrité (non-modification d'informations), et finalement la fiabilité des interactions.

Notre modèle s'intègre dans un environnement de construction de services distribués, pour guider, le plus automatiquement possible, la conception et le déploiement de tels systèmes, tout en garantissant le respect de contraintes de qualité. Il utilise un langage de description d'architectures [14], pour la spécification et l'analyse. Il s'associe à l'environnement Aster [10] pour le déploiement systématique de systèmes distribués au regard de contraintes de qualité, par spécialisation à l'aide de services des couches médiaticielles (c.-à-d. *middleware*, intergiciel ou intersticiel). L'intégration de notre modèle avec Aster a un double bénéfice. Premièrement, elle étend l'approche Aster avec l'utilisation des agents mobiles. Deuxièmement, les techniques de spécialisation propre à Aster pour le déploiement du système envers des critères de qualité nous permettent de garantir la synthèse d'une architecture. Nous étudions la mise en œuvre de notre proposition à l'aide de la plate-forme Grasshopper [8], qui offre, de manière uniforme, à la fois des interactions par invocations distantes et par agents mobiles.

Dans la suite de ce rapport, nous commençons par présenter le cadre formel pour la spécification de services distribués. Nous y proposons l'approche déclarative suivie, puis passons en revue les deux langages de spécification qui en découlent. La section 3 étudie certaines analyses possibles à partir de ce cadre formel. Elles portent sur la performance, la sécurité et la fiabilité. Certains de ces résultats sont ensuite illustrés à travers notre environnement de construction de service client en section 4. La définition de la structure globale de notre environnement est présentée, ainsi que son association avec Aster. Différentes étapes d'analyse offertes à l'architecte, au concepteur et au développeur de service distribués sont alors étudiées. Finalement, notre contribution est résumée, au regard des autres travaux du domaine, et quelques perspectives de recherche sont proposées.

2 Un cadre formel pour la spécification de services distribués

Les services primitifs peuvent être considérés comme des briques indépendantes.1 Des clients émettent des requêtes vers ces services. Les résultats de certains de ces services peuvent servir de données d'entrée pour d'autres. Les services primitifs peuvent ainsi être composés afin de constituer des services plus complexes. Un service complexe peut être construit par un client, à l'aide des services primitifs existants, et spécialisé à l'aide de traitement qui lui sont propres. De la même façon, un service complexe peut être proposé par un prestataire sous la forme d'un service primitif à ses clients.

Il est possible de s'appuyer sur une base fonctionnelle afin de spécifier des services complexes, fruit d'une composition de services primitifs. Un service primitif est alors représenté par une fonction, définie par ses interfaces. Une telle formalisation permet de spécifier la sémantique d'un service complexe sous la forme d'une combinaison de fonctions. Afin de permettre la spécification de services complexes, nous choisissons de considérer trois types d'objets :

- les données, qui sont les points d'entrée d'un service complexe et qui appartiennent au client (ses requêtes). Ce sont les objets de base des langages.
- les services primitifs, qui sont les fonctions offertes par les serveurs. Chacune porte un nom unique et offre un service primitif donné. Ces services sont décrits, au travers de leurs interfaces, par les prestataires de services. Le ou les résultats d'un service primitif peuvent dépendre des données arguments de la fonction.
- les traitements, qui sont les fonctions primitives définies par/pour les clients. Ces fonctions permettent d'étendre, par composition, les fonctionnalités primitives. Les traitements sont décrits dans des bibliothèques prédéfinies ou directement proposés par les clients.

Ces objets sont définis à l'aide d'un environnement (e), qui associe les fonctionnalités aux services ou traitements et leurs valeurs aux données.

À l'aide de ces entités, nous proposons deux langages pour définir un service distribué complexe. Dans le premier, dit abstrait, est spécifiée la fonctionnalité d'un service complexe. Seuls les flots de données entre les différents services et traitements apparaissent. Ainsi, une expression dans le langage abstrait permet simplement la spécification des dépendances entre les services/traitements utilisés. Elle ne fait pas état de modes d'interaction particuliers. Dans le second langage, dit concret, les modes d'interaction sont explicites (c.-à-d agent mobile, évaluation distante, invocation distante). Une expression abstraite doit être vue comme une spécification et une expression concrète comme une implémentation.

2.1 Langage abstrait

Le langage abstrait est présenté dans la figure 1. Une expression est soit une alternative (choix non déterministe) entre expressions, une fonction primitive f appliquée à une expression, un n-uplet d'expressions, ou simplement une donnée d fournie en entrée. Ces identificateurs de données, qui appartiennent à l'ensemble *Data*, prennent leurs valeurs dans un domaine $Value$ (type de données). Les fonctions (ensemble *Primitive*), sont soit des services primitifs, soit des traitements client. La construction **let** permet de partager les résultats $r_1, ..., r_n$ de sous expressions quand cela est nécessaire. Ce langage permet d'exprimer simplement la composition de services, le partage de résultat et le choix non-déterministe. En ce sens, nous approchons du pouvoir d'expression des langages *script* de flots de données [7].

$$E ::= E \square E \mid fE \mid (E, ..., E) \mid d \mid \textbf{let } (r_1, ..., r_n) = E \textbf{ in } E \mid r$$

où $f \in Primitive = Service \cup Treatment$ et $d \in Data$

FIG. 1 – *Langage abstrait*

2.2 Exemple

Les applications de commerce électronique fournissent de nombreux services complexes, par exemple les réservations combinées d'avion/train/hotel. Nous considérons ici un exemple de spécification de service complexe se rapportant aux activités d'un scientifique. Un client souhaite récupérer des rapports de recherche parus dans un laboratoire (p.ex. INRIA). À l'aide des requêtes d_1 et d_2, il interagit tout d'abord avec deux services $biblio_1$ et $biblio_2$, locaux à son Intranet (p.ex. bibliothèque), afin de récupérer deux listes. Les données d_1 et d_2 peuvent contenir des requêtes du type «donnez moi la liste des dernières publications internes depuis telle date». Par exemple, ces listes exhaustives récupérées contiennent respectivement les dernières parutions dans les thèmes génie logiciel et réseau, classées par auteurs et titres. Dans un second temps, ces listes sont filtrées et fusionnées à l'aide du traitement $filtre$, défini par le client, pour sélectionner par mots-clés les documents qu'il souhaite télécharger.

Ces documents sont sur le serveur distant du laboratoire, au format PostScript. Sur ce serveur, un premier service primitif *bibtex* fournit des fichiers BiBTEX en fonction des champs auteurs/titres fournis. Un second service *ps* fournit les fichiers PostScript. Un traitement *compress*, est utilisé dans l'intention de limiter le trafic engendré par le téléchargement. Ce service complexe peut être représenté par l'expression abstraite suivante :

$$\textbf{let } r_1 \ = \ filtre(biblio_1\ d_1, biblio_2\ d_2) \textbf{ in } (bibtex\ r_1, compress(ps\ r_1))$$

Il peut-être représenté par le diagramme de flot de données de la figure 2. Le client propose les données d_1 et d_2 qui sont indépendamment traitées par $biblio_1$ et $biblio_2$ pour produire un couple de résultats. Notons, par exemple, que le traitement $filtre$ doit être appliqué après les services $biblio_1$ et $biblio_2$. Par contre, aucun ordre n'est requis envers l'application des services *bibtex* et *ps*.

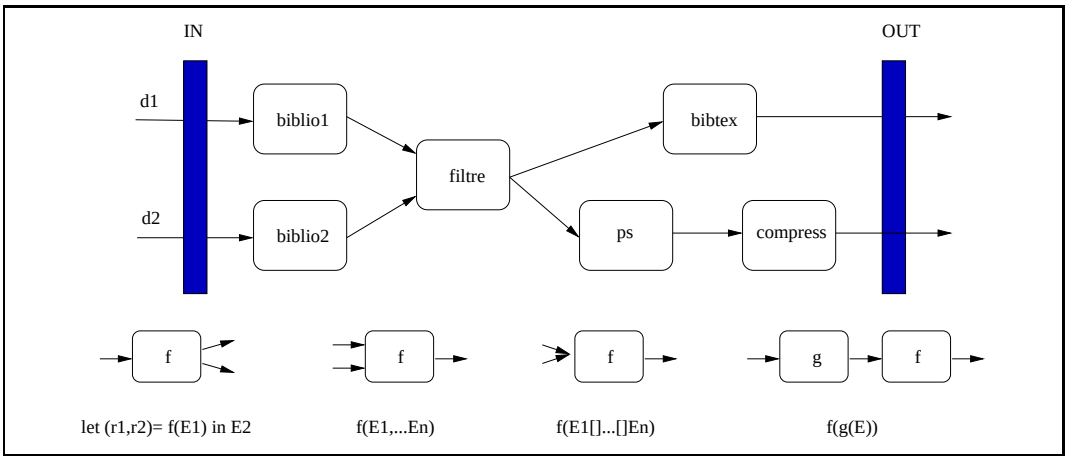

FIG. 2 – *Un exemple de service complexe.*

2.3 Langage concret

Le but du langage concret est de rendre le choix des interactions explicites pour la mise en œuvre du service complexe. En ce sens, il se rapproche d'un modèle d'exécution. Une expression concrète (voir figure 3) est une séquence de constructions **let**, où chacune représente explicitement une interaction. Le mode d'interaction peut être une invocation distante, une évaluation à distance ou une migration d'agent. Une représentation uniforme est utilisée pour les interactions. Les invocations distantes et les évaluations distantes sont ainsi représentées comme des versions dégénérées d'agents. Afin de représenter les parcours séquentiels d'agents, nous avons choisi d'utiliser les continuations [17] et une primitive de migration. L'utilisation des continuations nous permet d'encoder l'ordre d'application des

services et traitements. Les opérations se trouvent ainsi séquentialisées et facilitent les analyses ultérieures. Une fonction du niveau concret, à la différence des fonctions du niveau abstrait, utilise en plus un argument continuation A (la séquence des fonctions qui restent à être appliquées), appliqué au résultat de son évaluation sur les données. Ces fonctions appartiennent à l'ensemble $\overline{Primitive}$. La version CPS (*Continuation Passing Style*) d'une fonction abstraite f, telle que $f\ d\ =\ d'$, est une fonction $\overline{f}$ telle que $\overline{f}\ A\ d\ =\ A\ d'$. Deux fonctions particulières sont introduites. La primitive $go_{i,j}$ dénote la migration du site i au site j. Il est donc nécessaire, à ce niveau de spécification, de connaître la localisation des différents services et du client sur les sites. Nous utilisons un environnement particulier qui contient cette vue réseau sous la forme d'un graphe. La fonction *end* termine l'évaluation d'une interaction. La sémantique des expressions concrètes (figure 4) est relative à

$$
\begin{array}{lcl}
E & ::= & \textbf{let}\ (r_1,...,r_n) = A\ D\ \textbf{in}\ E \mid D \\
A & ::= & \overline{f}A \mid go_{id_1,id_2}A \mid end \\
D & ::= & (D_1,...,D_n) \mid d \mid r
\end{array}
$$

où $f\ \in\ \overline{Primitive},\ id_i\ \in Site$ et $d\ \in Data$

FIG. 3 – *Langage concret*

un environnement ($\overline{e}$) qui associe à un identificateur (c.-à-d. donnée, service ou traitement) sa définition fonctionnelle. Cet environnement, à la différence de l'environnement du niveau abstrait, contient la définition des fonctions sous forme de continuation. Les données gardent leurs interprétations abstraites. L'interprétation concrète $\mathcal{E}_c$ d'une expression (construction **let**) est usuelle et se rattache à l'équivalence :

$$\textbf{let}\ r = E_1\ \textbf{in}\ E_2 \quad \equiv \quad (\lambda r.E_2)E_1$$

Pour chaque séquence de fonctions d'une interaction (construction A), l'interprétation $\mathcal{A}_c$ correspond à la β-réduction des fonctions sous forme de continuations. L'application d'une fonction *go* est neutre du point de vue sémantique. En effet, les migrations ne changent en rien les résultats d'un service complexe, qu'il soit réalisé en local ou sur des sites distants. Toutefois, ces primitives de migration trouveront tout leur sens dans les interprétations non-fonctionnelles. L'identificateur de terminaison d'une interaction (*end*) est interprété comme l'identité, et fournit ainsi en résultat les données finales suite à l'interaction, c.-à-d. le sac à dos en terme de données de l'agent. L'interprétation $\mathcal{D}_c$ pour les données est standard.

Afin de donner l'intuition de cette sémantique, l'expression concrète suivante propose une implémentation possible du service complexe présenté précédemment.

$$
\begin{array}{l}
\textbf{let}\ r_1 = go_{c,b}(\overline{biblio_1}(go_{b,c}\ end))\ \ d_1\ \textbf{in} \\
\textbf{let}\ r_2 = go_{c,b}(\overline{biblio_2}(go_{b,c}\ end))\ \ d_2\ \textbf{in} \\
\textbf{let}\ r_3 = \overline{filtre}\ end\ \ (r_1,r_2)\ \textbf{in} \\
\textbf{let}\ (r_4,r_5) = go_{c,l}(\overline{bibtex}(\overline{ps}(\overline{compress}(go_{l,c}\ end))))\ \ r_3\ \textbf{in}\ (r_4,r_5)
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{E}_c \ : \ Exp_c \ \rightarrow \ Env \ \rightarrow \ Data & & \\
\mathcal{E}_c[\![\textbf{let } (r_1,...,r_n) = \ A \ D \textbf{ in } E]\!] \ e & = & (\lambda(r_1,\ldots,r_n).\mathcal{E}_c[\![E]\!] \ e)(\mathcal{A}_c[\![A]\!] \ e \ (\mathcal{D}_c[\![D]\!] \ e)) \\
\mathcal{E}_c[\![D]\!] \ e & = & \mathcal{D}_c[\![D]\!] \ e \\
 & & \\
\mathcal{A}_c[\![\overline{f} \ A]\!] \ e & = & e \ \overline{f} \ (\mathcal{A}_c[\![A]\!] \ e) \\
\mathcal{A}_c[\![go_{id_1,id_2} \ A]\!] \ e & = & \mathcal{A}_c[\![A]\!] \ e \\
\mathcal{A}_c[\![end]\!] \ e & = & \lambda d.d \\
 & & \\
\mathcal{D}_c \ : \ Exp_d \ \rightarrow \ Env \ \rightarrow \ Data & & \\
\mathcal{D}_c[\![(D_1,...,D_n)]\!] \ e & = & (\mathcal{D}_c[\![D_1]\!] \ e,...,\mathcal{D}_c[\![D_n]\!] \ e) \\
\mathcal{D}_c[\![d]\!] \ e & = & e \ d
\end{array}
$$

FIG. 4 – *Sémantique des expressions concrètes*

Différentes implémentations sont possibles suivant que l'on utilise des agents ou des invocations distantes. L'expression proposée représente respectivement deux invocations distantes des services $biblio_1$ et $biblio_2$ de la bibliothèque (site b), une application chez le client (site c) de $filtre$, et enfin une interaction de type agent mobile avec les deux services *bibtex* et *ps*, avec application de la compression, déportée sur le site l du laboratoire. Cette dernière interaction utilise le résultat r_3 produit par l'interaction précédente. Tout d'abord, l'agent migre ($go_{c,l}$) vers le site du laboratoire, avec sa donnée r_3 et les traitements présents dans la continuation (ici *compress*). Ensuite, il interagit avec *bibtex* puis avec *ps*. Il applique alors son traitement sur les données requises et finalement retourne chez le client ($go_{l,c}$) avant de rendre un couple de résultats (*end*).

2.4 Discussion

Les objets des langages sont définis statiquement. Le langage abstrait est simple et permet d'exprimer une large classe de services complexes. Le langage concret, qui explicite le type des interactions (c.-à-d. exécution locale, invocation distante, évaluation distante et migration d'agents), ne considère pas de primitives de communication entre les agents. Il va de soi que deux expressions dans chacun des deux langages, si elles correspondent à un même service distribué, sont fonctionnellement équivalentes et spécifient donc le même service complexe. Une expression concrète est alors le raffinement d'une expression abstraite. Il est possible de raffiner automatiquement [5] une expression abstraite en une concrète en choisissant le nombre, les places d'exécution des traitements et l'itinéraire des agents. Il est également possible (mais coûteux) de générer automatiquement toutes les expressions concrètes qui implémentent une expression abstraite donnée.

3 Analyses de qualité

Nous avons présenté notre modèle pour les spécifications abstraites et concrètes de services composites au travers de deux langages. Cette base formelle simple ouvre la voie à des analyses non-fonctionnelles qui visent à guider le concepteur de services distribués complexes. Ces analyses visent à déterminer la performance, la sécurité et la fiabilité des implémentations possibles d'un service complexe.

Dans cette section, nous présentons tout d'abord les trois propriétés de qualité considérées. Ensuite, nous présentons les analyses envisageables en nous focalisant sur celles qui ont un intérêt pratique pour le concepteur tout en restant effectives, certaines étant, dans la majorité des cas, trop coûteuses pour être réalisées.

3.1 Critères d'analyse

3.1.1 Performance

La notion de performance que nous considérons est en terme de volume de données échangées. Notre domaine de valeur est les entiers positifs, qui sont ordonnés par les relations classiques inférieur et supérieur. Une propriété est associée à chaque objet du langage :

- à une donnée d est associée une valeur, sa taille, dans le domaine des entiers.
- à un service primitif s ou traitement t est associé sa signature en terme de taille des données et des résultats. Ainsi, à l'identificateur *compress* pourrait être associé la signature $x \rightarrow x/3$, qui signifie que le document est compressible d'un facteur 3. Pour un traitement, il est également nécessaire de lui associer la taille de son code, susceptible de transiter sur le réseau dans le cas des interactions par agents ou évaluation distante.
- à un $go_{i,j}$ est associé un coefficient α qui permet de modéliser la capacité de transfert de la connexion entre les sites i et j.

3.1.2 Sécurité

Les notions de sécurité que nous considérons sont la confidentialité et l'intégrité. La confidentialité porte sur la non-divulgation des informations alors que l'intégrité porte sur leur non-modification. Dans le cadre du code mobile, la protection du site hôte envers du code malsain commence à être relativement bien traitée et repose largement sur des environnements d'exécution interprétés. Cependant, son dual à savoir la protection des agents envers des sites hôtes malintentionnés est encore loin d'avoir trouvé des solutions standardisées [15]. Nous nous intéressons donc au second cas. Pour cela, nous considérons simplement deux niveaux de sécurité dans le domaine : sûr et non-sûr. L'approche s'étend facilement à tout treillis de sécurité. Au même titre que pour la performance, des propriétés de sécurité sont associées aux objets de base :

- à une donnée d sont associés ses niveaux de confidentialité et d'intégrité.

- à un service primitif ou traitement est associée sa signature en terme de niveau de sécurité des données et des résultats. Ainsi pour la confidentialité, à l'identificateur $biblio_1$ pourrait être associé la signature $x \rightarrow$ non-sûr, qui signifie que quelque soit le niveau de confidentialité de la donnée reçue, le résultat sera non confidentiel. Pour un traitement, il est également nécessaire de lui associer les niveaux de son code source susceptible de transiter sur le réseau.
- à un $go_{i,j}$ est associé un niveau qui reflète la sécurité de la connexion (p.ex. cryptée ou non).
- à un site i sont associés les niveaux qui sont à même d'être respectés pour l'exécution du code des agents (niveaux de confiance).

3.1.3 Fiabilité

Les mécanismes de tolérances aux fautes pour le RPC sont maintenant efficacement traités. Dans le cas des agents, des problèmes nouveaux apparaissent à cause des migrations (étapes de calcul). Pour assurer la continuité du service complexe réalisé par un agent, il est nécessaire de répliquer leurs actions sur plusieurs serveurs distincts. Grossièrement, deux stratégies de réplication sont utilisables :

1. la réplication active [19] où, à chaque étape i de calcul, l'agent est exécuté simultanément sur n machines distinctes. Un site d'une étape i dispose des agents ayant effectués l'étape $i-1$. Un vote permet de ne conserver qu'un seul des agents reçus pour l'exécuter sur le site hôte. Après exécution, cet agent est répliqué pour être transmis aux sites de l'étape $i+1$.
2. la réplication passive [21, 11] où, à chaque étape, un agent mobile est répliqué en n exemplaires, mais une seule des n répliques effectue le calcul. Les $n-1$ autres répliques sont passives et ne prennent la relève que si la réplique active défaille (utilisation de transactions).

Pour les agents, nous nous intéressons aux mécanismes construits sur les sites serveurs (invisibles aux agents et développeurs d'agents). Le domaine de valeur que nous utilisons pour la fiabilité [13] s'appuie sur l'approche de Saridakis [18] qui traite de la tolérance aux fautes pour les interactions de type RPC. Cette approche établit un schéma de classification qui capture les relations de raffinement des propriétés de fiabilité. Les mécanismes de réplication propres aux agents [19, 21, 11] assurent, en partie, la propriété $fiable$ et disposent de mécanismes spécifiques par rapport à ceux du RPC. Cette propriété assure que l'état atteint après une faute inclut un état qui aurait du être atteint sans cette faute. Elle repose à la fois sur les propriétés de détection et de masquage.

Nous proposons d'utiliser un ensemble de ces propriétés de fiabilité où l'élément minimal est l'absence de mécanisme mis en œuvre, et le maximal est la propriété $fiable$. Un même mécanisme doit être utilisé sur tout l'itinéraire d'un agent pour garantir la propriété $fiable$. On associe simplement aux sites la propriété correspondant aux mécanismes implantés.

3.2 Proposition d'analyses

Les analyses que nous proposons se basent sur les techniques de spécification précédentes et traitent des trois propriétés de qualité. Lors de ces analyses, nous considérons que le concepteur dispose dans tous les cas de l'expression abstraite Exp_a. Elle représente le service complexe à étudier ou implanter. À ce service abstrait correspondent des expressions concrètes Exp_c (c.-à-d. les implémentations possibles), qui sont un raffinement de Exp_a. Afin d'estimer la qualité d'un service complexe, nous y associons une propriété de qualité P (P_{perf} pour la performance, P_{secu} pour la sécurité ou P_{fiab} pour la fiabilité), qui est une valeur dans le domaine de la propriété considérée (les entiers positifs pour la performance, un domaine binaire pour la sécurité et la fiabilité). Afin de calculer les trafics générés par les interactions d'une expression concrète, leurs niveaux de sécurité ou de fiabilité, il est nécessaire de connaître la vue du réseau, qui contient les informations sur les sites et leurs liaisons, ainsi que les propriétés de qualité associées. Cette vue est représentée sous la forme d'un graphe fini annoté.

L'analyse de base porte sur l'**inférence de la qualité d'implémentation**. Elle consiste à déterminer, à partir de Exp_c et du réseau, la valeur de la propriété P pour chacunes des interactions. Cette analyse [5], de complexité linéaire en fonction de la taille de Exp_c, se définit récursivement sur la grammaire du langage. La figure 5 présente une version de cette analyse dans le cas de la performance. $Cout_E$ permet l'analyse de chacunes des interactions. Le trafic généré par une interactions $A\ D$ est évalué par $Cout_A$, qui prend en arguments, en plus de l'environnement, la taille du code source de l'agent (somme des tailles des traitements) et les tailles des données transportées initialement. Une migration dans une continuation va induire du trafic si les sites source et destination sont différents. Le coût induit est alors la taille du code de l'agent plus la somme des tailles des données en transit. Ce sac à dos de l'agent pondéré par un facteur α qui peut permettre de représenter la bande passante de la connexion.

Afin de donner l'intuition de cette analyse, nous étudions grossièrement la dernière interaction de l'exemple proposé dans la section 2.3 pour P_{perf} :

$$go_{c,l}(\overline{bibtex}(\overline{ps}(\overline{compress}(go_{l,c}\ end)))) \quad r_3$$

Il est possible d'inférer le trafic total engendré par cette interaction par agent en connaissant la taille du code source de *compress*, le coefficient $\alpha_{c,l}$ (resp. $\alpha_{l,c}$), qui représente la bande passante entre les sites c et r, ainsi que α_{go} qui représente le coût induit par la migration d'un agent (c.-à-d sauvegarde de l'état). À la rencontre du premier *go*, le processus consiste à déterminer la taille des données (l'agent et son sac à dos) devant transiter sur le lien (c, l). Cette taille est la somme de la taille de r_3 (inférée par l'interaction précédente), la taille de *compress* et α_{go}. L'application de $\overline{bibtex}$ sur son argument r_3, puis de $\overline{ps}$ et $\overline{compress}$ permet de déterminer la taille des résultats (r_4 et r_5) qui vont être transmis sur le lien (r, c), les trois fonctions étant appliquées sur le même site (agent à migration unique). Finalement, le *end* clos l'analyse de l'interaction.

Dans le cas où la propriété P est déjà connue dans la spécification, en suivant la même approche, il est possible de s'assurer qu'elle est bien vérifiée par l'expression concrète. On

$$
\begin{array}{lcl}
Cout_E : Exp_c \rightarrow Env \rightarrow Entier & & \\
Cout_E[\![\textbf{let } (r_1,\ldots,r_n) = A\ D \textbf{ in } E]\!]\ e & = & \underline{let}\ (d_1,\ldots,d_n) = \mathcal{A}_c[\![A]\!]\ e\ (\mathcal{D}_c[\![D]\!]\ e)\ \underline{in} \\
 & & \quad Cout_E[\![E]\!]\ (e[r_i \leftarrow d_i]) \\
 & & + Cout_A[\![A]\!]\ e\ (Source[\![A]\!])\ (\mathcal{D}_c[\![D]\!]\ e) \\
Cout_E[\![D]\!]\ e & = & 0 \\
 & & \\
Cout_A[\![\overline{f}\ A]\!]\ e\ s\ d & = & e\ \overline{f}\ (Cout_A[\![A]\!]\ e\ s)\ d \\
Cout_A[\![go_{i,j}\ A]\!]\ e\ s\ d & = & Cout_A[\![A]\!]\ e\ s\ d + \\
 & & \underline{si}\ i = j\ \underline{alors}\ 0\ \underline{sinon}\ \alpha_{i,j}(s\ +\ Somme\ d) \\
Cout_A[\![end]\!]\ e\ s\ d & = & 0 \\
 & & \\
Source[\![\overline{f}\ A]\!] & = & taille(f)\ +\ Source[\![A]\!] \\
Source[\![go_{i,j}\ A]\!] & = & \alpha_{go} + Source[\![A]\!] \\
Source[\![end]\!] & = & 0
\end{array}
$$

FIG. 5 – *Évaluation du coût*

parle alors de **vérification de la qualité d'implémentation**. Cela permet par exemple de vérifier qu'une interaction respecte la confidentialité ou l'intégrité des données et traitements du client ou encore qu'elle est sûre de fonctionnement. Si le service complexe représenté par Exp_c ne vérifie pas la qualité d'implémentation, le concepteur peut retrouver les objets qui violent la propriété dans le cas de la sécurité et de la fiabilité.

Quels sont les autres problèmes que peut se poser le concepteur de services distribués complexes lorsqu'il dispose d'une expression abstraite Exp_a? Ces problèmes dépendent des connaissances dont il dispose sur l'expression concrète Exp_c, la propriété de qualité P et la vue réseau. Ces éléments peuvent être soit complètement définis (mode $+$), soit à synthétiser s'il manque des informations (mode $-$). On dispose donc théoriquement de huit (2^3) analyses possibles (voir figure 6). Toutes ces analyses peuvent se fonder sur une méthode d'interprétation des expressions concrètes, envers les critères non-fonctionnels. Toutefois, certaines de ces analyses sont complexes et hors d'atteinte pratiquement dans un cas général. C'est notamment le cas quand Exp_a n'est pas triviale ou quand le réseau est à « trous » en terme de spécification des placements des services, des liens ou des propriétés de qualité qui s'y attachent. Cependant, pour la sécurité et la fiabilité, le domaine de valeur étant binaire, certaines d'entre elles peuvent rester efficace. Le cas de la performance est généralement plus coûteux.

Nous choisissons ici de nous restreindre à la présentation des analyses qui ont un intérêt pratique et qui trouveront un écho dans la section 4.

Analyse	Propriété	Exp_c	Réseau
vérification de la qualité d'implémentation	+	+	+
inférence de la qualité d'implémentation	-	+	+
synthèse d'implantation minimale	-	-	+
synthèse d'implantation garantie	+	-	+
synthèse d'architecture	+	+	-
synthèse d'architecture minimale	-	+	-
synthèse d'architecture et d'implantation	+	-	-
synthèse d'architecture et d'implantation minimale	-	-	-

FIG. 6 – *Analyses*

3.2.1 Synthèse d'implantation minimale

Si l'expression concrète Exp_c n'est pas fournie, il est théoriquement possible de synthétiser la meilleure implémentation de Exp_a, au sens d'une ou des propriétés de qualité traitées. Pour la performance, on cherchera ainsi à fournir l'expression concrète qui génère le moins de trafic. Pour la sécurité, si elle existe, une expression qui assure la confidentialité ou l'intégrité forte. Pour la fiabilité, une expression sûre de fonctionnement. La complexité de cette analyse est directement dépendante de celle de l'expression abstraite et de l'environnement. Il est possible [5] de générer toutes les expressions concrètes correspondantes à une Exp_a. Naïvement, il suffit ensuite d'inférer la qualité d'implémentation de ces expressions, puis de sélectionner la meilleure. Malheureusement, pour une Exp_a non triviale, il existe un nombre important d'expressions concrètes qui en sont un raffinement. Deux types de complexité apparaissent si l'on recherche toutes les implémentations possibles d'un service abstrait :

1. une complexité de type `arrangements` qui correspond aux choix, pour chaque service présent dans l'expression abstraite, entre une interaction par agent (locale) ou par invocation distante.

2. une complexité de type `permutations` qui correspond aux choix des différents ordres possibles d'application des fonctions si elles sont indépendantes dans l'expression abstraite (cas de la construction tuple).

Théoriquement et au pire cas, à une expression abstraite constituée exclusivement d'un n-uplet de premier niveau, correspondent de l'ordre de $2^{n-1}n!$ implémentations possibles, où n est le nombre de fonctions présentes. Devant une telle complexité, il n'est pas envisageable de générer toutes les expressions concrètes correspondant à un service abstrait si le nombre de fonctions présentes est supérieur à une demi-douzaine. Le problème ne peut se décomposer sans perdre en précision car il est intrinsèquement global. Cependant, l'utilisation des critères non-fonctionnels et de techniques de tabulation (partage), lors d'une analyse « à la volée » du meilleur chemin, permettent de considérablement réduire l'espace de recherche et rendre l'approche effective pour des services complexes composés de plusieurs dizaines de service

de base. La mise en œuvre de cette technique, pour la performance, est présentée dans la section 4.2.1.

3.2.2 Synthèse d'architecture

Il peut arriver qu'un concepteur ait un service complexe Exp_c à implémenter sur un réseau dont les éléments ne sont pas tous déployés. Par exemple, certains services ou liens peuvent ne pas avoir de mécanismes propres à assurer des besoins de qualité. En ce sens, le réseau peux disposer de services ou liens qui n'ont pas encore de valeurs dans les environnements non-fonctionnels. Le réseau n'est donc pas encore entièrement instanciée. Si le concepteur dispose de la propriété P requise, il doit proposer une spécification du réseau totalement instanciée. Celle-ci doit respecter P pour le service complexe considéré. Au regard de la propriété P, une analyse basée sur l'inférence de la qualité d'implémentation de l'expression concrète va permettre d'instancier les éléments manquants. En effet, une expression concrète fournit explicitement les itinéraires des différents agents devant réaliser le service complexe.

La complexité de cette analyse dépend du nombre d'éléments entièrement définis dans le réseau. Si tout les éléments ont les instances requises, elle est linéaire et l'on retombe trivialement sur la vérification de la qualité d'implémentation. Sinon, il faut considérer les différentes alternatives, pour chaque entité, en fonction de la propriété P et ce pour chacune des interactions. Le cas extrême, où aucun élément n'a de propriété, peut trivialement se simplifier à tout localiser chez le client. Dans les cas intermédiaires, comme le graphe est par hypothèse fini, le nombre de choix reste borné. Dès qu'une proposition garantie P, l'analyse termine avec succès.

3.2.3 Synthèse de traitements

Une expression abstraite ou concrète représente un service complexe défini par l'architecte. Ajouter un traitement dans une telle expression, s'il est différent de l'identité, va modifier la sémantique du service complexe. Toutefois, si à un traitement t, il existe un traitement inverse t^{-1}, il va être envisageable de les composer au service complexe, sans en modifier la sémantique. L'application du traitement et de son inverse peux se faire en séquence dans une sous-expression (p.ex. $t^{-1}(t\ E)$). Chercher à étendre un service complexe revient à déterminer, dans une expression abstraite, les endroits où des traitements prédéfinis peuvent s'insérer. Des règles de typage (correspondance entre le type des arguments et le type des résultats) vont permettre de retrouver ces endroits. Si un traitement, et son inverse, se trouvent applicables, il est nécessaire de vérifier si l'extension fonctionnelle respecte les contraintes non-fonctionnelles requises. Si c'est le cas et si l'expression étendue apporte un gain global sur une propriété, l'introduction du traitement peut être validée. Dans un premier cas, si l'expression concrète est déjà définie, la vérification va étudier les différents choix de placement des traitements dans l'expression concrète (p.ex. traitement à distance, en local chez le client). Une analyse de type synthèse de la qualité d'implémentation sur chacunes des expressions concrètes étendues va alors permettre de déterminer la

plus favorable, et s'assurer qu'elle apporte un gain par rapport à l'expression initiale. Dans un second cas, si l'expression concrète n'est pas encore déterminée, des analyses de type synthèse d'implantation minimale sur l'expression abstraite et sur les expressions abstraites étendues, vont servir à produire, par comparaisons, une expression concrète optimale.

Pour le critère de performance, l'utilisation de compresseurs de fichiers (p.ex. codage de Huffmann, gzip, bzip2) peut permettre de réduire la taille de la donnée devant quitter un serveur, supportant les agents, avant d'être fournie au client. Par exemple, le service Exp_{a1} spécifié ci-dessous peut-être étendu en un service Exp_{a1+} en comprimant les résultats du service de rendu de documents. La décompression ayant bien sûr lieu ensuite afin de ne pas modifier la fonctionnalité du service complexe. L'expression concrète Exp_{c1+}, qui utilise un agent, pourra ainsi vraisemblablement réduire le trafic engendré par l'envoi du document.

$$ps(biblio\ d) \qquad (Exp_{a1})$$

$$uncompress(compress(ps(biblio\ d))) \qquad (Exp_{a1+})$$

$$\begin{array}{l}\textbf{let } r_1 = go_{c,b}(\overline{biblio}(go_{b,l}(\overline{ps}(\overline{compress}(go_{l,c}\ end))))) \ \ d \textbf{ in} \\ \textbf{let } r_2 = \overline{uncompress}\ end \ \ r_1 \textbf{ in } r_2\end{array} \qquad (Exp_{c1+})$$

Certains travaux portant sur la sécurité (cryptographie de données ou de programmes) et la fiabilité peuvent s'appliquer de la même manière.

3.3 Discussion

Nous avons traité de manière uniforme des analyses qui s'offrent à nous à partir de nos spécifications. Leur degré de précision est relatif à notre modélisation, qui reste abstraite (services, réseau, propriétés, etc.). Nous nous sommes restreints ici à présenter celles qui sont effectives pour des calculs non triviaux. Toutefois, il peut être envisagé de faire participer le concepteur à certains choix pour réduire la complexité d'autres analyses plus coûteuses. Certaines des analyses présentées (p.ex. vérification de la qualité d'implémentation ou synthèse d'implantation minimale) peuvent profiter de l'utilisation simultanée des trois propriétés de qualité afin d'être plus éfficaces. Pour d'autres (p.ex. synthèse d'architecture), nous ne considérons généralement qu'une propriété à la fois. Des choix de conception pour une propriété donnée peuvent influer sur une autre propriété et conduire à des problèmes de cohérence et consistance. Ce problème reste un axe de recherche à part entière.

Afin de cerner plus clairement la portée de notre modèle et certaines de ses analyses, nous proposons dans la section suivante son intégration dans un environnement de développement.

4 Environnement de développement

Cette section vise à valider l'approche proposée et certaines des analyses, par intégration dans un environnement de développement, qui s'appuie en partie sur l'environnement Aster [10]. Nous proposons un agencement possible des différentes analyses, en fonction des

propriétés de qualité étudiées. Il conduit à une spécification complète du réseau, du service complexe et des propriétés de qualité. La figure 7 propose une vue schématique de l'approche dans le cadre d'applications de type services client. Nous différencions cinq étapes d'analyses (flèches en pointillé) pour la construction de services distribués complexes. Les personnes qui participent sont l'architecte dans la phase de spécification, le concepteur qui profite des différentes analyses et le développeur qui se charge du déploiement.

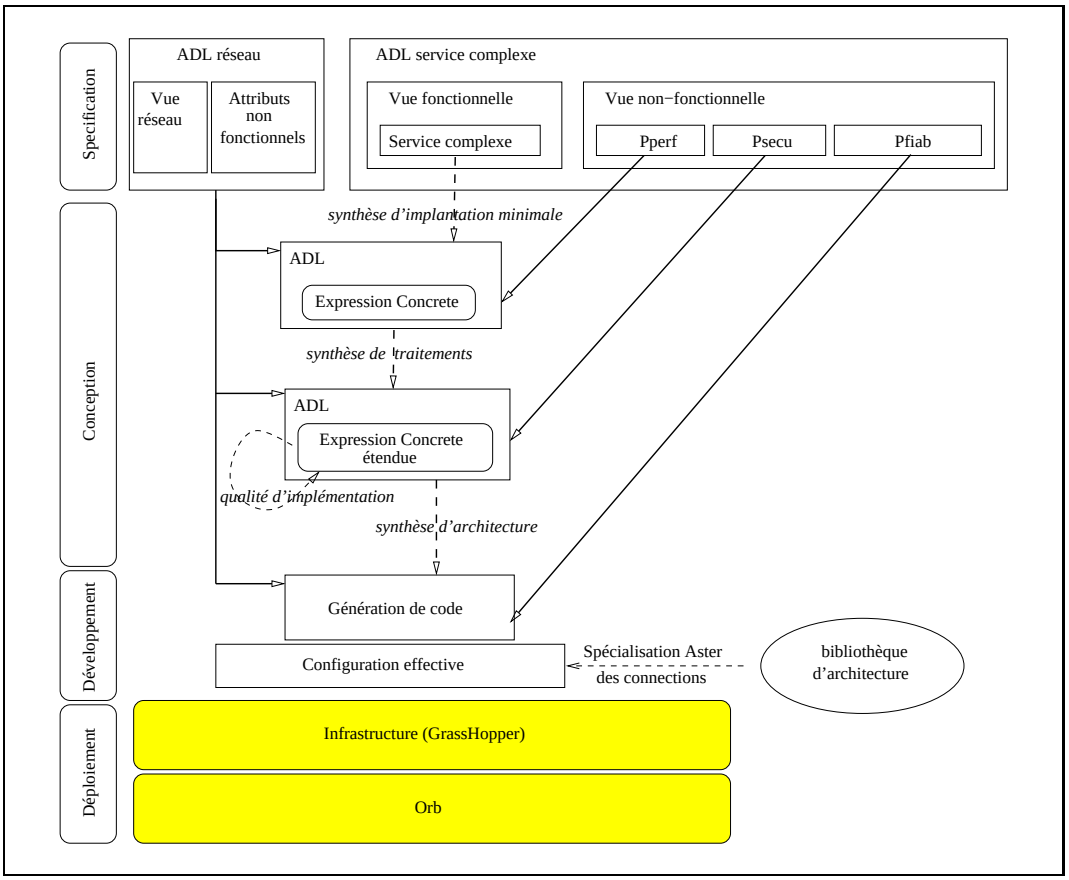

FIG. 7 – *Architecture de l'environnement*

Nous présentons dans la section 4.1 le rôle de l'architecte dans cet environnement. Le rôle du concepteur, assisté par nos analyses, est proposé dans la section 4.2. La section 4.3 s'intéresse à la participation du développeur qui se charge du déploiement d'un service complexe, sur une plate-forme médiaticielle particulière. Finalement nous présentons le cadre de spécification propre à l'environnement, basé sur un langage de description d'architectures.

4.1 Rôle de l'architecte

L'architecte fournit les spécifications du service complexe Exp_a (vue fonctionnelle), la vue non-fonctionnelle (propriétés P_{perf}, P_{secu} et P_{fiab}) qu'il souhaite voir vérifiée ainsi que la vue réseau et ses attributs non-fonctionnels. Ces spécifications sont fournies à l'aide d'un langage de description d'architectures (noté ADL et présenté en section 4.4), qui fournit une notation précise et structurée. Pour l'agencement étudié, l'environnement proposé par l'architecte contient la vue réseau où la localisation des services et liens est entièrement définie. Les propriétés de performance et de sécurité de ces éléments sont également fournies. Cependant, aucune information ne leur est associée pour la fiabilité. Dans l'exemple proposé ici, les propriétés requises par l'architecte sont : la performance optimale (P_{perf}), l'intégrité (P_{secu}) et la fiabilité (P_{fiab}). Pour un service complexe donné, le but des différentes étapes est d'obtenir une implémentation minimale en terme de trafic généré. Elle doit respecter l'intégrité des données et traitements du client et être sûre de fonctionnement. Les spécifications fournies par l'architecte vont donc servir de point d'entrée pour la conception.

4.2 Intervention du concepteur

À partir des spécifications fournies par l'architecte, le concepteur doit chercher à construire le service complexe demandé, c'est à dire proposer une Exp_c qui vérifie les trois propriétés de qualité sur le réseau donné. Dans un premier temps une analyse de synthèse d'implantation minimale permet d'obtenir, à partir de l'expression abstraite, une expression concrète Exp_c optimale en terme de performance. Ensuite, les analyses successives de synthèse de traitements, qualité d'implémentation et synthèse d'architecture vont permettre au concepteur, de manière automatique, de proposer une configuration effective.

4.2.1 Synthèse d'implantation par raffinement suivant la performance

En premier lieu, la synthèse d'implantation minimale (voir section 3.2.1), ou étape de raffinement, permet d'obtenir, à partir de la description abstraite du service complexe Exp_a, fournie par l'architecte, une spécification d'une implémentation possible de ce service. La propriété traitée lors de cette analyse est la performance. L'architecte requiert qu'elle soit minimale. La configuration du réseau est connue ainsi que les propriétés de performance associées à ses éléments. L'analyse va donc chercher à fournir l'expression concrète minimale au sens de l'ordre sur les trafics engendrés sur le réseau. Bien que la complexité théorique de cette analyse puisse paraître rédhibitoire pour la performance (le domaine de valeur étant infini, ce qui n'est pas le cas des domaines binaires des autres propriétés), l'utilisation du critère non-fonctionnel « à la volée » permet de considérablement réduire l'espace de recherche. En effet, il est possible, en cours de construction d'un graphe d'états des interactions, d'élaguer certaines branches de choix générées suite à l'introduction d'une nouvelle fonction (service ou traitement) à appliquer. Une application de fonction génère des transitions dans le graphe d'états. En suivant une construction de l'espace d'états en largeur d'abord, nous pouvons comparer au fur et à mesure les états qui définissent des interactions qui en sont sur un

même site et qui ont les mêmes fonctions appliquées, tout en ayant la même continuation. Ainsi, à chaque nouvelle transition, il est possible de partager des états équivalents.

Pour la performance, nous avons réalisé une mise en œuvre de cette analyse en XSB [23] qui permet de traiter ce partage grâce au mécanisme de tabulation. Des jeux d'essais ont montré qu'il est possible de dériver une implantation minimale pour des services complexes constitués de plusieurs dizaines de fonctions, voire plusieurs centaines pour des expressions linéaires du type $f_1(f_2(...))$. En effet, dans ce second cas, la complexité due aux parcours d'agents est réduite car la séquence d'application des fonctions est préfixée par l'expression abstraite. Le partage est donc maximal. Dans le cas où il est possible de raffiner également sur les autres critères de qualité de manière combinée, il est encore possible d'optimiser les temps de calcul. En effet, les critères de sécurité et de fiabilité ont des domaines de valeurs binaires qui accentuent le partage. Ils permettent de réduire d'autant plus l'espace de recherche en effectuant de simple tests booléens.

4.2.2 Synthèse de traitements, qualité d'implémentation et synthèse d'architecture

L'analyse de synthèse de traitement va par la suite permettre de spécialiser l'expression par l'utilisation de traitements (Exp_c+) qui peuvent faire décroître la taille des données devant transiter sur le réseau. Il est clair qu'il faut s'assurer durant cette étape qu'un serveur censé accueillir de nouveaux traitements propose un environnement d'exécution pour agents. Certaines extensions peuvent éventuellement remonter jusqu'à l'architecte dans la mesure où le concepteur juge judicieux de modifier un service primitif afin qu'il utilise un traitement supplémentaire. Ce traitement peut soit être composé directement sur le serveur (changement d'interface), soit proposé comme nouveau service primitif offert sur ce même serveur. Ensuite, à partir de l'expression concrète éventuellement étendue, il va vérifier que l'implémentation respecte les contraintes de sécurité requises par l'architecte (P_{secu}). Dans notre exemple, si la propriété d'intégrité n'est pas vérifiée, l'analyse permet de retrouver les objets qui violent la propriété. Le concepteur est alors amené, soit à considérer une autre expression concrète Exp_c, soit à demander à l'architecte un changement de niveau de sécurité dans le réseau. Finalement, une fois que le concepteur a pu déterminer une expression concrète performante et qui garanti l'intégrité, une analyse de synthèse d'architecture suivant la fiabilité va permettre d'instancier, si nécessaire, les propriétés de fiabilité des éléments du réseau (sites et liens). En effet, la configuration du réseau est connue mais pas les propriétés de fiabilité pour certaines de ses entités. À ce stade, le concepteur dispose d'une spécification complète du réseau, de l'implantation du service complexe et des propriétés de qualité. Il va fournir toutes ces informations au responsable du développement et du déploiement.

4.3 Rôle du développeur d'application

Le développeur d'application a en charge de développer et déployer le service complexe sur une plate-forme donnée. Il doit identifier les mécanismes à intégrer pour les entités

du réseau qui doivent assurer les propriétés de fiabilité requises, suite à la dernière analyse de synthèse d'architecture. Pour cela, il va suivre l'approche proposée dans le cadre de l'environnement Aster [18], qui, à partir d'une description de l'architecture, propose un ensemble d'outils pour systématiser l'intégration de l'application au sein de l'architecture distribuée. L'approche Aster s'appuie sur la standardisation de la couche logicielle qui se situe entre l'application et le système d'exploitation sous-jacent (c.-à-d couche médiaticielle). Cette couche fournit des solutions réutilisables pour la construction de logiciels complexes. Le travail du développeur quant à la mise en œuvre d'un système d'exécution pour une application donnée est la combinaison de services médiaticiels disponibles, de telle sorte que le système résultant satisfasse les exigences de l'application (la fiabilité dans l'exemple proposé). Pour un mécanisme requis, les services médiaticiels associés sont répertoriés dans une bibliothèque d'architecture. Ces architectures représentent des configurations médiaticielles concrètes. Aster permet ainsi la construction systématique de la couche médiaticielle, adaptée à l'application. Les propriétés de fiabilité requises sur les éléments du réseau sont utilisées durant cette phase pour spécialiser les interactions, directement sur la plate-forme médiaticielle. Si Aster ne peut pas spécialiser une interaction pour assurer la propriété requise, le développeur cherche à proposer un nouveau mécanisme dans la bibliothèque, ou s'en retourne vers le concepteur.

Bien qu'Aster supporte plusieurs médiaticiels répandus (c.-à-d. CORBA, DCOM, EJB), nous avons choisi de reposer sur la plate-forme d'agents mobiles Grasshopper [8], supportant aussi bien les interactions par invocation distante que par agents, pour mettre en œuvre les services définis par le concepteur. Grasshopper nous semble approprié dans la mesure où les invocations distantes et les agents mobiles sont traités de manière uniforme dans les mises en œuvre et il peut s'appuyer sur le médiaticiel CORBA. D'autres plate-formes auraient pu être choisies (p.ex. voir [1] pour une étude comparative). Avant de réaliser l'intégration des mécanismes de tolérance aux fautes pour agents au sein des bibliothèques Aster, nous reposons sur des services de persistance intégrés à l'infrastructure Grasshopper, qui permettent d'assurer les propriétés de fiabilité requises. Les services peuvent donc être déployés en tant que tel. Notre première mise en œuvre du déploiement génère des agents sous forme de code source. Cet outil de déploiement est implémenté en Java et produit un code source orienté vers Grasshopper. Il est possible que le développeur intervienne, en dernier lieu, sur le code source généré pour proposer une interface graphique, formater les résultats ou utiliser des traitements d'exceptions particuliers. Pour exécuter un service complexe sur un réseau existant en utilisant notre générateur, les étapes à suivre sont les suivantes :

1. création des services primitifs qui sont implémentés par des agents,
2. à partir de l'expression concrète, le compilateur crée les différents agents qui implémentent le service complexe,
3. exécution du service complexe par lancement chez le client, *via* l'environnement d'exécution Grasshopper, d'un agent collecteur qui lance les différents agents (un agent par construction **let**) devant interagir à travers le réseau,

4. enfin, une fois les résultats du service complexe récupérés, l'utilisateur peut relancer ce service complexe ou supprimer l'agent collecteur s'il n'en a plus besoin.

4.4 Intégration dans un langage de description d'architectures

Pour spécifier les notions de vue réseau et de service, nous avons choisi d'utiliser un langage de description d'architectures de logiciels [6]. Cette approche permet de reposer sur un cadre formel afin de raisonner sur la structure du système à construire. Elle différencie clairement les composants du système et leurs interactions (en terme de connecteurs). Les composants sont des unités de calcul et les connecteurs, un *médium* de communication entre ceux-ci. Pour la description de services, nous suivons une approche équivalente, où les interfaces de composants représentent les services primitifs et où les connecteurs représentent des protocoles d'interaction. Le langage proposé permet la description d'applications en terme d'interconnexion de composants logiciels, et associe des propriétés de qualité de service aux composants et connecteurs. Un composant peut héberger plusieurs services primitifs ou clients. Une invocation distante représente un protocole requête/résultat, reliant deux interfaces de composants, alors qu'un agent mobile, vu également comme une interaction, effectue la liaison ordonnée entre plusieurs interfaces de composants (hyperarc).

4.4.1 Spécification du réseau

Pour la spécification du réseau et de ses propriétés, nous proposons une syntaxe qui comprend :

- la notion de composant (**component**) et connecteur (**connector**) pour spécifier les serveurs et les liens,
- l'introduction d'attributs non-fonctionnels dans un champ **properties** de ces entités,
- une partie **networkbindings** pour décrire la configuration d'un réseau fini. Suivant le grain utilisé par l'architecte, le réseau peut être représenté au niveau de la couche physique, de la couche médiaticielle ou encore de la couche logique.

Des informations supplémentaires, telle que l'adresse `ip` et le port des serveurs, sont également introduites, afin de pouvoir générer automatiquement les interactions sur notre plateforme cible. La présence d'un environnement d'exécution d'agents est également stipulée. Un exemple de spécification de composant est proposé dans la figure 8. De manière similaire, un composant client est défini par l'interface de ses traitements, auxquels nous associons dans le champ **properties** la taille du code de la fonction, ainsi que ses niveaux de confidentialité et d'intégrité. Les types, la taille et les niveaux de sécurité des données sont également décrits. Les interfaces (voir figure 9) des fonctions offertes par les composants sont définies à part et contiennent les signatures des fonctionnalités, qui correspondent au prototype des fonctions implémentées dans un fichier source. Nous y associons, en plus, les signatures non-fonctionnelles.

component Bibliothèque {
provides biblio_1, biblio_2;
receiveMobileAgents non;
properties
confidentiality non-sûr;
integrity sûr;
faultolerance fiable;
address 194.2.94.60:7000;};

FIG. 8 – *Spécification d'un composant*

interface biblio_1 {
List $biblioInfo$(d: date);
properties
performance $x \rightarrow 1024$;
confidentiality $x \rightarrow$ non-sûr;
integrity $x \rightarrow$ sûr;
implementation biblioInfo.java;};

FIG. 9 – *Spécification d'une interface*

4.4.2 Spécification des expressions abstraites et concrètes

La configuration, dite abstraite, contient en plus des instances de composants, l'expression fonctionnelle du service complexe, associée aux propriétés non-fonctionnelles que l'architecte souhaite voir vérifiées par le service. La figure 10 présente un squelette de spécification de configuration du service complexe donné en exemple dans la section 2.2. On étend donc

configuration ServiceComplexe {
instances C:Client; B:Bibliothèque; L:Labo; C1:ConnecteurLan; C2:ConnecteurWan;
networkbindings C **to** B **through** C1; C **to** L **through** C2;
expression
let $r_1 = C.filtre(B.biblio_1\ C.d_1, B.biblio_2\ C.d_2)$ **in**
$(L.bibtex\ r_1, C.compress(L.ps\ r_1))$;
properties
performance min; **confidentiality** non-sûr; **integrity** sûr; **faultolerance** néant;
bindings
let $r_1 = go_{c,b}(\overline{B.biblio_1}(go_{b,c}\ end))\ \ C.d_1$ **in**
let $r_2 = go_{c,b}(\overline{B.biblio_2}(go_{b,c}\ end))\ \ C.d_2$ **in**
let $r_3 = \overline{C.filtre}\ end\ \ (r_1, r_2)$ **in**
let $(r_4, r_5) = go_{c,l}(\overline{L.bibtex}(\overline{L.ps}(\overline{C.compress}(go_{l,c}\ end))))\ \ r_3$ **in** (r_4, r_5)};

FIG. 10 – *Spécification d'un service complexe*

la partie **configuration** en y ajoutant l'expression abstraite et ses propriétés dans le champ **expression**. À ce stade de spécification, quelques analyses simples (p.ex. typage) peuvent permettre de vérifier une certaine cohérence entre la vue réseau et la vue du service complexe (à la fois fonctionnelle et non-fonctionnelle). Par exemple, un service s_i ne peut assurer un

niveau de sécurité élevé si le serveur sur lequel il est implanté est défini comme totalement non-sécurisé.

Le plus souvent, les descriptions d'architectures spécifient les liaisons entre composants directement à l'aide des connecteurs présents dans la vue réseau. Afin de pouvoir spécifier une configuration concrète qui fait apparaître les différents protocoles employés, nous intégrons directement l'expression concrète (imbrication de constructions **let**) dans la partie **bindings**. Nous avons choisi de faire apparaître les protocoles d'interactions sous forme de connecteurs. Nous obtenons ainsi un style architectural client-serveur qui définit une famille d'architectures, reposant sur les interactions par invocation distante, évaluation distante et agent. Ainsi, un agent mobile va être représenté par un connecteur, liant le client aux services primitifs engagés dans l'interaction. Ce type de connecteur est orienté, les migrations étant explicites dans une expression concrète.

4.5 Discussion

Il est clair que l'agencement des différentes analyses proposées dans l'environnement n'est qu'un choix parmi d'autres. Nous considérons qu'il est particulièrement utile au concepteur d'applications de services clients. Nous nous sommes restreints à traiter les propriétés indépendamment, sans s'inquiéter des impacts d'un choix d'implémentation d'un critère de qualité envers les autres. Par exemple, dans l'environnement proposé, le choix de l'expression concrète optimale en terme de performance peut être remis en cause suite à la synthèse de l'architecture où il peut être nécessaire d'imposer des mécanismes de tolérances aux fautes, qui induisent du trafic. Si nous avions traité la fiabilité pour l'étape de raffinement et la performance pour la synthèse de l'architecture, le problème aurait été moindre. Dans tous les cas, le concepteur peut utiliser les différents outils pour assister son expertise.

Nous avons présenté l'utilisation de l'environnement Aster pour la spécialisation. Cette association a un double bénéfice. Premièrement, dans le cadre d'architectures de service client, Aster ne traite que les interactions de type invocation distante. Notre approche permet l'introduction des agents mobiles comme un nouveau protocole d'interaction dans cet environnement. Deuxièmement, les techniques de déploiement envers des critères de qualité proposées dans les outils Aster, facilitent la construction systématique du système.

5 Conclusion

Nous avons présenté une approche qui porte sur un problème d'ingénierie, à savoir la construction de systèmes répartis, restreinte aux applications de type client-service. Nous avons défini un schéma de base pour la spécification des interactions. Il traite, sous un même formalisme, des interactions de type invocation distante, évaluation distante et agents mobiles. Ce formalisme ouvre la voie à des analyses qui permettent de construire une mise en œuvre garantissant des propriétés de performance, de sécurité et de fiabilité. Nous nous sommes placés du point de vue du client de manière à minimiser le trafic généré par son service complexe, sécuriser ses données et traitement et fiabiliser ses interactions avec les

services distants. Notre modèle nous a permis de mettre en valeur l'utilité de l'évaluation distante et des agents mobiles dans ce type d'applicatif. Plus particulièrement, le critère de performance, souvent mis en avant pour justifier l'usage des agents mobiles, est ici intégré dans des analyses effectives. De plus, la capacité des agents à effectuer des traitements déportés, à la différence des interactions classiques par invocation distante, permet de transformer le réseau de services en un véritable environnement d'exécution distribuée. Il devient alors possible de spécialiser à l'aide d'un traitement, directement chez le prestataire, un service primitif donné. Nous n'avons pas traité ici du code à la demande. Ce mécanisme d'interaction consiste à envoyer une requête à un service distant afin de récupérer un traitement (*applet*), à appliquer sur le site du client. Un tel mécanisme pourrait se modéliser par des fonctions d'ordre supérieur dans nos langages de spécification. Finalement, l'intégration de ce schéma avec l'environnement de développement Aster permet un déploiement systématique au regard des propriétés traitées.

5.1 Travaux connexes

Peu de travaux tentent d'évaluer les propriétés non-fonctionnelles des agents mobiles. Certains d'entre eux comparent les agents mobiles aux RPCs de manière expérimentale (p.ex. [9]). Carzaniga, Picco et Vigna [3] ont fourni des éléments de base pour comparer ces interactions en regard de la consommation en bande passante. Cette approche a été étendue pour une séquence mélangeant interactions par RPC et migration d'agents [22]. Dans notre modèle, cette approche revient à analyser directement une expression concrète où toutes les tailles de requêtes et de résultats sont fixées. Par ailleurs, beaucoup d'applications qui utilisent des agents mobiles ne pourront être utilisées tant que les problèmes de sécurité ne seront pas traités (protection des agents). Les recherches engagées dans la défense des agents mobiles envers les attaques proposent des mécanismes tels que les serveurs de confiance ou la cryptographie du code et des données de ces agents, afin de satisfaire au mieux les propriétés de confidentialité ou d'intégrité. Cependant, ils ne garantissent pas toujours les deux propriétés. En justifiant d'une connaissance à priori sur les environnements des serveurs (niveaux de confiance), notre approche permet de s'assurer qu'elles peuvent être garanties pour le service client. Pour la fiabilité, nous cherchons à vérifier si l'itinéraire d'un agent peut être considéré comme sûr pour son fonctionnement. Les mécanismes associés sont pris en charge par les serveurs hôtes. Notre approche permet de s'assurer qu'un service client peut vérifier la propriété requise.

Le nombre croissant de plate-formes de développement à base d'agents mobiles, qui permettent de traiter d'autres protocoles, ne fournissent pas d'outils de sélection parmi ceux-ci. Le concepteur doit lui-même choisir entre la migration et l'invocation distante. Par ailleurs, il existe de nombreux travaux de recherche sur la modélisation de systèmes à agents mobiles à partir de calculs de processus orientés vers la distribution [2]. Ils permettent, par exemple, de modéliser les sémantiques de défaillances, de sécurité ou de protection. Le calcul de Sekiguchi et Yonezawa [20], pour la description et la comparaison des mécanismes de migration (p.ex. Obliq, Telescript), est une extension du λ-calcul proche de notre proposition. Bien que ces différents modèles soient puissants, ces approches diffèrent de la nôtre, en ce

sens qu'elles ne traitent pas toujours d'interactions plus classiques telle que l'invocation distante (p.ex. Ambiant, Seal). Leurs analyses s'orientent vers la détection et le contrôle de comportement pour le respect d'une propriété.

5.2 Perspectives

Parmi les objectifs de recherche possibles suite à ce travail, notons à court terme la recherche d'autres propriétés non-fonctionnelles qui pourraient s'intégrer au modèle (p.ex. asynchronisme, ouverture, capacité de croissance). À plus long terme, nous pouvons nous intéresser à l'adaptation des services à un environnement changeant, l'approche étant jusqu'ici statique. Afin de permettre au service de s'adapter en fonction du contexte d'exécution ou des besoins plus spécifiques de l'application, on doit contrôler plus finement la spécification et le déploiement. Par ailleurs, étudier la cohérence des vues non-fonctionnelles est une piste intéressante à considérer dans notre approche. Par exemple, les propriétés de performance et de sécurité peuvent entrer en contradiction quand un mécanisme de sécurité induit une augmentation de trafic.

Remerciements

Ce travail a été réalisé, en partie, dans le cadre du projet européen Esprit C3DS (*Control and Coordination of Complex Distributed Systems*). L'auteur remercie Pascal Fradet et Valérie Issarny pour leurs commentaires sur les premières versions de ce rapport, ainsi que David Baudoin et Vincent Vesvard pour leurs contributions dans les phases de mise en œuvre de l'environnement.

Références

[1] Bernard (Guy). – Applicabilité et performances des systèmes d'agents mobiles dans les systèmes répartis. *In : Proc. 1ère conférence française sur les systèmes d'exploitation (CFSE)*, pp. 57–68. – Rennes, IRISA, juin 1999.

[2] Boudol (Gérard), Germain (Florence) et Lacoste (Marc). – *Analyse des langages et modèles de la mobilité.* – rapport technique n3930, INRIA, 2000.

[3] Carzaniga (Antonio), Picco (Gian Pietro) et Vigna (Giovanni). – Designing distributed applications with mobile code paradigms. *In : Proc. of the 19th Int. Conference on Software Engineering (ICSE'97)*, éd. par Taylor (R.). pp. 22–32. – ACM Press.

[4] Chess (David M.), Harrison (Colin G.) et Kershebaum (Aaron). – *Mobile Agents: Are They a Good Idea?* – rapport de recherche n19887, IBM Research Division, février 1994.

[5] Fradet (Pascal), Issarny (Valérie) et Rouvrais (Siegfried). – Analysing non-functional properties of mobile agents. *In : Proc. of the Third Int. Conference on Fundamental Approaches to Software Engineering (FASE 2000). Partie de ETAPS 2000*, éd. par Maibaum (Tom). pp. 319–333. – Springer.

[6] Garlan (David). – Software architecture. *In : Proc. of the 22th Int. Conference on Software Engineering (ICSE-00).* pp. 91–102. – ACM Press.

[7] Georgakopoulos (D.), Hornick (M.) et Sheth (A.). – An overview of workflow management: from process modeling to workflow automation infrastructure. *Journal on Distributed and Parallel Databases*, vol. 3, n2, avril 1995, pp. 119–153.

[8] IKV++ GmbH. – Grasshopper. *In :* `www.ikv.de`.

[9] Ismail (L.) et Hagimont (D.). – A performance evaluation of the mobile agent paradigm. *In : Proc. of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 306–313.

[10] Issarny (V.) et Bidan (C.). – Aster: A framework for sound customization of distributed runtime systems. *In : 16th Int. Conference on Distributed Computing Systems (ICDCS'96).* pp. 586–593. – Hong Kong, mai 1996.

[11] Johansen (Dag), Marzullo (Keith), Schneider (Fred B.), Jacobsen (Kjetil) et Zagorodnov (Dmitrii). – *NAP: Practical Fault-Tolerance for Itinerant Computations.* – rapport technique n98-1716, Cornell University, USA, novembre 1998.

[12] Lange (Danny B.) et Oshima (Mitsuru). – Seven good reasons for mobile agents. *Communications of the ACM, Multiagent Systems on the Net and Agents in E-commerce*, vol. 42, n3, mars 1999, pp. 88–89.

[13] Laprie (J. C.). – *Dependability: Basic Concepts and Terminology.* – Springer pour IFIP WG 10.4, 1994.

[14] Medvidovic (Nenad) et Taylor (Richard N.). – A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, vol. 26, n1, janvier 2000, pp. 70–93.

[15] Moore (Jonathan T.). – *Mobile Code Security Techniques.* – rapport technique nMS-CIS-98-28, University of Pennsylvania, USA, mai 1998.

[16] Picco (Gian Pietro). – *Understanding, Evaluating, Formalizing, and Exploiting Code Mobility.* – Thèse de doctorat, Politecnico di Torino, Italie, février 1998.

[17] Reynolds (John C.). – The discoveries of continuations. *Lisp and Symbolic Computation*, vol. 6, n3/4, novembre 1993, pp. 233–248.

[18] Saridakis (Titos) et Issarny (Valérie). – Developing Dependable Systems Using Software Architecture. *In : Proc. of the 1st Working IFIP Conference on Software Architecture*, pp. 83–104. – San Antonio, USA, février 1999.

[19] Schneider (Fred B.). – *Towards Fault-tolerant and Secure Agentry.* – rapport technique n97-1636, Cornell University, USA, juillet 1997.

[20] Sekiguchi (T.) et Yonezawa (A.). – A calculus with code mobility. *In : Proc. 2nd IFIP Workshop on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, éd. par Chapman et Hall, pp. 21–36. – Londres, 1997.

[21] Strasser (M.) et Rothermel (K.). – Reliability concepts for mobile agents. *Int. Journal of Cooperative Information Systems*, vol. 7, n4, 1998, pp. 355–382.

[22] Straßer (Markus) et Schwehm (Markus). – A Performance Model for Mobile Agent Systems. *In : Proc. of the Int. Conference on Parallel and Distributed Processing Techniques and Applications PDPTA'97*, pp. 1132–1140. – Las Vegas, USA, 1997.

[23] Warren (D.). – The xsb programming system. *In :* `xsb.sourceforge.net`.

[24] Xmethods, web services list. *In :* `www.xmethods.net`.

Table des matières

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399