N 0 Ordre :
de la thèse

UNIVERSITE DE VERSAILLES
SAINT-QUENTIN-EN-YVELINES

THÈSE

présentée

DEVANT L'UNIVERSITÉ DE VERSAILLES SAINT-QUENTIN-EN-YVELYNES

pour obtenir

le grade de : DOCTEUR DE L'UNIVERSITE DE VERSAILLES

mention : Informatique

PAR

Yérom-David BROMBERG

Equipe d'accueil : INRIA, Projet ARLES

TITRE DE LA THÈSE :

Résolution de l'hétérogénéité des intergiciels d'un environnement ubiquitaire

COMPOSITION DU JURY

Yolande Berbers,
Professeur à Katholieke Universiteit Leuven – Belgique – Rapporteur
Isabelle Demeure,
Professeur à l'école Nationale Supérieure des Télécommunication de Paris – Rapporteur
Nicole Levy,
Professeur à l'Université de Versailles Saint-Quentin-en-Yvelines – Examinateur
Luc Bouganim,
Chargé de recherche à l'INRIA – Examinateur
Laurent Réveillère,
Maître de conférence à l'ENSERB – Examinateur
Valérie Issarny,
Directeur de recherche à l'INRIA – Directrice de thèse

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE | INRIA

Résumé

L'informatique diffuse devient aujourd'hui une réalité grâce à la mise en réseau d'un nombre croissant de dispositifs informatiques par le biais de technologies réseaux sans fil basées ou non sur des infrastructures (WLAN, Bluetooth, GSM, GPRS, UMTS). Une des problématiques majeures de l'informatique diffuse est de faire communiquer de façon dynamique, spontanée et transparente ces différents dispositifs entre eux indépendamment de leurs hétérogénéités matérielle et logicielle. Les intergiciels ont été introduits dans cet objectif, cependant étant donné leur diversité, une nouvelle source d'hétérogénéité de plus haut niveau apparaît, notamment au niveau de leur protocole d'interaction. Actuellement, deux méthodes permettent de résoudre ces incompatibilités : la *substitution* et la *traduction de protocoles*. La première requiert la conception de nouveaux intergiciels capables de s'adapter en fonction de leur environnement d'exécution afin de résoudre dynamiquement l'hétérogénéité des intergiciels existants. L'avantage de cette méthode est de fournir une interopérabilité dynamique. En revanche, son inconvénient est d'être non transparente : elle crée une nouvelle source d'hétérogénéité entre ces nouveaux intergiciels, et nécessite de développer des applications qui leur sont spécifiques. La seconde méthode, quant à elle, est transparente : elle ne requiert ni la conception de nouveaux intergiciels, ni le développement de nouvelles applications. Cependant, elle reste statique et planifiée contrairement à la précédente méthode. Dans le contexte de l'informatique diffuse, ces deux méthodes sont complémentaires. Notre contribution consiste à combiner ces deux approches. A l'aide des langages de processus, nous proposons, dans un premier temps, une spécification formelle de notre solution qui permet de résoudre l'hétérogénéité des intergiciels quels que soient la spécificité de leurs caractéristiques, de leurs protocoles et de leurs technologies. Dans un second temps, nous présentons deux systèmes, basés sur cette spécification, conçus pour résoudre : (i) les incompatibilités des protocoles de découverte de services, (ii) les incompatibilités des protocoles de communication. Leur particularité est d'assurer une interopérabilité dynamique et transparente sans requérir de modifications des applications et des intergiciels existants. A partir de nos différentes expérimentations, il apparaît que le surcoût de cette solution pour résoudre les incompatibilités de protocoles est raisonnable.

Abstract

The advent and the phenomenal growth of low-cost, lightweight, portable computers concomitant with the advances in wireless networking technologies (eg.,WLAN,GPRS, UMTS) are making ubiquitous computing a reality. Devices from various application domains, e.g., home automation, consumer electronics, mobile and personal computing domains, need to dynamically interoperate irrespectively of the heterogeneity of their underlaying hardware and software. Middleware have been introduced in order to overcome this issue by specifying a reference interaction protocol enabling so compliant software systems to interoperate. However the emergence of different middleware to address requirements of specific application domains leads to a new heterogeneity issue among interaction protocol. Thus, at a given time and/or at a specific place, devices hosting the wrong middleware become isolated. First, this thesis investigates this issue by adopting an approach based on process algebras to reason on middleware heterogeneity in order to be independent of their underlaying technology. We provide a formal modelling of our solution that overcomes dynamically middleware interaction protocol mismatch using protocol conversion. Then, we introduce two systems implementing our formal model in order to overcome respectively the communication protocols mismatch and the service discovery protocols mismatch used by middleware. The introduced systems achieve interoperability among existing middleware without modifying them and their related applications. Finally, from our experimental results, the efficiency of our solution, both in terms of resource usage and latency, is reasonable.

Table des Figures

Liste des Tableaux

1 Introduction

En 1991, Mark Weiser présentait sa vision futuriste de l'informatique du 21eme siècle en établissant les fondements de ce que nous appelons aujourd'hui l'informatique diffuse (*Ubiquitous Computing*) [58]. L'*informatique diffuse* se définit comme un environnement saturé de dispositifs informatiques susceptibles de coopérer de façon autonome et transparente afin d'améliorer l'interactivité et/ou l'expérience de l'utilisateur [57]. L'essence même de cette vision consiste à mettre les nouvelles technologies de l'information et de la communication au service des utilisateurs et non l'inverse. L'objectif est de permettre à ces derniers d'accéder aux différentes fonctionnalités offertes par les divers dispositifs informatiques hétérogènes présents dans leur environnement immédiat, à partir de n'importe quel terminal, comme par exemple leur téléphone portable, ou leur PDA (*Personal Digital Assistant*).

Les fonctionnalités offertes par les dispositifs informatiques peuvent être abstraites sous forme de services [48]. L'informatique diffuse consiste alors à offrir aux utilisateurs un accès aux services numériques de leur environnement immédiat quelle que soit l'hétérogénéité matérielle et logicielle des dispositifs informatiques sous-jacents. Ainsi, l'un des défis majeurs de l'informatique diffuse, que cette thèse tente de relever, est de garantir l'interopérabilité entre les différents dispositifs informatiques afin d'assurer, dans tous les cas, la disponibilité des services déployés dans l'environnement de l'utilisateur [56].

Dans la suite de ce chapitre, nous présentons dans un premier temps un scénario illustrant les principes de l'informatique diffuse afin de mettre en perspective l'importance qu'il y a à résoudre l'hétérogénéité des différents dispositifs informatiques. Dans un second temps, nous introduisons notre contribution et exposons le contenu de ce document.

1.1 Principes de l'informatique diffuse

Nous présentons un scénario concret, illustrant l'informatique diffuse, dans lequel un utilisateur Paul interagit avec différents services de son environnement :

Paul participe à une réunion qu'il doit malheureusement quitter plus tôt pour se rendre à un rendez-vous au bureau de Suzanne situé à l'autre bout de la ville. Dès que Paul quitte la réunion, le flux audio de cette dernière est automatiquement transféré sur son téléphone mobile. Paul prend alors sa voiture. L'ordinateur de bord détecte la conversation téléphonique en cours et la rediffuse immédiatement sur les haut-parleurs du véhicule jusqu'à ce que la conversation se termine. Grâce à un service d'information trafic auquel le véhicule a accès via une infrastructure réseau du réseau routier, un message s'affiche sur l'écran du tableau de bord indiquant à Paul l'existence d'un accident situé à 2 km. Ne pouvant pas se rendre en voiture à

son rendez-vous, Paul décide alors de se garer et de poursuivre à pieds. Ne connaissant pas le quartier, il consulte via son PDA un service d'information local, accessible dans son environnement immédiat grâce au point d'accès Wi-Fi de la ville, qui lui affiche le plan du quartier et l'itinéraire le plus court pour se rendre à son rendez-vous. Arrivé à temps à ce dernier, Paul sort son ordinateur portable et le connecte, faute de réseau sans fil, au réseau local, afin de découvrir un service permettant de projeter sur un écran la présentation de son projet et termine son rendez-vous en synchronisant des documents de son portable avec ceux d'un dispositif de stockage de données appartenant à Suzanne.

Ce scénario met en valeur plusieurs concepts clés de l'informatique diffuse, dont celui de permettre aux utilisateurs d'accéder à des ressources et/ou à des services numériques n'importe où et à tout instant, à travers différents dispositifs. Paul découvre et accède aux services de son environnement, alternativement, à l'aide d'un téléphone mobile, d'un ordinateur portable, d'un PDA, ou de dispositifs embarqués en fonction du contexte dans lequel il se trouve. Un autre principe clé de l'informatique diffuse est l'aspect dynamique de l'environnement de l'utilisateur. Par exemple, l'accident annoncé par le service d'information trafic est un évènement que l'on ne peut pas prédire. Cela a amené Paul à utiliser son PDA pour analyser son environnement immédiat afin, éventuellement, de trouver un service lui permettant d'obtenir un plan du quartier où il se trouve. Une troisième caractéristique de l'informatique diffuse correspond au fait que le comportement de l'environnement de l'utilisateur est non déterministe : les interactions initiées par l'utilisateur n'aboutissent pas systématiquement à un même résultat étant donné l'aspect dynamique de son environnement.

Un point important à souligner est que le développement de ces différents concepts de l'informatique diffuse n'est possible qu'à la seule et unique condition que les différents dispositifs, terminaux ou services de l'environnement de l'utilisateur soient capables d'interagir indépendamment de leurs hétérogénéités matérielle et logicielle. Dans notre exemple, le PDA, le téléphone mobile, les dispositifs informatiques embarqués, le dispositif de stockage de données sont soumis potentiellement à des contraintes matérielles différentes et sont supportés par des systèmes logiciels hétérogènes. Ainsi, un des défis de l'informatique diffuse est donc d'intégrer, de façon transparente pour l'utilisateur, des technologies hétérogènes [56], [44]. C'est dans ce contexte que s'inscrit le travail réalisé dans cette thèse.

1.2 Contribution et structure du document

La résolution de l'hétérogénéité matérielle et logicielle des dispositifs, et implicitement des services de l'environnement de l'utilisateur, est réalisée par la conception d'intergiciels [29], [2], [3]. Toutefois, la diversité des intergiciels disponibles conçus dans cet objectif est à l'origine d'une nouvelle source d'hétérogénéité de plus haut niveau [134]. Une solution consiste alors à standardiser un intergiciel particulier, destiné à être adoptée par la majorité des acteurs industriels

et académiques du domaine de l'informatique diffuse, afin d'homogénéiser les technologies employées par les différents dispositifs garantissant ainsi leur interopérabilité. Le problème majeur de cette approche est qu'il existe différentes plateformes, incompatibles entre elles, conçues dans cet objectif comme par exemple WSAMI [165], GAIA [167], AURA [166] recréant encore une fois une nouvelle source d'hétérogénéité.

Par conséquent, une autre alternative, que nous adoptons, qui consiste à résoudre l'hétérogénéité entre différents intergiciels, s'impose. Elle revient à considérer qu'au niveau réseau les seules parties visibles des différents services de l'environnement de l'utilisateur sont leurs protocoles d'interactions. De ce fait, garantir leur interopérabilité se résume à résoudre l'incompatibilité de leurs protocoles. Actuellement, deux méthodes, ayant chacune leurs propres avantages et inconvénients, permettent de résoudre cette problématique : (i) *la substitution de protocoles*, et (ii) *la traduction de protocoles*. Chacune de ces deux méthodes présente des inconvénients dans le cadre de l'informatique diffuse. En conséquence, nous proposons et évaluons une fusion de ces deux approches afin de mettre à profit leurs avantages sans leurs inconvénients.

Ainsi, ce document est structuré de la manière suivante :

Dans le chapitre 2, après avoir donné la définition d'un environnement et d'une *architecture de services ubiquitaires*, nous présentons le rôle des intergiciels dans la résolution de l'hétérogénéité matérielle et logicielle des dispositifs informatiques. Nous proposons ensuite un panorama des intergiciels existants en mettant particulièrement l'accent sur leur diversité et leur hétérogénéité.

Dans le chapitre 3, nous proposons une modélisation de l'*environnement ubiquitaire* en utilisant des concepts issus des architectures logicielles dans le but de pouvoir considérer l'hétérogénéité entre intergiciels comme une incompatibilité de couplage entre composants (unités de calcul) et connecteurs (protocoles d'interaction). Cette modélisation nous permet alors de nous focaliser sur les problèmes d'incompatibilités de protocoles que nous formalisons à l'aide d'une algèbre de processus telle que FSP [105]. Cette formalisation rend alors possible la présentation des avantages et des inconvénients des concepts des solutions existantes tout en restant indépendante des technologies les mettant en œuvre. Enfin, nous terminons le chapitre en analysant quelques unes de ces technologies.

Dans le chapitre 4, nous proposons une spécification formelle, à l'aide de l'algèbre de processus FSP, de notre propre solution à la résolution de l'incompatibilité entre protocoles des services numériques ubiquitaires. Par la suite, nous évaluons notre modèle et proposons une architecture logicielle d'un système permettant d'implémenter notre solution.

Dans le chapitre 5, nous mettons en œuvre deux systèmes basés sur la spécification formelle et l'architecture logicielle de notre solution introduite dans le chapitre 4, afin

de résoudre les incompatibilités entre les protocoles de découvertes de services et de communication d'une *architecture de services ubiquitaires*.

Enfin, le chapitre 6 résume ce document et notre contribution pour finalement explorer quelques perspectives futures de nos travaux.

2 Intergiciels pour l'environnement ubiquitaire

Dans ce chapitre, nous introduisons, tout d'abord, la définition d'un environnement et d'une *architecture de services ubiquitaires.* Nous présentons ensuite les rôles d'un intergiciel pour ces architectures. Dans un second temps, nous proposons un panorama d'intergiciels susceptibles d'être déployés dans un *environnement ubiquitaire*.

2.1 Définition d'un environnement ubiquitaire

Grâce aux progrès réalisés ces dernières années dans la miniaturisation de composants électroniques et à l'émergence des technologies réseaux sans fil, un nombre croissant d'objets de notre quotidien intègre des dispositifs électroniques grâce auxquels ils deviennent communicants [55]. Ces *objets communicants* peuvent être des dispositifs informatiques, des périphériques, ou des équipements de domaines aussi variés que l'électronique grand public, l'électroménager, les télécommunications, la domotique ou l'automobile [56]. Ces objets sont, selon le cas, soit mobiles ou immobiles. Ils sont *mobiles* lorsqu'ils sont par exemple portés (téléphones, PDAs, ordinateurs portables, vêtements, etc.) ou conduits (véhicules) par leur(s) utilisateur(s). Ils sont en revanche *immobiles* et invisibles lorsqu'ils sont intégrés ou enfouis dans des objets fixes de l'environnement de l'utilisateur (télévisions, lampes, machines à laver, réfrigérateurs, etc.). Tout objet physique de l'environnement de l'utilisateur peut être potentiellement communicant et être ainsi capable d'interagir avec l'utilisateur et/ou, de façon autonome, avec d'autres objets. Chacun de ces objets dispose de sa propre unité de traitement, puissance de calcul, capacité mémoire, et connectivité réseau. Finalement, l'utilisateur se retrouve au centre d'un espace composé d'objets physiques hétérogènes, dotés de capacités de communication filaire ou non. Ces objets doivent tous pouvoir communiquer avec l'utilisateur et/ou spontanément avec d'autres objets aussi bien localement (dans l'environnement immédiat), qu'à distance, en utilisant l'infrastructure de réseau local. En effet, un tel espace est un espace qui fédère des objets disséminés dans l'environnement de l'utilisateur, indépendamment de leur hétérogénéité. Cette dernière ne doit pas être un handicap à la collaboration des objets, bien au contraire. Par exemple, les *objets communicants* disposant de très peu de ressources (capacité de traitement et/ou d'espace mémoire) doivent être en mesure de profiter de celles d'objets plus puissants. Ces derniers, à leur tour, doivent pouvoir utiliser les fonctionnalités apportées par les premiers. Cette coopération entre objets est dynamique et spontanée dans le sens où : (i) elle n'est pas prévue à l'avance, et (ii) elle doit s'adapter aux changements qui peuvent survenir dans l'environnement de l'utilisateur par l'apparition ou la disparition d'objets.

L'environnement numérique de l'utilisateur se définit comme un *environnement ubiquitaire* formé d'une fédération spontanée et dynamique d'objets communicants. Cette fédération est possible si l'on est capable de faire abstraction de l'hétérogénéité

physique et matérielle des objets numériques pour ne considérer que les fonctionnalités qu'ils sont susceptibles de fournir à leur environnement. Pour faire abstraction de leur hétérogénéité, les *objets communicants* peuvent être perçus comme des entités abstraites qui requièrent et/ou fournissent un ou plusieurs services. Dans la suite de ce document, nous assimilons l'*environnement ubiquitaire* à une *architecture de services ubiquitaires*.

2.2 Définition d'une architecture de services ubiquitaires

Une architecture de services permet de standardiser l'accès aux ressources et/ou aux fonctionnalités des *objets communicants* en les représentant sous formes de services [48]. Un *service* est défini par un *contrat* (appelé aussi *interface*) qui est une spécification abstraite de ses fonctionnalités. Ce contrat décrit : (i) ce que le service fournit, (ii) comment y accéder, et (iii) éventuellement, quelles sont ses propriétés non fonctionnelles. L'architecture se compose de consommateurs de services (ou clients) qui interagissent avec des fournisseurs de services, mettant à disposition un ou plusieurs services. Un client et un service communiquent *via* des interactions synchrones ou asynchrones suivant le contrat du service, exprimé dans un langage déclaratif indépendant de tout langage de programmation, ce qui permet de s'abstraire de l'implémentation du service. Les consommateurs et les fournisseurs de services ne connaissent rien de leur implémentation respective. Les clients ne référencent pas directement des instances particulières de services qui leur sont nécessaires mais leur font indirectement référence par l'intermédiaire de la description de leurs caractéristiques. A l'aide de cette dernière, les clients sont en mesure de localiser/découvrir dynamiquement les instances de services disponibles dans leur environnement en interrogeant un service de découverte *via* un *protocole de découverte de services* (SDP pour *Service Discovery Protocol*) [51]. La découverte de service est soit centralisée ou distribuée [50]. Dans le premier cas, la découverte se fait à l'aide d'un annuaire qui centralise les descriptions des services, tandis que dans le second cas, les fournisseurs de services participent à la découverte de services suivant un modèle de découverte de service passif ou actif. La découverte est passive lorsque ce sont les fournisseurs de services qui envoient périodiquement des annonces dans l'environnement des services qu'ils mettent à disposition, tandis qu'elle est active lorsque ce sont les clients qui diffusent des requêtes décrivant les caractéristiques des services dont ils ont besoin. Ainsi, les interactions entre clients et services se font en plusieurs étapes (Figure 1):

- Le fournisseur de services publie la description de ses services auprès du service de découverte (voir Figure 1, étape❶).

- Le consommateur de services interroge le service de découverte en lui soumettant la description (partielle) du ou des services requis (voir Figure 1, étape❷).

- Le service de découverte renvoie le contrat du service et la référence d'une ou plusieurs instances de services correspondant (voir Figure 1, étape❷).

- Le consommateur de services initie les interactions avec le fournisseur de service suivant les termes du contrat du service (voir Figure 1, étape❸).

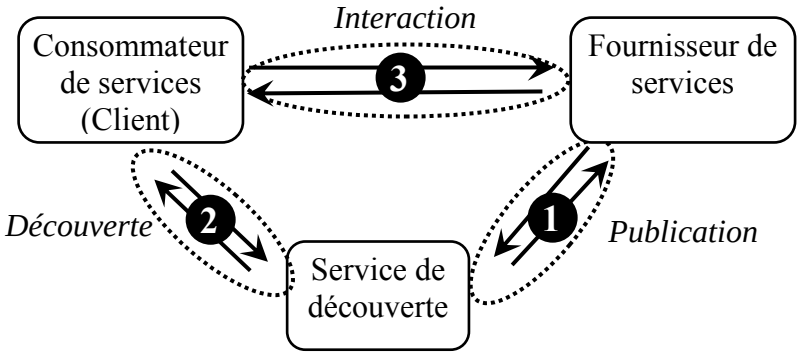


Figure 1. Architecture de services

Une *architecture de services ubiquitaires* représente un *environnement ubiquitaire* où les *objets communicants* se comportent aussi bien comme un consommateur et/ou un fournisseur de service. Au niveau du réseau, les seules parties visibles de leur implémentation sont leurs protocoles de communication respectifs. L'interaction est possible si le comportement du service est connu, c'est-à-dire si son *contrat*/*interface* est standardisée et si le consommateur et le fournisseur de services utilisent un même *protocole de découverte de services* et d'interaction.

La mise en œuvre et le support des clients et des services sont faits *via* l'utilisation d'intergiciels. Par conséquent, le langage de description des services, et leurs protocoles d'interaction dépendent de l'intergiciel utilisé.

2.3 Définition et rôle d'un intergiciel

Un intergiciel se situe au dessus du système d'exploitation (OS pour *Operating System*) et en dessous des applications (c'est-à-dire des clients et/ou des services) de son hôte [29]. L'utilisation d'un intergiciel permet de développer des applications *via* la réutilisation/assemblage de services déployés au travers de différents *objets communicants* indépendamment de leur localisation, langage de programmation, système d'exploitation et matériel [2], [3]. Un intergiciel fournit différentes fonctionnalités/facilités pour l'implémentation d'applications distribuées. Dans la littérature, les fonctionnalités offertes d'un intergiciel sont souvent présentées comme étant les services de l'intergiciel à ne pas confondre, dans notre contexte, avec les services de l'*architecture de services ubiquitaires* (ces derniers étant développés à l'aide des services de leur intergiciel). Quel que soit le type des *objets communicants*

(mobile ou statique, filaire ou non) sur lesquels un intergiciel est déployé il doit fournir les fonctionnalités/services élémentaires suivants [44], [2], [3], [4] :

Protocole d'interaction

Les différents services d'un *environnement ubiquitaire* résidant sur différents objets communicants, les interactions s'effectuent *via* l'utilisation de protocoles réseaux, classifiés selon le modèle de référence OSI [1]. Les intergiciels sont usuellement conçus au dessus de la couche transport [5]. Les couches inférieures, c'est-à-dire la couche physique, liaison, et réseau sont implémentées par le système d'exploitation tandis que les couches supérieures (session et présentation) sont implémentées par l'intergiciel. Avec la couche session, l'intergiciel gère de façon transparente la coordination entre les différents *objets communicants* ; avec la couche présentation, l'intergiciel transforme les données à transmettre selon un format adapté au protocole de transport.

Coordination

Les clients et les services étant distribués sur des hôtes différents, il peut être nécessaire de coordonner leurs interactions. Une interaction est synchrone, si les parties impliquées dans l'interaction doivent attendre que l'interaction se termine avant d'entamer une autre tâche. En d'autres termes, le processus de l'appelant est bloqué jusqu'à la réception d'une réponse du processus de l'appelé [2], [3]. L'un et l'autre sont fortement couplés ou dépendants, étant donné que leur interaction est opérationnelle uniquement si tous les deux sont simultanément en ligne au moment de l'appel et jusqu'à ce que l'interaction se termine. L'attente active de l'appelant peut constituer une perte de temps et de ressources significatives dès lors que l'obtention d'une réponse de l'appelé prend du temps. Cependant, suivant les services que l'on souhaite mettre en œuvre, une interaction synchrone n'est pas toujours nécessaire. Une interaction asynchrone peut être plus appropriée. Avec ce paradigme d'interaction, la communication est non-bloquante. Une fois qu'une requête est envoyée, au lieu d'attendre activement une réponse correspondante, l'émetteur continue son activité. Si le modèle d'interaction asynchrone est réalisé *via* une communication par envoi de messages (*message passing*), le destinataire doit être disponible et en ligne (connecté) au moment de la réception du message autrement ce dernier est perdu. En revanche, si le modèle d'interaction asynchrone se fait à l'aide d'une communication par file de messages (*message queuing*), tant que le destinataire est indisponible, les messages qui lui sont destinés sont sauvegardés dans une file d'attente. Cette dernière peut être localisée sur un intermédiaire plutôt que sur le destinataire pour éviter la perte de messages, dans le cas où le destinataire est hors ligne (déconnecté), assurant ainsi une communication fiable. Les interactions asynchrones réduisent le couplage entre les clients et les services [3], [30]. Cependant, l'utilisation de files d'attentes est nécessaire pour garantir la fiabilité des interactions. Par ailleurs, l'asynchronisme ouvre la voie à des communications autres que celle du type requête-réponse comme la diffusion d'information ou la notification d'évènements [27], [31]. Les interactions synchrones et asynchrones répondent toutes

deux à des besoins différents, et implicitement, le choix d'un intergiciel synchrone plutôt qu'asynchrone dépend du type d'applications/services que l'on souhaite déployer.

Fiabilité

Les protocoles réseaux ont différents degrés de fiabilité. Ils ne garantissent pas nécessairement que toutes les données transmises par l'émetteur soient correctement réceptionnées par le destinataire ou que l'ordre dans lequel les données sont émises soit préservé. C'est l'un des rôles de l'intergiciel de prendre en charge une détection des erreurs et/ou des mécanismes de corrections supplémentaires ou complémentaires selon les cas pour pallier au manque de fiabilité de certains protocoles. La fiabilité d'une interaction a un impact non négligeable sur les performances. Ainsi, dans la pratique, différents types d'interaction, de fiabilité variable, sont nécessaires en fonction du service/application que l'on souhaite concevoir [4]. Une requête émise par un client à destination d'un service peut être délivrée selon le mode *au-mieux* (*best-effort*), *au-plus-une-fois* (*at-most-once)*, *au-moins-une-fois* (*at-least-once)*, ou *exactement-une-fois* (*exactly-once*) [2], [3], [4]. La sémantique *au-mieux* indique qu'il n'y a aucune garantie de la bonne réception/exécution de la requête par le service. La sémantique *au-plus-une-fois* garantit que la requête est reçue/exécutée au plus une fois. De plus, dans l'éventualité où la réception/exécution de la requête échoue, l'émetteur en est notifié. La sémantique *au-moins-une-fois* signifie que la requête est toujours reçue/exécutée au moins une fois, cependant, elle peut l'être plusieurs fois. Enfin le plus haut niveau de fiabilité est délivré par le mode *exactement-une-fois* qui assure qu'une requête est reçue/exécutée une fois exactement. Par ailleurs, il est possible d'étendre cette notion de fiabilité non plus à des requêtes individuelles mais également à un ensemble de requêtes regroupées pour former une transaction [6], [7]. Les transactions sont caractérisées par les propriétés dites ACID (*atomicity*, *consistency*, *isolation*, et *durability*). Cela signifie que toutes les requêtes d'une transaction, sont soit toutes reçues/exécutées ou pas du tout. Bien qu'à l'origine la notion de transaction ait été développée dans le contexte des bases de données, elle est cruciale pour les intergiciels utilisés dans des applications critiques, qui doivent distribuer des données à travers de multiples systèmes hétérogènes de manière atomique [8]. La fiabilité peut également être améliorée *via* la réplication de services [170], [10], [3] : plusieurs exemplaires d'un même service sont déployés sur différents hôtes. Ainsi, si l'un d'entre eux est inaccessible, un *réplica* sur un hôte différent est en mesure d'assurer la continuité des fonctionnalités offertes par le service. Toutefois, certains services maintiennent un ou plusieurs états internes requérant des intergiciels qui prennent en charge la réplication de sorte que leurs états soient synchronisés entre les différentes copies d'un même service.

Hétérogénéité

Les services d'un *environnement ubiquitaire* sont susceptibles d'être déployés sur des *objets communicants* hétérogènes. Nous sommes témoins depuis plusieurs années d'une prolifération d'*objets communicants* de différentes sortes susceptibles de

communiquer entre eux [11] : PC, téléphone portable, PDA *etc*. Ces objets sont potentiellement hétérogènes à plusieurs niveaux : matériel, système d'exploitation, langages de programmation, et l'intergiciel lui-même. En fonction du processeur embarqué sur un *objet communicant*, la représentation des nombres entiers, longs ou flottants peut changer : elle est de type *big-endian* (l'octet le plus significatif est à droite) sur certains processeurs tandis qu'elle est de type *little-endian* (l'octet le plus significatif est à gauche) sur d'autres. Ainsi, lorsqu'un nombre est envoyé d'un hôte *little-endian* à un hôte *big-endian* et *vice versa,* l'ordre des octets doit être inversé. De façon similaire, l'encodage des caractères alphanumériques diffère selon le système d'exploitation employé, nécessitant une transformation préalable des données afin de les interpréter [22]. Quant aux services, ils peuvent être programmés selon différents langages de programmation, impératifs, déclaratifs, orientés objet, *etc*. Chaque langage de programmation dispose de ses propres spécificités : modèle objet ou non, et/ou typage des données différent. Toutes ces différences doivent être prises en charge au niveau de l'intergiciel qui abstrait ces hétérogénéités [4], [5]. Enfin, comme décrit dans les sections suivantes, il existe une très grande variété d'intergiciels. Cette diversité est due au fait qu'il existe plusieurs méthodes pour assurer les fonctionnalités offertes par ces derniers. En fonction du domaine d'application, du type d'*objets communicants* considérés (mobiles ou statiques, sans fil ou non), certains intergiciels sont plus adaptés que d'autres [53]. De ce fait, dans un *environnement ubiquitaire*, différents types d'intergiciels se retrouvent déployés et ils doivent être interopérables les uns avec les autres, de façon à de ne pas constituer un obstacle à l'établissement d'une *architecture de services ubiquitaires*.

Cecilia Mascolo *et al.* distinguent deux catégories d'intergiciels [53] : (i) les intergiciels traditionnels, et (ii) les intergiciels mobiles qui constituent la base des intergiciels ubiquitaires [44]. Les premiers sont adaptés à des applications distribuées déployées sur des hôtes fixes, riches en ressources, ayant une connectivité filaire et interconnectés *via* une infrastructure réseau plutôt statique. Les seconds, qui sont une adaptation des premiers, sont destinés à des hôtes mobiles, limités en ressources, dotés d'une connectivité sans fil, comme le WLAN [59], et interconnectés suivant un modèle dit *ad hoc* [171] ou *infrastructure*. Avec un modèle *infrastructure*, les hôtes mobiles se connectent à un point d'accès qui coordonnent leur communication. En général, un point d'accès est connecté à un réseau filaire permettant ainsi aux hôtes sans fil de se connecter au réseau fixe. En revanche, le modèle *ad hoc* permet des communications spontanées entre les différents hôtes sans fil, sans passer par un point d'accès.

Etant donné le nombre conséquent et la diversité des intergiciels disponibles [172], plutôt que de faire une étude individuelle de tous les intergiciels existants, nous présentons dans la section suivante, les concepts clés des intergiciels traditionnels hérités par les intergiciels ubiquitaires, pour ensuite présenter les concepts majeurs de ces derniers.

2.4 Propriétés élémentaires des intergiciels traditionnels

Un des critères qui permet de différencier les différents intergiciels traditionnels est le modèle de communication utilisé pour interconnecter les applications distribuées. Les modèles de communications synchrones et asynchrones étant les plus utilisés, on distingue deux types d'intergiciels traditionnels suivant ces modèles [12].

Les intergiciels synchrones sont usuellement fondés suivant le modèle des RPC (*Remote Procedure Call*) ou de concepts similaires. Historiquement, les RPC ont été introduits en 1976 par James E. White de l'institut de recherche de Stanford comme un modèle permettant de réaliser des appels de procédures, localisées sur différentes machines distantes, comme si elles étaient locales en faisant abstraction des détails relatifs à la programmation réseau [13]. Au début des années 1980, Birell et Nelson publient leurs travaux sur l'implémentation des RPC [14] et introduisent les concepts de base sur lesquels sont fondés une grande partie des systèmes distribués d'aujourd'hui [4]. Le modèle RPC établit la notion de client, qui effectue un appel de procédure (c.-à-d. une invocation synchrone ayant une fiabilité du type *at-most-once*), et la notion de serveur (le fournisseur de service), qui implémente la procédure distante invoquée (le service dans notre contexte). Il introduit également, entre autres, les concepts de langages de description d'*interface* (*Interface Description Langage* ou IDL), d'*annuaire* de services, d'*interface*s de services, et de *bindings*.

A l'origine, la première étape dans la construction d'une application distribuée est de définir l'*interface* de la procédure distante que l'on souhaite invoquer. Cela est réalisé au moyen d'un IDL qui fournit une représentation abstraite de la procédure (c'est-à-dire, sa signature) en décrivant quels sont les paramètres pris en arguments et ceux renvoyés en retour. Cette description IDL constitue la spécification du service. La deuxième étape est de compiler cette description afin de produire :

- ***Des talons pour le client*** : un talon est un morceau de code compilé à lier avec le code du client. Ainsi, dès que ce dernier effectue un appel de procédure distante, l'appel est exécuté comme s'il s'agissait d'un appel local sur le talon. Ce dernier prend alors soin, de localiser et de se connecter au serveur distant (*binding)*, de formater les données de façon adéquate (*serializing*), d'interagir avec le serveur, d'obtenir une réponse (c'est-à-dire la valeur de retour de la procédure distante invoquée) et de transmettre cette dernière au client. En d'autres termes, un talon encapsule un *proxy* qui masque complètement les mécanismes mis en œuvre pour interagir avec le serveur pour exécuter la procédure distante.

- ***Des talons pour le serveur*** : ils fonctionnement selon le même principe que ceux du client, à l'exception du fait qu'ils contiennent le code permettant de recevoir l'invocation en provenance du talon du client. Ils permettent en particulier d'effectuer les opérations inverses à celles réalisées du côté client (c'est-à-dire, *deserializing*), d'invoquer la

procédure locale, et enfin de transférer la valeur de retour de cette dernière au talon du client.

L'association du talon client à celui du serveur distant (ou *binding*) peut être statique ou dynamique. Lorsque l'association est statique, cela signifie que la localisation du serveur, qui contient la procédure distante recherchée, est connue à l'avance, favorisant ainsi une interaction directe entre le client et le serveur. La localisation de ce dernier est codée en dur dans le talon du client, impliquant un couplage fort entre le client et le serveur. De ce fait, si l'accès au serveur échoue (par exemple, si la localisation du serveur change, ou si le client se trouve dans un nouvel environnement), le client devient non opérationnel. Il est alors nécessaire de recompiler un nouveau talon client pointant sur la nouvelle localisation du serveur. L'association dynamique permet de résoudre cet inconvénient *via* l'utilisation d'un intermédiaire qui assure la fonction d'un annuaire de service. Ainsi, lorsqu'un client invoque une procédure, son talon interroge l'annuaire afin de trouver le serveur adéquat. L'annuaire renvoie l'adresse de ce dernier au talon du client qui à son tour invoque alors la procédure distante. L'utilisation d'un annuaire rajoute un niveau d'indirection entre le client et le serveur : l'un devant consulter, l'autre devant s'enregistrer auprès de l'annuaire avant de pouvoir communiquer. Cependant, cette opération d'association est cachée aussi bien aux clients qu'aux services *via* l'utilisation de talon. Ce dernier, couplé avec l'utilisation d'un IDL, résout également l'hétérogénéité entre plateformes. L'IDL ne sert pas uniquement à décrire la signature des procédures, mais aussi à définir une représentation intermédiaire des données échangées entre les clients et les serveurs, permettant de faire abstraction des formats des données et des langages de programmations spécifiques aux plateformes sur lesquelles les clients et les serveurs sont susceptibles d'être déployés. Indépendamment de leur plateforme sous-jacente, ces derniers sont alors capables d'interagir dès lors qu'ils savent comment interpréter la représentation intermédiaire des données. Cette interprétation est réalisée de manière transparente *via* leur talon respectif qui assure les fonctions de *serialization/deserialization*.

Il existe de nombreux intergiciels fondés sur le modèle RPC, on peut citer par exemple, parmi les intergiciels commerciaux les plus utilisés, CORBA (*Common Object Request Broker Architecture*) [16], les services Web [20] et RMI (*Remote Method Invocation*) [21]. Avec CORBA, les *interfaces* sont spécifiées selon le langage de description CORBA IDL à partir duquel sont générés, selon la terminologie CORBA, un *stub* (talon client) et un *skeleton* (talon serveur). Les données échangées sont formatées selon la spécification CDR (*Common Data Representation*), et véhiculées à l'aide du protocole d'invocation GIOP (*General Inter-ORB Protocol*). Quant aux services Web, ils utilisent WSDL [26] comme IDL, SOAP [25] comme protocole d'invocation, XML [24] pour le formatage des données. RMI contrairement à CORBA et aux services Web ne prend en charge qu'un seul et unique langage de programmation, c'est-à-dire, Java. Ce dernier est également utilisé comme IDL, ce qui implique implicitement que les clients et les serveurs doivent être implémentés en Java. Ces trois intergiciels prennent également en charge la découverte de service (*binding* dynamique) *via* un annuaire. Par exemple, les services

Web utilisent UDDI [23] tandis que les intergiciels RMI et CORBA utilisent respectivement un *Rmi Registry* [21] et un *Corba Naming Service* [16].

Une alternative à l'interaction synchrone est la communication asynchrone où les clients et les fournisseurs de services communiquent *via* un échange de messages ou d'évènements. La classe des intergiciels prenant en charge ce type de communication orientée message sont appelés MOM (*message oriented middleware*) [32]. La notion de client et de service devient purement conceptuelle, étant donné qu'au niveau de l'intergiciel, tous deux sont considérés comme des producteurs et/ou des consommateurs de messages. Les modèles de communication asynchrones les plus souvent pris en charge par les MOMs sont le *message queuing* et/ou le *publish-subscribe*. Pour rappel, avec le modèle *message queuing*, les messages envoyés par le producteur sont placés dans une file d'attente, identifiée par un nom et liée à un consommateur spécifique. Lorsque ce dernier est prêt à traiter un nouveau message, il invoque la fonction de l'intergiciel lui permettant de rapatrier le premier message de la file. Avec le modèle *publish-subscribe*, les producteurs et les consommateurs interagissent en publiant des évènements et en souscrivant à des classes d'évènements auxquels ils se sont déclarés intéressés auprès de leur intergiciel. Contrairement au modèle *message queuing*, la communication est *multipoint*, *anonyme*, *implicite*, et *sans états*. Elle est *multipoint* (un-à-plusieurs ou plusieurs-à-plusieurs [27], [28]) car les évènements sont envoyés à l'ensemble des consommateurs qui se sont déclarés intéressés. Elle est *anonyme* étant donné que le producteur ne connait pas l'identité des consommateurs. Elle est *implicite* puisque les consommateurs sont déterminés par les souscriptions et non pas explicitement par les producteurs. Enfin, la communication est *sans états* car les évènements générés ne sont pas sauvegardés : ils ne sont envoyés qu'aux consommateurs ayant souscrit avant leur publication.

Traditionnellement, les MOMs requièrent l'utilisation d'intermédiaires, appelés aussi bus évènementiel (*event broker*), routeurs, ou serveurs dans la littérature. Dans le cas du modèle *message queuing*, le bus évènementiel est utilisé pour héberger les files de messages de sorte que même si le consommateur est déconnecté, il lui est toujours possible de rapatrier ultérieurement les messages qui lui sont destinés [5]. Dans le cas du modèle *publish-subscribe*, le bus évènementiel permet de collecter les souscriptions des différents consommateurs, et de répartir les évènements de façon adéquate en réduisant la charge du réseau [36], [37], [31]. Dans les deux cas, l'architecture du bus évènementiel est soit centralisée (un seul serveur) soit distribuée (topologie de serveurs interconnectés qui coopèrent). Il existe de nombreux intergiciels implémentant ces deux modèles, et en particulier de nombreux intergiciels industriels. En effet, les MOMs sont le plus souvent utilisés pour intégrer des applications et des données hétérogènes dans les systèmes d'informations en entreprise. Le choix des MOMs, comme intergiciel privilégié dans l'entreprise, s'explique par la forte capacité des MOMs à gérer, *via* leur bus évènementiel, de façon fiable et robuste simultanément un nombre conséquent d'applications et un trafic élevé de messages. Par exemple :

- Pour le modèle *message-queuing*, on peut citer MQSeries d'IBM [33], MSMQ de Microsoft [35], JMQ de Sun [34].

- Pour le modèle *publish-subscribe*, on peut évoquer SIENA [31], JEDI [45], ELVIN [46], TIBCO/Rendezvous, CEA [30], CORBA Notification Service [19], [18], REBECA [71], et Griphon [28].

Les MOMs diffèrent selon la topologie du bus évènementiel, les méthodes de filtrage des évènements, les algorithmes de distribution de messages (routage), et les propriétés non fonctionnelles offertes comme la fiabilité, l'ordonnancement ou la sécurité. Pour une taxonomie détaillée des MOMs, on peut se référer à [47].

Une grande partie des intergiciels traditionnels, qu'ils soient synchrones ou asynchrones, sont conçus selon l'hypothèse que les hôtes sur lesquels ils sont déployés sont, dans la plupart des cas, stationnaires, et interconnectés *via* une infrastructure réseau fixe. Par conséquent, ils ne sont donc pas adaptés à la mobilité de leur hôte, et aux environnements ubiquitaires [53], [60], [61], [62], [72]. La miniaturisation des objets communicants, leur connectivité sans fil, et leur mobilité ont introduit de nouveaux défis [65], [54], [44] nécessitants une adaptation des intergiciels traditionnels (les principes de base introduits par ces derniers perdurant), favorisant l'émergence d'intergiciels ubiquitaires.

2.5 Propriétés élémentaires des intergiciels ubiquitaires

Usuellement, les intergiciels traditionnels sont à usages généraux et embarquent le plus possible de fonctionnalités les rendant complexes, gourmands en ressources (mémoires, puissance du processeur) et non-adaptés à des hôtes limités en ressources [40]. Une des solutions possibles consiste à adapter les intergiciels traditionnels en réduisant leur taille et en minimisant au maximum leurs fonctionnalités [38], [39], [67]. L'inconvénient de ce principe réside dans son inflexibilité : il n'offre pas la possibilité de sélectionner les fonctionnalités de l'intergiciel à exclure ou à inclure en fonction des services que l'on souhaite déployer et/ou des ressources de l'*objet communicant*. Les auteurs de [40], [41], [42], [66], [70] préconisent l'emploi d'intergiciels réflexifs qui disposent d'une structure interne modulable de manière à les configurer/personnaliser en fonction des besoins.

Avec une connectivité sans fil, la connexion réseau est plus sensible aux interférences qu'avec une connectivité filaire [63]. La bande passante étant variable et les taux d'erreurs plus importants, la communication entre *objets communicants* est moins stable et potentiellement interrompue par des déconnections intermittentes. Le modèle de communication synchrone, qui requiert un couplage fort entre les différentes parties communicantes, n'est pas le modèle le plus adéquat et nécessite d'être adapté, de manière à faire face à la variabilité de la connectivité sans fil [53]. Un modèle mixte entre le synchrone et l'asynchrone (plus précisément le *message queuing* pour sa capacité à rendre plus fiable les interactions grâce aux files

d'attentes), semble être plus approprié. Certains intergiciels pour les hôtes mobiles utilisent un protocole d'invocation de type *semi-asynchrone* fondé sur le modèle RPC qui met en cache les données non émises ou non acquittées [68], [69]. Les intergiciels fondés sur le modèle de communication asynchrone, et en particulier sur le modèle *publish-subscribe* se sont quand à eux montrés particulièrement adaptés aux contraintes des réseaux sans fil [61], [77], [45]. En effet, le modèle asynchrone fournit un découplage dans le temps entre les consommateurs et les producteurs de messages, et grâce au modèle *publish-subscribe*, ces derniers communiquent indépendamment de leur localisation *via* un échange de messages en fonction de leur type ou de leur contenu, plutôt que des adresses de destination. Si les intergiciels *publish-subscribe* traditionnels sont plutôt bien adaptés aux communications sans fil, ils ne prennent toutefois pas en charge la mobilité des objets communicants.

La gestion de la mobilité des *objets communicants* sans fil diffère suivant leurs modèles d'interconnexion : *infrastructure* ou *ad hoc*. Dans le premier modèle, les *objets communicants* sont interconnectés *via* plusieurs points d'accès (aussi appelés bornes ou stations de base) localisés à la périphérie d'une infrastructure réseau dont le cœur est composé de routeurs fixes. Chaque point d'accès dispose d'une couverture réseau sans fil limitée. Ainsi, lorsqu'un *objet communicant* se déplace, il arrive un moment où la qualité de communication entre lui et le point d'accès avec lequel il interagit se dégrade l'obligeant à se connecter à un nouveau point d'accès plus proche de sa nouvelle localisation. Ce changement de point d'accès (*handover*) doit se faire sans entrainer une perte des connexions ou des souscriptions en cours, suivant que le modèle de communication de l'intergiciel de l'*objet communicant* est synchrone [67], ou asynchrone [72]. L'infrastructure réseau et les intergiciels doivent assurer la continuité des communications malgré le changement de localisation. On peut citer, de façon non-exhaustive, ALICE pour CORBA [67], DOLMEN [68], Wireless CORBA [76], Mobile RMI [74], ALICE pour RMI [75] qui sont autant d'intergiciels différents, conçus selon un modèle de communication synchrone, qui prennent en charge le *handover via* la dissémination de passerelles (*gateway*) à la périphérie du réseau fixe pour gérer de façon transparente la mobilité. Concernant les MOMs dépendants d'un intermédiaire, étant donné leur faculté à mémoriser temporairement les messages sur leur bus évènementiel en cas de déconnection des destinataires, ils se révèlent plutôt bien adaptés à la gestion de la mobilité. Toutefois, il leur est nécessaire de détecter la mobilité des producteurs/consommateurs de messages de manière à dispatcher les messages vers leur nouvelle localisation. Par exemple, les intergiciels comme REBECA ou ELVIN ont été adaptés en ce sens (comme indiqué respectivement dans [72], [73]) ; d'autres intergiciels comme JEDI [45], ou CEA [30] ont étés conçus nativement pour prendre en charge cette notion de mobilité. Les intergiciels asynchrones gérant la mobilité diffèrent, non seulement dans leur représentation des messages, mais aussi dans les algorithmes de mise à jour dynamique des tables de routage et la détection de la mobilité qui peut être implicite (détectée par le bus évènementiel) ou explicite (notifiée explicitement par le producteur/consommateur auprès du bus évènementiel) [72].

Avec une interconnexion de type *ad hoc*, les *objets communicants* sont interconnectés *via* un réseau sans fil spontanément et directement sans infrastructure. Dès lors qu'ils se retrouvent dans une même zone géographique, ils forment des groupes d'objets capables d'interagir spontanément entre eux. Le modèle de communication asynchrone, et la notion de groupe de communication basée sur la proximité se révèlent particulièrement adaptés dans ce contexte [79], [80], [81], [84] et requièrent des intergiciels spécialisés comme par exemple LIME [77], [78], STEAM [61], AdHocFs [64], EMMA [60], TOTA [83] et Mobile Gaia [82].

2.6 Synthèse

Bien que les intergiciels existants remplissent leurs rôles, qui est d'abstraire la diversité des systèmes d'exploitation de réseaux et de protocoles d'interaction, ils sont le plus souvent incompatibles entre eux et représentent une nouvelle source d'hétérogénéité de plus haut niveau comme nous le détaillons dans le chapitre suivant. Il s'agit du paradoxe de l'intergiciel [134]. Les différentes approches d'abstractions des divers intergiciels existants rendent difficile l'établissement d'une *architecture de services ubiquitaires*. Un des défis de l'informatique diffuse est de rendre les intergiciels interopérables entre eux.

3 Intergiciels interopérables : solutions existantes

Pour déterminer et raisonner sur les différentes solutions possibles pour résoudre l'hétérogénéité des intergiciels il est nécessaire de considérer un *environnement ubiquitaire* comme un système distribué de systèmes. Un tel système peut être modélisé au moyen des concepts issus des architectures logicielles [89] : les composants abstrayant les services (resp. les clients) et les connecteurs abstrayant les intergiciels (voir Figure 2).

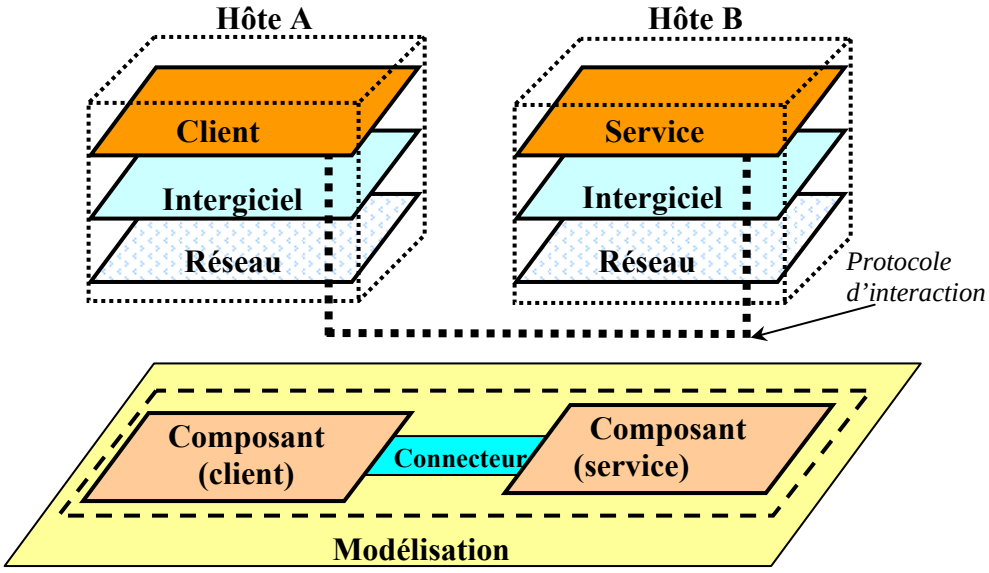


Figure 2. Modélisation des intergiciels, des clients et des services

L'*environnement ubiquitaire* est alors modélisé sous la forme d'un graphe. Les nœuds du graphe reliés par des arcs sont des composants interconnectés *via* des connecteurs. Etant donné l'aspect dynamique d'un *environnement ubiquitaire*, le graphe évolue au cours du temps. L'apparition et/ou la disparition de nouveaux services (resp. de clients) correspond à une modification dynamique du graphe *via* une suppression ou une adjonction de composants et/ou de connecteurs. Cependant, l'évolution du graphe est soumise à une contrainte forte qu'il est nécessaire de surmonter : les composants ne peuvent être associés qu'à des connecteurs pour lesquels ils ont été conçus, ou plus précisément à des connecteurs avec lesquels ils sont compatibles [87]. Ainsi, résoudre l'hétérogénéité des intergiciels revient à résoudre, à un niveau plus abstrait, les incompatibilités d'interconnexions entre les composants et les connecteurs.

Dans ce chapitre, nous présentons dans un premier temps, dans la section 3.1, les concepts inhérents aux architectures logicielles, c'est-à-dire les concepts relatifs aux composants, connecteurs, et configurations. Nous nous focalisons par la suite, dans la

section 3.2, exclusivement sur le langage de processus FSP pour décrire les incompatibilités d'interconnexion entre les connecteurs et les composants. Dans la section 3.3, nous présentons les principales méthodes pour résoudre ces incompatibilités. Dans la section 3.4, nous décrivons, comme exemples d'implémentation, diverses technologies mettant en œuvre ces méthodes pour conclure, dans la section 3.5, sur les limitations des solutions existantes.

3.1 Modélisation de l'environnement ubiquitaire

Les langages de description d'architecture (ou ADL pour *Architecture Description Langage*), employés dans le domaine des architectures logicielles, reposent sur un certain nombre de concepts fondamentaux pour modéliser les systèmes logiciels, dont principalement les composants, les connecteurs et les configurations [90], [88], que nous réutilisons dans le contexte de ce document pour modéliser un *environnement ubiquitaire*. Bien qu'il existe un consensus sur le rôle fondamental de ces trois concepts, il subsiste des divergences sur leur définition respective selon l'ADL considéré [91]. Dans cette section, lorsque nous faisons référence aux notions de composants, de connecteurs, et de configurations nous nous référons à la définition générique donnée par l'ADL ACME [94], [95], que nous rappelons brièvement ci-après, qui est un ADL pivot fédérant les ADL existants.

Les *composants* se présentent comme des briques logicielles dotées d'une ou de plusieurs *interfaces* de communication (Figure 3). Chacune d'entre elles correspond à un *port* de communication. Un composant dispose alors d'un ensemble de *ports* qui sont autant de points d'interaction avec le monde extérieur. Les ports peuvent également être perçus comme des points d'accès au composant.

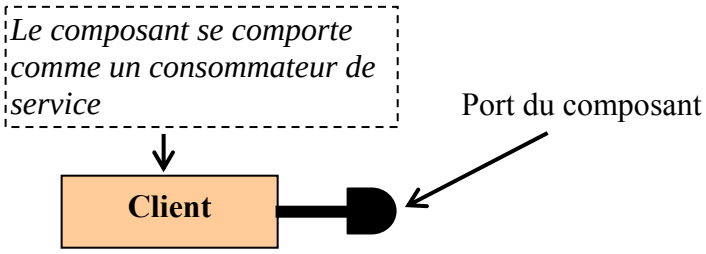


Figure 3. Schématisation d'un composant

A chaque port correspond un protocole d'interaction dont la spécification peut être décrite *via* un langage de processus tel que CSP [97], CCS [98], ou FSP [105] que nous utilisons par la suite. La description d'un composant inclut éventuellement, la spécification d'un comportement global qui coordonne celui de l'ensemble de ses ports, si le composant en possède plusieurs.

Les *connecteurs*, quant à eux, modélisent les interactions entre les composants en définissant les règles qui régissent ces interactions [99], comme illustré sur la Figure 4. Les connecteurs possèdent plusieurs *interfaces* de communication ou *rôles*. Chaque *rôle* définit le comportement attendu du participant à l'interaction. La spécification d'un connecteur inclut également la définition d'un protocole de coordination (appelé *glue*) qui spécifie la coordination des différents rôles du connecteur. Le comportement des rôles et de la *glue* sont exprimés, comme pour les composants, à l'aide d'un langage de processus.

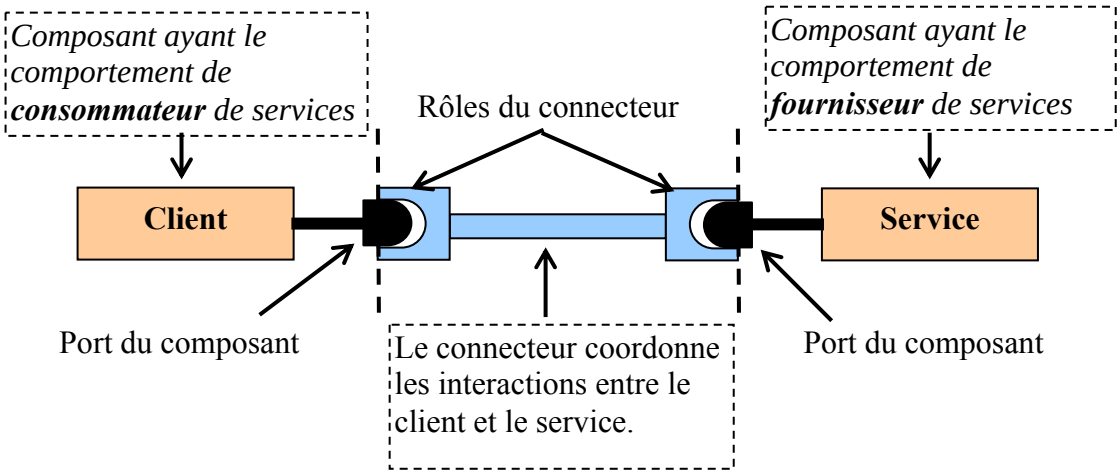


Figure 4. Schématisation d'un connecteur

Les différents modèles de communication réseau des différents intergiciels passés en revue dans le chapitre précédent (synchrone, semi-synchrone, asynchrone et les différentes variantes), sont tous modélisés *via* un connecteur adapté [101], [91]. Par exemple, le connecteur représentant un intergiciel dont le modèle de communication est de type RPC, est un connecteur synchrone, bloquant, et point-à-point qui coordonne deux rôles distincts : *l'appelant* et *l'appelé*. Pour un modèle de communication du type *message-passing*, le connecteur adéquat est un connecteur asynchrone, non-bloquant, et point-à-point qui transfert les messages en provenance de *l'émetteur* au *récepteur* de messages. Certains connecteurs peuvent avoir plusieurs rôles. Par exemple, un connecteur pour une communication asynchrone du type notification d'évènements diffuse les évènements reçus en provenance d'un rôle *diffuseur d'évènements* vers un nombre arbitraire de rôles *récepteurs d'évènements*. D'autres connecteurs modélisent également le modèle de communication *publish-subscribe*.

Enfin, la *configuration* décrit l'architecture d'un système sous la forme d'un assemblage de composants et de connecteurs formant ainsi un graphe. Plus particulièrement, la configuration d'un système peut être *statique* ou *évolutive*, selon la spécificité de l'ADL considéré pour modéliser un système. La configuration est par

exemple statique si l'on considère l'ADL Wright [92], [87] ou évolutive si l'on considère sa version dynamique [93]. L'aspect statique d'une configuration ne correspond pas à notre vision de l'*environnement ubiquitaire* où l'assemblage des composants et des connecteurs est dynamique. Notre perception de la configuration est plus proche des ADL où la configuration est évolutive. On distingue plus précisément deux types d'évolution possible : planifiée, et non planifiée [88], [103]. Une évolution planifiée signifie que le système est uniquement en mesure de s'accommoder à des changements dynamiques prévus à l'avance (par exemple, l'approche de l'ADL Darwin [104] ou de la version dynamique de l'ADL Wright [93]) tandis qu'une évolution non-planifiée caractérise l'aptitude générale du système à s'adapter à des changements dynamiques impromptus (par exemple, l'approche de l'ADL C2 [102]). Cette dernière définition caractérise parfaitement la conception que l'on a d'un *environnement ubiquitaire*. Celui-ci est donc un graphe de composants et de connecteurs qui évolue et/ou réagit au cours du temps *via* des assemblages/désassemblages, des apparitions/disparitions de composants et de connecteurs. L'évolution du système ainsi modélisé n'étant pas planifiée, il y a potentiellement des incohérences de couplages, qui se traduisent par l'incapacité d'interconnecter des composants, étant donné l'incompatibilité de leurs ports respectifs avec les connecteurs qui les interconnectent [96]. L'objectif de la section suivante est de définir ce qu'est exactement une incompatibilité et de mettre en relief les différents problèmes d'incompatibilités susceptibles de survenir (en analysant les ports des composants, les rôles des connecteurs, ainsi que leur *glue*) afin d'introduire par la suite les méthodes existantes pour surmonter ces problèmes de couplage.

3.2 Formalisation des problèmes d'incompatibilités

L'ADL ACME nous a permis d'introduire une définition générique des concepts des architecturels logicielles. Toutefois, pour modéliser un *environnement ubiquitaire* sous la forme d'un système, des définitions plus spécifiques des connecteurs et des composants telles que fournies par l'ADL Wright nous semblent être plus adaptées. En effet, le fait que Wright considère les connecteurs comme des entités de première classe constitue un avantage d'une part parce que les connecteurs peuvent être typées, nommées, et réutilisées contrairement à d'autres ADL où les connecteurs sont implicites et représentés simplement sous formes de lignes pour indiquer que des composants sont interconnectés [87]. Lorsque les connecteurs n'ont pas d'existence propre, les interactions entre composants sont décrites à travers une description globale de l'architecture et par conséquent, il devient difficile de raisonner sur l'interchangeabilité, la réutilisation, et l'adaptation de connecteurs. Et d'autre part parce que la définition explicite de rôles et de *glues* permet de définir expressément le comportement des participants à l'interaction, ainsi que le protocole d'interaction. De plus, nous utilisons l'algèbre de processus FSP pour décrire les ports, les rôles et les *glues* au lieu de CSP utilisé par Wright pour des raisons de simplicité, FSP étant d'une manipulation plus aisée [106].

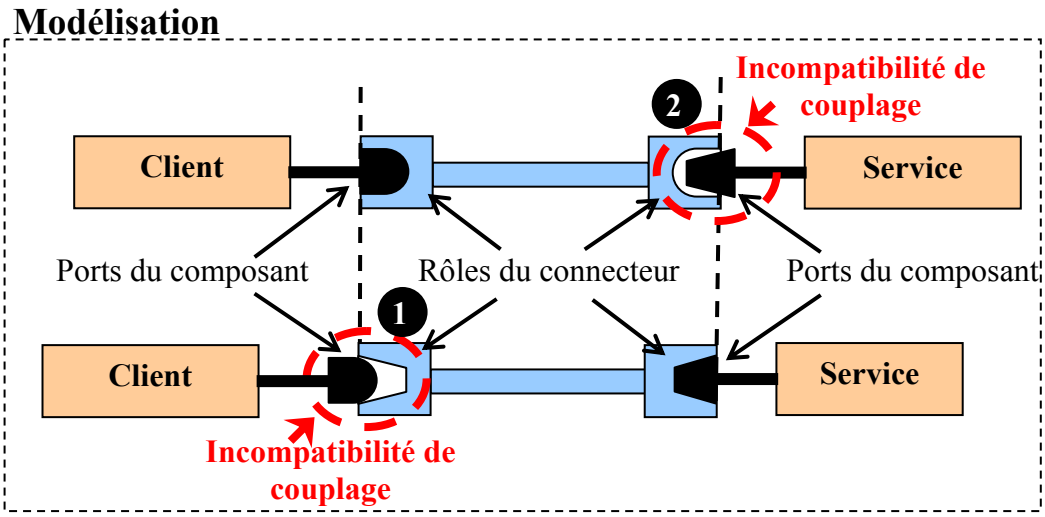


Figure 5. Incompatibilité entre ports et rôles

Ainsi, en ce qui concerne la formalisation des problèmes d'incompatibilités, autoriser le couplage de composants ayant des ports incompatibles revient à résoudre l'incompatibilité de leurs ports respectifs vis-à-vis du connecteur qui les interconnecte. Dans la Figure 5, pour interconnecter deux composants, l'un se comportant comme un client, l'autre comme un service distant, deux cas de figure se présentent : soit le connecteur utilisé est compatible avec le port du client mais incompatible avec le port du service, soit il est compatible avec le port du service mais incompatible avec le port du client. Le comportement des composants et celui des connecteurs étant décrit par l'algèbre de processus FSP, il est nécessaire de présenter au préalable brièvement les bases de FSP (ses principaux opérateurs) afin de spécifier leur sémantique (ou spécification) respective pour illustrer leurs incompatibilités.

Un processus décrit le comportement d'une entité (comme le port d'un composant ou le rôle et la *glue* d'un connecteur) *via* une séquence d'évènements [105]. Un évènement est un message *initié* (c'est-à-dire *émis*) ou *observé* (c'est-à-dire *reçu*) par un processus. L'ensemble des évènements d'un processus P, définit l'alphabet du processus, ou αP. On note $a \rightarrow P$ un processus qui engage (ou émet) un évènement *a* pour se comporter par la suite comme un processus P. Lorsque des processus sont composés en parallèle *via* l'opérateur de composition ||, ils se synchronisent sur les évènements qu'ils partagent dans leur alphabet respectif. Par exemple, lorsque deux processus P et Q sont composés (P||Q), seuls les évènements présents uniquement dans l'alphabet de l'un des processus peuvent survenir indépendamment de l'autre processus. Concernant les évènements présents dans l'alphabet des deux processus, ils ne peuvent survenir que si les deux processus sont simultanément prêts à engager le même évènement.

On note P = (a→ Q | b→ R) un processus qui engage un évènement *a* ou un évènement *b* pour se comporter par la suite respectivement comme un processus Q ou R. Par ailleurs, le processus P = (a→ Q | a → R) engage un évènement *a* pour se comporter de façon non déterministe comme le processus Q ou R. Contrairement à CSP, il n'y a pas de distinction en FSP entre un choix *interne* (choix réalisé par le processus lui-même) et *externe* (choix réalisé par l'environnement d'un processus). Par conséquent, nous considérons par défaut que l'opérateur de choix noté | effectue un choix *externe*. Ainsi, dans l'expression P = (a→ Q | b→ R), le choix d'engager l'évènement *a* ou *b* est réalisé par les autres processus avec lequel P est composé. Pour exprimer un choix *interne* en FSP, nous combinons l'opérateur | avec l'opérateur \ permettant de cacher un évènement, comme par exemple :

$$P = (a\rightarrow Q \mid a \rightarrow R)\backslash\{a\}$$

Ainsi, l'évènement *a* ne pouvant survenir qu'au sein du processus P, le processus effectue un choix *interne* non déterministe pour se comporter comme Q ou R.

Deux autres opérateurs FSP importants sont les opérateurs de renommage de processus et d'évènements. Le renommage d'un processus P par une étiquette *tag*, noté (tag : P), permet de préfixer tous les évènements du processus P par l'étiquette *tag*. Par exemple, si P = (a→ Q | b→ R | c → S) alors le processus devient, suite au renommage (tag : P), P = (tag.a→ Q | tag.b→ R | tag.c → S). Les évènements *tag.a*, *tag.b*, *tag.c* sont les nouveaux évènements de l'alphabet du processus P. Il est également possible de renommer un évènement particulier d'un processus *via* l'opérateur de renommage d'évènements noté /. Ainsi, P/{d/a} remplace tous les évènements *a* du processus P par les évènements *d*.

Enfin, il est parfois utile de définir des ensembles d'évènements afin de synthétiser la spécification du comportement de certain processus. Par exemple, soit le processus P suivant :

$$P= (a\rightarrow P \mid b\rightarrow P \mid c \rightarrow P \mid d \rightarrow P \mid e \rightarrow P \mid f \rightarrow P)$$

En définissant l'ensemble d'évènements S = {a, b, c, d, e}, on obtient une définition plus synthétique du processus P : P= [e : S] → P.

Nous rappelons les principaux operateurs de FSP dans la Figure 6. Par ailleurs, pour une description plus détaillée de FSP, le lecteur peut se référer à [105]. Intéressons nous dorénavant à la spécification d'un connecteur en FSP.

a→ P	Engagement d'un évènement
a→ Q | b→ R	Choix
P||Q	Composition parallèle
label:P	Re-nommage du processus
P/{new/old}	Re-nommage d'évènements
P\{hidden}	Evènement caché
Set S = {a,b,c}	Définition d'un ensemble S
[v :S]	Associe la variable à une valeur de S

Figure 6. Syntaxe de l'algèbre de processus FSP

La spécification d'un connecteur est définie par un ensemble de processus [92], [91]. A chaque rôle correspond un processus. Ces processus sont indépendants les uns des autres, toutefois, ils sont coordonnés *via* le processus de la *glue*. Ainsi, le comportement d'un connecteur est le résultat de la composition en parallèle des processus de ses rôles et de celui de sa *glue*.

Connector RPC-Call-1
Role Client = request → response → Client
Role Service = request → response → Service
Glue = c.request → s.request → Glue
| s.response → c.response → Glue

‖RPC-Call= c : Client ‖ s : Service ‖ Glue

Figure 7. Spécification d'un connecteur de type RPC

Formellement, la sémantique d'un connecteur ayant n rôles $R_1 \ldots R_n$ se note :

$$\text{Glue} \parallel R_1 \parallel R_2 \parallel \ldots \parallel R_3$$
$$\text{Et } \alpha\text{Glue} = \alpha R_1 \cup \alpha R_2 \cup \ldots \cup \alpha R_n$$

L'alphabet de chacun des rôles est choisi de façon à ce qu'aucun d'entre eux n'ait d'évènements en commun. Seul l'alphabet de la *glue* inclut (ou partage) les évènements des alphabets des rôles. Ainsi la coordination entre les différents rôles se fait exclusivement *via* le processus de la *glue*. La Figure 7 illustre la spécification d'un connecteur de type RPC. Le connecteur dispose de deux rôles *client* et *service*. Dans la Figure 7, le rôle *client* précise qu'une fois que le client a émis une requête à destination d'un service distant, il attend une réponse de ce dernier tandis que le rôle *service* indique que le service attend la réception d'une requête avant d'émettre une réponse. La *glue* quant à elle coordonne les deux rôles : elle attend que le client initie

une requête afin de la transmettre au service, ou que le service émette une réponse pour la transmettre au client. Ce choix est dit externe car il est réalisé par l'environnement de la *glue* (c'est-à-dire par le client ou le service) et non par la *glue* elle-même.

De façon comparable aux connecteurs, la sémantique du composant est donnée par les processus de ses ports. Un composant se retrouve connecté à un connecteur en associant les ports de l'un aux rôles de l'autre [91]. Plus précisément l'association port/rôle se fait en remplaçant le protocole du rôle d'un connecteur par celui du port du composant connecté. Nous introduisons l'opérateur P ➲ Q qui autorise le remplacement d'un processus P par un processus Q. Ainsi, formellement, l'association de n ports $P_1 \ldots P_n$ aux rôles $R_1 \ldots R_n$ d'un connecteur est le processus suivant :

$$\text{Glue} \parallel R_1 \rightarrowtail P_1 \parallel R_2 \rightarrowtail P_2 \parallel \ldots \parallel R_n \rightarrowtail P_n$$

Le protocole du connecteur est donc le résultat de la composition obtenu par le remplacement des processus des rôles par ceux des ports. Déterminer la compatibilité ou l'incompatibilité entre un composant et un connecteur se résume donc à savoir si le comportement du port est similaire à celui du rôle qu'il remplace, et s'il diffère, de déterminer s'il répond aux contraintes de l'interaction [92]. Cette notion de compatibilité s'exprime dans les langages de processus par le concept de raffinement [91]. Un processus P_1 est le raffinement d'un processus R_1 (se note $R_1 \subseteq P_1$) si et seulement si : (i) P_1 et R_1 partagent le même alphabet, et (ii) les comportements de R_1 incluent ceux de P_1. Ainsi, un port P_1 est compatible avec un rôle R_1 si et seulement si $R_1 \subseteq P_1$. On peut se reporter aux références [87], [92] pour avoir une démonstration de cette définition. Implicitement, un composant est donc incompatible avec un connecteur si le format des messages échangés (implémentation de l'alphabet) et les règles de coordination entre ces messages (implémentation du protocole de coordination) sont non compatibles.

Considérons, par exemple, que le client et le service de la Figure 5 sont respectivement conçus à l'aide d'un intergiciel RMI [21] et SOAP [25]. Les spécifications des connecteurs qui modélisent ces intergiciels sont illustrées dans la Figure 8a et 8b. Celles des composants SOAP et RMI ne sont pas représentées. En effet, les spécifications de leurs ports respectifs correspondent à celles des rôles des connecteurs SOAP et RMI.

La comparaison des spécifications, et en particulier de leur alphabet, des connecteurs de la Figure 8 met en évidence qu'un client SOAP est dans l'incapacité d'interagir avec un service RMI non pas parce qu'ils diffèrent dans leur modèle de communication (tous les deux suivent un modèle RPC) mais parce qu'ils diffèrent dans le formatage des messages échangés.

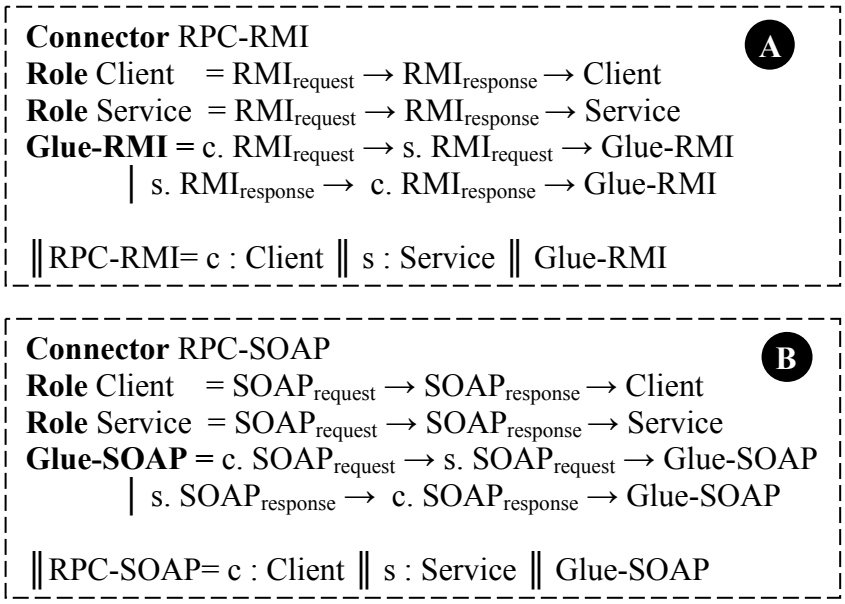


Figure 8. Spécifications d'un connecteur SOAP et RMI

Formellement, un client SOAP connecté à un service RMI *via* un connecteur RMI (Figure 9) se note :

$$\text{Glue-RMI} \parallel P_{\text{Client-SOAP}} \parallel P_{\text{Service-RMI}} \qquad (1)$$

Le processus résultant de l'équation (1) se bloque (*deadlock*) car la *glue* RMI n'a aucun évènement en commun avec le processus du port du client RMI.

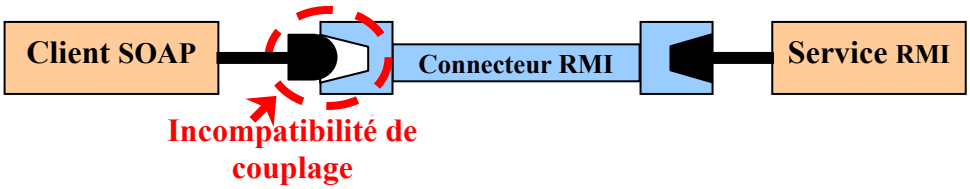


Figure 9. Incompatibilité de couplage entre un client SOAP et un connecteur RMI

De façon similaire, si le client SOAP est connecté à un service RMI *via* un connecteur SOAP (Figure 10) on obtient :

$$\text{Glue-SOAP} \parallel P_{\text{Client-SOAP}} \parallel P_{\text{Service-RMI}} \qquad (2)$$

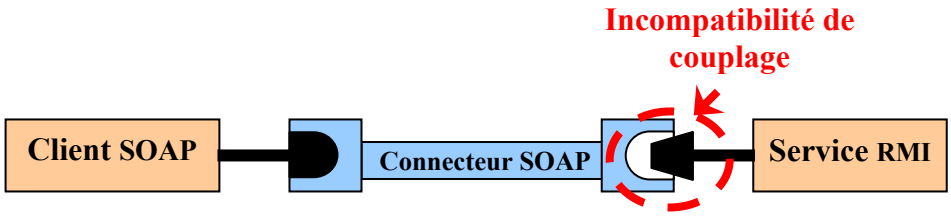


Figure 10. Incompatibilité de couplage entre un client SOAP et un connecteur RMI

Le processus résultant de l'équation (2) se bloque pour les mêmes raisons que précédemment : la *glue* SOAP ne partage aucun évènement en commun avec le processus du port du service RMI.

Comme précisé antérieurement, l'incompatibilité entre composants et connecteurs survient également pour des raisons de comportements incompatibles. Considérons dorénavant que le client et le service de la Figure 5 sont conçus respectivement à l'aide d'un intergiciel RMI et d'un intergiciel transactionnel RMI. Une spécification simplifiée du connecteur représentant ce dernier est donnée dans la Figure 11a. Elle se différencie de celle du connecteur RMI traditionnel par ses propriétés ACID : les requêtes RPC se font au sein d'une session et ne deviennent persistantes qu'une fois la session validée.

A

Connector RPC-Transaction-RMI
Role Client = open → SESSION,
SESSION= $RMI_{request}$ → $RMI_{response}$ → SESSION | close → Client
Role Service = open → SESSION,
SESSION= $RMI_{request}$ → $RMI_{response}$→ SESSION | close → Service
Glue-TRMI = c.open → s.open → Glue-TRMI,
| c.$RMI_{request}$ → s.$RMI_{response}$ → Glue-TRMI,
| s.$RMI_{response}$ → c.$RMI_{response}$ → Glue-TRMI,
| c.close → s.close → Glue-TRMI,

‖RPC-Transaction-RMI = c : Client ‖ s : Service ‖ Glue-TRMI

B

Connector Message-Passing
Role Client = request → Client | response → Client
Role Service = request → Service | response → Service
Glue-MP = c.request → s.request → Glue-MP,
| s.response → c.response → Glue-MP
‖Message-Passing = c : Client ‖ s : Service ‖ Glue-MP

Figure 11. Spécifications d'un connecteur transactionnel et orienté message.

Au niveau de la spécification du connecteur transactionnel, les propriétés ACID se traduisent par le fait que le rôle *client* doit engager un évènement *open* avant de générer des requêtes RPC pour ouvrir une session et clôturer cette dernière par un évènement *close* afin que le service rende les requêtes du client persistantes[1]. Deux cas de figures se présentent suivant que le client est connecté au service distant *via* un connecteur transactionnel ou non.

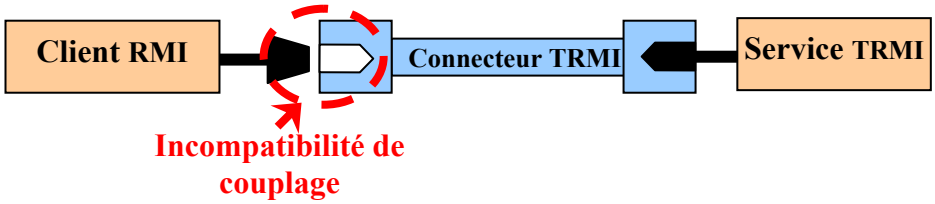


Figure 12. Incompatibilité de couplage entre un client RMI et un connecteur TRMI

Dans le premier cas (Figure 12), on obtient formellement l'équation suivante :

$$\text{Glue-TRMI} \| \; P_{\text{Client-RMI}} \| P_{\text{Service-TRMI}} \qquad (3)$$

Le processus résultant de l'équation (3) se bloque alors que les processus du port du client $P_{\text{Client-RMI}}$, de celui du service $P_{\text{Service-TRMI}}$ et de la *glue* Glue-TRMI partagent des évènements identiques comme $\text{RMI}_{\text{request}}$ et $\text{RMI}_{\text{response}}$. En effet, le processus de la *glue* et de celui du port du service sont incapables de se coordonner : $P_{\text{Service-TRMI}}$ ne peut engager un évènement $\text{RMI}_{\text{request}}$ que s'il reçoit préalablement un évènement *open* que le processus client $P_{\text{Client-RMI}}$ n'engage jamais avant de d'émettre une requête RPC (voir le rôle *client* Figure 8a).

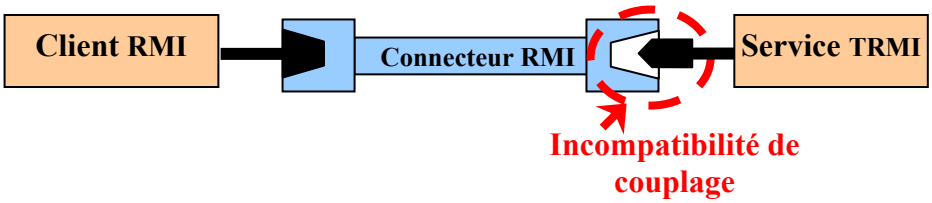


Figure 13. Incompatibilité de couplage entre un service TRMI et un connecteur RMI

Dans le second cas (Figure 13), l'interconnexion du client RMI et du service TRMI *via* un connecteur non transactionnel RMI se note :

$$\text{Glue-RMI} \| \; P_{\text{Client-RMI}} \| P_{\text{Service-TRMI}} \qquad (4)$$

[1] Par ex., un service bancaire rend persistant les modifications occasionnées par les requêtes du client.

Le processus de l'équation (4) est bloquant pour les mêmes raisons que celui de l'équation (3) : l'incompatibilité est due à une incompatibilité de comportements. Par ailleurs, dans les deux équations (3&4) l'alphabet du service est plus riche que celui du client et représente de fait une source d'incompatibilité supplémentaire.

Enfin si l'on compare la spécification d'un connecteur asynchrone (Figure 11b) avec un connecteur synchrone (Figure 7), on constate de manière évidente un autre exemple d'incompatibilité de couplage pour faute d'incompatibilité de comportement. Après l'émission d'une requête auprès d'un service distant asynchrone, un client synchrone est susceptible d'attendre indéfiniment la réception d'une réponse qui ne parvient pas[2].

En nous appuyant sur la définition de l'incompatibilité de couplages entre composants et connecteurs, nous présentons dans la section suivante les solutions permettant de les résoudre.

3.3 Résolution des incompatibilités

Lorsque deux composants sont incapables de communiquer *via* le connecteur qui les interconnecte, deux solutions sont possibles [108] : soit on crée un nouveau connecteur, soit on modifie le connecteur existant afin de résoudre les incompatibilités de couplage. Pour Spitznagel, la transformation d'un connecteur est réalisable en lui appliquant une série d'opérations élémentaires de transformation [106], que l'auteur classifie dans [107]. En fonction de la combinaison de ces opérations, on obtient la transformation désirée. L'idée est d'associer à un connecteur existant un adaptateur (ou *wrapper*), qui regroupe l'ensemble des opérations à appliquer, pour obtenir le protocole d'interaction souhaité. Suivant ce principe, pour résoudre l'hétérogénéité des intergiciels, et sur la base des solutions existantes, nous distinguons deux types majeurs de transformations : la *substitution* et la *traduction de protocoles*, dont nous donnons ci-après une interprétation en FSP. Nous considérons que chacune de ces deux transformations constitue une approche distincte à la résolution des incompatibilités de couplage.

- **Substitution de protocoles**

Le principe de la *substitution de protocoles* consiste à remplacer à la volée le protocole d'un composant par un autre protocole plus adapté en fonction de son environnement d'exécution. Ce fonctionnement est illustré dans la Figure 14 : le composant C1 est alternativement associé au connecteur 1 et 4 suivant le protocole d'interaction du composant avec lequel il interagit. La *substitution de protocoles* se modélise aisément *via* un nouveau connecteur qui agrège un ensemble de *n*

[2] Plus exactement, le client attend l'expiration du délai d'attente d'une réponse provoquant ainsi une erreur.

connecteurs et qui se comporte, selon le contexte, comme l'un d'entre eux. Suivant les travaux de Spitznagel [107], [106], ce nouveau connecteur (C-SUB) est obtenu par la combinaison de trois opérations de transformation : commutation, agrégation de rôles et de *glues*, que nous appliquons à *n* connecteurs.

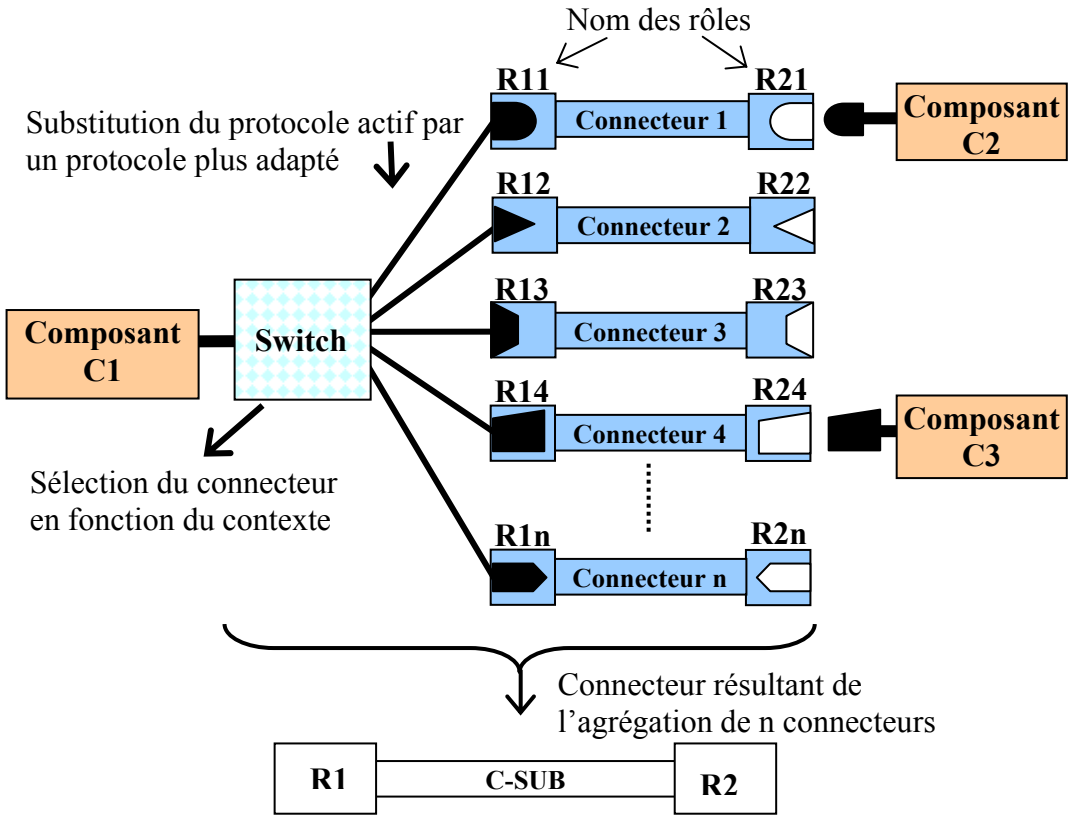


Figure 14. Principe de la substitution de protocoles

Intéressons-nous dorénavant à la spécification de C-SUB, et notamment à sa *glue*, notée SUB. Tout d'abord, SUB est composée des *glues* de *n* connecteurs, soit :

$$SUB = \|_{i \in [1,..,n]} \ \mathbf{Glue}i \qquad (1)$$

Une seule *glue* à la fois, parmi Glue1,…, GlueN, doit être active. Le résultat escompté de SUB n'est pas une composition des *glues* des connecteurs agrégés mais une substitution de *glue*. Par conséquent, il est nécessaire de s'assurer que l'alphabet de chacune des *glues* n'ait aucun évènement en commun de sorte que les *glues* ne puissent pas se synchroniser. On marque (à l'aide d'une étiquette) alors l'alphabet de chacune des *glues* par un identifiant unique :

$$SUB = \|_{i \in [1,..,n]} \ \mathbf{tag}i \ \mathbf{:Glue}i \qquad (2)$$

Maintenant que les *glues* sont indépendantes les unes des autres, il est indispensable de pouvoir en activer une à un instant donné. L'élection de la *glue* est réalisée *via* un processus *Switch* qui émet un évènement pour indiquer la *glue* active. On obtient :

$$\text{SUB} = \mathbf{Switch} \;\|_{i \in [1,..,n]}\; \mathbf{tag}\boldsymbol{i} : \mathbf{Glue}\boldsymbol{i}, \qquad (3)$$

avec,

$$\mathbf{Switch} = (\text{election} \rightarrow \text{reset} \rightarrow \mathbf{Switch} \;|_{i \in [1,..,n]}\; \text{election} \rightarrow \text{glue}\boldsymbol{i} \rightarrow \mathbf{Switch}) \setminus \{\text{election}\}$$

L'évènement *election* du processus Switch est un évènement interne non observable (par les autres processus de SUB). La décision d'activer la *glue**n*** à un instant t se fait exclusivement par le processus *Switch* lui-même (choix interne), selon sa propre stratégie de sélection. Par ailleurs, le processus *Switch* a la possibilité d'engager un évènement *reset* pour réinitialiser les autres processus de SUB afin de réitérer une sélection de *glue*.

La spécification des rôles R1 et R2 inclut respectivement l'ensemble des spécifications des rôles R1*i* et de R2*i* des *n* connecteurs agrégés, pour $i \in [1,.., n]$. Les processus $R1i_{\,i \in [1,..,n]}$ et $R2i_{\,i \in [1,..,n]}$ deviennent des processus locaux de R1 et de R2 sélectionnés/activés par ces derniers selon l'évènement *glue**n*** engagé par *Switch*. Ainsi, pour R1 et R2, on note :

$$\mathbf{Role}\ R1 = |_{\,i \in [1,..,n]}\ (\text{glue}\boldsymbol{i} \rightarrow R1\boldsymbol{i}),$$

$R1\boldsymbol{i}_{\,i \in [1,..,n]}$ = *spécification initiale du rôle R1**i*** *tel que donnée par le connecteur i* | reset → R1,

$$\mathbf{Role}\ R2 = |_{\,i \in [1,..,n]}\ (\text{glue}\boldsymbol{i} \rightarrow R2\boldsymbol{i}),$$

$R2\boldsymbol{i}_{\,i \in [1,..,n]}$ = *spécification initiale du rôle R2**i*** *tel que donnée par le connecteur i* | reset → R2,

On compose R1 et R2 ainsi définis avec l'équation (3) de SUB et l'on obtient :

$$\text{SUB} = \| \mathbf{r1:R1} \;\|\; \mathbf{r2 : R2} \;\|\; \mathbf{Switch} \;\|_{i \in [1,..,n]}\; \mathbf{tagi : Gluei} \qquad (4)$$

Toutefois l'équation (4) n'est pas encore fonctionnelle car les alphabets des processus locaux de R1 et de R2 et de celui des *glues* marquées par un identifiant unique sont disjoints. Il est donc nécessaire de composer les processus R1 et R2 respectivement à l'aide des processus W1 et W2 dont leur rôle est de faire la jonction avec la *glue* sélectionnée par le processus Switch. La spécification complète de C-SUB est donnée dans la Figure 15.

//Définition des connecteurs pris en charge par C-SUB

Role R1 = | $_{\mathbf{i} \in [1,..,n]}$ (glue***i*** → R1***i***),

R1***i*** $_{i \in [1,..,n]}$ = *spécification initiale du rôle R1**i** tel que donnée par le connecteur i* | reset → R1

Role R2 = | $_{\mathbf{i} \in [1,..,n]}$ (glue***i*** → R2***i***),

R2***i*** $_{i \in [1,..,n]}$ = *spécification initiale du rôle R2**i** tel que donnée par le connecteur i* | reset → R2

Glue1 = *spécification de la glue**1** qui coordonne les rôles R11 et R12*
Glue2 = *spécification de la glue**2** qui coordonne les rôles R12 et R22*

Glue*n* = *spécification de la glue**n** qui coordonne les rôles R1**n** et R2**n***

//Définition des évènements initiés et observés pour chaque rôle

Set E1***i*** $_{i \in [1,..,n]}$ = { αR1***i*** ∩ αGlue***i*** }
Set E2***i*** $_{i \in [1,..,n]}$ = { αR2***i*** ∩ αGlue***i*** }

//Définitions des processus de jonction (*wrapper*)

W1 = | $_{\mathbf{i} \in [1,..,n]}$ (glue***i*** → ToGlue***i***),

ToGlue***i*** $_{i \in [1,..n]}$ = [e:E1***i***] → tag***i***.e → ToGlue***i*** | a.reset → W1,

W2 = | $_{\mathbf{i} \in [1,..,n]}$ (glue***i*** → ToGlue***i***),

ToGlue***i*** $_{i \in [1,..n]}$ = [e:E1***i***] → tag***i***.e → ToGlue***i*** | a.reset → W1,

//Définitions de sélection du connecteur

Switch = (election→ reset →**Switch**
| $_{\mathbf{i \in [1,..,n]}}$ (election → glue***i*** → **Switch**))\ {election}

//Glue du connecteur

C-SUB=|| (R1 || W1 || R2 || W2 || Switch ||$_{i \in [1,..,n]}$ tagi:Gluei

Figure 15. Spécification complète de la substitution de protocoles

Etant donné la capacité du rôle R1 du connecteur C-SUB à muter, les composants destinés à lui être connectés doivent être réimplémentés et spécifiquement adaptés. Implicitement le port P1 d'un tel composant est conçu de telle sorte que P1=R1. Ainsi, l'inconvénient majeur de la *substitution de protocoles* est de résoudre l'incompatibilité d'interconnexion entre composants de façon non transparente contrairement à *traduction de protocoles* présenté ci-après.

- **Traduction de protocole**

Le principe de la *traduction de protocoles* consiste à ajouter à l'une des extrémités d'un connecteur, un adaptateur afin de corriger l'incompatibilité de l'un de ses rôles. Ce principe est illustré dans la Figure 16A : si le connecteur 1 (resp. le connecteur 2) est le connecteur utilisé, pour interconnecter deux composants C1 et C2, il faut adapter son rôle R11 (resp. R22) afin qu'il se comporte comme le rôle R12 (resp. R21) du connecteur 2 (resp. du 1) usuellement utilisé par le composant C1 (resp. C2). En d'autres termes, le connecteur-traducteur (C-TRAD) est conçu à partir d'un connecteur de départ auquel on a ajouté une certaine combinaison d'opérations de transformations. Contrairement à l'approche précédente, il n'y a pas d'automatisation de résolution de l'incompatibilité de couplage car les opérations de transformation à appliquer se font au cas par cas en fonction des ports des composants à interconnecter et implicitement des connecteurs à *fusionner* (Figure 16B).

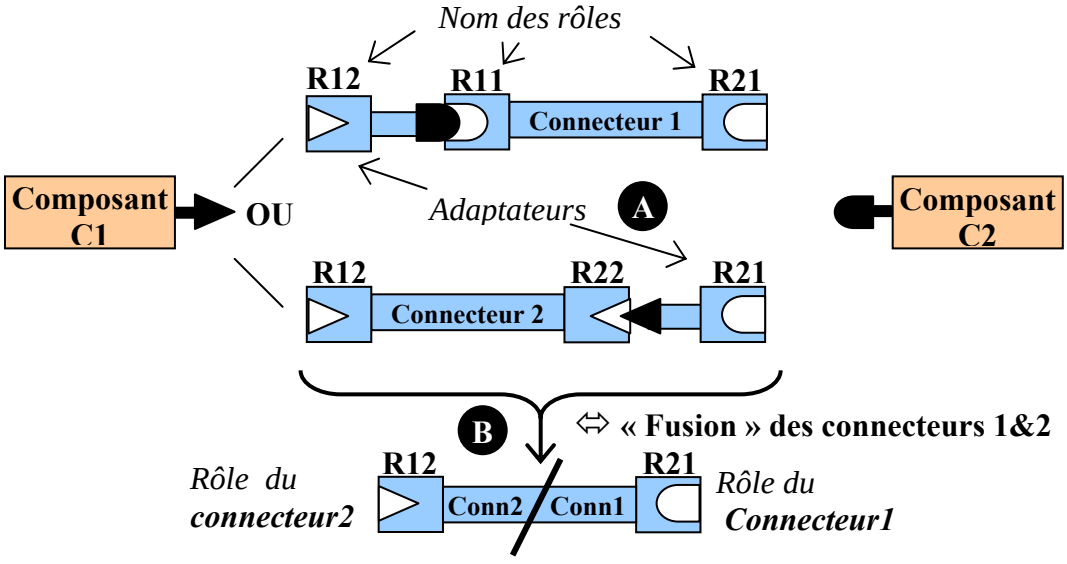


Figure 16. Principe de la traduction de protocoles

Adapter le rôle R11 du connecteur 1 pour qu'il se comporte comme le rôle R12 du connecteur 2 ou adapter le rôle R22 du connecteur 2 pour qu'il se comporte comme le rôle R21 du connecteur 1 revient, entre autres transformations, *à fusionner* le connecteur 1 avec le connecteur 2 (Figure 16B). Par conséquent, parmi les diverses opérations à combiner pour concevoir C-TRAD, l'opération de jointure [107], qui permet de faire la jonction entre les rôles des deux connecteurs fusionnés (Figure

16B), est la seule opération que l'on est toujours sûr d'utiliser. Il est possible d'en déduire une spécification minimale de C-TRAD, donnée dans la Figure 17.

//Connecteur1
Role R11 = *Spécification du rôle R11 du connecteur 1*
Role R21 = *Spécification du rôle R21 du connecteur 1*
Glue1 = *Spécification de la glue du connecteur 1*

//Connecteur2
Role R12 = *Spécification du rôle R12 connecteur 2*
Role R22 = *Spécification du rôle R22 connecteur 2*
Glue2 = *Spécification du rôle de la glue du connecteur 2*

Set I1= *évènements initiés par les rôles R11 et R12*
Set I2= *évènements initiés par les rôles R21 et R22*

Bridge = tag1.[e:I1] → tag2.e → Bridge
| tag2.[e:I2] → tag1.e → Bridge

⊤ = *processus à définir suivant les adaptations à apporter au connecteur1 ou au connecteur2*

||C-TRAD = tag1:(r1:RoleX||Glue1)|| ⊤ || Bridge ||tag2: (r2:RoleY || Glue2)

Figure 17. Spécification minimale de la traduction de protocoles

L'opération de jointure assure simplement une redirection des évènements issus des rôles des deux connecteurs à *fusionner* sans y apporter de transformations. Par exemple, dans le cas de la Figure 16B, l'opération de jointure consiste simplement à transférer les évènements du rôle R12 du connecteur 2 vers le rôle R21 du connecteur 1 et *vice versa*. Ainsi, la spécification minimale de C-TRAD (Figure 17) est à compléter par celles de processus supplémentaires, opérant des transformations, en fonction des adaptations à effectuer sur l'un des rôles d'un des deux connecteurs à *fusionner*. De plus, la position de ces processus additionnels au sein de la *glue* de C-TRAD est variable et à définir suivant, par exemple, qu'il faille transformer ou non un rôle avant de le composer avec sa *glue*. Afin d'illustrer l'utilisation de la spécification de C-TRAD, reprenons un des exemples d'incompatibilité de la section précédente où un client RMI est interconnecté à un service distant RMI qui a la particularité d'être transactionnel. Pour rappel, deux problèmes d'incompatibilité ont été identifiés entre les connecteurs RMI et TRMI: (i) l'incompatibilité de comportement de leurs rôles, et (ii) l'asymétrie de leur alphabet. Le connecteur que l'on souhaite concevoir est illustré dans la Figure 18 et sa spécification complète est donnée dans la Figure 19.

Figure 18. Traducteur RMI/TRMI

Avant de traiter une requête, le service TRMI requiert au préalable la réception d'un évènement *open* pour initier une session transactionnelle et un évènement *close* pour la clôturer. Le client RMI n'engageant jamais ces deux évènements, et n'ayant aucune connaissance de la notion de transaction, il est nécessaire :

1. D'enrichir en conséquence son alphabet.

2. De faire précéder chacune de ses requêtes (évènement $RMI_{request}$) par l'émission d'un évènement *open* et succéder chacune des réponses (évènement $RMI_{response}$) par l'émission d'un évènement *close*.

Il est alors nécessaire d'interposer entre le rôle *Client* du connecteur RPC-RMI et sa *glue* un processus $\top$ qui succède et précède les évènements $RMI_{request}$ et $RMI_{response}$ émis et reçus par le client par des évènements *open* et *close*. Comme indiqué dans la Figure 19, le processus $\top$ se note :

$$\top = c.RMI_{request} \rightarrow tag1.c.open \rightarrow tag1.c.RMI_{request} \\ \mid tag1.c.RMI_{response} \rightarrow tag1.c.close \rightarrow c.RMI_{response}$$

L'évènement $c.RMI_{request}$ initié par le rôle *Client* est transformé en deux évènements successifs *tag1.c.open* et $tag1.c.RMI_{request}$. L'évènement *tag1.c.open* ne faisant pas partie de l'alphabet de la *glue Glue-RMI*, le processus $\top$ se synchronise avec le processus *Bridge* qui, en renommant l'évènement *tag1.c.open* en *tag2.c.open,* se synchronise à son tour avec la *glue Glue-TRMI,* qui enfin transfère cet évènement au rôle *TService*. Ce dernier, ayant engagé l'évènement *open,* devient prêt à recevoir une requête, qui arrive selon un cheminement similaire à partir du processus $\top$ avec l'évènement $tag1.c.RMI_{request}$. Suivant un principe analogue, le rôle *Client* ne reçoit l'évènement $c.RMI_{response}$ qu'après que le processus $\top$ ait préalablement émis un évènement *tag1.c.close* pour fermer la session transactionnelle. Ainsi, grâce au processus $\top$, le rôle *Client* du connecteur RPC-RMI étant en mesure d'initier les évènements *open* et *close*, il devient alors possible d'altérer le processus *Bridge* pour lui indiquer de les transférer vers le rôle *TService* du connecteur RPC-Transaction-RMI. Il devient évident que la spécification de C-TRAD (Figure 17) a été personnalisée suivant la particularité des composants à interconnecter.

L'avantage de la *traduction de protocole* est de résoudre l'incompatibilité d'interconnexions entre composants sans requérir une modification de leurs ports. En revanche, cette traduction n'est pas automatisée, elle est spécifique à un couple

particulier de composants. Par ailleurs, avec le principe de la *traduction de protocole*, la sémantique du connecteur-traducteur peut parfois ne pas tout à fait correspondre à la sémantique de l'un des ports interconnectés. Par exemple, dans le cas précédent, à moins d'altérer son port, le client RMI ne peut émettre, au sein d'une même session, qu'une seule et unique requête alors qu'une session est à l'origine conçue pour qu'il y ait au contraire plus d'une requête.

```
//Connecteur RPC-RMI
Role Client  = RMI_request → RMI_response → Client
Role Service  = RMI_request → RMI_response → Service
Glue-RMI = c. RMI_request → s. RMI_request → Glue-RMI
         | s. RMI_response →  c. RMI_response → Glue-RMI

//Connecteur RPC-Transaction-RMI
Role TClient = open → SESSION,
        SESSION= RMI_request → RMI_response → SESSION | close → Tclient
Role TService = open → SESSION,
        SESSION= RMI_request → RMI_response→ SESSION | close → TService
Glue-TRMI = c.open → s.open,
         | c.RMI_request → s.RMI_response → Glue-TRMI,
         | s.RMI_response →  c.RMI_response → Glue-TRMI,
         | c.close →  s.close → Glue-TRMI

⊤ = c.RMI_request → tag1.c.open → tag1.c.RMI_request
        | tag1.c.RMI_response → c.RMI_response → tag1.c.close

Bridge = tag1.c.open → tag2.c.open → Bridge
                 | tag1.c.close → tag2.c.close → Bridge
                 | tag1.s.RMI_request → tag2.s.RMI_request → Bridge
                 | tag2.c.RMI_response → tag1.c.RMI_response → Bridge

||ClientToTService=
   || c:Client || ⊤ || tag1:Glue-RMI || Bridge || tag2: (s:TService || Glue-TRMI)
```

Figure 19. Spécification complète du traducteur RMI/TRMI

La *substitution* et la *traduction de protocoles* constituent deux solutions à la résolution de l'hétérogénéité des intergiciels, chacun ayant des avantages et des inconvénients respectifs. Quoi qu'il en soit, le type de connecteur qu'elles introduisent (C-SUB et C-TRAD) modélise des intergiciels qualifiés d'*interopérables* du fait de leur capacité d'interopérer avec des intergiciels hétérogènes. Dans la section suivante, différents exemples d'intergiciels *interopérables* concrétisant les principes de ces connecteurs sont présentés.

3.4 Exemples d'intergiciels interopérables

Comme exemples de concrétisation du connecteur C-SUB, nous présentons dans la section 3.4.1 ReMMoC[3] [109], [120] puis OSGI[4] [117], le premier appartenant à la dernière génération des intergiciels réflexifs, le second étant un des intergiciels ubiquitaires industriels le plus souvent utilisé, tandis que pour le connecteur C-TRAD nous introduisons dans la section 3.4.2 les ESBs[5] [118] qui définissent les bases de la *traduction de protocoles* à grande échelle.

3.4.1 Implémentation de la substitution de protocoles

La spécification de la *substitution de protocole* donnée dans la section précédente dénote le caractère dynamique et reconfigurable du connecteur C-SUB et implicitement celui de l'intergiciel ainsi modélisé. Dans la solution présentée dans les références [41], [42], la capacité d'un intergiciel à se reconfigurer dynamiquement est une fonctionnalité clé de la prochaine génération d'intergiciels afin qu'ils puissent s'adapter aux changements susceptibles de survenir dans leur environnement. L'auteur de la référence [42] prône la création d'intergiciels reconfigurables *via* le mariage des techniques de réflexion [112] et des technologies à composants logiciels [111]. Ces composants, sont des briques logicielles préfabriquées et indépendantes conçues pour être composées les unes avec les autres [110]. Il existe plusieurs technologies à composants : COM [121], EJB [122], CCM [123], FRACTAL [124], .Net [125]. Bien que la définition d'un composant varie d'une technologie à une autre, les concepts clés sont les *interfaces*, les *réceptacles* et les connexions (voir Figure 20). Un composant est décrit par des *interfaces* et des *réceptacles* précisant respectivement les fonctions (ou opérations) fournies et requises par le composant. Une connexion représente une liaison entre deux composants ayant une *interface* et un *réceptacle* identiques.

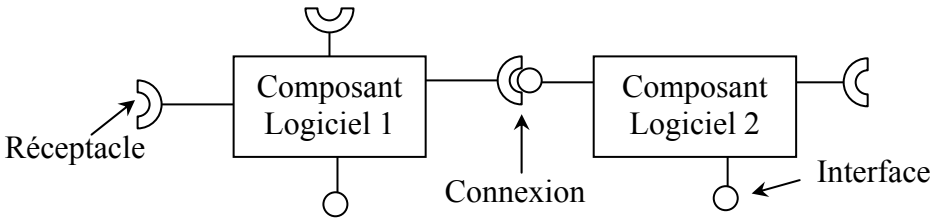


Figure 20. Composants logiciels

L'utilisation de composants logiciels pour construire la structure interne d'un intergiciel favorise sa reconfiguration, et la réflexion le dote de la capacité à s'inspecter et à se modifier lui-même au cours de son exécution. Un groupe d'intergiciels réflexifs expérimentaux implémentant ces principes ont émergé ces dix

[3] Reflective Middleware for Mobile Computing.
[4] Open Service Gateway Initiative.
[5] Enterprise Service Bus.

dernières années [113], tel que OpenOrb [42], DynamicTAO [114], OpenCORBA [115], et OpenCOM [126] et constituent le point de départ de la conception d'intergiciels réflexifs conçus pour résoudre l'hétérogénéité d'intergiciels [109].

C'est dans ce contexte que nous présentons, comme premier exemple, ReMMoC [120], héritier d'OpenCOM, qui appartient à la dernière génération d'intergiciels réflexifs développés à l'université de Lancaster [116]. Comme second exemple, nous introduisons, OSGi, pour deux raisons : (i) c'est un intergiciel reconfigurable provenant du milieu industriel usuellement embarqué dans divers *objets communicants* de notre quotidien [117], et (ii) c'est un intergiciel Java souvent réutilisé dans des *architectures de services ubiquitaires* comme Gravity [119] et Amigo [127].

- **ReMMoC**

Un objet communicant d'un *environnement ubiquitaire* ne sait pas à l'avance avec quels intergiciels les services distants, avec lesquels il est susceptible d'interagir, sont implémentés. Etant donné la diversité d'intergiciels existants, il est fort probable que cet *objet communicant* ait à interagir avec d'autres objets reposant sur des intergiciels incompatibles avec le sien. ReMMoC est un intergiciel réflexif conçu pour résoudre cette problématique : il est capable d'interagir avec divers intergiciels malgré l'hétérogénéité de leurs fonctions de communication et de découverte de services suivant le principe de la *substitution de protocole*. ReMMoC a la capacité de se reconfigurer dynamiquement, afin de sélectionner le protocole de découverte services, et le protocole de communication adapté suivant son environnement d'exécution.

Pour atteindre cet objectif, la structure interne de ReMMoC se décompose en deux conteneurs de composants (*composant framework*) [110] qui en fonction de leur configuration respective, permettent de gérer différents types de protocoles. Chaque *framework*, gère le cycle de vie de ses composants : leur activation, désactivation, la validité de leur assemblage/réassemblage et de leurs dépendances [126]. En d'autres termes, un *framework* contrôle l'évolution dynamique de l'intergiciel et s'assure que ce dernier reste dans un état stable quelles que soient les reconfigurations engagées.

ReMMoC se compose donc :

1. D'un premier *framework* (Figure 21A), dénommé *discovery framework*, qui rassemble les composants implémentant divers SDPs tels que SSDP, SLP.

2. D'un second *framework* (Figure 21B), appelé *binding framework*, qui regroupe, les composants implémentant différents protocoles de communication tel que IIOP, SOAP.

La reconfiguration de l'intergiciel, (c'est-à-dire, la modification de l'assemblage des composants des différents *frameworks*), est réalisée par ReMMoC suivant la détection du SDP en cours d'utilisation dans l'environnement réseau, et du protocole de communication utilisé par le ou les services distants nouvellement découverts. Le mécanisme de détection de protocoles de ReMMoC correspond à une implémentation possible du processus *Switch* de la spécification de C-SUB donnée dans la Figure 15 de la section précédente.

Techniquement, pour découvrir le SDP utilisé dans l'environnement réseau, le *discovery framework* doit disposer d'autant de composants logiciels que de protocoles de découverte de services (SDP) existants. Ainsi, par exemple, pour découvrir si UPnP/SSDP est utilisé dans l'environnement, ReMMoC doit : (i) charger et activer le composant UPnP/SSDP dans le *discovery framework*, (ii) générer une requête *multicast* de recherche de service, (iii) attendre une réponse. Si, pendant un certain laps de temps, aucune réponse ne parvient, cela signifie qu'aucun service n'annonce sa présence avec UPnP/SSDP. Pour découvrir l'ensemble des SDP utilisés dans l'environnement ambiant, ReMMoC doit réitérer ce processus en trois phases pour chaque SDP dont il possède le composant. La découverte de SDP est relativement coûteuse en termes de puissance de calcul pour charger/décharger les différents composants, de consommation de bande passante pour générer les différentes requêtes, et en temps d'attente puisqu'il faut attendre les réponses aux requêtes générées pour chaque SDP que l'on souhaite découvrir. Afin d'éviter les chargements/déchargements intempestifs des composants du *discovery framework*, la dernière version de ReMMoC utilise un composant, appelé *discover discovery*, qui vient se greffer sur le *discovery framework*. Ce composant implémente très partiellement les fonctionnalités des différents SDP. Son rôle est de générer les en-têtes des requêtes de recherche de service des différents SDP, et de les envoyer vers leurs adresses *multicasts* respectives. En retour, dès qu'il reçoit une réponse, il indique au *discovery framework* le SDP couramment utilisé afin de charger et d'activer le composant adéquat, qui est une implémentation complète du SDP découvert. Cette solution reste toutefois limitée. Elle nécessite toujours de générer un trafic additionnel pour découvrir le ou les SDP de l'environnement et elle n'améliore pas le temps d'attente nécessaire pour découvrir l'ensemble des SDP utilisés. Enfin, l'environnement réseau étant potentiellement dynamique, le ou les SDP courants peuvent changer rapidement, ce qui n'évite pas les reconfigurations successives des composants du *discovery framework*.

Enfin, ReMMoC détermine le protocole de communication des services distants grâce aux informations obtenues lors de leurs découvertes. Si le service recherché par une application cliente est présent dans l'environnement ambiant, celle-ci obtient, *via* le SDP adéquat, la description du service. A partir de cette description, ReMMoC détermine le protocole de communication en extrayant soit l'URI (*Universal Resource Identifiers*) du service, soit les attributs du service. Une URI est un schéma standardisé d'identification qui permet à la fois de localiser et de déterminer le protocole de communication du service. Cependant, étant donnée l'hétérogénéité des SDPs, l'URI n'est pas toujours la méthode utilisée pour identifier les services. Quant

aux attributs extraits de la description d'un service, il n'est pas garanti qu'ils indiquent toujours le protocole de communication de ce dernier. La détection du protocole de communication peut donc échouer.

Par ailleurs, comme modélisé par le connecteur C-SUB, et confirmé dans [120] et [119], l'utilisation d'un intergiciel dynamiquement reconfigurable n'est pas transparente pour le développeur d'applications, il doit explicitement programmer ses applications pour qu'elles prennent en charge chaque changement dynamique. Pour rappel, cette contrainte est modélisée par le fait que le port P1 d'un composant utilisant le connecteur C-SUB doit être conçu de façon à ce qu'il soit équivalent au rôle R1 de ce dernier, qui est sujet à des mutations (commutation dynamique de rôle). Par exemple, plus concrètement, si un service distant découvert est de type SOAP, l'application doit effectuer une invocation SOAP, tandis que si le service distant est de type RMI, il doit effectuer une invocation RMI. Afin de simplifier la tâche des développeurs, ReMMoC fournit une interface de programmation (API) unifiée qui lui est spécifique pour le développement de ses applications[6], comme illustré dans la Figure 21C. Ainsi, l'API ReMMoC permet de concevoir des applications capables de découvrir et de communiquer avec des services de l'environnement ambiant indépendamment de la configuration du *binding* et du *discovery framework* et donc indépendamment des protocoles utilisés.

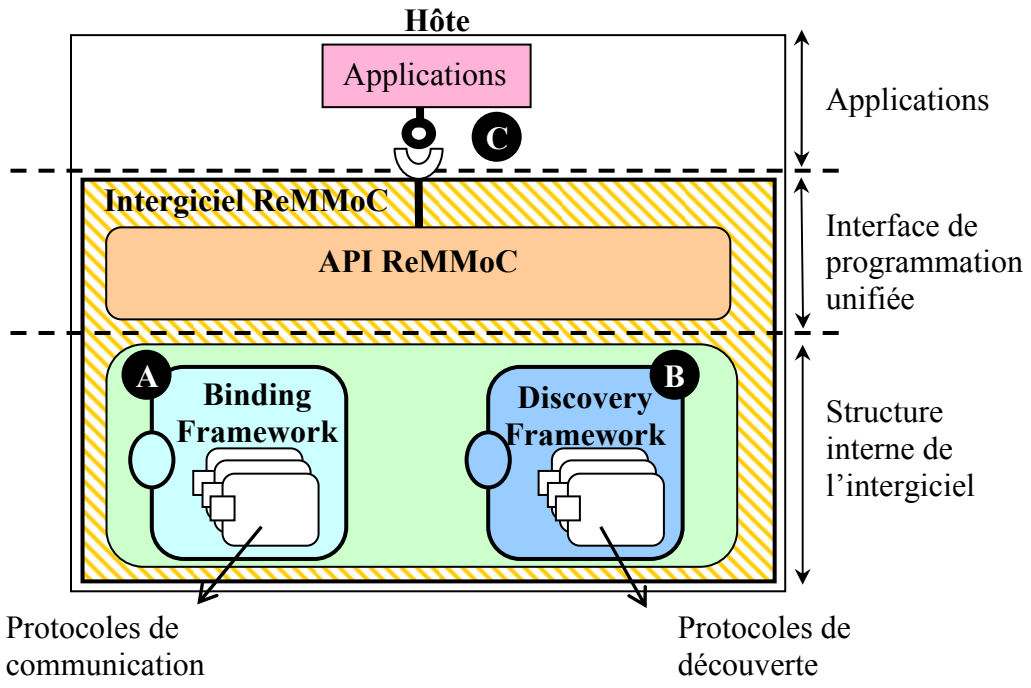


Figure 21. Structure globale de l'intergiciel ReMMoC

[6] Au niveau du modèle, cela revient à remplacer le rôle R1 par (P1 || I) avec I processus synchronisant les actions unifiées de P1 avec C-SUB.

- **OSGi**

OSGi est un consortium d'industriels formé en 1999 ayant pour objectif de définir et de promouvoir la spécification d'un intergiciel OSGi permettant le déploiement de services applicatifs, à l'origine, dans des équipements domotiques, et aujourd'hui, dans toutes sortes d'équipements, du PDA à la voiture [117]. L'intergiciel OSGi se superpose au-dessus d'une JVM (Java Virtual Machine) pour former un intergiciel Java composé d'un *framework* de composants (Figure 22A, B, C). Dans la terminologie OSGi, une application est une composition de briques logicielles appelées *bundles*, théoriquement similaires aux composants présentés précédemment. Chaque *bundle* fournit un certain nombre de fonctions (ou opérations) qui peuvent être réutilisées par d'autres *bundles*. Concrètement, un *bundle* est une archive de type JAR qui contient :

  - Une description abstraite du *bundle* (*manifest file*). Cette description décrit ses caractéristiques : ses différentes propriétés, les packages Java qu'il fournit, et/ou les packages Java dont il dépend.

  - Le code Java et les ressources associées (librairies externes, images, etc.) permettant d'instancier le *bundle*.

OSGi fournit différentes fonctions d'administration permettant d'installer, de désinstaller, d'activer, de désactiver et/ou de mettre à jour les différents *bundles* de son *framework*. Ces fonctions d'administration peuvent être utilisées : (i) directement par les *bundles* grâce à l'API OSGi (Figure 22G) ou (ii) par l'utilisateur *via* une console d'administration locale ou distante. Lorsqu'un *bundle* est installé puis activé, il peut publier ses fonctions ou découvrir des fonctions venant d'autres *bundles* grâce à un annuaire intégré à l'intergiciel. Si un *bundle* activé dépend de fonctions venant d'autres *bundles* non activés, il lui est également possible de les activer s'ils ont été préalablement installés. Par ailleurs, la publication, la découverte de fonctions et la composition de *bundles* ne peut se faire qu'entre *bundles* d'un même intergiciel : OSGi n'est pas un système distribué mais centralisé (Figure 22B, C). Toutefois, il est possible d'enrichir à la volée l'intergiciel OSGi en téléchargeant de nouveaux *bundles* à partir d'une URL (Figure 22E).

Au niveau de l'implémentation, comparativement à un intergiciel réflexif, OSGi est un intergiciel reconfigurable non dynamique [128] : la gestion du cycle de vie des *bundles* du *framework* est laissée à la charge du développeur des *bundles* et des applications. Chaque application doit gérer elle-même, l'activation/désactivation, le contrôle et la gestion des dépendances des *bundles* qu'elle utilise (Figure 22F, G). Le projet Gravity [119] dote l'intergiciel OSGi d'un mécanisme, dénommé *ServiceBinder*, automatisant la gestion du cycle de vie des *bundles*, proche de celui employé dans les intergiciels réflexifs. Le *ServiceBinder* analyse automatiquement les dépendances des *bundles* utilisés afin de : (i) activer/désactiver en conséquence

les *bundles* dont ils dépendent, (ii) enregistrer/désenregistrer les fonctions qu'ils fournissent auprès de l'annuaire du *framework*.

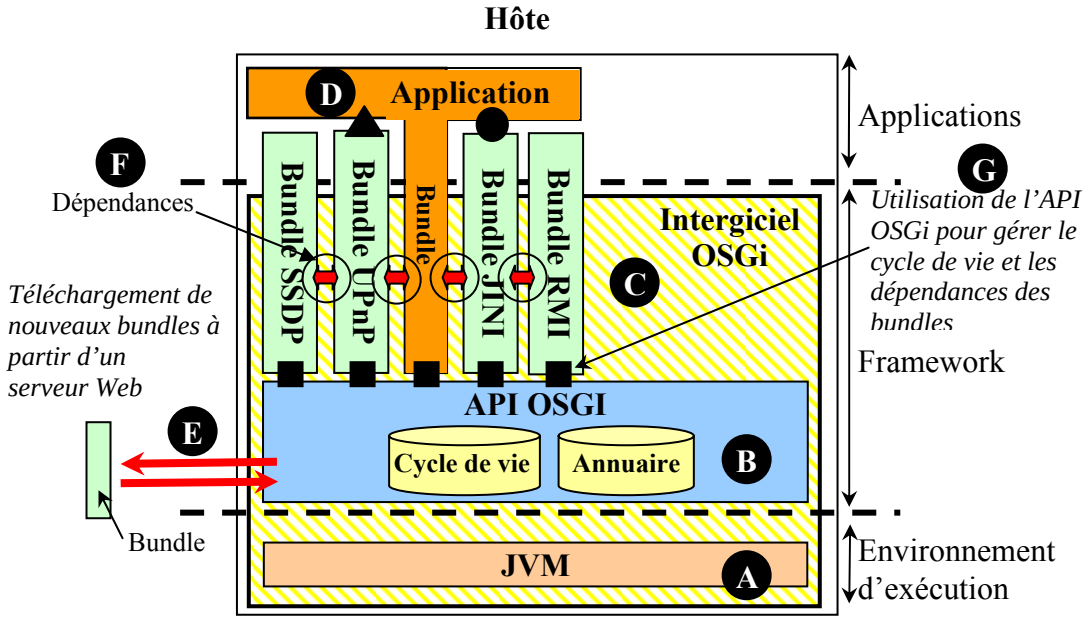


Figure 22. Architecture de l'intergiciel OSGi

En ce qui concerne les protocoles de communication et de découverte de services pris en charge par OSGi, cela dépend des *bundles* installés. Par ailleurs, OSGI n'intègre pas, par défaut, un mécanisme permettant de détecter les protocoles en cours d'utilisation dans l'*environnement ubiquitaire* afin d'activer en conséquence les *bundles* adéquats. L'implémentation de cette fonctionnalité est laissée à la charge du développeur de l'application.

Le projet européen Amigo [127], dont l'objectif est de promouvoir la spécification d'un intergiciel adapté aux *objets communicants* d'un *environnement ubiquitaire*, propose, entre autres, une extension de OSGi, qui couplée au *serviceBinder*, permet l'intégration de la détection de protocoles et de la sélection de *bundle* au niveau de l'intergiciel plutôt qu'au niveau applicatif. L'objectif de cette extension est de concevoir des applications, pour des *objets communicants* équipés d'OSGi, qui profitent automatiquement des capacités de l'intergiciel à opérer une *substitution de protocoles* suivant son contexte d'exécution. Ainsi, contrairement à la Figure 22D, où les applications font référence explicitement aux *bundles* correspondants aux différents protocoles pris en charge, les applications doivent être liées à deux *bundles* génériques :

- **Un *bundle* dédié à la découverte de service.** Ce dernier fournit une interface générique de découverte de service. Ce *bundle* gère, en interne, de l'activation/désactivation automatique d'autres *bundles*

implémentant chacun d'entre eux un *protocole de découverte de services* différents. Dès qu'un service est découvert, un *bundle* générique le représentant est activé.

- **Un *bundle* générique représentant un service.** Chaque service découvert a une représentation locale au sein de l'intergiciel sous forme de *bundle* générique. Ce *bundle* s'apparente à une sorte de talon donnant l'impression que le service distant est un *bundle* local. Ce dernier fournit une interface générique d'invocation de service à distance et se charge d'activer le *bundle* implémentant le protocole de communication adéquat pour interagir avec le service distant.

Le *framework* OSGi contient finalement deux catégories de *bundles,* l'une dédié à la découverte de service (équivalente au concept du *discovery framework* de ReMMoC) et l'autre dédiée à la communication (équivalente au concept du *communication framework* de ReMMoC). Ainsi, OSGi peut être considéré comme l'alter ego de ReMMoC dédié à la plateforme Java.

Quelles que soient leurs facultés d'adaptation, les intergiciels adaptatifs nécessitent à la fois de concevoir des composants qui leur sont propres et de redévelopper intégralement les applications pour profiter de leurs fonctions d'adaptation.

3.4.2 Implémentation de la traduction de protocoles

Une des parties essentielle de la spécification du connecteur C-TRAD (Figure 17) repose sur les processus ⊤||*Bridge* de la *glue* qui permettent de faire la jonction entre deux intergiciels hétérogènes. L'implémentation de ces processus correspond à l'implémentation de ponts (*bridges*) ou d'adaptateurs (*wrappers*) logiciels. Il existe de très nombreux ponts conçus pour interconnecter, par exemple, des intergiciels commerciaux bien établis comme SOAP2CORBA [129], IIOP-.NET [132], DCOM-CORBA [130], RMI-IIOP [131]. L'inconvénient majeur de ces ponts est de ne permettre la *traduction de protocoles* uniquement entre deux intergiciels hétérogènes. Ainsi, à plus grande échelle, n intergiciels hétérogènes déployés dans un *environnement ubiquitaire* requiert l'utilisation de n^2 ponts pour assurer leur interopérabilité. Une alternative plus efficace consiste à utiliser un intergiciel comme intermédiaire afin d'éviter n^2 conversions. Cette approche est typiquement celle des EAIs [144] (Enterprise Application Integration) qui utilisent des ponts (ou adaptateurs suivant la terminologie adaptée) pour traduire le trafic des n intergiciels vers un protocole intermédiaire (ou canonique), réduisant ainsi le nombre de ponts nécessaires à n. Concrètement, un EAI est un MOM architecturé autour d'un serveur (*message broker*) faisant office de mandataire grâce auquel des intergiciels *incompatibles* interagissent à l'aide d'un adaptateur adéquat (Figure 23). Le *message broker* véhicule les messages échangés entre les différents adaptateurs selon un protocole canonique propre à l'EAI. Ainsi, un EAI est un *routeur multi-intergiciel* qui prend en charge l'interopérabilité entre des intergiciels hétérogènes [136].

Selon Vinoski [136], l'approche empruntée par les EAIs présente deux inconvénients : (i) les performances de toutes les interactions se retrouvent pénalisées par la traduction systématique des protocoles, et (ii) le protocole canonique est propriétaire, spécifique à un EAI, et est sujet à des évolutions fréquentes, ne réduisant pas ainsi véritablement le problème des n^2 conversions. La première contrainte est inhérente à la *traduction de protocoles*, quant à la seconde, elle peut être résolue en ayant recours aux services Web afin de standardiser les EAIs [136].

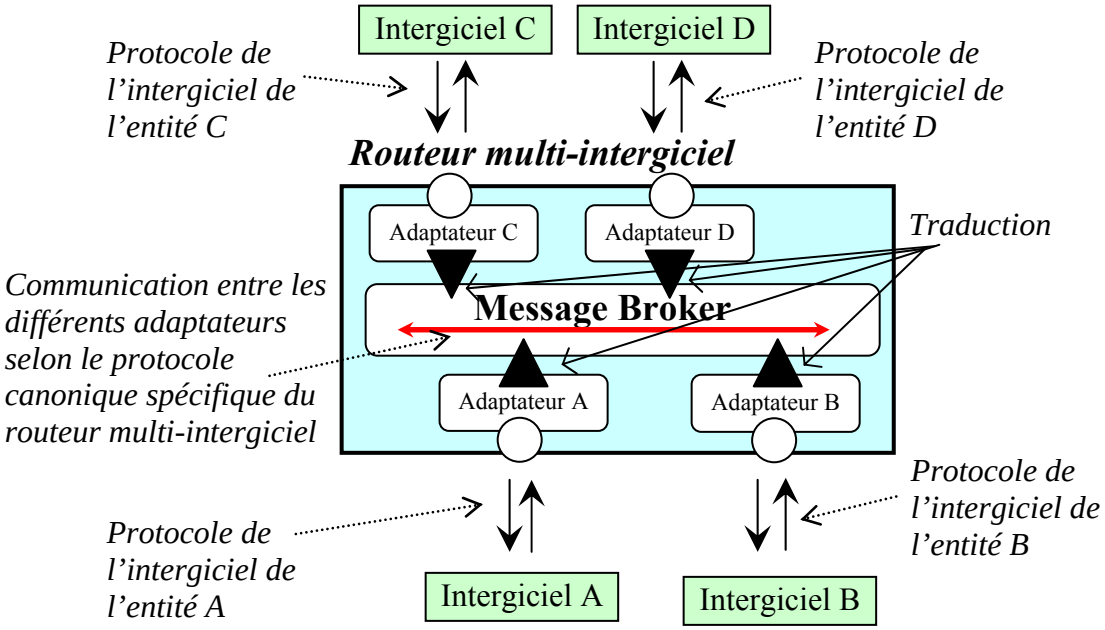


Figure 23. Architecture d'un EAI

Il existe deux visions différentes des services Web [135] : ils peuvent être considérés soit comme une nouvelle façon de concevoir un intergiciel, soit comme une évolution des EAIs, appelés alors des ESBs (*Enterprise Service Bus*) [137]. Dans le Chapitre 2, nous avons présenté les services Web selon la première tendance, dorénavant nous introduisons maintenant les services Web selon la seconde tendance.

Pour rappel, les services Web sont fondés sur le triptyque SOAP, UDDI, WSDL. Bien que WSDL ait un rôle similaire aux IDLs des intergiciels traditionnels, il diffère de ces derniers : WSDL spécifie, en plus des opérations offertes par un service, les protocoles permettant l'accès au service qu'il décrit [135]. WSDL n'est lié à aucun intergiciel spécifique contrairement aux IDLs traditionnels qui n'incluent pas ce genre d'information car le protocole employé est implicite puisque c'est celui de l'intergiciel pour lequel l'IDL est conçu. Dans le contexte des services Web, on peut accéder à chaque service *via* différents protocoles et il est donc nécessaire que cette information fasse partie de la description d'un service. Un document WSDL se

décompose globalement en deux parties majeures : une partie, qualifiée d'abstraite, analogue à celles des IDLs traditionnels et une seconde partie, concrète, qui spécifie les protocoles d'accès du service (*service bindings*). La structure d'un ESB est similaire à celle d'un EAI : il s'agit d'un *routeur multi-intergiciel* dont les adaptateurs sont générés à partir d'un document WSDL et dont le protocole canonique est SOAP.

Lorsque des clients et des services dépendants d'intergiciels hétérogènes existent déjà, l'intérêt d'un *routeur multi-intergiciel* devient déterminant [136]. Toutefois, l'interopérabilité reste statique au sens où elle est planifiée à l'avance : elle exige l'intervention d'un administrateur qui doit rédiger le document WSDL des services existants en fonction des adaptateurs disponibles pris en charge par le routeur utilisé. Par ailleurs, l'interopérabilité entre intergiciels hétérogènes dans une *architecture de services ubiquitaires* dépend de la présence ou non d'un *routeur multi-intergiciel*. Or, il n'est pas raisonnable de considérer qu'un tel routeur soit systématiquement déployé quel que soit l'*environnement ubiquitaire* traversé par les objets communicants. En effet, l'utilisation d'un routeur présuppose l'existence d'une infrastructure réseau minimale qui n'est pas systématique dans le domaine de l'informatique diffuse. Une alternative à cette problématique est que les objets communiquants gèrent eux-mêmes *l'interopérabilité* à travers leur propre intergiciel. Cela revient à concevoir des intergiciels répondant au principe de la *substitution de protocoles* [138], appelé également *middleware switching* [136].

3.5 Synthèse

Dans ce chapitre, nous avons identifié et modélisé deux concepts permettant de résoudre l'hétérogénéité des intergiciels : la *substitution* et la *traduction de protocoles*. L'avantage des intergiciels conçus suivant la *substitution de protocoles* réside dans leur capacité à s'adapter en fonction de leur environnement d'exécution, ce qui résout dynamiquement l'hétérogénéité des intergiciels. En revanche, leur inconvénient est de fournir une interopérabilité non transparente étant donné la nécessité de redévelopper leurs applications. Implicitement, une nouvelle source d'hétérogénéité se crée et concerne cette fois-ci les intergiciels *interopérables*. A l'inverse, la mise en œuvre du principe de la *traduction de protocoles* ne requiert ni la conception de nouveaux intergiciels, ni la modification des applications existantes : l'interopérabilité est transparente. Mais, elle reste statique au sens où elle est planifiée. Il apparaît donc nécessaire d'optimiser la résolution de l'hétérogénéité des intergiciels en combinant les deux principes de *substitution* et de *traduction de protocoles*. On peut noter que le produit IONA Artix [139] présente une certaine avancée dans cette perspective. En effet, ce produit se caractèrise par une plateforme composée, d'une part, d'un *routeur multi-intergiciel* qui gère l'interopérabilité entre les applications déjà existantes et, d'autre part, d'un intergiciel adaptif pour les nouvelles applications afin qu'elles soient nativement interopérables. Ce qui fait qu'Artix constitue une solution permettant alternativement la *substitution* ou la *traduction de protocoles* mais jamais simultanément. Notre contribution dans ce

document a pour objet de fusionner les deux approches afin, suivant le point de vue choisi, de rendre transparente la *substitution de protocoles* ou de rendre la *traduction de protocoles* dynamique. L'idée est de fusionner les avantages de la *substitution* et de la *traduction de protocoles* sans les inconvénients.

4 Traduction dynamique de protocoles

Dans un *environnement ubiquitaire*, la coopération entre *objets communicants* doit être dynamique et spontanée dans le sens où : (i) elle ne peut être prévue à l'avance, (ii) elle doit s'adapter aux changements qui peuvent survenir dans l'environnement de l'utilisateur par l'apparition ou la disparition d'objets, (iii) elle ne doit pas nécessiter d'intervention et être transparent pour l'utilisateur. Cette coopération est possible si ces objets sont capables d'interagir indépendamment de l'hétérogénéité de leur intergiciel respectif. Au niveau du réseau, l'hétérogénéité des intergiciels se manifeste par l'incompatibilité de leurs protocoles d'interaction (comme les protocoles d'accès et de découverte de services). Comme présenté dans le chapitre précédent, l'aspect dynamique de la *substitution de protocoles* et la transparence de la *traduction de protocoles* sont deux fonctionnalités à combiner afin de résoudre l'hétérogénéité des intergiciels d'un *environnement ubiquitaire*.

Dans ce chapitre, à partir des spécifications de la *substitution* et de la *traduction de protocoles*, nous introduisons les principes de la *traduction dynamique de protocoles* que nous modélisons sous la forme d'un nouveau connecteur. Dans la section 4.1, nous proposons une spécification formelle de ce dernier à l'aide de l'algèbre de processus FSP. Par la suite, dans la section 4.2, nous évaluons qualitativement les limites et les contraintes de notre solution. Dans la section 4.3, nous présentons l'application des principes de *la traduction dynamique de protocoles* aux protocoles réseaux. Enfin, dans la section 4.4, nous proposons l'architecture logicielle d'un système implémentant le connecteur modélisant notre solution.

4.1 Principe et formalisation

Le principe de la *traduction dynamique de protocoles* consiste à adapter dynamiquement les rôles d'un connecteur de façon à les rendre compatibles avec les ports des composants à interconnecter. Dans une *architecture de services ubiquitaires,* un client interagit usuellement avec un service, et *vice versa*, *via* un seul protocole d'interaction, (celui correspondant à leur intergiciel ubiquitaire respectif). Par conséquent, dans la formalisation que nous proposons ci-après, les connecteurs disposent seulement de deux rôles et les composants d'un unique port. Toutefois notre modélisation s'adapte aisément au cas le plus général où un composant est doté de plusieurs ports (modélisant ainsi le fait qu'un client a la capacité d'utiliser alternativement ou simultanément plusieurs protocoles d'interactions distincts). En effet, dans ce contexte, selon notre modélisation, chaque port d'un tel composant est assemblé à un connecteur distinct, plutôt qu'à un unique connecteur ayant autant de rôles distincts qu'il existe de ports.

Ainsi, comme illustré dans la Figure 24, si le port d'un composant C1 est compatible avec le rôle R1*i* d'un connecteur *i* avec $i \in [1,.., n]$ (Figure 24A) et que le port d'un

composant C2 est compatible avec le rôle R2*k* d'un connecteur *k* avec $k \in [1,.., m]$ avec $i \neq k$ (Figure 24B), il faut adapter dynamiquement :

- Soit le rôle R2*i* du connecteur *i* pour qu'il se comporte comme le rôle R2*k* du connecteur *k* (Figure 24C),
- Soit le rôle R1*k* du connecteur *k* pour qu'il se comporte comme le rôle R1*i* du connecteur *i* (Figure 24D).

L'adaptation du rôle R2*i* ou R1*k* respectivement des connecteurs *i* ou *k* aboutit dans les deux cas à un nouveau connecteur dont les rôles sont R1*i* et R2*k* (Figure 24E). Ce nouveau connecteur correspond à une fusion des connecteurs *i* et *k*.

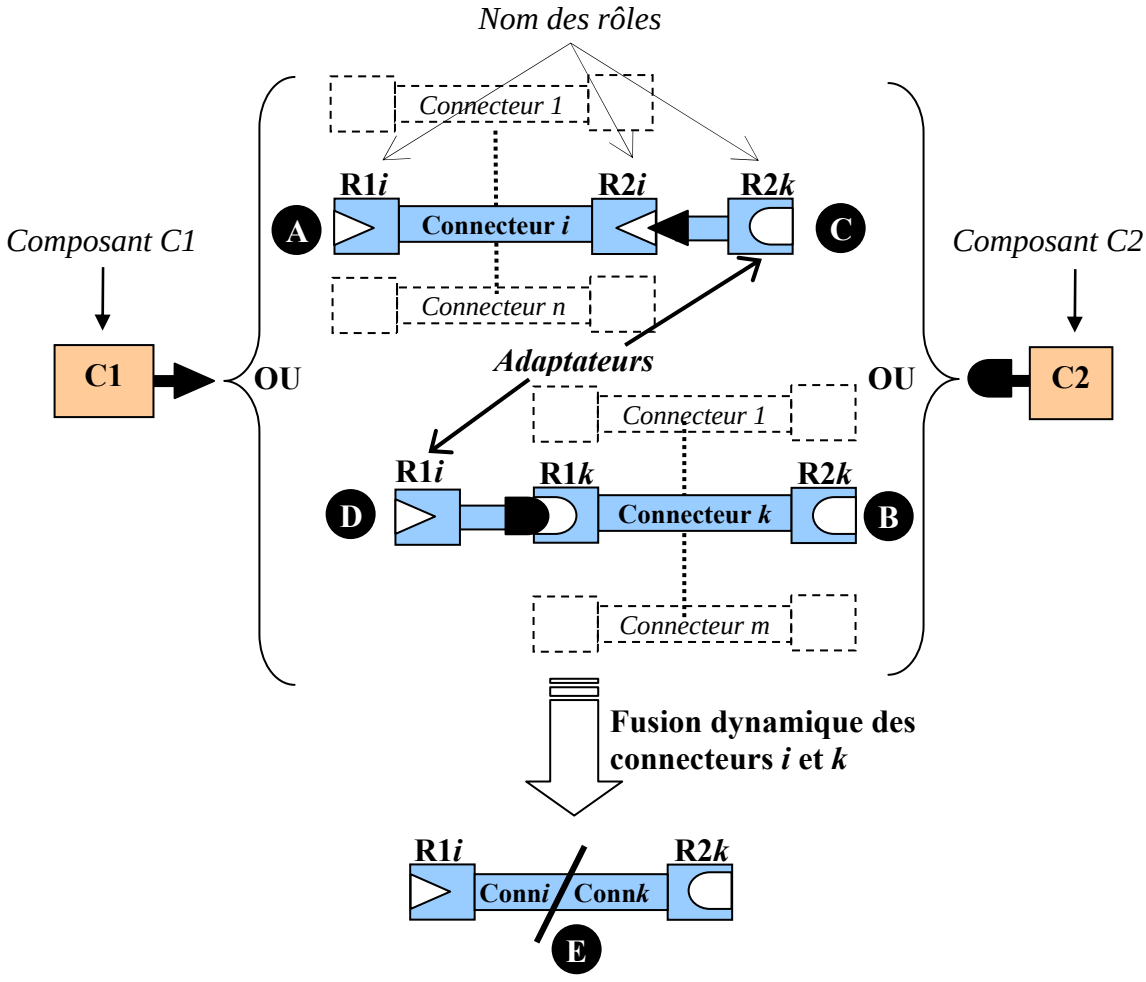


Figure 24. Principe de la traduction dynamique de protocoles

Ainsi, la *traduction dynamique de protocoles* se modélise *via* un nouveau connecteur (Figure 25A), dénommé C-UNIV, qui fusionne dynamiquement deux connecteurs

distincts parmi l'ensemble des connecteurs existants en fonction des ports des deux composants à interconnecter. De ce fait, le comportement des rôles R1 et R2 de C-UNIV varie, de manière analogue au connecteur C-SUB (*cf* chapitre précédent), selon la spécificité des ports des composants (Figure 25 B, C). Toutefois C-UNIV et C-SUB diffèrent.

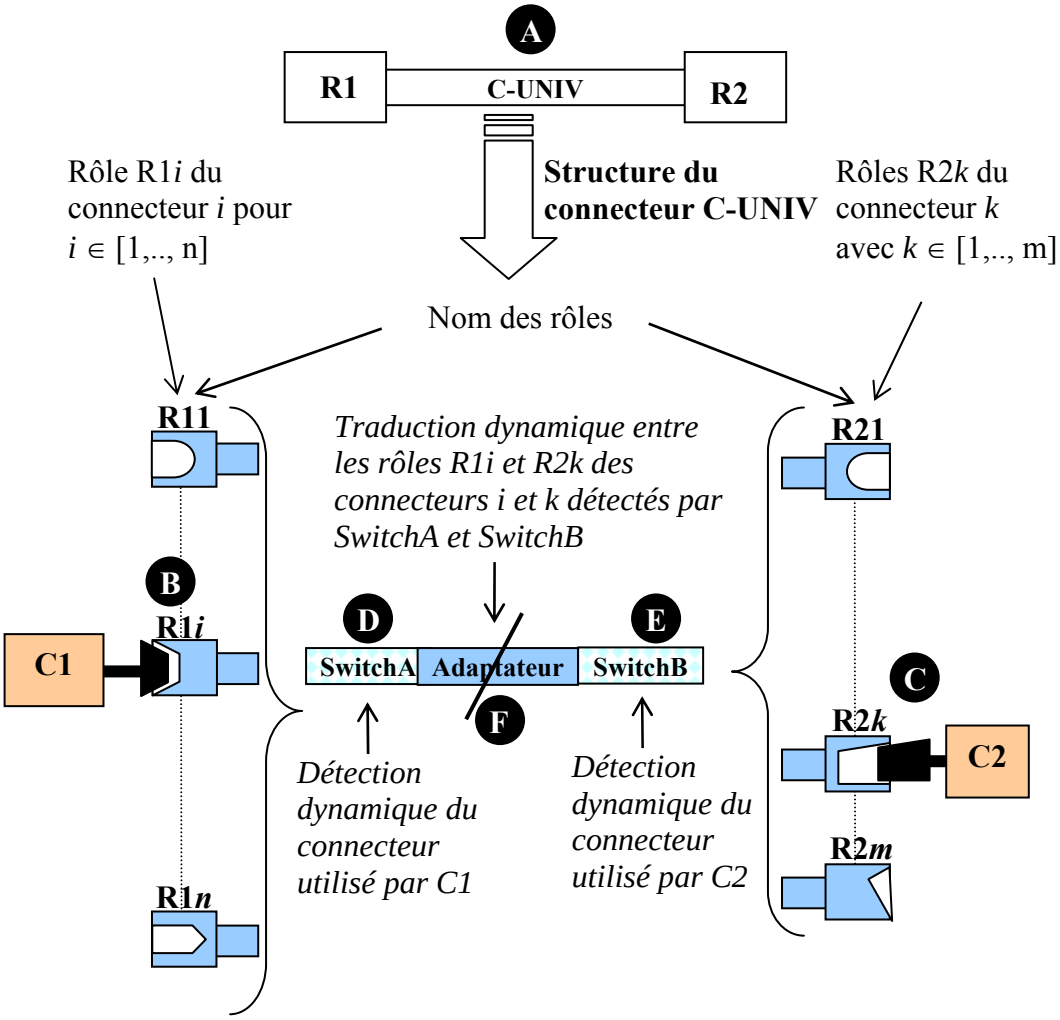


Figure 25. Connecteur modélisant la traduction dynamique de protocoles

En effet, pour interconnecter deux composants C1 et C2 :

- **C-SUB** sélectionne un connecteur suivant le port du composant C2 tout en nécessitant une altération du port du composant C1 afin qu'il soit compatible avec les rôles du connecteur dynamiquement sélectionné. Le changement de comportement des rôles de C-SUB n'est pas transparent pour l'un des deux composants.

- **C-UNIV** détecte, quant à lui, les deux connecteurs utilisés respectivement par les composants C1 et C2 (Figure 25 D, E) afin de les combiner *via* un adaptateur adéquat (Figure 25F). Ainsi, les rôles de C-UNIV sont compatibles avec les ports des deux composants à interconnecter sans susciter leur altération (Figure 25 B, C).

Par conséquent, contrairement à C-SUB, C-UNIV dispose de deux rôles dont le comportement respectif est déterminé dynamiquement selon la spécificité des ports des composants à interconnecter. En d'autres termes, le changement dynamique du comportement des rôles du connecteur C-UNIV est transparent pour les deux composants à interconnecter. La *traduction dynamique de protocoles* est réalisée par l'adaptateur de C-UNIV qui résout l'incompatibilité des protocoles des rôles des connecteurs dynamiquement fusionnés (Figure 25F). De façon similaire au connecteur C-TRAD, présenté dans le chapitre précédent, l'adaptateur effectue une opération de jointure qui permet de faire la jonction entre les différents rôles de C-UNIV en plus des opérations de transformation suivant la spécificité de ces derniers. La fusion de n connecteurs avec m connecteurs requiert n x m opérations de transformation pour résoudre les incompatibilités des $n + m$ protocoles. Afin d'éviter n x m opérations, la traduction réalisée par C-UNIV se fait *via* un protocole intermédiaire. C-UNIV applique le principe de la projection de protocoles [142], [143] de façon à projeter les protocoles incompatibles en provenance de ses rôles sur un *protocole image*, *via* une fonction de projection *f*, afin de résoudre leurs incompatibilités. En d'autres termes, le *protocole image* établit dynamiquement une correspondance sémantique entre les protocoles en cours d'utilisation par les rôles de C-UNIV. Plus précisément, contrairement au principe du protocole intermédiaire tel que défini, par exemple, par les EAIs, le *protocole image* dépend des similitudes[7] partagées à un instant donné par les protocoles incompatibles des rôles R1 et R2, le protocole de ces derniers variant suivant les ports des composants à interconnecter.

Détaillons la spécification formelle de C-UNIV permettant de *fusionner* un connecteur i et un connecteur k, avec $i \in [1,.., n]$, $k \in [1,.., m]$ et $i \neq k$. Tout d'abord, la *glue* de C-UNIV est composée des $n + m$ *glues* des $n + m$ connecteurs pris en charge par C-UNIV. Parmi ces $n + m$ *glues*, seules les deux *glues* correspondant aux deux connecteurs fusionnés à un instant donné doivent se synchroniser (ou être actives). Ainsi, afin que les $n + m$ *glues* de C-UNIV ne se synchronisent pas directement, on marque l'alphabet de chacune des *glues* par un identifiant unique (une étiquette) afin de s'assurer qu'elles ne partagent aucun évènement en commun. Cet identifiant est préfixé par *a* pour chaque *glue* provenant du connecteur i pour tout $i \in [1,.., n]$ et par *b* pour chaque *glue* provenant du connecteur k pour tout $k \in [1,.., m]$. Formellement, on obtient alors :

$$\|\mathbf{C\text{-}UNIV} = \|_{i \in [1,..,n]}\ \mathbf{a.tag}i\ \mathbf{:Glue}i\ \|_{k \in [1,..,m]}\ \mathbf{b.tag}k\ \mathbf{:Glue}k$$

[7] Deux protocoles incompatibles partagent des similitudes si les processus les représentant ont des évènements de leur alphabet ayant une sémantique identique et un certains nombre de comportements compatibles.

La sélection des deux *glues* à synchroniser est réalisée par deux processus distincts, nommés *SwitchA* et *SwitchB*. L'un et l'autre émettent un évènement *a.gluei* et *b.gluek* indiquant aux autres processus de la *glue* de C-UNIV de se synchroniser avec les *glues* des connecteurs *i* et *k*. La *glue* de C-UNIV devient :

||C-UNIV= SwitchA $||_{i \in [1,..,n]}$ **a.tag*i* :Glue*i*** || **SwitchB** $||_{k \in [1,..,m]}$ **b.tag*k* :Glue*k***

Avec :

SwitchA = (electionA → a.reset → **SwitchA**
$|_{i \in [1,..,n]}$ electionA → a.glue***i*** → **SwitchA)** \ {electionA}

SwitchB = (electionB → b.reset → **SwitchB**
$|_{k \in [1,..,m]}$ electionB → b.glue***k*** → **SwitchB)** \ {electionA}

La stratégie de sélection des deux *glues* est propre aux processus *SwitchA* et *SwitchB* et tous deux sont libres de réitérer une sélection de *glue*, de façon indépendante, suite à l'émission d'un évènement *a.reset* et/ou *b.reset* pour réinitialiser les autres processus de C-UNIV. En effet, les évènements *electionA* et *electionB* sont des évènements internes non observables respectivement des processus *SwitchA* et *SwitchB*.

La spécification des rôles R1 et R2 de C-UNIV inclut respectivement la spécification des rôles R1*i* et R2*k* des connecteurs *i* et *k* avec $i \in [1,..,n]$ et $k \in [1,..,m]$. Les processus $R1i_{\ i \in [1,..,n]}$ deviennent des processus locaux de R1 et les processus $R2k_{\ k \in [1,..,m]}$ deviennent des processus locaux de R2. Le comportement de R1 et R2, c'est-à-dire la sélection/activation d'un de leurs processus locaux, est déterminé en fonction des évènements *a.gluei* et *b.gluek* engagés respectivement par les processus *SwitchA* et *SwitchB*. Les rôles R1 et R2 peuvent donc être spécifiés formellement ainsi :

Role R1 = $|_{i \in [1,..,n]}$ (a.glue***i*** → R1***i***),
R1***i*** $_{i \in [1,..,n]}$ = *spécification initiale du rôle R1**i** tel que donnée par le connecteur i* | a.reset → R1,

Role R2 = $|_{k \in [1,..,m]}$ (b.glue***k*** → R2***k***),
R2***k*** $_{k \in [1,..,m]}$ = *spécification initiale du rôle R2**k** tel que donnée par le connecteur k* | b.reset → R2,

Les spécifications initiales des rôles $R1i_{\ i \in [1,..,n]}$ et $R2k_{\ k \in [1,..,m]}$ sont enrichies de telle sorte qu'elles prennent en considération les évènements *a.reset* et *b.reset*. Ces deux évènements autorisent la réinitialisation des rôles R1 et R2 de telle sorte que ces derniers puissent changer leur comportement, c'est-à-dire réitérer une nouvelle sélection/activation d'un de leurs processus locaux suivant les évènements engagés

par les processus *SwitchA* et *SwitchB*. La sélection du comportement des rôles R1 et R2 se faisant en concordance avec celle des *glues*, il apparaît naturellement que la détermination des connecteurs à *fusionner* est réalisée par les processus *SwitchA* et *SwitchB*.

La *glue* de C-UNIV devient :

|| **C-UNIV=**
R1|| SwitchA (||$_{\mathbf{i} \in [1,..,\ n]}$ a.tag***i*** :Glue***i***)|| SwitchB (||$_{\mathbf{k} \in [1,..,\ m]}$ b.tag***k*** :Glue***k***)||**R2**

Les alphabets du processus local de R1, du processus local de R2, ainsi que celui des deux *glues* correspondant aux connecteurs *i* et *k,* sélectionnés par *SwitchA* et *SwitchB,* sont disjoints. Par conséquent, selon le même principe que C-SUB, il est nécessaire de composer R1 et R2 respectivement avec des processus W_1 et W_2[8] afin qu'ils se synchronisent avec les *glues* correspondant aux connecteurs à *fusionner*. Plus précisément, le processus W_1 (resp. W_2) assure deux fonctions :

1. Il redirige les évènements initiés par le processus local R1*i* (resp. R2*k*) sélectionné par R1 (resp. R2) vers la *glue* du connecteur *i* (resp. du connecteur *k*).

2. Il redirige les évènements observés en provenance de la *glue* du connecteur *i* (resp. du connecteur *k*) vers le processus local R1*i* de R1 (resp. R2*k* de R2).

On définit les ensembles I1*i* $_{i\in[1,..,n]}$ (resp. I2*k* $_{k\in[1,..,m]}$) correspondants aux évènements initiés (ou émis) par les processus R1*i* $_{i\in[1,..,n]}$ (resp. R2*k* $_{k\in[1,..,m]}$) et les ensembles O1*i* $_{i\in[1,..,n]}$ (resp. O2*k* $_{k\in[1,..,m]}$) regroupant les évènements observés (ou reçus) par les processus R1*i* $_{i\in[1,..,n]}$ (resp. R2*k* $_{k\in[1,..,m]}$). La spécification formelle de W_1 et W_2 se notent alors :

$W_1 = |_{\mathbf{i} \in [1,..,\ n]}$ (a.glue***i*** → ToGlue***i***),

ToGlue***i*** $_{i\in[1,..n]}$ = [e:I1***i***] → a.tag***i***.e → ToGlue***i***
| a.tag***i***. [e:O1***i***] → [e] → ToGlue***i***
| a.reset → W_1,

$W_2 = |_{\mathbf{k} \in [1,..,\ m]}$ (b.glue***k*** → ToGlue***k***),

ToGlue***k*** $_{k\in[1,..,m]}$ = b.tag***k***. [e:O2***k***]→ [e] → ToGlue***k***
| [e:I2***k***] → b.tag***k***.[e] → ToGlue***k***
| b.reset → W_2,

[8] Les processus sont dénotés W pour signifier *Wrapper*.

La spécification de la *glue* de C-UNIV devient :

$$
\begin{aligned}
&\| \textbf{ C-UNIV=} \\
&\quad \text{R1} \| \boldsymbol{W_1} \\
&\qquad \| \text{ SwitchA } (\|_{\mathbf{i} \in [1,..,n]} \text{ a.tag}\boldsymbol{i} \text{ :Glue}\boldsymbol{i}) \\
&\qquad \| \text{ SwitchB } (\|_{\mathbf{k} \in [1,..,m]} \text{ b.tag}\boldsymbol{k} \text{ :Glue}\boldsymbol{k}) \\
&\quad \| \boldsymbol{W_2} \| \text{R2}
\end{aligned}
\qquad \textbf{(1)}
$$

L'équation (1) révèle sans ambiguité l'héritage de certains concepts de C-SUB par C-UNIV. Toutefois, la différence majeure entre C-SUB et C-UNIV réside dans le fait que C-UNIV sélectionne simultanément, de façon décorrélée, deux connecteurs distincts à *fusionner*. De plus, les incompatibilités de protocoles de ces derniers ne sont pas encore résolues par la *glue* définie par l'équation (1). Les protocoles des rôles de C-UNIV, comme indiqué précédemment, doivent être projetés sur un *protocole image* commun, grâce à une fonction de projection, afin de résoudre leurs incompatibilités. Nous formalisons ci-après la projection de protocoles.

Le *protocole image* commun est formalisé par un processus dont :

- **L'alphabet** correspond à l'intersection de l'ensemble des évènements des alphabets αGluei' et αGluek' images de αGlue$i$ et αGlue$k$ *via* une fonction de projection *f* bijective. Cette fonction établit une correspondance sémantique entre les évènements des alphabets des différents processus Glue$i_{\ i\in[1,..,n]}$ et Glue$k_{\ k\in[1,..,m]}$: les évènements de ces processus sont sémantiquement équivalents dès lors que leurs images à travers *f* sont identiques.

- **Le comportement** correspond à celui résultant de la composition des deux processus Glue$i$ et Glue$k$.

Ainsi, les évènements initiés par les processus R1$i_{\ i\in[1,..,n]}$ et R2$k_{\ k\in[1,..,m]}$ à destination de leurs *glues* respectives sont projetés suivant une fonction de projection *f*. Inversement, les évènements observés par les processus R1$i_{\ i\in[1,..,n]}$ et R2$k_{\ k\in[1,..,m]}$ en provenance de leurs *glues* respectives sont projetés *via* la fonction f^{-1}, fonction inverse de *f*. La projection est réalisée à l'aide de deux processus distincts $\boldsymbol{M}_1$ et $\boldsymbol{M}_2$[9] qui appliquent la fonction *f* sur les processus R1$i_{\ i\in[1,..,n]}$ et R2$k_{\ k\in[1,..,m]}$ ainsi que sa fonction inverse f^{-1} afin d'assurer une projection bidirectionnelle d'évènements entre les rôles et les *glues* de C-UNIV. Par ailleurs, les processus R1i et R2k étant sélectionnés de façon décorrélée, les processus $\boldsymbol{M}_1$ et $\boldsymbol{M}_2$ ne connaissent pas à l'avance quels vont être les évènements observés par R1i et R2k. Par conséquent, nous sommes amenés à définir l'ensemble d'évènements $\sum_{O1n} = \cup_{\ \mathbf{i} \in [1,..,n]}$ O1$\boldsymbol{i}$ et

[9] Les processus sont dénotés $\boldsymbol{M}$ pour signifier *Mapping*.

l'ensemble $\sum_{O2m} = \cup_{\mathbf{k} \in [1,..,\ m]}$ O2$\boldsymbol{k}$ regroupant l'ensemble des évènements susceptibles d'être observés et projetés par la fonction de projection f^{-1} appliquée par les processus M_1 et M_2. De plus, sachant que les évènements projetés (ou *évènements images*) par *f* sont préfixés par le mot clé *map* pour les distinguer des évènements non projetés, on déduit de ce qui précède la spécification formelle de M_1 et M_2 de la façon suivante :

$M_1 = |_{\mathbf{i} \in [1,..,\ n]}$ (a.glue$\boldsymbol{i}$ → ToMap$\boldsymbol{i}$),

ToMap$\boldsymbol{i}_{\ i \in [1,..n]}$ = a.tag$\boldsymbol{i}$.[e: I1$\boldsymbol{i}$] → a.tag$\boldsymbol{i}$.map.[e] → ToMap$\boldsymbol{i}$
| a.tag$\boldsymbol{i}$.map.[e: $\sum_{O1n}$] → a.tag$\boldsymbol{i}$.[e:O1$\boldsymbol{i}$] → ToMap$\boldsymbol{i}$
| a.reset → M_1,

$M_2 = |_{\mathbf{k} \in [1,..,\ m]}$ (b.glue$\boldsymbol{k}$ → ToMap$\boldsymbol{k}$),

ToMap$\boldsymbol{k}_{\ k \in [1,..,m\ ]}$ = b.tag$\boldsymbol{k}$.map.[e: $\sum_{O2m}$] → b.tag$\boldsymbol{k}$.[e:O2$\boldsymbol{k}$] → ToMap$\boldsymbol{k}$
| b.tag$\boldsymbol{k}$. [e:I2$\boldsymbol{k}$] → b.tag$\boldsymbol{k}$.map.[e] → ToMap$\boldsymbol{k}$
| b.reset → M_2,

La projection altère également l'alphabet des *glues* qui doivent se synchroniser aux processus M_1 et M_2, par l'intermédiaire des évènements projetés *via f*, appelés aussi *évènements images,* en provenance des rôles R1 et R2. Ainsi, l'alphabet de chacune des *glues* est remplacé par leur image à travers *f*. Ce remplacement se traduit par un renommage de l'alphabet des *n + m glues* en préfixant leur alphabet respectif par le mot clé *map* pour les distinguer des évènements non projetés par *f.* Par conséquent, on obtient la spécification de la *glue* de C-UNIV suivante :

$$
\begin{aligned}
\|\textbf{C-UNIV} = R1 \ \|\ W_1 \ \|\ \boldsymbol{M_1} \\
&\|\ \text{SwitchA} \ (\|_{\mathbf{i} \in [1,..,\ n]}\ \text{a.tag}\boldsymbol{i}\ \text{:Glue}\boldsymbol{i}\ /\ \{\text{map.[r : } \alpha\text{Glue}\boldsymbol{i}\text{]/[r]}\}) \\
&\|\ \text{SwitchB} \ (\|_{\mathbf{k} \in [1,..,\ m]}\ \text{b.tag}\boldsymbol{k}\ \text{:Glue}\boldsymbol{k}\ /\ \{\text{map.[r : } \alpha\text{Glue}\boldsymbol{k}\text{]/[r]}\}) \\
\|\ W_2 \ \|\ \boldsymbol{M_2} \ \|\ R2
\end{aligned}
\qquad \textbf{(2)}
$$

L'équation (2) n'est pas encore fonctionnelle car les *évènements images* sont marqués par un préfixe différent *a* ou *b* suivant qu'ils proviennent du rôle R1 ou R2. Par conséquent, de façon similaire au connecteur C-TRAD, présenté dans le chapitre précédent, il est nécessaire d'avoir recours à une opération de jointure afin d'assurer la redirection des évènements issus de R1 à destination de R2 et *vice versa*. La particularité de C-UNIV par rapport à C-TRAD tient au fait que l'opération de jointure se fait dynamiquement entre les deux connecteurs sélectionnés suivant les ports des composants à interconnecter. Cette opération de jointure est alors réalisée à l'aide de deux processus distincts *Bridge1* et *Bridge2* qui redirigent respectivement les *évènements images*, projetés par M_1 et M_2, entre les connecteurs *i* et *k* sélectionnés par les processus *SwitchA* et *SwitchB*.

Plus précisément, les *évènements images* obtenus en provenance du connecteur *i* (resp. du connecteur *k*) sont toujours préfixés par le label *a.tag**i*** (resp. *a.tag**k***). Ainsi, le processus *Bridge1* (resp. *Bridge2*) réalise deux opérations suivant la provenance des évènements :

1. Il supprime le préfixe a.tag***i*** (resp. *a.tag**k***) des *évènements images* en provenance du connecteur *i* (resp. du connecteur *k*),

2. Il ajoute le préfixe a.tag***i*** (resp. *a.tag**k***) des *évènements images* en provenance du connecteur *k* (resp. du connecteur *i*).

De ce fait, en composant les processus *Bridge1* et *Bridge2*, les *évènements images* en provenance du connecteur *i* et à destination du connecteur *k*, étiquetés par le préfixe ***a.tagi*,** sont automatiquement relabélisés par le préfixe *a.tag**k*** et *vice versa*, permettant ainsi la synchronisation de l'ensemble des processus de la *glue* de C-UNIV**.** Les processus *Bridge1* et *Bridge2* transférant aussi bien les évènements à la fois observés et initiés par les rôles R1 et R2, il devient nécessaire de définir les ensembles d'évènements suivants :

- E1$\boldsymbol{i}_{\,i\in\,[1,..,n]}$ = { αR1***i*** $\cap$ αGlue***i*** }. E1*i* est un ensemble regroupant les évènements initiés et observés par le rôle R1*i*.

- E2$\boldsymbol{k}_{\,k\in\,[1,..,m]}$ = { αR2***k*** $\cap$ αGlue***k*** }. E2*k* est un ensemble regroupant les évènements initiés et observés par le rôle R2*k*

- $\sum_{E1n} = \cup_{\,\mathbf{i}\,\in\,[1,..,\,n]}$ E1***i***. L'ensemble $\sum_{E1n}$ contient les évènements initiés et observés de tous les processus locaux que R1 est susceptible de sélectionner, c'est-à-dire des processus R1$i_{\,[1,..,n]}$.

- $\sum_{E2m} = \cup_{\,\mathbf{k}\,\in\,[1,..,\,m]}$ E2***k***. L'ensemble $\sum_{E2m}$ contient les évènements initiés et observés de R2, c'est-à-dire des processus R2$k_{\,[1,..,m]}$.

Formellement, on obtient la spécification suivante de *Bridge1* et *Bridge2* :

Bridge1 = $|_{\,\mathbf{i}\,\in\,[1,..,\,n]}$ (a.glue***i*** → ToBridge***i***),

ToBridge$\boldsymbol{i}_{\,i\in\,[1,..,n]}$ = a.tag***i***.map.[e: $\sum_{E2M}$]→ map.e → ToBridge***i***
| map.[e: $\sum_{E1N}$]→ a.tag***i***.map.[e] → ToBridge***i***
| a.reset → **Bridge1**

Bridge2 = $|_{\,\mathbf{k}\,\in\,[1,..,\,m]}$ (b.glue***k*** → ToBridge***k***),

ToBridge$\boldsymbol{k}_{\,k\in\,[1,..,m]}$ = map.[e: $\sum_{E2M}$]→ b.tag***k***.map.[e] → ToBridge***k***
| b.tag***k***.map.[e: $\sum_{E1N}$]→ map.[e] → ToBridge***k***
| b.reset → **Bridge2**

On en déduit enfin la spécification complète de C-UNIV :

Spécification complète du connecteur C-UNIV

// Définition des ensembles d'évènements de C-UNIV

Set E1***i*** $_{i\in[1,..,n]}$ = { αR1***i*** ∩ αGlue***i*** }.
Set E2***k*** $_{k\in[1,..,m]}$ = { αR2***k*** ∩ αGlue***k*** }

Set $\sum_{E1n} = \cup_{\mathbf{i} \in [1,..,n]}$ E1***i***
Set $\sum_{E2m} = \cup_{\mathbf{k} \in [1,..,m]}$ E2***k***.

Set I1*i* $_{i \in [1,..,n]}$ = *Ensembles d'évènements initiés par le rôle R1i*
Set I2*k* $_{k \in [1,..,m]}$ = *Ensembles d'évènements initiés par le rôle R2k*

Set O1*i* $_{i \in [1,..,n]}$= *Ensemble d'évènements observés par le rôle R1i*
Set O2*k* $_{k \in [1,..,m]}$= *Ensemble d'évènements observés par le rôle R2k*

Set $\sum_{O1n} = \cup_{\mathbf{i} \in [1,..,n]}$ O1***i***
Set $\sum_{O2m} = \cup_{\mathbf{k} \in [1,..,m]}$ O2***k***

// Définition des connecteurs pris en charge par C-UNIV

Role R1 = | $_{\mathbf{i} \in [1,..,n]}$ (a.glue***i*** → R1***i***),
R1***i*** $_{i \in [1,..,n]}$ = *spécification du rôle R1**i*** | a.reset → R1,

Role R2 = | $_{\mathbf{k} \in [1,..,m]}$ (b.glue***k*** → R2***k***),
R2***k*** $_{k \in [1,..,m]}$ = *spécification du rôle R2**k*** | b.reset → R2,

Glue*i* $_{i \in [1,..,n]}$ = *spécification de la gluei qui coordonne les rôles R1i et R2i*
Glue*k* $_{k \in [1,..,m]}$ = *spécification de la gluek qui coordonne les rôles R1k et R2k*

// Définitions des processus de détection

SwitchA = (electionA → a.reset → **SwitchA**
| $_{\mathbf{i} \in [1,..,n]}$ electionA → a.glue***i*** → **SwitchA**) \ {electionA}

SwitchB = (electionB → b.reset → **SwitchB**
| $_{\mathbf{k} \in [1,..,m]}$ electionB → b.glue***k*** → **SwitchB**) \ {electionA}

// Définitions des processus de jonction entre les glues et les rôles actifs

W_1 = | $_{\mathbf{i} \in [1,..,n]}$ (a.glue***i*** → ToGlue***i***),

ToGlue***i*** $_{i \in [1,..n]}$ = [e:I1***i***] → a.tag***i***.e → ToGlue***i***
| a.tag***i***. [e:O1***i***] → [e] → ToGlue***i***
| a.reset → W_1,

W_2 = | $_{\mathbf{k} \in [1,..,m]}$ (b.glue***k*** → ToGlue***k***),

ToGlue***k*** $_{k \in [1,..,m]}$ = b.tag***k***. [e:O2***k***]→ [e] → ToGlue***k***
| [e:I2***k***] → b.tag***k***.[e] → ToGlue***k***
| b.reset → W_2,

// Définitions des processus de projection

M_1 = | $_{\mathbf{i} \in [1,..,n]}$ (a.glue***i*** → ToMap***i***),

ToMap***i*** $_{i \in [1,..n]}$ = a.tag***i***.[e: I1***i***] → a.tag***i***.map.[e] → ToMap***i***
| a.tag***i***.map.[e: $\sum_{O1n}$] → a.tag***i***.[e:O1***i***] → ToMap***i***
| a.reset → M_1,

M_2 = | $_{\mathbf{k} \in [1,..,m]}$ (b.glue***k*** → ToMap***k***),

ToMap***k*** $_{k \in [1,..,m]}$ = b.tag***k***.map.[e: $\sum_{O2m}$] → b.tag***k***.[e:O2***k***] → ToMap***k***
| b.tag***k***. [e:I2***k***] → b.tag***k***.map.[e] → ToMap***k***
| b.reset → M_2,

// Définitions des processus de jonction

Bridge1 = | $_{\mathbf{i} \in [1,..,n]}$ (a.glue***i*** → ToBridge***i***),

ToBridge***i*** $_{i \in [1,..,n]}$ = a.tag***i***.map.[e: $\sum_{E2m}$]→ map.e → ToBridge***i***
| map.[e: $\sum_{E1n}$]→ a.tag***i***.map.[e] → ToBridge***i***
| a.reset → Bridge1,

Bridge2 = | $_{\mathbf{k} \in [1,..,\ m]}$ (b.glue***k*** → ToBridge***k***),

ToBridge***k*** $_{k \in [1,..,m]}$ = map.[e: $\sum_{E2m}$]→ b.tag***k***.map.[e] → ToBridge***k***
| b.tag***k***.map.[e: $\sum_{E1n}$]→ map.[e] → ToBridge***k***
| b.reset → Bridge2,

// **Glue de C-UNIV**

||**C-UNIV**=R1 || W_1 || M_1 || SwitchA
|| SwitchA ||$_{\mathbf{i} \in [1,..,\ n]}$ a.tag***i*** :Glue***i*** / {map.[r : αGlue***i***]/[r]}
|| Bridge1
|| Bridge2
|| SwitchB ||$_{\mathbf{k} \in [1,..,\ m]}$ b.tag***k*** :Glue***k*** / {map.[r : αGlue***k***]/[r]}
|| M_2 || W_2 || R2

A partir de sa spécification, il apparaît que le connecteur C-UNIV hérite bien des avantages de C-TRAD et de C-SUB : la *glue* de C-UNIV réutilise le principe de fonctionnement des processus R1, R2, W1, W2 et *Switch* du connecteur C-SUB et celui du processus *Bridge* du connecteur C-TRAD. Ainsi, C-UNIV dispose alors des propriétés de transparence de C-TRAD et des propriétés d'adaptation dynamique de C-SUB. De ce fait, la fusion dynamique de deux connecteurs se réalise sans altérer les ports des composants à interconnecter. Cependant, la combinaison de C-SUB et C-TRAD ne se fait pas sans quelques inconvénients. Nous évaluons dans la section suivante les limites de la traduction de C-UNIV.

4.2 Evaluation qualitative

D'une manière générale, la *traduction de protocoles*, qu'elle soit statique ou dynamique, constitue une solution à la résolution de leur incompatibilité si et seulement si ils partagent un minimum de similitudes [142], [143]. En d'autres termes, la traduction opérée par C-UNIV est fonctionnelle et valide si et seulement si les protocoles des rôles R1 et R2 de C-UNIV peuvent être dynamiquement projetés sur un *protocole image* commun établissant entre ces derniers une correspondance sémantique. Cette traduction n'est possible que si les processus des *glues* des deux connecteurs à *fusionner* possèdent : (i) une sémantique des évènements de leur alphabet identique (évènements projetés *via* la fonction de projection *f* ayant une même image), et (ii) des comportements compatibles.

Il s'ensuit que, si l'on considère deux connecteurs *i* et *k*, sélectionnés à un instant donné par *SwitchA* et *SwitchB*, la traduction dynamique réalisée par C-UNIV peut échouer dans les deux cas suivants :

1. Lorsque la projection des alphabets des processus Glue*i* et Glue*k* par la fonction de projection *f* de C-UNIV est disjointe, c'est-à-dire si $\{\alpha Glue i'\} \cap \{\alpha Glue k'\} = \varnothing$. Cela signifie que Glue*i* et Glue*k* n'ont aucun *évènement image* en commun *via f* et donc aucun évènement sémantiquement équivalent. De ce fait, les rôles R1 et R2 sont dans l'incapacité de se synchroniser, l'équation de la *glue* du connecteur C-UNIV se bloque et la traduction échoue.

2. Lorsque les processus Glue*i* et Glue*k* correspondant aux *glues* des connecteurs *i* et *k* ne possèdent aucun comportement compatible.

A partir des propriétés précédentes, il découle que la *traduction dynamique* est toujours valide si Glue*i* est un raffinement de Glue*k* *via* la fonction *f* appliquée sur leur alphabet, c'est-à-dire si et seulement si :

$$Glue i_{|\alpha Glue i'} \subseteq Glue k_{|\alpha Glue k'}$$

Nous qualifions cette condition de validité de *forte* pour deux raisons précises dues aux propriétés du principe du raffinement [91] : (i) l'alphabet $\alpha Glue i'$ doit inclure tous les évènements de l'alphabet $\alpha Glue k'$, et (ii) les comportements de Glue*i* doivent inclure tous ceux de Glue*k*.

Toutefois dans la pratique, cette condition de validité peut ne pas toujours être respectée sans pour autant nuire au fonctionnement de C-UNIV à condition que Glue*i* et Glue*k* raffinent *via f* un même processus Q, qui définit un comportement minimal garantissant une traduction valide, c'est-à-dire si et seulement si:

$$Glue i_{|\alpha Glue i'} \subseteq Q \text{ et } Glue k_{|\alpha Glue k'} \subseteq Q$$

L'alphabet αQ devient un sous ensemble d'évènements communs obligatoires de $\alpha Glue i'$ et de $\alpha Glue k'$ à partir desquels Glue*i* et Glue*k* peuvent toujours se synchroniser. De ce fait, afin que C-UNIV garantisse une traduction minimale valide, quels que soient les connecteurs fusionnés, les processus $Glue i_{[1,..,n]}$ et $Glue k_{[1,..,m]}$ de C-UNIV doivent tous raffiner le processus Q. Nous qualifions cette condition de validité de *faible* par opposition à la précédente. En d'autres termes, Q détermine les critères minimums que les connecteurs doivent respecter pour être pris en charge par C-UNIV.

Par ailleurs, si la Glue d'un connecteur ne raffine pas le processus Q, il est possible de façon similaire à C-TRAD, de composer les processus *Bridge1* et *Bridge2* avec des processus T_1 et T_2 opérant des transformations sur sa Glue afin qu'ils raffinent Q. Ces transformations sont actives uniquement si une des *glues* sélectionnée par

SwitchA ou *SwitchB* correspond à la Glue de ce nouveau connecteur. Les processus T_1 et T_2 s'insèrent de part et d'autre des processus réalisant la jonction :

$$\|\textbf{C-UNIV} = \ldots \| T_1 \| \text{Bridge1} \| \text{Bridge2} \| T_2 \| \ldots$$

Avec :

$T_1 = |_{i \in [1,..,n]} (a.glue_i \rightarrow ToT_i)$,

$ToT_{i\ [1,..n]}$ = *Spécification de la transformation sur* $Glue_i$ $|_{\alpha Glue_i'}$

$T_2 = |_{k \in [1,..,m]} (b.glue_k \rightarrow ToT_k)$,

$ToT_{k\ [1,..m]}$ = *Spécification de la transformation sur* $Glue_k$ $|_{\alpha Gluek'}$

Reprenons les exemples d'incompatibilités des protocoles de communication hétérogènes de type RPC tel que SOAP, RMI, et TRMI du chapitre précédent afin d'illustrer et d'évaluer le principe de C-UNIV. La spécification de leur connecteur respectif est donnée dans la Figure 26. Comme indiqué dans le chapitre 2, les protocoles de types RPC suivent un paradigme de communication de type requête/réponse synchrone. On en déduit aisément la spécification du processus Q définissant le comportement minimal que les *glues* des connecteurs détectés par C-UNIV doivent raffiner afin de garantir une traduction minimale valide des différents protocoles RPC :

$$Q = \text{map.c.request} \rightarrow \text{map.s.request} \rightarrow \text{map.s.response} \rightarrow \text{map.c.response} \rightarrow Q$$

Le processus Q décrit la nature de la séquence (ou l'ordonnancement) minimale des *évènements images* obligatoires (c'est-à-dire αQ) qui doit être échangée entre les rôles R1 et R2 de C-UNIV. Par ailleurs, la fonction de projection *f* est définie de façon à établir, lorsque cela est possible, une correspondance sémantique entre les évènements des alphabets de $Glue_{RPC\text{-}RMI}$, $Glue_{RPC\text{-}SOAP}$, $Glue_{RPC\text{-}Transaction\text{-}RMI}$ et Q. Comme illustré dans la Figure 27, les *évènements images* de ces alphabets obtenus *via f* sont préfixés par le mot clé *map* pour les distinguer des évènements non projetés. Ainsi les évènements de chacun des ensembles suivants sont sémantiquement équivalents : $\{c.RMI_{request}, c.SOAP_{request}\}$, $\{s.RMI_{request}, s.SOAP_{request}\}$, $\{s.RMI_{response}, s.SOAP_{response}\}$, $\{c.RMI_{response}, c.SOAP_{response}\}$

Une fois la fonction *f* définie, il apparaît que la traduction dynamique entre les protocoles SOAP et RMI est garantie suivant la condition de validité forte :

$$Glue_{RPC\text{-}RMI}|\alpha Glue_{RPC\text{-}RMI}' \subseteq Glue_{RPC\text{-}SOAP}|\alpha Glue_{RPC\text{-}SOAP}'$$

$$\text{et } Glue_{RPC\text{-}SOAP}|\alpha Glue_{RPC\text{-}SOAP}' \subseteq Glue_{RPC\text{-}RMI}|\alpha Glue_{RPC\text{-}RMI}'$$

En revanche, ce n'est pas le cas du processus $Glue_{RPC\text{-}Transaction\text{-}RMI}$ qui, ne raffinant pas Q, ne peut pas être traduit dynamiquement *via* C-UNIV sans avoir recours au processus de transformation T_1 et T_2. En effet, le comportement de la *glue* du connecteur RPC-Transaction-RMI diffère de celui de Q et dépend d'évènements dont les images *via f* n'ont aucune correspondance sémantique avec aucun des *évènements images* des alphabets projetés des autres *glues*.

Connector RPC-RMI (A)
Role Client = $RMI_{request} \rightarrow RMI_{response} \rightarrow$ Client
Role Service = $RMI_{request} \rightarrow RMI_{response} \rightarrow$ Service
Glue-RMI = c. $RMI_{request} \rightarrow$ s. $RMI_{request} \rightarrow$ Glue-RMI
| s. $RMI_{response} \rightarrow$ c. $RMI_{response} \rightarrow$ Glue-RMI
‖RPC-RMI= c : Client ‖ s : Service ‖ Glue-RMI

Connector RPC-SOAP (B)
Role Client = $SOAP_{request} \rightarrow SOAP_{response} \rightarrow$ Client
Role Service = $SOAP_{request} \rightarrow SOAP_{response} \rightarrow$ Service
Glue-SOAP = c. $SOAP_{request} \rightarrow$ s. $SOAP_{request} \rightarrow$ Glue-SOAP
| s. $SOAP_{response} \rightarrow$ c. $SOAP_{response} \rightarrow$ Glue-SOAP
‖RPC-SOAP= c : Client ‖ s : Service ‖ Glue-SOAP

Connector RPC-Transaction-RMI (C)
Role Client = open → SESSION,
SESSION= $RMI_{request} \rightarrow RMI_{response} \rightarrow$ SESSION | close → Client
Role Service = open → SESSION,
SESSION= $RMI_{request} \rightarrow RMI_{response} \rightarrow$ SESSION | close → Service
Glue-TRMI = c.open → s.open,
| c.$RMI_{request} \rightarrow$ s.$RMI_{response} \rightarrow$ Glue-TRMI,
| s.$RMI_{response} \rightarrow$ c.$RMI_{response} \rightarrow$ Glue-TRMI,
| c.close → s.close → Glue-TRMI,
‖RPC-Transaction-RMI = c : Client ‖ s : Service ‖ Glue-TRMI

Figure 26. Exemple de spécification de connecteurs à fusionner dynamiquement

$f(c.RMI_{request})$	=	map.c.request
$f(s.RMI_{request})$	=	map.s.request
$f(c.RMI_{response})$	=	map.c.response
$f(s.RMI_{response})$	=	map.s.response
$f(c.SOAP_{request})$	=	map.c.request
$f(s.SOAP_{request})$	=	map.s.request
$f(c.SOAP_{response})$	=	map.c.response
$f(s.SOAP_{response})$	=	map.s.response
$f(c.open.RMI_{request})$	=	map.c.open
$f(s.open.RMI_{request})$	=	map.s.open
$f(c.close.RMI_{request})$	=	map.c.close
$f(s.close.RMI_{request})$	=	map.s.close

Figure 27. Définition de la fonction de projection

Au chapitre 3, nous avons défini la fonction de transformation τ réalisant une fusion des connecteurs RPC-RMI et RPC-Transaction-RMI :

$$\tau = c.RMI_{request} \rightarrow tag1.c.open \rightarrow tag1.c.RMI_{request} \\ \quad | \; tag1.c.RMI_{response} \rightarrow c.RMI_{response} \rightarrow tag1.c.close$$

Dans le contexte de C-UNIV, cette fonction devient un processus local de τ_1 et τ_2 qui se note :

$$ToT_x = map.c.request \rightarrow map.c.open \rightarrow map.c.request \\ \quad | \; map.c.response \rightarrow map.c.response \rightarrow map.c.close$$

La valeur de x correspond au numéro de la *glue* du connecteur RPC-Transaction-RMI. Par ailleurs, la spécification du processus ToT_x ne se fait plus en fonction de l'alphabet de $Glue_{RPC\text{-}Transaction\text{-}RMI}$ mais en fonction de son image *via f*. Ainsi, il est possible de *fusionner* le connecteur RPC-Transaction-RMI aussi bien avec le connecteur RPC-RMI que RPC-SOAP. Contrairement à C-TRAD, la traduction n'est pas spécifique à deux connecteurs particuliers mais devient dynamique avec n'importe quel connecteur pris en charge par C-UNIV.

Dans la section suivante, nous appliquons les principes de la *traduction dynamique de protocoles*, modélisés par le connecteur C-UNIV, aux protocoles réseaux afin de résoudre leur incompatibilité.

4.3 Application aux protocoles réseaux

Afin de faire face à la complexité des spécifications de certains protocoles réseaux, et dans un souci de standardisation de ces derniers, l'ISO (*International Standardisation Organization*) a conçu un modèle de référence OSI (*Open Systems Interconnection*)

qui décompose un protocole réseau en couches de façon à classifier les différentes fonctions qu'un protocole réseau doit fournir pour interconnecter des systèmes distants [1]. Les différentes couches sont délimitées par une interface (Figure 28A) qui définit les opérations ou les fonctions qu'une couche inférieure offre à la couche immédiatement supérieure [140]. Les fonctions d'une couche sont réalisées *via* un protocole définissant un ensemble de règles qui déterminent le format et la signification des messages qui sont échangés par des entités paires au sein d'une couche (Figure 28B, C, D). Ainsi, si l'on considère deux entités E1 et E2 interconnectées *via* un protocole réseau (Figure 28), la *couche n* de l'entité A interagit avec la *couche n* de l'entité B *via* le *protocole n*. Par ailleurs, les fonctions de la *couche n* dépendent des fonctions de la *couche n-1*, celles de la *couche n-1* dépendent à leur tour de celles de la *couche n-2* et ainsi de suite. Un protocole réseau est ainsi représenté par une pile de couches ou une pile de protocoles [147].

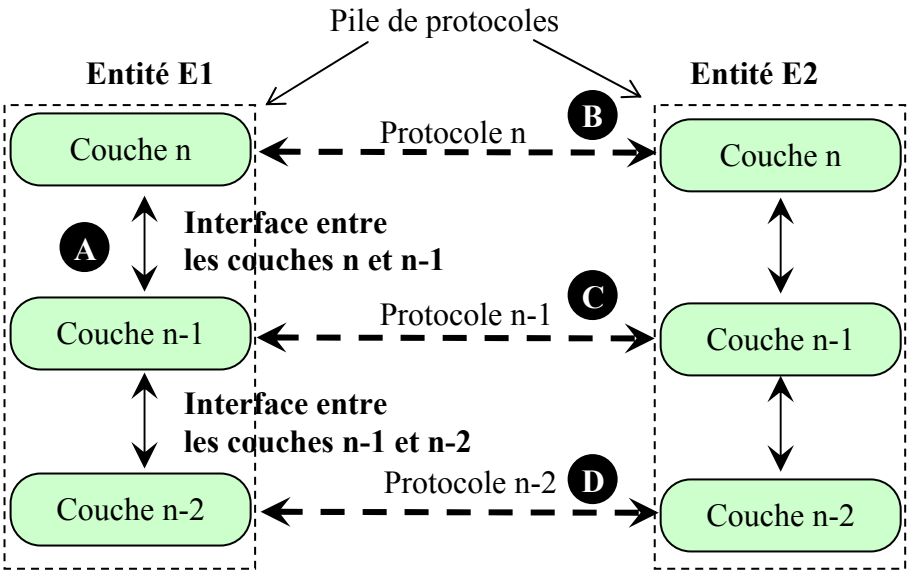
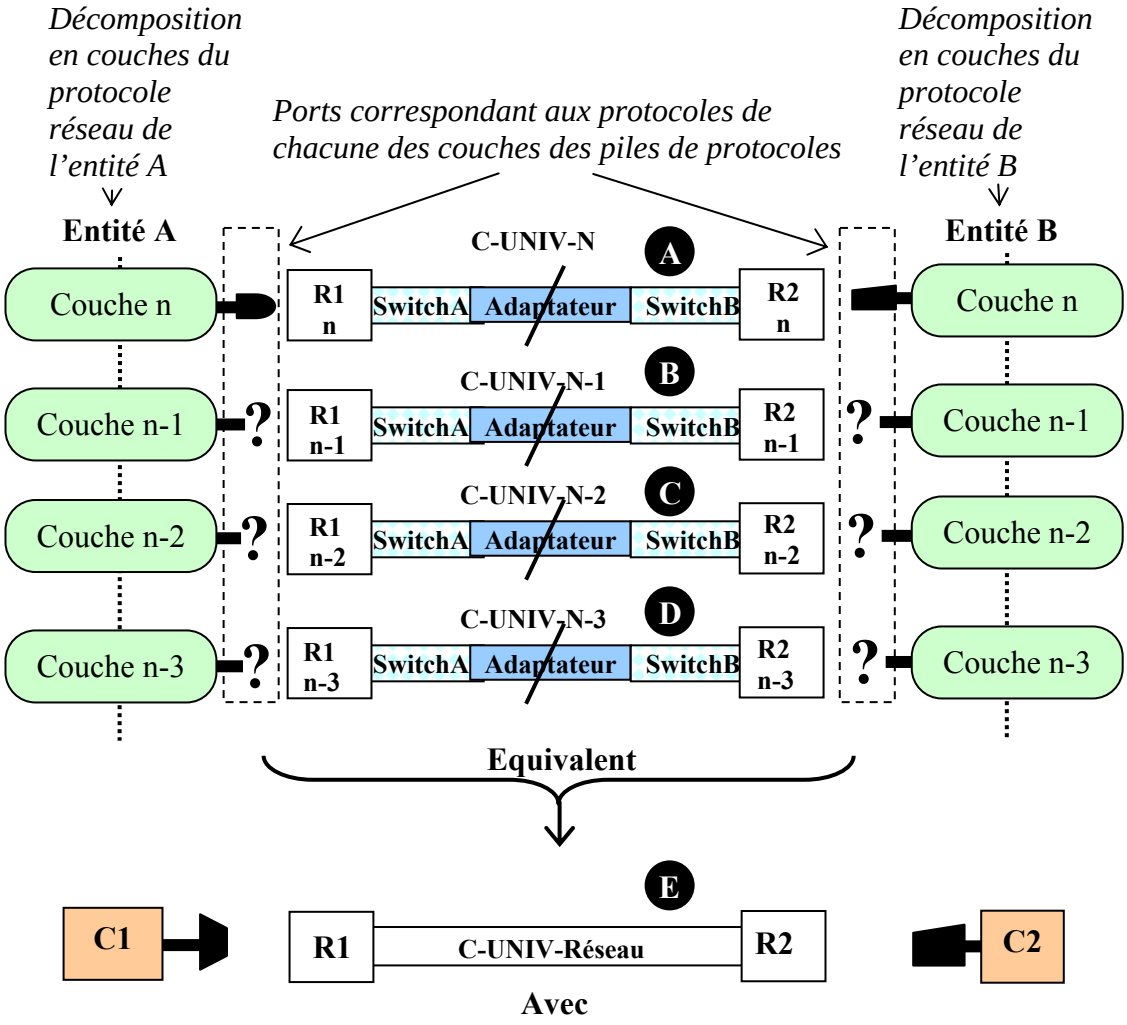


Figure 28. Décomposition en couches d'un protocole réseau.

La spécification d'un protocole réseau est définie par la spécification de chacune de ses couches. Le nombre de couches d'un protocole réseau doit être suffisamment grand pour que des fonctions très distinctes ne soient pas regroupées dans une même couche, l'objectif étant que chaque couche assure une fonction bien définie, simplifiant implicitement le protocole de cette dernière [140]. Le modèle de référence OSI préconise la décomposition des protocoles réseaux en sept couches [1] : (1) physique, (2) liaison, (3) réseau, (4) transport, (5) session, (6) présentation, (7) application. La source des incompatibilités entre les protocoles réseaux vient du fait qu'il existe plusieurs protocoles différents standardisés pour chacune de ces couches, c'est-à-dire plusieurs façons d'implémenter la fonction d'une couche. Par exemple, au niveau réseau, les intergiciels hétérogènes sont dans l'incapacité d'interagir entre eux car ils implémentent de façon différente une ou plusieurs de ces couches.

Naturellement, résoudre l'incompatibilité des protocoles réseaux équivaut à résoudre l'incompatibilité des protocoles utilisés par leur(s) couche(s) hétérogène(s) qui sont dans le pire des cas au nombre de *n*, ou au nombre de 7 suivant le modèle de référence OSI. Les différents protocoles possibles pour une même couche, bien que différents, présentent des similitudes étant donné qu'ils sont tous dédiés à une même couche, et qu'ils implémentent donc tous des fonctions identiques. Ainsi, en assimilant les différentes couches d'un protocole réseau à des composants, et les protocoles de chacune d'elles à des connecteurs, résoudre l'incompatibilité des protocoles réseaux revient à interconnecter chacune des couches de leur pile respective par un connecteur C-UNIV (Figure 29).



$$||\text{C-UNIV-Réseau} = ||_{k \in [1,..,n]} \text{C-UNIV-}k$$

Figure 29. Résolution des incompatibilités entre protocoles réseaux

Comme C-UNIV requiert que les protocoles des connecteurs qu'il est susceptible de *fusionner* partagent un minimum de similitudes, il s'ensuit qu'à chaque *niveau* ou *étage* d'une pile de protocoles correspond un connecteur C-UNIV spécifique (Figure 29A, B, C, D) : soit n le nombre de niveaux d'une pile de protocoles, et $k \in [1, n]$, alors le connecteur C-UNIV-k résout dynamiquement les incompatibilités entre les différents protocoles susceptibles d'exister entre les diverses *couches k* des protocoles réseaux hétérogènes. On définit alors un nouveau connecteur C-UNIV-Réseau résultant de la composition des *glues* des connecteurs C-UNIV-k avec $k \in [1, n]$ (Figure 29E). Par conséquent, deux composants C1 et C2, utilisant des protocoles réseaux incompatibles, peuvent être interconnectés à l'aide de C-UNIV-Réseau dont la *glue* se note :

$$||\text{C-UNIV-Réseau} = ||_{k \in [1,..,n]} \text{C-UNIV-}\boldsymbol{k}$$

Maintenant que nous avons modélisé le fonctionnement de la *traduction dynamique de protocoles* sous la forme du connecteur C-UNIV et fourni une spécification formelle de ce dernier, nous présentons dans la section suivante l'architecture logicielle d'un système implémentant un tel connecteur.

4.4 Architecture logicielle du système de traduction

Dans cette section, lorsque nous parlons de *composants* nous faisons référence au concept de composants tel que défini par les ADL tandis que lorsque nous parlons de *composants logiciels* nous nous référons aux technologies orientées composants utilisées pour le développement de systèmes logiciels. Nous avons vu au chapitre 3, et en particulier dans la section 3.4.1, que les composants logiciels constituent un choix technique privilégié pour concevoir des systèmes reconfigurables. De ce fait, étant donné l'aspect dynamique du connecteur C-UNIV, l'utilisation d'un ensemble de composants logiciels, dynamiquement assemblables, nous semble adéquat pour implémenter les différents principes de la spécification formelle de C-UNIV.

Nous présentons dans un premier temps l'architecture générique de notre système, dénommé SysTDyP (Système de Traduction Dynamique de Protocoles) puis nous introduisons une spécialisation de ce dernier afin de résoudre les incompatibilités des protocoles réseaux.

Architecture générique.

Les auteurs de la référence [148] introduisent l'utilisation de composants logiciels particuliers, tels que des analyseurs (*parsers*) et des composeurs (*composers*), afin de concevoir un système permettant de résoudre les incompatibilités entre plusieurs versions d'un même protocole. Pour les auteurs de [148], la moindre modification de la spécification d'un protocole déjà existant, comme le format d'un message, est une condition suffisante responsable de l'apparition d'une nouvelle version d'un

protocole incompatible avec la précédente. Bien que leur vision soit plus restrictive que la nôtre, puisqu'elle est focalisée sur la résolution des incompatibilités des différentes versions d'un même protocole, les analyseurs et les composeurs peuvent être réutilisés pour la *traduction dynamique de protocoles* et constituent ainsi la base de notre système.

Un *analyseur* est un composant logiciel qui effectue une analyse syntaxique de la structure des messages d'un protocole spécifique. En particulier, un analyseur prend en entrée un message et génère en sortie un ensemble d'évènements représentant la structure du message. Un *composeur* est un composant logiciel qui effectue l'opération inverse : c'est un générateur de messages, spécifique à un protocole, qui génère, à partir d'un ensemble d'évènements, des messages. Ainsi, la résolution des incompatibilités entre deux versions *k* et *i* d'un protocole P_1 est effectuée en connectant un analyseur, spécifique à la version *k*, à un composeur, dédié à la version *i*, c'est-à-dire, en transférant les évènements générés par le premier vers le second. Les évènements permettent d'établir une correspondance sémantique entre des messages issus de différentes versions d'un même protocole. En d'autres termes, la communication entre un analyseur et un composeur ne dépend d'aucun détail syntaxique étant donné qu'ils communiquent à un niveau sémantique à l'aide des évènements.

Notre objectif étant d'implémenter les différents processus de la *glue* de C-UNIV par un ensemble de composants logiciels, nous réutilisons les parseurs et les composeurs, définis ci-dessus, pour implémenter la projection des alphabets des protocoles incompatibles des rôles R1 et R2 suivant une fonction de projection *f*.

L'équation de processus résultant de la fusion d'un connecteur *i* avec un connecteur *k* suivant la spécification de C-UNIV se note (Voir Figure 30):

(R1*i* || ToMap*i*)||(ToGlue*i* || ToT*i*) || a.tag***i*** :Glue***i*** / {map.[r : αGlue***i***]/[r]}|| ToBridge***i*** || ToBridge***k*** || ToMap*k* || ToGlue*k* || ToT*k* || a.tag***k*** :Glue***k*** / {map.[r : αGlue***k***]/[r]}|| R2*k*

Le processus R1i || ToMapi, est aisément implémenté par l'analyseur *i* et le composeur *i* spécifiques au rôle R1i (Figure 30A). La synchronisation des *évènements images* entre l'analyseur *i* et le composeur *i* est effectuée à l'aide du processus Glue*i* éventuellement altérée *via* le processus ToT*i* selon que Glue*i* raffine ou non le processus Q. L'implémentation du processus ToT*i* || Glue*i* se fait à l'aide d'un nouveau composant logiciel que l'on appelle unité *i,* qui utilise un automate à état fini (AEF) pour implémenter le comportement de ToT*i* || Glue*i* (Figure 30C). L'AEF d'une unité se définit comme suit :

Un AEF est un 5-tuple (Ψ, C, Φ, T, q_0, F)

- Ψ est un ensemble fini d'états.

- C est un ensemble d'évènements susceptibles de changer l'état de l'AEF.
- Φ est un ensemble d'actions (dont l'émission et la réception d'évènements).
- $T : \Psi \times \Phi \times C \Rightarrow \Psi$ est la fonction de transition.
- $q_0 \in \Psi$ est l'état de départ.
- $F \subset \Psi$ est l'ensemble des états terminaux.

Par conséquent, la synchronisation entre l'analyseur *i* et le composeur *i* se fait en connectant ces derniers à l'unité *i* (Figure 30B). De façon similaire, le processus R1*k* || ToMap*k* est implémenté par l'analyseur *k* et le composeur *k* qui se retrouvent connectés à une unité *k* qui synchronise les *évènements images* échangés entre ces derniers et qui implémente le processus ToT*k* || Glue*k* (Figure 30E, F, G). En ce qui concerne les comportements des processus ToBridge*i* et ToBridge*k*, ils correspondent tout simplement à la composition dynamique de l'unité *i* avec l'unité *k* (Figure 30D). Cette composition dépend du comportement des processus *SwitchA* et *SwitchB* de C-UNIV implémenté à un composant logiciel appelé *moniteur* qui, selon le domaine d'applications pour lequel le système SysTDyP est conçu, utilise sa propre stratégie de sélection.

Par ailleurs, les différents composants logiciels de SysTDyP sont connectés les uns avec les autres *via* des connexions évènementielles. L'architecture évènementielle du système ainsi obtenue (Figure 30H) réduit les contraintes de couplages entre les différents composants. De ce fait, SysTDyP est un système reconfigurable dynamique et extensible : les interactions de ses composants logiciels se faisant *via* des évènements, les composants logiciels deviennent indépendants les uns des autres et sont capables d'opérer sans aucune connaissance préalable des composants logiciels avec lesquels ils interagissent. De ce fait, un changement dynamique d'analyseurs, de composeurs et/ou d'unités peut survenir pendant l'exécution de SysTDyP sans altérer son fonctionnement.

Raffinement de l'architecture du système pour les protocoles réseaux.

Tout d'abord, il est important d'introduire de plus amples détails sur le fonctionnement d'une pile de protocoles avant de présenter le raffinement de l'architecture du système SysTDyP pour les protocoles réseaux afin de comprendre son fonctionnement. Si l'on considère deux entités E1 et E2 interconnectées *via* un protocole réseau (Figure 28), en réalité, la *couche n* de l'entité E1 n'interagit pas directement avec la *couche n* de l'entité E2 : aucune donnée n'est directement transmise entre les *couches n* des deux entités. En effet, dans la pratique, chaque couche transmet les données à émettre à la couche immédiatement inférieure jusqu'à la couche la plus basse, qui est le support physique, grâce auquel les données sont physiquement transmises [147], [140]. Chaque couche traversée rajoute une entête contenant des informations spécifiques à son protocole. Ainsi, les données émises provenant de la *couche n,* communément appelées les (N)-PDU (Protocole Data Unit) [140], sont transmises à la *couche n-1* qui complète les (N)-PDU par une entête, dénommée (N-1)-PCI (Protocole Control Information) [140], pour former des (N-1)-

PDU qui sont à leur tour transférés à la couche inférieure et ainsi de suite jusqu'à la couche physique pour finalement devenir un message réseau transmis en direction de l'entité B qui effectue le processus inverse [147], [140].

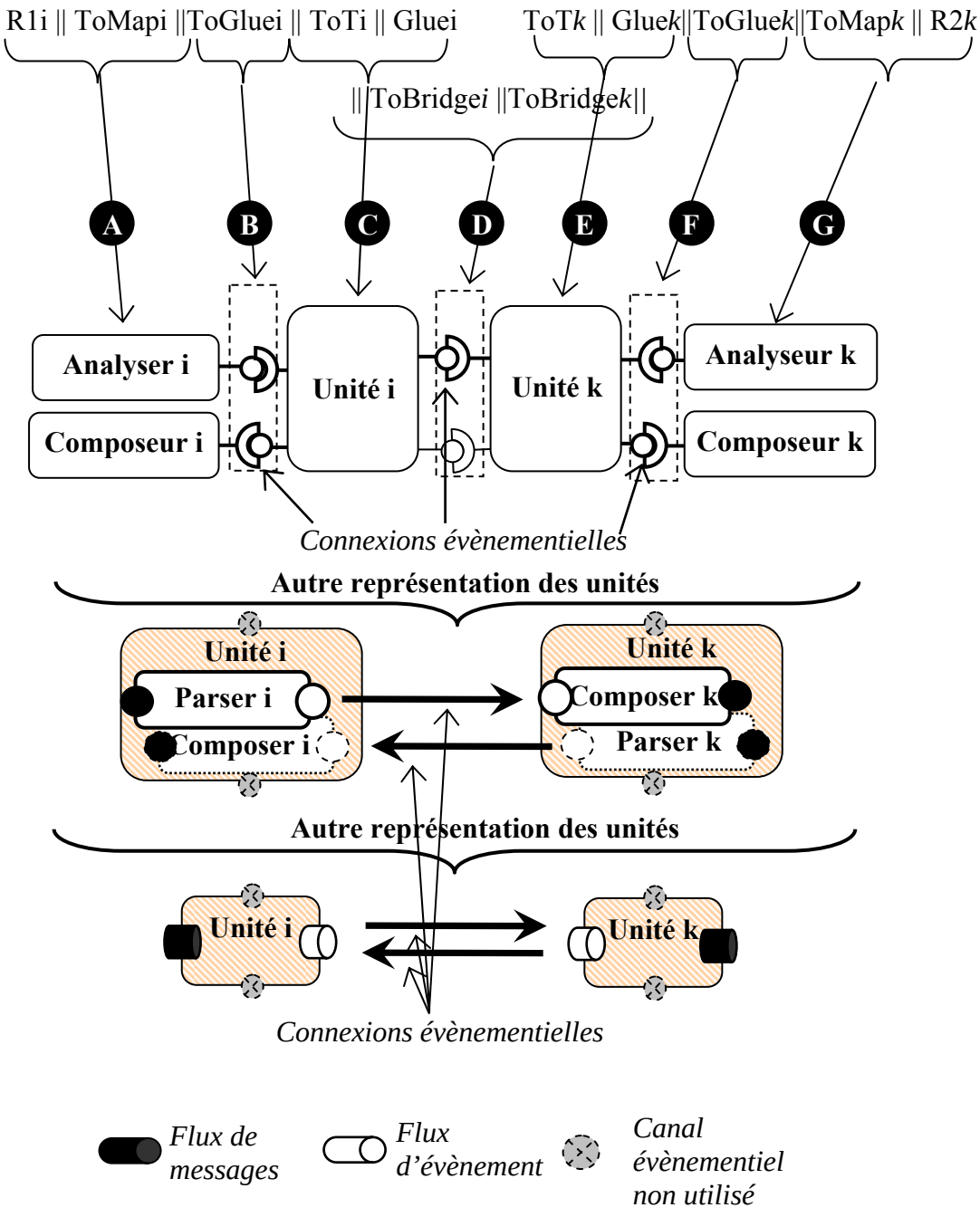


Figure 30. Passage du formalisme de C-UNIV à une architecture logicielle

De ce fait, la traduction d'un protocole réseau résulte de la composition successive d'unités au fur et à mesure de la détection des protocoles associés à chacune de ses couches. La détection des protocoles de ces dernières est réalisée *via* l'analyse des

entêtes successives inclues dans un message réseau, une entête étant spécifique à un protocole donné. En d'autres termes, l'ensemble des entêtes d'un message constitue une signature qui révèle les protocoles des couches de la pile dont le message réseau est issu. Conformément à la spécification de C-UNIV-Réseau, la résolution des incompatibilités des protocoles réseaux s'effectue grâce à la composition de *n* connecteurs C-UNIV, soit de 2 x *n* unités, *n* étant le nombre de couches des piles des protocoles réseaux.

La Figure 31 illustre l'architecture logicielle de C-UNIV-Réseau permettant de traduire les messages issus de la pile de protocoles d'une entité E1 en messages compatibles avec la pile de protocoles d'une entité E2. La traduction est réalisée par la composition de deux chaînes d'unités A et B. La chaîne A transforme les messages réseaux en provenance de l'entité E1 en un flux d'évènements tandis que la chaîne B de l'entité E2 génère, à partir de ce flux, des messages réseaux compréhensibles par l'entité E2. Dans l'étape❶ de la Figure 31, le message en provenance de l'entité E1 est d'abord analysé par l'analyseur de l'unité n-3, premier maillon de la chaîne A, qui représente le protocole de la dernière couche traversée par le message avant sa transmission. L'analyseur décompose le message en deux parties distinctes : l'entête ((N-3)-PCI) et les données ((N-2)-PDU). La première partie est transformée en un flux d'évènements qui est transféré vers le composeur de l'unité n-3 de la chaîne B. La deuxième partie est quant à elle transmise à l'analyseur de l'unité n-2 qui est l'unité suivante de la chaîne A. Récursivement, l'unité n-2 de la chaîne A extrait des données qu'elle reçoit, d'une part, une nouvelle en-tête ((N-2)-PCI) traduit en un flux d'événements envoyés au composeur de l'unité n-2 de la chaîne B et d'autre part, de nouvelles données ((N-1)-PDU) redirigées vers l'analyseur de l'unité suivante de la chaine A et ainsi de suite jusqu'à ce qu'il n'y ait plus d'en-têtes à extraire. Les évènements produits par l'analyseur de chaque unité de la chaîne A sont séquentiellement transférés au composeur de chaque unité de la chaîne B (Figure 31, étape❷).

Les composeurs des unités de la chaîne B ne peuvent pas générer de messages avant que l'analyseur de la dernière unité de la chaîne A ne termine intégralement l'analyse syntaxique du message reçu générant les derniers événements représentant la structure du message. En fait, ces derniers évènements vont permettre au composeur de l'unité située à l'une des extrémités de la chaîne B de générer un fragment de message requis par le maillon suivant de la chaîne et ainsi de suite jusqu'au composeur de l'unité située à l'autre extrémité de la chaîne B (Figure 31, étape❸). Le message finalement généré, résultant de la composition des composeurs des unités de la chaîne B, est compatible avec la pile de protocole de l'entité B (Figure 31, étape ❹). Bien que la Figure 31 mette en perspective principalement la traduction d'un message en provenance de l'entité E1 vers l'entité E2, l'inverse est également possible de façon symétrique (la traduction inverse est représentée par des flèches en pointillées).

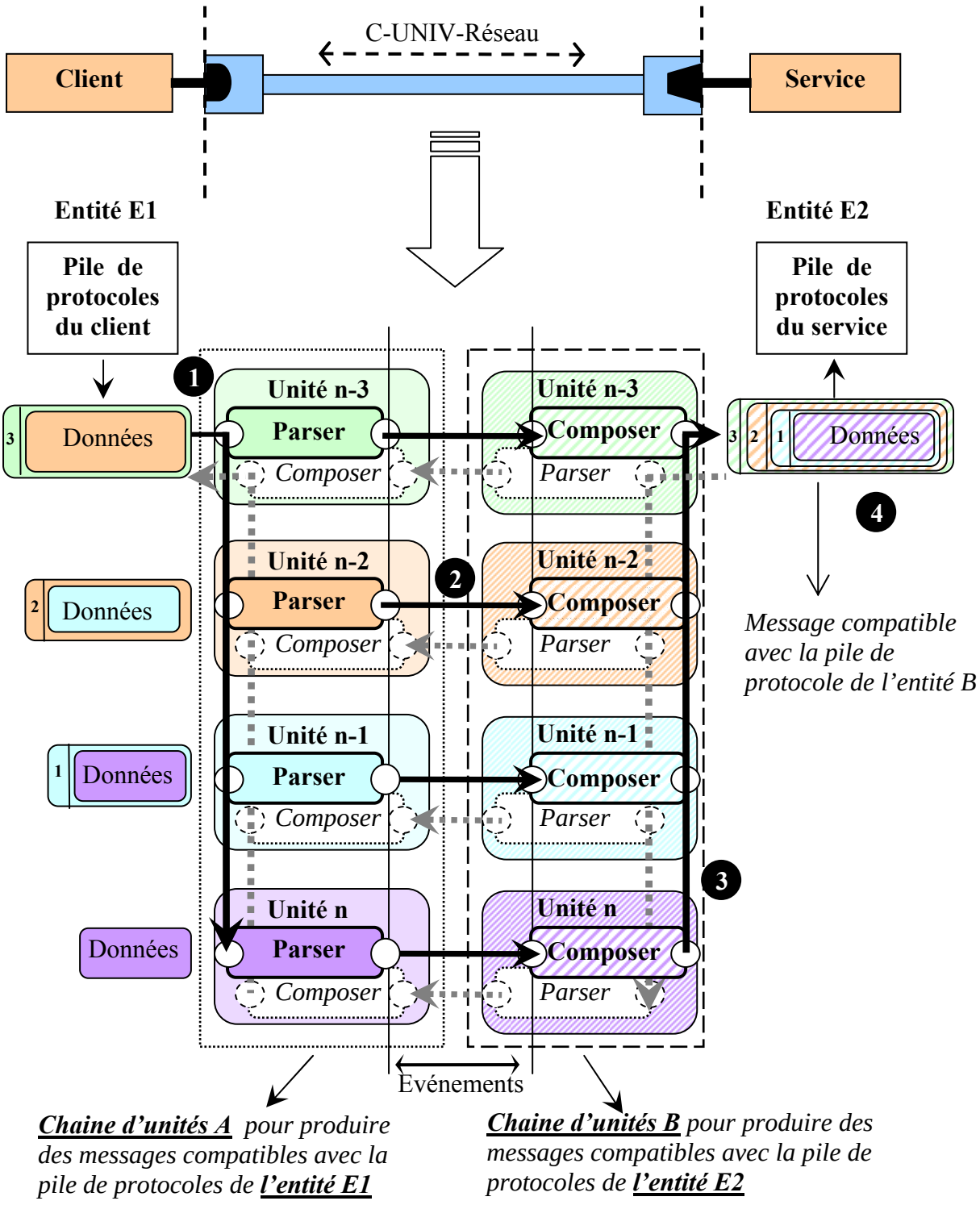


Figure 31. Composition des unités correspondantes aux différentes couches

Il ressort de l'architecture logicielle du système SysTDyP que la *traduction dynamique de protocoles* est réalisée à travers une composition dynamique d'unités à la fois verticale et horizontale. La composition verticale, c'est-à-dire le chaînage vertical d'unités, transforme les messages d'un protocole réseau en un flux d'évènements, tandis que la composition horizontale d'unités, traduit ce flux d'évènements en messages suivant la spécification d'un autre protocole réseau. Enfin, comme indiqué dans la spécification C-UNIV-Réseau, le système SysTDyP n'altère ni ne remplace les intergiciels existants : il effectue une traduction transparente.

4.5 Synthèse

Dans ce chapitre, nous avons présenté notre solution, que nous qualifions de *traduction dynamique de protocoles*, permettant de résoudre les incompatibilités de protocoles entre des intergiciels hétérogènes. Notre démarche consiste à fusionner les avantages de la *substitution* et de la *traduction de protocoles* usuellement utilisées dans ce contexte. Nous avons, en particulier proposé dans ce chapitre :

1. Une spécification formelle de notre solution selon le langage de processus FSP. Notre solution peut s'appliquer à n'importe quel type de protocoles.

2. Une évaluation qualitative des limites de notre approche.

3. Une architecture logicielle d'un système, nommé SysTDyP, définie à partir de notre spécification formelle.

La spécification formelle de la *traduction dynamique de protocoles*, modélisée sous la forme d'un connecteur appelé C-UNIV, fournit une solution qui permet de résoudre l'hétérogénéité des intergiciels indépendamment de leurs spécificités, et des technologies employées. L'évaluation de notre approche a mis en perspective que la *traduction dynamique de protocoles* assure toujours une *traduction minimale* mais ne garantit jamais une *traduction totale* entre deux protocoles incompatibles.

Dans la suite de ce document, nous nous intéressons à la mise en œuvre du système SysTDyP implémentant le connecteur C-UNIV dans le contexte d'une *architecture de services ubiquitaires*.

5 Différentes mises en œuvre de la traduction dynamique de protocoles

Dans une *architecture de services ubiquitaires* les clients (c'est à dire les consommateurs de services) sont confrontés à deux problèmes majeurs. Ils doivent successivement : (i) découvrir, sans aucune connaissance préalable, la localisation des services distants, et (ii) communiquer avec les services nouvellement découverts, indépendamment de l'hétérogénéité matérielle et logicielle de leur hôte. Comme présenté dans le chapitre 2, ces deux problèmes sont résolus par leur intergiciel respectif *via* l'utilisation de *protocoles de découverte de services* et de *communication*. Toutefois, un client et un service ne peuvent interopérer qu'à la seule et unique condition que leur intergiciel utilise des protocoles d'interaction identiques.

Dans ce chapitre, nous résolvons ces deux problématiques (l'incompatibilité des protocoles de découverte de services et de communication) à l'aide des principes de *la traduction dynamique de protocoles* afin que les clients et les services de l'*environnement ubiquitaire* deviennent interopérables de façon transparente (c'est-à-dire sans avoir à altérer leur implémentation) et dynamique (c'est-à-dire sans prévoir à l'avance les interactions susceptibles d'avoir lieu). Plus précisément, nous introduisons deux systèmes, chacun d'eux raffinant l'architecture logicielle du système SysTDyP, présenté dans le chapitre précédent, suivant les spécificités de l'une et de l'autre de ces problématiques. Dans la section 5.1, nous présentons le système INDISS (*Interoperable Discovery System for Networked Services*) [155] permettant de résoudre l'hétérogénéité entre SDPs, tandis que dans la section 5.2 nous exposons le système NEMESYS (Network Metacommunication System for Middleware Interoperability) spécifiquement conçu pour résoudre celle des protocoles de communication.

5.1 INDISS

Le système INDISS [155] est un raffinement spécifique de l'architecture du système SysTDyP pour résoudre les incompatibilités entre différents SDPs et hérite de ce fait de ses caractéristiques. INDISS est alors un système, dynamiquement reconfigurable en fonction de son contexte d'exécution, qui effectue une traduction dynamique des SDPs hétérogènes en cours d'utilisation dans son voisinage. Une des propriétés clés d'INDISS, héritée des concepts du connecteur C-UNIV-Réseau, est d'opérer de façon transparente au niveau réseau, en capturant et en traduisant à la volée les messages réseaux relatifs aux différents SDPs sans que les intergiciels des fournisseurs et consommateurs de services en soient conscients. Grâce à son architecture modulable, INDISS peut : (i) s'adapter aux ressources potentiellement limitées (mémoires, CPU, bande passante) de certains périphériques et (ii) être déployé aussi bien du côté d'un fournisseur que d'un consommateur de services ou sur un intermédiaire comme une passerelle.

Enfin, la résolution de l'incompatibilité entre différents SDPs, selon les principes de la *traduction dynamique de protocoles*, est valide si et seulement si les SDPs peuvent être projetés sur un *protocole image* commun. La projection de protocole est possible si les SDPs partagent un minimum de similitudes, c'est-à-dire s'ils possèdent des règles de coordinations compatibles et un ensemble de messages sémantiquement équivalents. De ce fait, dans la section 5.1.1, dans l'objectif de déterminer leurs similitudes, nous faisons un panorama des SDPs. Cette étude nous permet de réaliser, dans la section 5.1.2, un raffinement de l'architecture logicielle du système SysTDyP à partir duquel nous définissons, dans la section 5.1.3, le système INDISS. Nous considérons, par la suite, dans la section 5.1.4, plusieurs scénarios d'utilisation de ce dernier dans le contexte d'un *environnement ubiquitaire* afin de mettre en perspective son fonctionnement. Enfin, dans la section 5.1.5, nous présentons l'implémentation du prototype INDISS et évaluons ses performances

5.1.1 Détermination des similarités entre divers SDPs

Un SDP fonctionne suivant un *mode centralisé*, un *mode distribué*, ou un *mode mixte* [51] selon que la découverte de services dans une *architecture de services ubiquitaires* est *centralisée, distribuée, ou mixte* (*centralisée* et *distribuée*). Comme introduit dans le chapitre 2, lorsque la découverte de services est *centralisée*, la découverte se fait par l'intermédiaire d'un annuaire *via* un protocole de communication en mode *point-à-point*. En revanche, lorsqu'elle est *décentralisée*, la découverte se fait directement entre les clients et les services à l'aide d'un protocole *multicast* basé sur le principe des groupes de diffusion afin de réaliser la découverte et la localisation de services suivant un modèle qui peut être *actif*, *passif* ou *mixte* (*actif* et *passif*). Avec un *modèle actif*, les clients diffusent périodiquement des requêtes *multicast* afin de découvrir les services distants. Ces derniers, quant à eux, attendent passivement la réception de ces requêtes et répondent aux clients qui les sollicitent par un message *unicast* annonçant leur présence/localisation. En revanche, avec un *modèle passif*, les clients se mettent en attente passive d'annonces *multicast* en provenance des services. Ces derniers doivent périodiquement diffuser des notifications indiquant leur présence/localisation dans l'environnement.

Le protocole SLP [50], implémente les modes de fonctionnement *centralisé* et *distribué* [181]. Dans le *mode décentralisé*, la découverte se fait selon le *modèle mixte*, les requêtes des clients (resp. les annonces des services) sont diffusées dans le groupe *multicast* identifié par l'adresse IP de classe D [156] suivante 239.255.255.253 :427, tandis que dans le *mode centralisé*, les clients et les services interagissent avec l'annuaire selon une communication *point-à-point* au dessus de TCP ou d'UDP. Le protocole UPnP/SSDP [157] fonctionne uniquement avec le *mode décentralisé*. En revanche, il implémente les deux modèles de découverte *actif* et *passif* : les services annoncent leur présence et les clients diffusent leurs requêtes de recherche dans le groupe *multicast* 239.255.255.250:1900. Le SDP de ZeroConf [161], [162] a un comportement similaire à celui d'UPnP, et utilise le groupe *multicast* 224.0.0.251:5353. En revanche, le protocole Jini [149], [150] implémente

exclusivement le *mode centralisé*, et la découverte de services se fait à l'aide d'un protocole de communication de type RPC comme RMI. Toutefois, Jini offre à ses clients et services l'opportunité de découvrir dynamiquement l'annuaire, (et uniquement ce dernier) *via* un SDP qui lui est dédié à l'aide du groupe *multicast* 224.0.1.85:4160. Un annuaire, comme UDDI [23], peut également être considéré comme un SDP, comparable à Jini, dont la découverte de services est *centralisée* et dont le protocole est de type RPC (par exemple, SOAP pour UDDI). Cependant, contrairement à Jini, les clients et les services doivent être préconfigurés afin de connaître à l'avance la localisation de l'annuaire. Enfin, le protocole JESA [159], dont l'objectif est de fournir des fonctionnalités similaires à celle de Jini indépendamment de la présence d'un annuaire, est quant à lui opérationnel aussi bien en *mode centralisé*, qu'en *mode décentralisé* selon un *modèle mixte*.

Les différents modes de fonctionnement, *centralisé*, *décentralisé actif*, *décentralisé passif*, ou *décentralisé mixte (*c'est-à-dire *mode décentralisé* selon respectivement le *modèle actif, passif, ou mixte)*, implémentés par un SDP conditionnent les différents types de requêtes/réponses que ce dernier est susceptible de générer. A partir des travaux réalisés par les auteurs des références [178], [179], [180], [181] on déduit la classification suivante :

- Avec le *mode centralisé*, les SDPs requièrent l'utilisation de requêtes/réponses permettant l'enregistrement et la découverte de services auprès d'un annuaire suivant un protocole *point-à-point*.

- Avec le *mode décentralisé actif*, les SDPs définissent les requêtes/réponses permettant aux clients de découvrir des services distants suivant un protocole *multicast*.

- Avec le *mode décentralisé passif*, les services distants diffusent deux types d'annonces : celles émises pour notifier leur présence, et celles émises pour indiquer leur intention de disparaître.

Un autre aspect important de la découverte de services est le format/contenu des requêtes/réponses échangées. En effet, dans le *mode centralisé* et *décentralisé actif*, pour qu'un client découvre un service distant, il lui faut générer une requête contenant une description partielle ou complète de l'*interface* du service recherché qui soit compatible avec celle des services distants déployés dans l'*environnement ubiquitaire*. Dans le *mode décentralisé passif*, la description de l'*interface* du service incluse dans les annonces des services doit être compatible à celle qu'attendent leurs clients potentiels.

Avec le protocole SLP, la description des services se fait en employant des modèles de service (*service templates*) [158] standardisés par l'IETF (*Internet Engineering Task Force*) et publiés par l'IANA (*Internet Assigned Number Authority*). Les modèles spécifient le type, le nom, la version, la langue, une description, et une liste

d'attributs d'un service. Un attribut est un couple (clé, valeur), la clé étant le type de l'attribut, et la valeur pouvant prendre la forme d'un entier, d'une chaîne de caractères, d'un Booléen, ou de données binaires. Lorsqu'un service SLP s'enregistre auprès d'un annuaire, il intègre dans sa requête son URL et sa description en attribuant, suivant ses caractéristiques, des valeurs aux attributs du modèle lui correspondant. Quant aux clients SLP, ils découvrent la présence/localisation des services distants en intégrant dans leurs requêtes des prédicats sur les valeurs d'un certain nombre d'attributs correspondant aux modèles des services qu'ils recherchent. L'enregistrement et la découverte de services Jini (resp. UDDI) fonctionnent de façon analogue à SLP ; cependant, la description des services est réalisée par le biais d'*interface* Java compilée (resp. de documents WSDL).

Avec le protocole UPnP/SSDP, qui fonctionne sans annuaire, les services s'annoncent en intégrant dans leurs messages : (i) une URI (*Universal Resource Identifier*) afin de s'identifier de façon unique et (ii) une URL non pas du service mais d'un document XML contenant la description complète du service. Si le service qui s'annonce correspond à celui que le client recherche, ce dernier doit rapatrier la description complète du service afin d'obtenir entre autres, l'URL et les attributs du service distant. Comme pour SLP, les clients UPnP recherchent des services distants en intégrant dans leur requête des prédicats sur leur URI.

Le SDP de ZeroConf réutilise les messages définis dans le protocole DNS pour annoncer et rechercher des services. En effet, ce SDP est une adaptation du protocole DNS (*Domain Name System*) afin de découvrir la présence de services distants *via* un protocole *multicast* plutôt qu'*unicast* afin d'éviter l'utilisation d'un serveur DNS. Ainsi, les services s'annoncent *via* des messages DNS incluant le nom, le type, le protocole d'accès, la description et l'adresse IP du service. Les clients découvrent, comme précédemment, les services en intégrant dans leur requête des prédicats sur leur description.

Un point essentiel qu'il est nécessaire d'évoquer ici est la notion de compatibilité entre les descriptions des services recherchés ou annoncés inclus dans les messages des différents SDPs. En effet, nous distinguons deux types de compatibilité : la *compatibilité syntaxique* et *la compatibilité sémantique*. La description d'*interfaces* de services selon des langages de description différents sont compatibles syntaxiquement s'il est possible de résoudre l'hétérogénéité de leur format, cependant cela n'implique pas nécessairement leur *comptabilité sémantique*. En effet, deux descriptions de services distincts, syntaxiquement équivalentes, peuvent avoir une signification différente. Dans ce document, nous considérons que les *interfaces* des services d'une *architecture de services ubiquitaires* sont standardisées bien qu'elles puissent être décrites par des langages de description différents. En d'autres termes, nous considérons que la *compatibilité syntaxique* implique systématiquement la *compatibilité sémantique* afin de s'abstraire du défi scientifique que représente cette

dernière[10]. En effet, l'interprétation sémantique de la description d'un service se réalise au niveau applicatif plutôt qu'au niveau de l'intergiciel [175], et nous nous intéressons exclusivement à l'hétérogénéité des intergiciels et non pas de celle des applications. Le travail des auteurs des références [173], [177], [182] sur la *compatibilité sémantique* des services d'un *environnement ubiquitaire* est complémentaire à nos travaux [175] et s'intègre parfaitement au système de traduction SysTDyP comme présenté dans section 5.3.

Par ailleurs, les incompatibilités entre des SDPs fonctionnant selon le *mode centralisé*, relèvent davantage des problèmes d'incompatibilités entre protocoles de communication étant donné que leurs protocoles de découverte utilisés pour interagir avec un annuaire est de type RPC (à l'exception des annuaires SLP). Par conséquent, les incompatibilités entre SDPs fonctionnant selon le *mode centralisé* sont résolues à l'aide de NEMESYS présenté dans la section 5.2, qui considère un annuaire comme un service quelconque accédé *via* un protocole de communication, plutôt qu'à l'aide d'INDISS qui se retrouve spécialisé pour les SDPs fonctionnant selon le *mode décentralisé*, c'est-à-dire pour les SDPs *multicast*.

Enfin, les SDPs fonctionnant selon le mode *décentralisé passif* opèrent en général également selon un mode *décentralisé actif* [178], [181]. Par conséquent, quel que soient leurs modèles, les SDPs partagent dans le pire des cas seulement deux types de messages en commun, c'est-à-dire les requêtes de découverte de services et leurs réponses associées. Dans le meilleur des cas, les SDPs partagent, en plus de ces messages, ceux correspondants aux annonces générées par les services distants pour notifier leur présence ou leur disparition. Bien que le format de ces messages diffère suivant les différents SDPs, la description des services recherchés ou annoncés, qu'ils véhiculent, contient toujours le nom, le type, l'URL et les attributs d'un service. En ce qui concerne le comportement des différents SDPs, il reste relativement simple et correspond soit à un paradigme d'interaction de type requête/réponse asynchrone, soit à un paradigme d'interaction de type diffusion.

Les différentes similitudes entre les SDPs fonctionnant selon le *mode décentralisé* étant dorénavant établies, nous sommes en mesure, dans la section suivante, d'utiliser le connecteur C-UNIV-Réseau pour résoudre leur incompatibilité et en particulier de raffiner l'architecture logicielle du système SysTDyP.

5.1.2 Raffinement du système de traduction

Nous définissons dans un premier temps le fonctionnement du composant logiciel moniteur, notamment sa stratégie pour détecter les SDPs en cours d'utilisation dans l'environnement. Dans un second temps, nous réalisons une décomposition en couche des SDPs afin de déterminer les différentes unités à composer. Enfin, nous déterminons quels sont les messages sémantiquement équivalents et les règles de

[10] L'interprétation sémantique adresse une problèmatique qui, bien que complémentaire à nos travaux, ne fait pas parti du sujet de ce document

coordination que tous les SDPs pris en charge par le système SysTDyP ont l'obligation de partager (c'est-à-dire quel doit être le *protocole image* commun des SDPs). En d'autres termes, nous définissons les critères minimums que les SDPs doivent respecter pour être dynamiquement traduits. Ces critères sont exprimés sous la forme d'un processus FSP qui correspond au processus Q défini dans le chapitre précédent.

Le composant moniteur.

Lorsque les clients, et les services d'un même SDP n'ont aucune connaissance de leur localisation mutuelle, ils se découvrent dynamiquement en diffusant des messages à destination d'un groupe *multicast* identifié par une adresse *multicast* propre à leur SDP. Une adresse *multicast* est composée d'une adresse IP de classe D associée à un port UDP/TCP. Chacune de ces adresses, délivrée par une autorité d'assignation des numéros Internet (IANA pour *Internet Assigned Numbers Authority*), identifie sans aucune ambigüité un seul et unique SDP. L'adresse IP et le port UDP/TCP forment alors une paire unique qui peut être interprétée comme un tag d'identification permanent d'un SDP. Ainsi, en souscrivant simultanément à plusieurs groupes *multicast*, le composant logiciel moniteur est capable de déterminer les SDPs utilisés dans son environnement suivant le groupe dans lequel ils sont générés. La découverte des SDPs utilisés dans l'environnement se fait par une écoute passive aux ports UDP/TCP des différents groupes *multicast* et dépend des annonces ou des requêtes *multicast* générées respectivement par les services ou les clients. La Figure 32 décrit le fonctionnement du composant logiciel moniteur pour détecter les protocoles de découvertes de services quel que soit leur modèle actif ou passif en l'absence d'annuaire. Le composant logiciel moniteur, qui peut être déployé aussi bien du coté du client, du service, ou d'une passerelle, s'inscrit aux groupes *multicast* correspondant aux protocoles SDP1 et SDP2 et écoute passivement à leur ports UDP/TCP respectifs. Ces deux protocoles sont identifiés par leurs adresses *multicast* respectives. Par ailleurs, le protocole SDP1 suit un modèle de découverte de services actif, tandis que le protocole SDP2 suit un modèle passif.

Deux cas de figures se présentent alors :

- Avec le protocole SDP1 (*modèle actif*) (Figure 32A), ce sont les clients qui émettent des requêtes *multicast* au groupe *multicast* SDP1 pour découvrir les services présents dans leur voisinage. Le composant logiciel moniteur, faisant partie lui aussi du groupe *multicast* SDP1, reçoit les requêtes en provenance des clients et est ainsi capable de détecter la présence du protocole SDP1 dans l'environnement. Quel que soit leur contenu, dès que des messages arrivent à destination de l'adresse *multicast* du protocole SDP1, le moniteur identifie le protocole de découverte.

- Avec le protocole SDP2 (*modèle passif*) (Figure 32B), ce sont les services qui diffusent des messages annonçant leur présence/localisation aux membres du

groupe *multicast* spécifique au protocole SDP2. De façon similaire au protocole SDP1, dès que des messages arrivent à destination de l'adresse *multicast* du protocole SDP2, le composant logiciel moniteur détecte la présence d'un protocole SDP2 dans l'environnement.

Le composant logiciel moniteur est capable de déterminer le SDP actuellement utilisé dans l'environnement uniquement en fonction de l'arrivée de messages aux ports des groupes *multicast* dont il est passivement à l'écoute (Figure 32C). La détection des SDPs se fait sans effectuer de calculs, d'interprétation ou de transformations de données. Le modèle du protocole SDP n'a aucune influence sur la détection réalisée par le moniteur. Cette dernière est indépendante du contenu des messages et ne dépend que de l'arrivée de données brutes aux adresses *multicast* des SDPs. Par conséquent, le coût de la détection en termes de ressources (bande-passante, mémoire et CPU) est réduit au minimum.

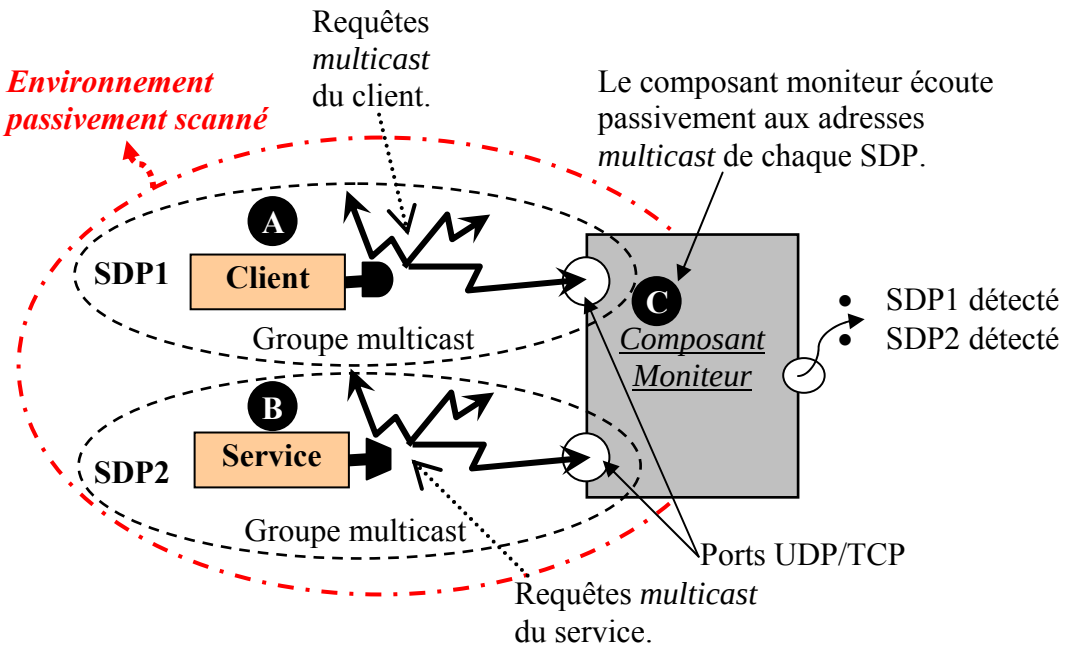


Figure 32. Detection de SDP passif et actif *via* le composant *moniteur*

La détection des SDPs utilisés dans un environnement ambiant n'est qu'une première étape vers la résolution des incompatibilités entre SDPs. Le flux de données, correspondant aux messages réseaux qui atteignent les ports UDP/TCP du composant logiciel moniteur, reste à traduire. La traduction se fait en appliquant les concepts présentés précédemment, c'est-à-dire *via* la composition d'unités de protocoles différents.

Décomposition en couches des SDPs.

Comparativement au modèle OSI, qui décompose les protocoles réseaux en sept couches, et à l'analyse des SDPs de la section 5.1.1, on modélise les SDPs en les décomposant en cinq couches comme illustré dans la Figure 33. Le *support physique* (correspondant aux couches OSI *physique* et *liaison*) détermine le type de réseau sur lequel les entités communicantes peuvent transmettre de l'information. Tout comme le modèle OSI, la couche *réseau* détermine comment sont transférées les données entre les différentes entités réseaux tandis que la couche *transport* spécifie comment est gérée la délivrance des messages de bout en bout. La couche OSI *session* est inutile dans le contexte de la découverte de services puisque les différentes entités communicantes sont amenées à générer aléatoirement des requêtes de découverte de services sans avoir l'obligation d'établir préalablement une connexion aux services distants. Enfin, la couche *application* (correspondant aux couches OSI *présentation* et *application*) fournit aux applications d'une entité réseau une *interface* regroupant les fonctions permettant de réaliser des requêtes de découverte de services. Ces fonctions sont implémentées par la couche *découverte* sous-jacente.

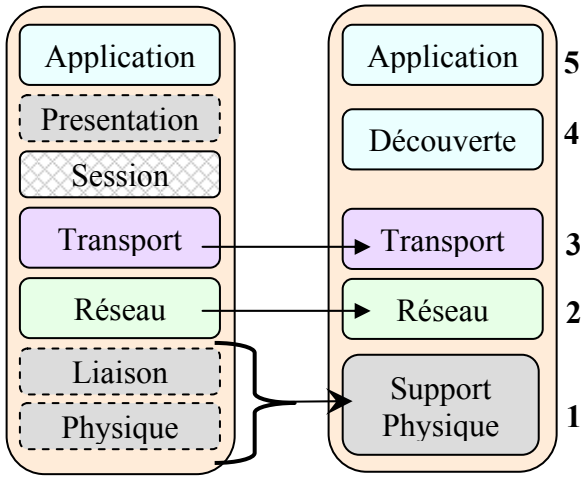


Figure 33. Décomposition en couches des SDPs

Les SDPs partagent toujours un certain nombre de couches identiques réduisant ainsi le nombre d'unités impliquées dans la traduction. Par conséquent, il n'y a pas autant d'unités à composer qu'il existe de couches dans une pile de protocoles. Par exemple, les SDPs fondés sur TCP/IP ont leur couche physique, réseau et transport identiques. Par ailleurs, les *objets communicants* de l'*environnement ubiquitaire* requièrent, au minimum, l'utilisation d'un même support physique pour s'échanger des données. Pour les SDPs basés sur TCP/IP, les couches identiques ne nécessitant pas de traduction, le connecteur C-UNIV-Réseau ne se compose que d'un seul connecteur C-UNIV dédié à la quatrième couche. Le système SysTDyP adapté pour les SDPs n'effectue alors qu'une composition horizontale d'unités (voir Figure 34). Par

exemple, comme indiqué dans la Figure 34A, la résolution des incompatibilités entre les SDPs SLP et UPnP consiste à composer une unité SLP avec une unité UPnP. Les différentes unités, dynamiquement composées en fonction de la détection effectuée par le *moniteur*, communiquent à l'aide du *protocole image* que nous définissons ci-après.

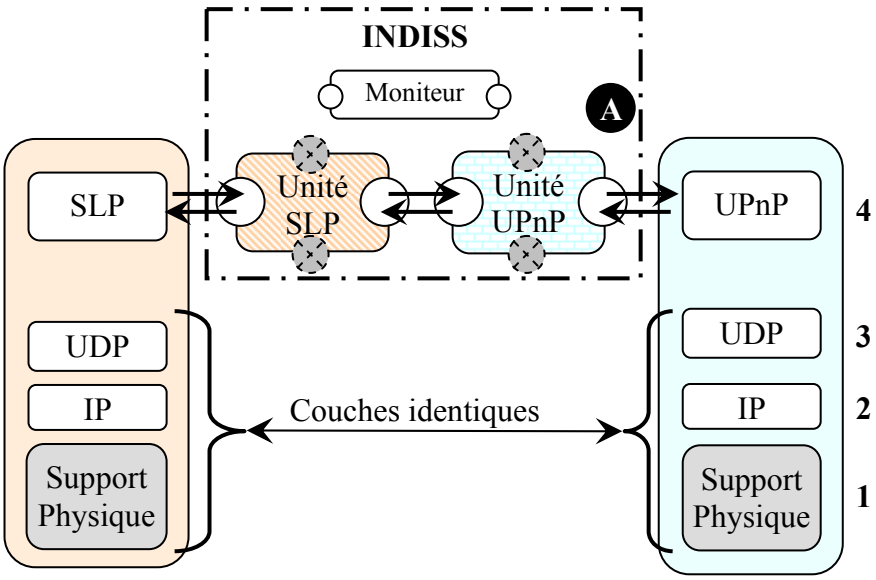


Figure 34. Composition d'unités horizontale

Protocole image commun aux SDPs.

Tel qu'introduit dans la section 5.1.1, quatre types de messages sont susceptibles d'être générés par les SDPs.

1. Les requêtes de découvertes de services.
2. Les réponses aux requêtes de découvertes de services.
3. Les annonces des services pour indiquer leur présence.
4. Les annonces des services pour indiquer leur disparition.

Chacun de ces messages contient les informations suivantes :

1. Les propriétés du protocole de transport sous-jacent, c'est à dire l'adresse du destinataire et de l'expéditeur du message, ainsi que le mode d'interaction *unicast* ou *multicast*.

2. Le type du message parmi les suivants : requête, réponse ou annonce.

3. La description du service recherché si le message est une requête, ou la description du service annoncé ou découvert si le message est respectivement une annonce ou une réponse.

Chacune de ces informations est projetée sous forme d'évènements *via* les analyseurs des unités du système SySTDyP afin de faire abstraction de l'hétérogénéité de leur format. On définit en conséquence un ensemble d'évènements S_{evt} qui se décompose en cinq sous ensembles. Soit :

$$S_{evt} = \text{"SDP\_Control\_Events"} \cup \text{"SDP\_NetworksEvents"} \cup \text{"SDP\_Service\_Events"} \cup \text{"SDP\_Requests\_Events"} \cup \text{"SDP\_Response\_Events"}$$

Avec,

1. "SDP_Control_Events" : cet ensemble d'évènements définit les évènements permettant de synchroniser les différents composants du système SySTDyP. Par exemple, il définit les évènements indiquant les SDPs détectés, le début et la fin d'un flux d'évènements extrait à partir d'un message reçu. Le flux d'évènements est le résultat de l'analyse syntaxique opérée par l'analyseur de l'unité qui reçoit le message.

2. "SDP_NetworksEvents" : cet ensemble regroupe les évènements qui décrivent les propriétés du protocole de transport sous jacent. Ces évènements indiquent par exemple : l'adresse source, l'adresse de destination, le mode de transmission (*unicast* ou *multicast*).

3. "SDP_Service_Events" : chacun de ces évènements définit les types des messages des SDPs.

4. "SDP_Requests_Events" : cet ensemble définit des évènements permettant de décrire le contenu des requêtes, et en particulier, les descriptions des services recherchés par les clients.

5. "SDP_Response_Events" : ces évènements, quant à eux, décrivent les différentes réponses possibles aux requêtes de découverte de services : un acquittement positif, négatif, la localisation du service recherché ou la description d'un service qui s'annonce.

Par ailleurs, un évènement est un n-uplet de la forme (Type,(clé$_i$: valeur$_i$)$_{i\in\{1,..,n\}}$) dont le premier élément correspond au type de l'évènement, et dont les éléments suivants définissent les attributs spécifiques de l'évènement. Un évènement décrit aussi bien le type d'un message que le contenu des données inclu dans le message.

Enfin, à chaque ensemble, correspond une série d'évènements obligatoires que les unités de protocoles prises en charge par SysTDyP doivent impérativement gérer pour assurer une *traduction minimale* (voir Tableau 1). Ces évènements obligatoires

correspondent, au niveau conceptuel, aux évènements du processus Q (voir section 4.2) définissant les critères minimum pour garantir une traduction valide.

Ensemble d'évènements	évènements	Sémantique
SDP Control Events	sdp.c.start sdp.c.stop sdp.c.sdp.type	Reception d'un message Fin de reception Type du SDP
SDP Network Events	sdp.net.unicast sdp.net.multicast sdp.net.source.addr sdp.net.dest.addr	Message unicast Message multicast Adresse de l'émetteur Adresse du destinataire
SDP Service Events	sdp.service.request sdp.service.response sdp.service.alive sdp.service.byebye	Recherche de service Reponse à une recherche Annonce d'existence Annonce de disparition
SDP Request Events	sdp.req.name sdp.req.lang sdp.req.type sdp.req.attr	Nom du service Langage du service Type du service Attribut du service
SDP Response Events	sdp.res.ok sdp.res.err sdp.res.name sdp.res.ttl sdp.res.url sdp.res.type sdp.res.attr	Service découvert Service non découvert Nom du service ttl entre deux réponses Url du service Type du service découvert Attribut du service découvert

Tableau 1. Evènements obligatoires

Par ailleurs, il est également nécessaire de définir les règles de coordination obligatoires que doivent respecter les SDPs pris en charge par SysTDyP. Les SDPs implémentant le *modèle actif*, *passif* ou *mixte* il peut être utile de s'assurer que l'un de ces modèles est toujours au moins respecté. Il est intéressant de noter que les évènements décrivant aussi bien le type des messages que leur contenu, il est possible d'établir à la fois des règles sur le type des messages échangés et sur leur contenu respectif. Par exemple, lorsque le modèle est actif, la réponse qui suit la requête de recherche d'un client doit toujours contenir l'URL du service recherché. Ainsi, on peut définir avec précision le comportement minimal à respecter pour garantir une traduction valide. On exprime ce comportement à l'aide de l'algèbre de processus FSP :

Q = sdp.c.start → message,
message = (sdp.net.unicast | sdp.net.multicast |
sdp.net.source.addr | sdp.net.dest.addr) → message | sdp.service.request →
request | sdp.service.alive → alive | sdp.service.response → response,
request = (sdp.req.name | sdp.req.type |sdp.req.lang |sdp.req.attr) →request
| sdp.stop→ Q,
response = sdp.res.ok → sdp.res.url → sdp.stop → Q,
alive = (sdp.res.name | sdp.res.type |sdp.res.url |sdp.res.attr) →alive
| sdp.stop→ Q,

A partir du processus FSP Q on obtient aisément un automate à état fini que toute unité du système SysTDyP doit inclure dans son propre automate.

Par ailleurs, bien que l'analyseur et le composeur de chaque unité doivent respectivement comprendre les évènements de S_{evt}, ils peuvent aussi produire et interpréter des évènements additionnels relatifs aux fonctionnalités avancées du protocole pour le lequel ils ont été conçus. La génération d'évènements n'appartenant pas à l'ensemble S_{evt}, n'altère pas le fonctionnement de la composition d'unités de protocoles différents. En effet, les composeurs de chaque unité ignorent les évènements additionnels inconnus. Par conséquent, les évènements de S_{evt} permettent aux SDPs les plus sophistiqués d'utiliser leur fonctionnalité avancée sans pour autant être incompris des SDPs plus basiques et *vice versa* tant que les critères minimums représentés par le processus Q sont respectés. Par exemple, bien que SLP soit un SDP plus rudimentaire qu'UPnP, il est toujours possible de trouver un service SLP à partir d'une requête UPnP. Le composeur d'une unité SLP ignore tout simplement les évènements spécifiques à UPnP. D'autre part, un composeur d'une unité JESA peut comprendre un certain nombre d'évènements spécifiques à UPnP. Le système SysTDyP, comme indiqué dans la section 4.4, est facilement extensible ; il est possible d'introduire de nouveaux évènements *a posteriori* pour augmenter la qualité de la traduction dynamique ou de nouveaux SDPs sans déclencher une cascade de modifications des composants du système. Cette flexibilité du système provient de son style architectural évènementiel.

Dans la section suivante, nous présentons le système INDISS qui est une implémentation du raffinement de l'architecture logicielle du système SysTDyP pour les SDPs.

5.1.3 Le système INDISS

Le système INDISS se compose du composant logiciel moniteur (présenté dans la section 5.1.2) et d'un ensemble d'unités, toutes dédiées à un SDP différent. L'architecture logicielle du système est décrite dans un fichier de configuration à partir duquel est généré le système INDISS. Ce fichier de configuration se définit en termes d'unités de protocoles à instancier et s'organise en deux parties (Figure 35).

La première partie (Figure 35A) précise : (i) les différentes unités que le système est susceptible d'instancier et de composer au cours de son exécution et (ii) les différents groupes *multicasts* auquel le moniteur doit s'abonner pour détecter les SDPs en cours d'utilisation. La deuxième partie (Figure 35B) définit la structure interne de chacune des unités du système. Cela consiste à indiquer les analyseurs et les composeurs susceptibles d'être connectés à chacune des unités, ainsi que leur automate à état fini respectif (AEF). L'AEF définit le comportement d'une unité, c'est à dire les règles de synchronisation des évènements qui transitent par l'unité (voir Figure 35C). L'AEF doit inclure l'automate correspondant au processus Q définissant les règles de coordination obligatoires. Il est important de noter que l'analyseur ou le composeur d'une unité ne lui est pas nécessairement spécifique, l'un ou l'autre peut être réutilisé dans d'autres unités. C'est l'AEF de ces dernières qui spécialise un analyseur et un composeur vis à vis d'un protocole particulier. Par exemple, au niveau de l'implémentation, l'analyseur et/ou le composeur HTTP de l'unité UPnP peut être réutilisé par toute autre unité dédiée à un autre protocole différent, basé lui aussi sur HTTP.

```
System INDISS = {
// Définition de la configuration globale du système
Component Moniteur = {
ScanGroup = { 239.255.255.253 :427,
              239.255.255.250:1900,
              224.0.0.251:5353,
              …
        }
Component Unit SLP(port=427) ;
Component Unit UPnP(port=1900) ;
Component Unit ZeroConf(port=5353) ;
}
// Définition des différents unités et de leur automates respctifs
Component Unit UPnP = {
        setAEF(aef,UPnP-AEF)
        AddParser(component, SSDP) ;
        AddComposer(component, SSDP) ;
        AddParser(component, HTTP) ;
        AddComposer(component, HTTP) ;
        … } …
AEF UPnP-AEF = {
        AddTuple(CurrentState, triggers, condition-guards, NewState, actions)
        …
        }
…}//fin
```

A

B

C

Figure 35. Exemple de spécification d'unités

Une fois le système INDISS instancié à partir de son fichier de configuration, la traduction dynamique de SDPs devient fonctionnelle (Figure 36 B). L'architecture interne du système évolue au cours du temps suivant les SDPs en cours d'utilisation qui peuvent changer en fonction de l'arrivée et/ou le départ aléatoire de nouveaux *objets communicants* de l'*environnement ubiquitaire* (Figure 36 C, D, E). L'architecture du système n'est donc pas figée dès sa spécification. En effet, si son fichier de configuration indique les unités dont le système dispose (Figure 35A), il ne décrit pas comment et/ou quand les unités sont composées. La composition d'unités est réalisée dynamiquement en fonction du contexte d'exécution en fonction : (i) des applications embarquées sur le périphérique sur lequel INDISS est déployé, (ii) des SDPs en cours d'utilisation dans l'environnement.

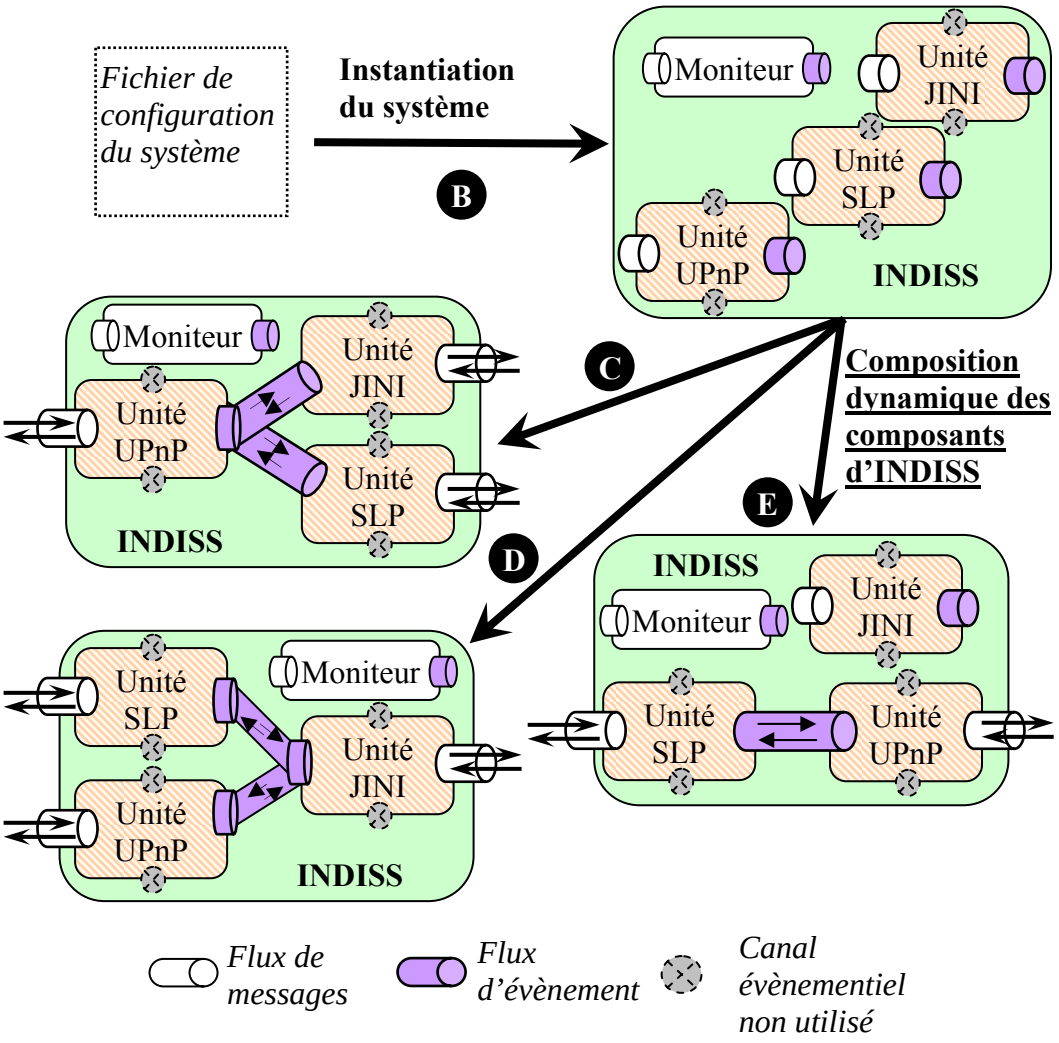


Figure 36. Evolution du système INDISS

Par ailleurs, le système INDISS se déploie et opère au dessus du système d'exploitation et en dessous de l'intergiciel (Figure 37). En effet, INDISS réalise

uniquement une traduction dynamique du protocole de la couche 4 de la pile des SDPs. Les couches transport et réseau étant implémentées à l'aide de pilotes intégrés au cœur du système d'exploitation, INDISS intercepte les messages en provenance du réseau *via* le système d'exploitation, les traduit et les redirige vers l'intergiciel sans qu'il soit nécessaire de modifier et/ou de redévelopper les applications déjà existantes. La présence d'INDISS n'est nécessaire que sur une des entités impliquées dans l'interaction, c'est-à-dire sur l'hôte du client ou du service. INDISS peut éventuellement être déployé sur une passerelle au profit de l'ensemble des *objets communicants* de l'*environnement ubiquitaire*.

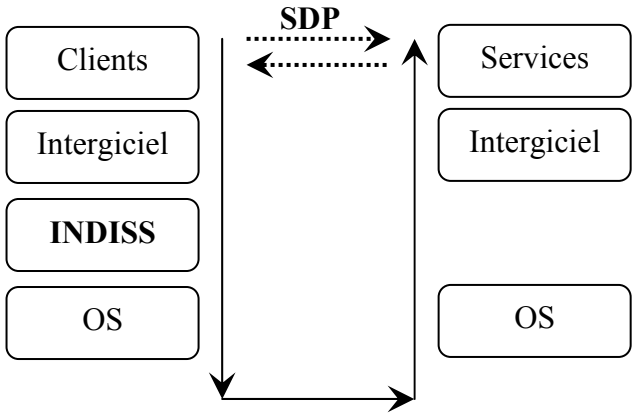


Figure 37. Localisation du système INDISS

Le système INDISS étant défini, nous étudions dans la section suivante le comportement d'INDISS dans un *environnement ubiquitaire* selon différents scénarios d'utilisation en fonction de l'hôte sur lequel INDISS est déployé (celui d'un consommateur de service, ou d'un fournisseur de service) ainsi que du *modèle actif*, *passif,* ou *mixte* des SDPs utilisés dans l'environnement.

5.1.4 Scénario d'utilisation

Avec INDISS, les clients et les services ont toujours l'illusion qu'ils utilisent un même SDP. En réalité, les SDPs sont hétérogènes et fonctionnent selon différent modèles (*actif*, *passif*, *mixte*), INDISS peut donc être localisé aussi bien du côté du client, du service que sur une passerelle. Par conséquent, nous devons considérer plusieurs scénarios d'évaluation et déterminer pour chacun d'entre eux les performances du système, en particulier son impact sur la consommation de ressources qui peuvent être limitées sur certains périphériques sur lequel INDISS peut être déployé.

En fonction du modèle des SDPs utilisés respectivement par les clients et par les services, on distingue quatre scénarios. Pour chacun de ces scénarios, on considère deux cas d'utilisation d'INDISS selon qu'il soit hébergé sur l'hôte du client ou sur l'hôte du service.

Scénario 1 : les clients et les services utilisent un SDP *décentralisé passif*

Dans ce scénario, les SDPs des clients et des services fonctionnent suivant un *modèle passif*. Les services diffusent alors périodiquement des annonces *multicast* tandis que les clients attendent passivement la réception de ces annonces. En d'autres termes, les services sont des annonceurs et les clients des récepteurs.

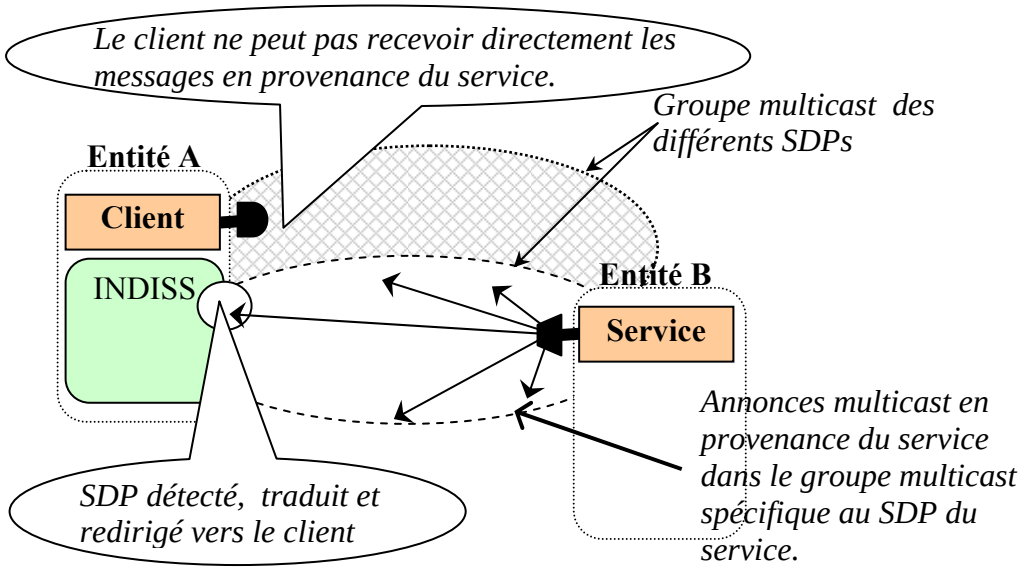


Figure 38. Découverte de services passive avec INDISS côté client

Dans ce contexte, si INDISS est embarqué du côté des clients, ces derniers sont en mesure de recevoir tous les messages générés par les services distants indépendamment de leur appartenance aux différents groupes *multicast* et de l'hétérogénéité du format de leurs messages (Figure 38). L'interopérabilité entre SDPs est alors garantie. En revanche, si INDISS est placé du côté des services, l'interopérabilité n'est plus garantie car, par définition du modèle passif, les clients sont en attente, et ne génèrent aucun message. Par conséquent, étant donné l'hétérogénéité des SDPs (différence de groupes *multicast*, et de formats de messages), les clients sont incapables de recevoir dans cette situation les annonces des services (voir Figure 39B).

Pour détecter et résoudre cette situation bloquante, on définit un seuil de trafic réseau en dessous duquel, INDISS intercepte les messages générés par les services locaux (c'est-à-dire des services de l'hôte sur lequel INDISS est déployé) afin de traduire les annonces de ces derniers vers tous les SDPs connus, c'est-à-dire les SDPs dont INDISS possède les unités (voir Figure 39A). Bien que cette *traduction multiple*, d'un SDP en plusieurs autres SDPs, illustre la flexibilité du système INDISS, elle n'est pas sans conséquence sur la consommation de ressources. La reconfiguration dynamique du système pour activer ce mode de traduction, requiert des ressources processeur conséquentes (puisqu'il faut instancier toutes les unités du système). La diffusion des messages traduits respectivement vers les groupes *multicasts* des

différents SDPs augmente également la consommation de la bande passante. Toutefois, l'*interopérabilité* entre SDPs est assurée sans pour autant saturer la bande passante, étant donné que ce mode de *traduction multiple* est activé uniquement lorsque le trafic réseau est faible (en dessous du seuil fixé).

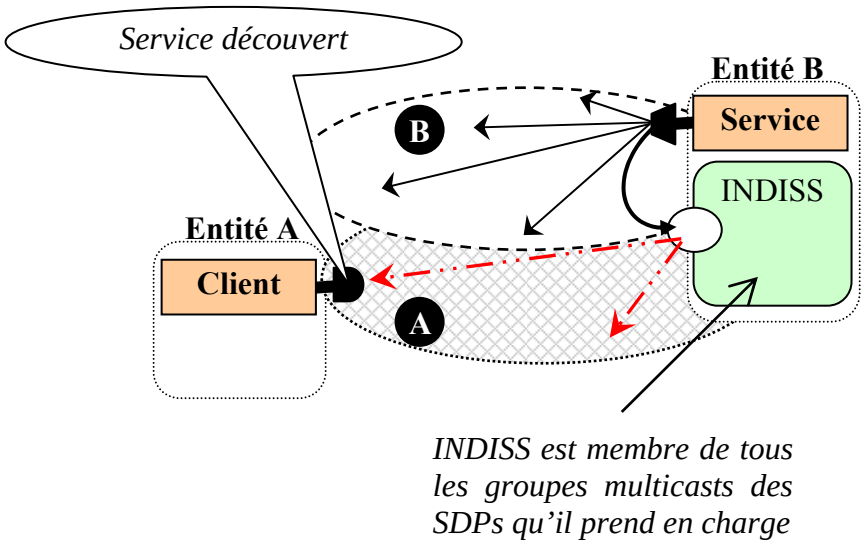


Figure 39. Découverte de services passive avec INDISS du côté du service

Scénario 2 : les clients et les services utilisent un SDP *décentralisé actif*

Dans ce scénario, les SDPs des clients et des services fonctionnent tous deux suivant un *modèle actif*. Contrairement au scénario précédent, cela signifie, cette fois ci, que les clients sont des émetteurs et les services des récepteurs. Afin d'optimiser la consommation de la bande passante et des ressources processeur, l'emplacement le plus adéquat du système INDISS se trouve sur l'hôte hébergeant les services de façon à recevoir les requêtes des clients. Dans le cas contraire, si INDISS est déployé du côté des clients, il doit intercepter les requêtes locales de ces derniers et les traduire vers tous les SDPs connus de sorte que les services distants non équipés d'INDISS puissent recevoir leurs requêtes. Dans ce contexte, l'interopérabilité entre SDPs, bien que fonctionnelle, peut ne pas se révéler suffisamment efficace en termes de consommation de ressources (particulièrement au niveau de la bande passante) pour des raisons similaires au scénario précédent à cause des conséquences de la *traduction multiple*. De manière générale, lorsque les clients et les services utilisent un même modèle de découverte de services, le meilleur emplacement d'INDISS se trouve sur l'hôte dont les applications se comportent comme des récepteurs.

Scénario 3 : les clients utilisent un SDP *décentralisé actif* tandis que les servicesutilisent un SDP *décentralisé passif*

La particularité de ce scénario résulte de l'utilisation d'un *modèle actif* de découverte de services pour les clients, et d'un *modèle passif* pour les services. Dans ce contexte, les clients et les services génèrent tous deux des requêtes, l'un pour faire des recherches, l'autre pour s'annoncer. Quel que soit l'emplacement d'INDISS, l'interopérabilité entre SDPs est toujours garantie et l'activation de la *traduction multiple* n'est à aucun moment nécessaire. (Figure 40).

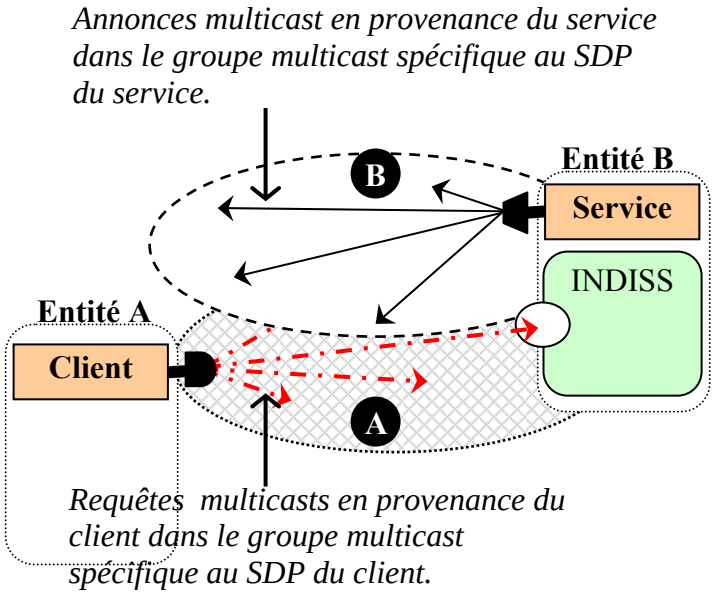


Figure 40. Découverte de services à la fois passive et active

L'interopérabilité n'est pas bidirectionnelle (Figure 40). Lorsqu'INDISS est déployé du côté du service, les annonces des services locaux ne sont pas traduites par défaut (Figure 40B). En particulier, les clients distants qui ne sont pas interopérables (c'est-à-dire non-équipés d'INDISS), ne sont pas capables de recevoir ces annonces. En revanche, les requêtes de découverte de services des clients sont traduites *via* INDISS pour les services de l'entité B (Figure 40A). De façon symétrique, lorsqu'INDISS est déployé du côté du client, seules les annonces des services sont traduites. Bien que l'interopérabilité ne soit pas parfaite, elle est suffisante pour découvrir les services de l'environnement. Par ailleurs, si le taux d'occupation de la bande passante est faible, la qualité de l'interopérabilité peut être améliorée dès lors qu'INDISS passe dans un mode de *traduction multiple*, comme illustré dans le premier et le second scénario.

Scénario 4 : les clients utilisent un SDP *décentralisé passif* tandis que les services utilisent un SDP *décentralisé actif*

Lorsque le modèle du SDP des clients est *passif* et celui des services est *actif*, les clients et les services sont tous deux des récepteurs : ils se retrouvent tous à attendre les messages des uns et des autres. Nous sommes de nouveau confrontés à une situation bloquante. Seulement, cette fois-ci, la reconfiguration dynamique d'INDISS ne peut pas résoudre le problème étant donné que ni les clients et ni les services n'amorcent la découverte de services. Notre objectif étant de fournir une interopérabilité entre SDPs sans altérer le comportement de leurs clients et de leurs services, en particulier de leur intergiciel respectif, il n'y aucune solution à ce problème. Néanmoins, cette situation particulière se produit rarement car dans la pratique les clients sont toujours capables de générer des requêtes de découverte de services (leur SDP suit en général un *modèle mixte*) [173], [177], [182].

Suivant la localisation d'INDISS, la bande passante requise pour assurer l'interopérabilité est plus ou moins importante et dépend de la nécessité d'activer ou non le mode de *traduction multiple* d'INDISS. Par ailleurs, si les ressources processeur et mémoire d'un ou de plusieurs périphériques sont trop limitées pour embarquer INDISS, il est toujours possible de déployer ce dernier sur une passerelle. Dans ce cas, INDISS capture tous les messages relatifs à la découverte de services de son environnement ambiant et les traduit vers tous les SDPs qu'il prend en charge *via* ses différentes unités. Bien évidemment, ce mode de *traduction multiple* génère du trafic additionnel et reste valide tant que la bande passante est suffisante (inférieure au seuil fixé).

Dans la section suivante, nous présentons notre implémentation d'un premier prototype du système INDISS composé d'unités UPnP et SLP afin de mettre en évidence les performances de notre approche selon les différents scénarios d'utilisation que nous venons de présenter.

5.1.5 Implémentation du prototype et évaluation

INDISS est développé en Java et peut donc être déployé sur toute sorte de périphériques à condition que ces derniers soient équipés d'une machine virtuelle Java (JVM pour *Java Virtual Machine*). Cependant, INDISS n'est pas pour autant dépendant de la plateforme Java et peut être développé aussi bien en C, ou tout autre langage de programmation plus proche du système d'exploitation, afin d'obtenir une vitesse d'exécution optimum et une empreinte mémoire réduite au minimum. Néanmoins, nous obtenons déjà en Java des résultats très encourageants. Comme décrit dans le Tableau 2, l'ensemble du système INDISS se résume à 39 classes Java et à 2910 lignes de code source non commentés (NCSS). Cela correspond à une taille du système égale à 218 ko. Cette taille inclut 125 ko pour l'unité UPnP, 49 ko pour celle de SLP et 44 ko pour le cœur du système.

Tailles requises du système *INDISS*				
	Size (KB)	**Classes**	**NCSS**	**Overhead**
Core framework	44	15	789	-
UPnP Unit	125	18	1515	-
SLP Unit	49	6	606	-
Total	**218**	**39**	**2910**	-
Tailles requises pour les différentes librairies SDPs				
OpenSlp	126	21	1361	-
Cyberlink UPnP	372	107	5887	-
Total	**498**	**128**	**7248**	-
Tailles requises pour fournir l'interopérabilité avec et sans INDISS				
Librairie SLP et UPNP + clients SLP & UPnP	514	-	-	-
Librairie et client UPnP + INDISS	598	-	-	**14%**
Librairie et client SLP + INDISS	352	-	-	**-31.5%**

Tableau 2. Tailles en Koctets des librairies SDPs et d'INDISS

Selon notre approche, pour être *interopérables*, les périphériques utilisant UPnP (resp. SLP) doivent héberger la librairie UPnP (resp. SLP), implémentant la pile de protocole UPnP (resp. SLP), en plus du système INDISS. Ce dernier traduit un protocole en un autre sans implémenter pour autant leurs piles de protocoles respectives. En d'autres termes, INDISS est un traducteur de protocoles indépendant des piles de protocoles embarquées ou non sur un périphérique. Comparativement, sans l'utilisation d'INDISS, un périphérique effectue une découverte de services *interopérable* :

1. S'il embarque autant de piles de protocoles que de protocoles existants.

2. S'il embarque autant de versions différentes d'une même application que de piles de protocoles différentes.

Implicitement, l'intergiciel non équipé d'INDISS requiert, d'une part, l'utilisation de la librairie UPnP en plus de celle de SLP, et d'autre part, le développement de ses applications en double exemplaires : l'une dédiée à SLP et l'autre dédiée à UPnP. Sans INDISS, indépendamment de la taille des applications, la taille minimum requise pour l'intergiciel est de 498 Ko (taille de la libraire UPnP plus celle de SLP). Avec INDISS, cette taille minimum est de 344 Ko si l'intergiciel est SLP (taille de INDISS plus celle de la librairie SLP) et de 590 Ko s'il est UPnP (taille de INDISS plus celle de la librairie UPnP). De prime abord, du point de vue de l'espace mémoire nécessaire, l'utilisation d'INDISS peut sembler désavantageuse dans certains cas (notamment avec UPNP). Cependant, l'espace mémoire disponible sur un

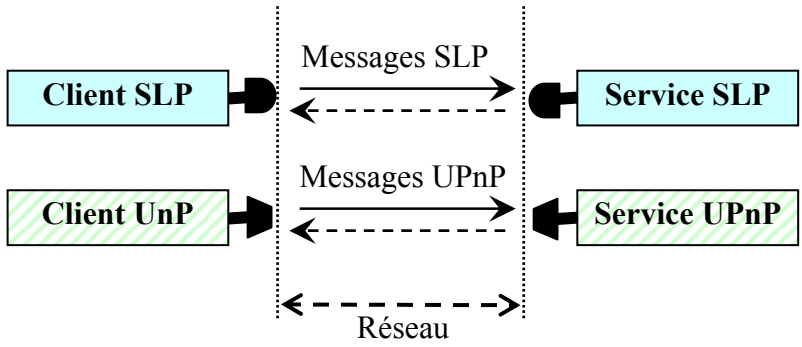
périphérique diminue proportionnellement avec le nombre d'applications/services hébergés. Ainsi, sans INDISS, cette taille augmente deux fois plus vite qu'avec INDISS. En effet, dans le premier cas, il faut ajouter deux versions d'une même application ou d'un même service (l'une pour SLP et l'autre pour UPnP) tandis que dans le deuxième cas, une seule version est nécessaire.

Ainsi, la taille supplémentaire potentielle induite par l'utilisation d'INDISS, par exemple lorsque les applications sont UPnP, disparaît avec le nombre de services hébergés. Etant donné qu'INDISS n'est qu'un traducteur, c'est-à-dire qu'il n'implémente pas les piles de protocoles qu'il traduit, un intergiciel doit toujours fournir à ses applications les librairies des SDPs dont elles dépendent. Dans le cas où ces applications dépendent de différents SDPs, l'intérêt d'INDISS peut être remis en cause, car cela implique que l'intergiciel fournisse les librairies correspondantes. Toutefois, il ne faut pas oublier que l'un des avantages majeurs d'INDISS est de réduire, dans tous les cas, de façon conséquente (par deux au minimum, et plus, en fonction du nombre d'unités embarquées dans une instance d'INDISS) le nombre d'applications à développer pour un intergiciel si ce dernier prend en charge plusieurs SDPs.

Pour évaluer les performances de la *traduction dynamique de protocoles* réalisée par INDISS, nous mesurons le temps nécessaire à INDISS pour traduire les requêtes d'un client dédié à un SDP et découvrir un service distant dépendant d'un autre SDP. Plus précisément, l'objectif de l'expérimentation consiste à comparer le temps d'attente d'un client pour obtenir une réponse d'un service distant lorsque tous deux utilisent un même SDP avec le temps d'attente obtenu lorsque le service distant utilise un SDP différent de celui du client. Dans le premier cas, le client et le service interagissent directement tandis que dans le deuxième cas, ils interagissent *via* INDISS qui traduit les requêtes et les réponses de l'un et de l'autre. Les expérimentations sont effectuées avec les protocoles SLP et UPnP. Le modèle de découverte de services de ces deux protocoles est aussi bien *passif* qu'*actif* : les clients et les services sont respectivement capables de générer des requêtes et de diffuser des annonces. De ce fait, il n'y a pas de situation de blocage en perspective et l'activation du mode de *traduction multiple* n'est pas nécessaire. Ainsi, quel que soit l'hôte sur lequel est déployé INDISS (sur celui hébergeant un client ou sur celui hébergeant un service) l'interopérabilité est assurée sans générer de trafic additionnel. Lors des expérimentations, nous n'avons donc pas évalué la bande passante étant donné qu'INDISS n'introduit pas de nouveaux SDPs et ne modifie pas les SDPs existants. Le trafic généré est, ni plus ni moins celui de clients (resp. de services) SLP et/ou UPnP.

Les mesures ont été réalisées sur des stations de travail équipées de 256 Mo de RAM et d'un processeur Intel PIV cadencé à 1.8GHz. Le système d'exploitation utilisé est un Linux Fedora Core 2 de Red Hat, la machine virtuelle java employée est une JDK1.4.2 de Sun et les outils de mesure proviennent de la suite Hyades de la fondation Eclipse. Les clients SLP (resp. UPnP) et les services SLP (resp. UPnP) sont hébergés sur différents hôtes interconnectés *via* un réseau LAN de 10Mo/s. Les

clients et les services SLP dépendent de la librairie OpenSlp[11] tandis que les clients et les services UPnP utilisent Cyberlink[12] pour Java. Les mesures relevées sont données en ms et correspondent à la médiane de trente tests consécutifs pour éviter une moyenne biaisée par une valeur anormalement élevée ou basse.



	SLP -> SLP	UPnP -> UPnP
Médiane (ms)	0.7	40

Figure 41. Découverte native entre les clients et les services.

La Figure 41 indique le temps d'attente d'un client SLP (resp. UPnP) pour découvrir un service distant SLP (resp. UPnP) suite à une requête de découverte de services. Avec SLP, la découverte s'effectue en 0.7 ms tandis qu'avec UPnP, elle s'effectue en 40 ms. Il apparaît clairement que SLP est un SDP plus efficace qu'UPnP. Ces mesures sont considérées comme des mesures de référence qui vont nous permettre d'interpréter par comparaison les résultats des expérimentations suivantes :

Expérience 1 : INDISS est déployé sur l'hôte du service

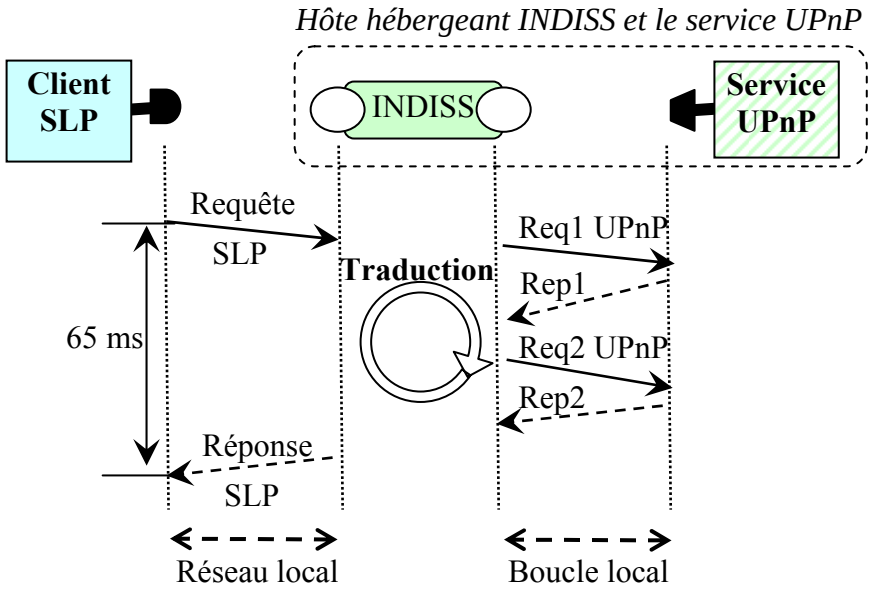
Le SDP du client est différent de celui du service et INDISS est déployé sur l'hôte du service. Ce service peut alors être découvert quel que soit le SDP utilisé par le client. On considère deux cas selon que le SDP utilisé par le client et le service est SLP ou UPnP:

- ***Cas 1 : le SDP du client est SLP, celui du service est UPnP***

Lorsque le SDP du client est SLP alors que celui du service est UPnP, la découverte se fait en 65 ms (Figure 42). Ce temps paraît particulièrement élevé comparativement au cas où il n'y pas de traduction (0.7 ms). Cela s'explique par le fait que le processus de traduction entre SLP et UPnP ne se fait pas message par message mais

[11] www.openslp.org/
[12] www.cybergarage.org/net/upnp/java/

se fait selon les évènements qu'il est possible d'extraire des messages. Par exemple, INDISS extrait de la requête SLP le fait que le client souhaite obtenir l'URL du service distant. Avec UPnP, l'obtention de cette URL se traduit par la génération de deux requêtes. La première permet d'obtenir l'URL de l'hôte ayant la description du service et non pas la localisation du service recherché. A partir de cet URL, la deuxième requête permet d'obtenir le contenu de la description du service recherché afin d'y extraire l'URL de sa localisation. Comparativement avec SLP, une seule requête est requise pour obtenir dans une même réponse, la description du service distant et son URL.



	Slp->[Slp-UPnP]
Médiane (ms)	**65**

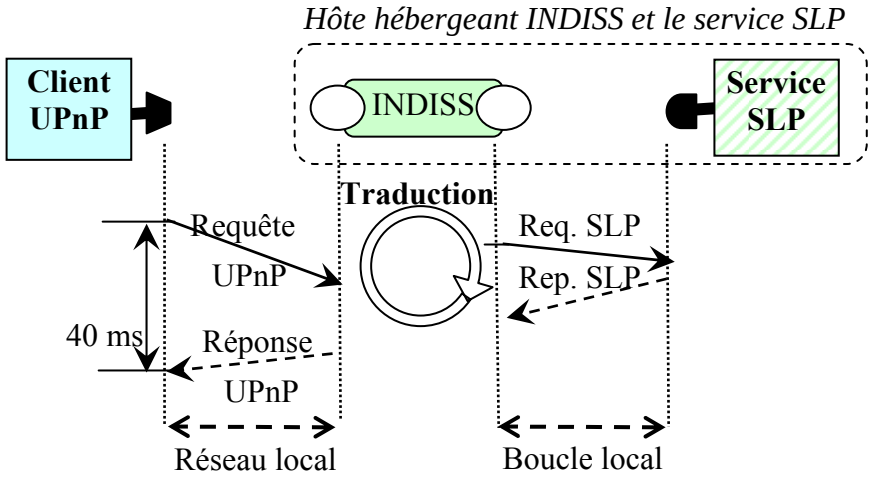
Figure 42. Client SLP, service UPnP et INDISS côté service

Par conséquent, INDISS a traduit la requête SLP en deux requêtes UPnP locales[13] pour obtenir auprès du service local suffisamment d'informations pour générer sur le réseau une réponse SLP valide correspondant à la requête SLP reçue. Plus précisément, cela signifie qu'INDISS a en 65 ms : (i) analysé la requête SLP, (ii) généré deux requêtes UPnP, (iii) attendu les réponses à ces requêtes, (iv) analysé les réponses UPnP, et (v) généré une réponse SLP (voir Figure 42). Localement, INDISS s'est comporté auprès du service UPnP comme un client UPnP qui a généré deux requêtes UPnP. Or, nativement, avec UPnP, l'obtention d'une réponse à la première requête UPnP est dejà de 40 ms. Dans ce contexte, le résultat obtenu par INDISS est raisonnable.

[13] Dans la pratique l'hôte du service est aussi celui ayant la description du service.

- ***Cas 2 : le SDP du client est UPnP, celui du service est SLP***

  Lorsque le SDP du client est UPnP alors que celui du service est SLP, la découverte se fait en 40 ms (voir Figure 43). On constate que cette valeur est égale à celle obtenue lorsque le client et le service sont tous deux UPnP. Plusieurs raisons expliquent ce résultat. Tout d'abord, la requête UPnP se traduit en une seule requête SLP. En effet, la réponse SLP correspondante véhicule suffisamment d'informations pour générer la réponse UPnP sans avoir à générer de requêtes SLP supplémentaires. Par ailleurs, le temps écoulé entre l'émission de la requête SLP et la réception de sa réponse devient *quasi* nul car cette dernière est diffusée en local. Enfin, le temps de traduction devient négligeable car la requête SLP est générée aussitôt qu'INDISS extrait suffisamment d'évènements de la requête UPnP. Par conséquent, la réponse à la requête SLP générée est reçue avant même que l'analyse de la requête UPnP ne se termine.



	UPnP->[UPnP -SLP]
Médiane (ms)	**40**

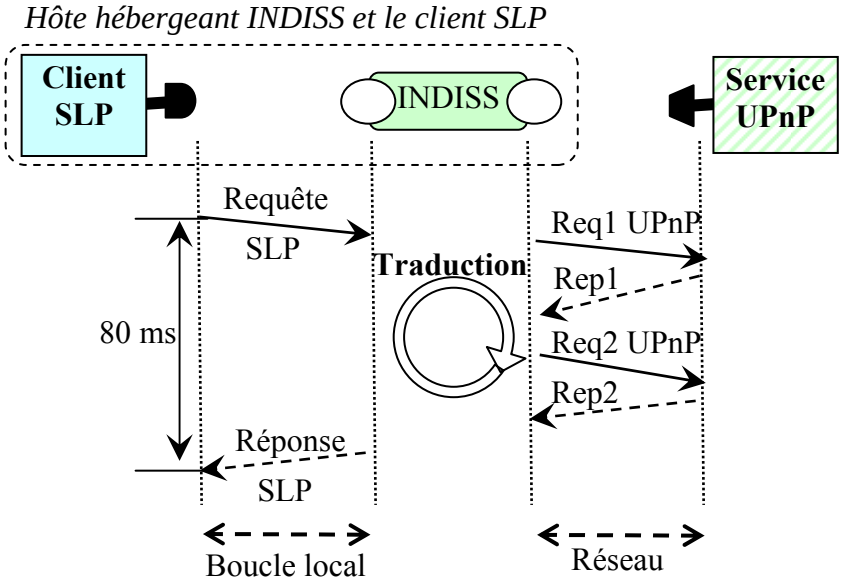
Figure 43. Client UPnP, service SLP et INDISS côté service

Expérience 2 : INDISS est déployé sur l'hôte du client

Le SDP du client est toujours différent de celui du service mais cette fois-ci INDISS est déployé sur l'hôte du client. Le client peut alors être découvert quel que soit le SDP utilisé par le service. On considère à nouveau deux cas selon la nature du SDP utilisé par le client et le service.

- ***Cas 1 : le SDP du client est SLP, celui du service est UPnP***

Lorsque le SDP du client est SLP alors que celui du service est UPnP, la découverte se fait en 80 ms contre 65 ms dans l'expérimentation précédente (voir Figure 44). Avec INDISS déployé sur l'hôte du client, le trafic UPnP résultant de la requête SLP est émis sur le réseau plutôt qu'en local. Ainsi, les temps de latences entre le moment où les requêtes UPnP sont émises et les réponses correspondantes sont reçues augmentent et se répercutent sur le temps de traduction d'INDISS.



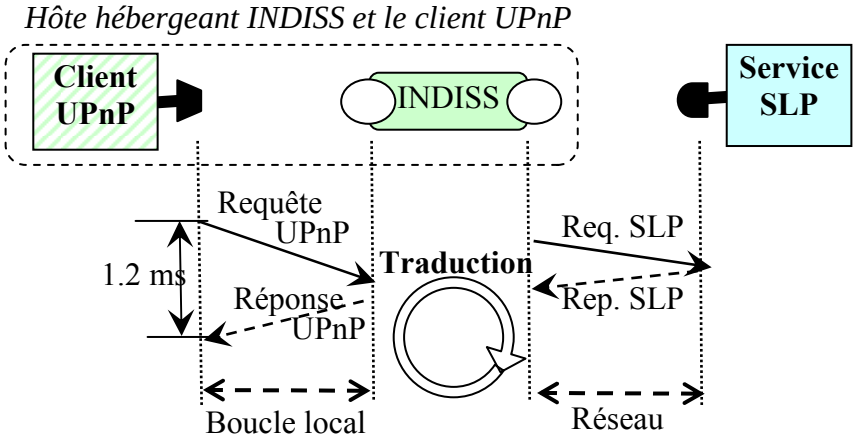
	SLP->[SLP –UpnP]
Médiane (ms)	**80**

Figure 44. Client SLP, service UPnP et INDISS côté client

- ***Cas 2 : le SDP du client est UPnP, celui du service est SLP***

Lorsque le SDP du client est UPnP alors que celui du service est SLP, la découverte se fait en 1.2 ms (voir Figure 45). Cette faible valeur s'explique, en partie, par le fait que le trafic UPnP est local alors que le trafic transmis sur le réseau est celui de SLP qui est particulièrement très peu gourmand en bande passante (ce qui implique une transmission particulièrement rapide sur le réseau (0.7ms)). De plus, les unités d'INDISS génèrent des requêtes et des réponses SLP et/ou UPnP dès qu'elles obtiennent suffisamment d'évènements de sorte qu'elles puissent être générées avant même que l'analyse des messages SLP et/ou UPnP reçus, effectuée par les analyseurs du système, ne soit terminée. Enfin, il est également important de noter qu'INDISS, étant indépendant des piles de protocoles embarquées par l'hôte sur lequel il est déployé, n'utilise pas les librairies UPnP et/ou SLP de l'intergiciel (Cyberlink et/ou

OpenSLp dans notre cas). Les unités de protocoles qu'utilisent INDISS sont optimisées et spécialisées pour la traduction, ce qui n'est pas le cas des librairies telles que celles de Cyberlink et d'OpenSlp.



	UPnP->[UPnP –SLP]
Médiane (ms)	**1.2**

Figure 45. Client UPnP, Service SLP, et INDISS du côté client

Ces deux expériences démontrent que la traduction dynamique de SDPs augmente de peu le temps d'attente d'un client pour découvrir un service distant. Bien évidemment une fois qu'un service distant est découvert, encore faut il que ce dernier utilise le même protocole de communication que le client qui vient de le découvrir. Dans la section suivante, nous introduisons NEMESYS qui est conçu, de façon similaire à INDISS, suivant les concepts introduit dans le chapitre 4. Le rôle de NEMESYS consiste à résoudre les incompatibilités des protocoles de communications entre les clients et les services.

5.2 NEMESYS

NEMESYS est une implémentation de l'architecture logicielle du système SySTDyP spécifiquement raffinée pour la résolution des incompatibilités entre protocoles de communication. NEMESYS est conçu comme INDISS sur les principes du connecteur C-UNIV-Réseau et par conséquent ne peut réaliser une *traduction dynamique de protocoles* que si les protocoles qu'il prend charge partagent des similitudes. De ce fait, nous avons conçu NEMESYS exclusivement pour des protocoles de communication hétérogènes de type RPC. Ces derniers sont encore largement utilisés dans les *environnements ubiquitaires* même s'ils ne sont pas toujours les plus adaptés. Enfin, les systèmes NEMESYS et INDISS ayant une architecture logicielle commune, ils possèdent des caractéristiques similaires : ce sont tous deux des systèmes de *traduction dynamique de protocoles* qui peuvent être déployés sur des périphériques aussi bien riches que limités en ressources (mémoire, processeur, etc.) et qui capturent, de façon transparente, les messages réseaux qui transitent par le système d'exploitation de l'hôte sur lesquels ils sont déployés. Toutefois, NEMESYS capture uniquement les messages réseaux relatifs aux protocoles de communication RPC pour les traduire dynamiquement, *via* une composition dynamique d'unités, en un autre protocole RPC en fonction de son environnement d'exécution. La résolution des incompatibilités de protocoles de communication entre intergiciels hétérogènes est réalisée sans requérir de modifications de ces derniers ni de leurs applications respectives. Ce qui distingue NEMESYS d'INDISS est la capacité du système NEMESYS à traduire dynamiquement des protocoles réseaux *via* une composition à la fois verticale et horizontale d'unités de protocoles. En d'autres termes, au niveau formel, contrairement à INDISS, NEMESYS est l'implementation d'un connecteur C-UNIV-Réseau résultant de la composition d'au moins deux connecteurs C-UNIV.

Avant d'introduire le système NEMESYS, nous exposons les différentes similitudes existantes entre les protocoles RPC, afin de déterminer un *protocole image* commun (section 5.2.1). Par la suite, nous adaptons en conséquence la spécification du connecteur C-UNIV-Réseau aux RPCs et proposons un raffinement de l'architecture logicielle du système SySTDyP (section 5.2.2) à partir duquel nous présentons une implémentation possible du système NEMESYS (section 5.2.3), suivi de différents scénarios d'utilisation (section 5.2.4), et nous terminons par une évaluation des performances du prototype du système NEMESYS (section 5.2.5).

5.2.1 Détermination des similarités entre les protocoles RPC

Comme indiqué dans le chapitre 2, les protocoles de communication RPC partagent les deux caractéristiques suivantes :

1. Ce sont des protocoles qui suivent par défaut un paradigme de communication de type requête/réponse synchrone. L'appelant est bloqué suite à sa requête jusqu'à la réception d'une réponse de l'appelé [2], [3].

2. Ce sont des protocoles qui ont été conçus pour réaliser des appels (ou invocations) de procédures (ou méthodes) localisées sur des machines distantes en faisant abstraction des détails relatifs à l'hétérogénéité de leurs plateformes. Plus précisément, un protocole RPC définit une représentation intermédiaire des données échangées entre les clients et les services, permettant de faire abstraction de leurs formats de données spécifiques aux plateformes sur lesquelles les clients et les services sont déployés [13]. Parmi les différentes fonctions qu'un protocole RPC fournit, on s'intéresse particulièrement à deux d'entres elles :

    - L'invocation qui consiste à invoquer une méthode distante et,
    - La sérialisation qui consiste à formater (ou à sérialiser) les données nécessaires à l'invocation suivant une représentation intermédiaire de référence des données définie par le protocole.

La source des incompatibilités entre protocoles RPC intervient lorsque chacun d'entre eux utilise une représentation intermédiaire des invocations des méthodes distantes qui leur est spécifique. L'analyse des protocoles RPC, tels que SOAP [25], CORBA [16], RMI [21], et de leur IDL respectif, révèle qu'une requête correspondant à l'invocation d'une méthode distante contient toujours indépendamment de sa représentation : le nom de la méthode invoquée, la liste des arguments requis par la méthode, et le type de la valeur de retour attendue. L'argument d'une méthode est un couple (clé, valeur), la clé correspondant au type de la valeur. Par ailleurs, la réponse à une telle requête contient la valeur de retour du type attendu de la méthode exécutée par le service distant. Une requête d'invocation contient toujours les informations suivantes :

$$NomMéthode\ ((type_i, valeur_i)_{|\ i \in [\ 1,..,n\ ]})\ :\ (typeRetour)$$

Les différents types des arguments d'une méthode peuvent différer d'un IDL à un autre, toutefois des correspondances existent toujours et ont été notamment définies, et même standardisées pour certaines, dans le cadre de la réalisation de ponts entre intergiciels commerciaux, précédemment présentés dans la section 3.4.2, tels que par exemple SOAP2CORBA [129], IIOP-.NET [132], DCOM-CORBA [130], RMI-IIOP [131]. On distingue principalement les types primitifs tels que *integer, long, double, string, char* et les types complexes qui constituent une combinaison de plusieurs types primitifs. Par conséquent, les différents protocoles RPC, ayant à la fois un comportement similaire (communication *via* un paradigme de type requête/réponse synchrone) et des informations communes à échanger, peuvent être projetés sur un protocole commun. La section suivante présente l'adaptation de la spécification de

C-UNIV-Réseau adaptée aux protocoles RPC et un raffinement en conséquence de l'architecture du système SySTDyP.

5.2.2 Raffinement de l'architecture du système de traduction

Nous présentons tout d'abord les différentes stratégies possibles que le composant moniteur peut implémenter pour détecter les protocoles RPCs en cours d'utilisation. Par la suite, nous décomposons ces derniers en couches afin de déterminer l'architecture du système SySTDyP adaptée aux protocoles RPC. Enfin, nous présentons les critères minimums, c'est-à-dire les messages sémantiquement équivalents et les règles de coordination, que les protocoles RPC doivent respecter pour être pris en charge par le système SySTDyP. De façon analogue à INDISS, ces critères sont exprimés sous la forme d'un processus FSP, qui est traduit par un automate que les unités du système doivent inclure. Ce processus FSP définit les règles de coordination du *protocole image* sur lequel les protocoles RPCs sont projetés.

Le composant moniteur.

La détection des protocoles de communication RPC en cours d'utilisation peut se faire de deux façons différentes :

1. En analysant les en-têtes inclues dans les messages réseaux interceptés (Voir section 4.4). Pour rappel, le contenu d'un message est préfixé par un ensemble d'entêtes qui correspond à une signature déterminant les protocoles de chacune des couches de la pile de protocoles dont est issu le message.

2. En se référant, lorsque le service distant recherché est découvert dynamiquement *via* un SDP, au SDP qui a été utilisé pour découvrir/annoncer le ou les services recherché(s). En effet, le plus souvent, à partir des informations obtenues *via* un SDP, il est possible de déduire le protocole de communication d'un service annoncé et/ou découvert (cette information, lorsqu'elle existe, fait partie d'un des attributs de la description du service). L'implémentation de cette stratégie correspond à l'utilisation du système INDISS comme un composant moniteur et implique implicitement une coopération entre INDISS et NEMESYS comme détaillée dans la section 5.2.4.

A de rares exceptions près, la pile de protocoles d'un protocole RPC est figée, connue à l'avance et standardisée. De ce fait, nous considérons la première stratégie comme une solution permettant de pallier aux déficiences de la seconde, c'est à dire lorsque le SDP utilisé pour trouver le service distant recherché ne permet pas de déterminer le protocole RPC du service.

Décomposition en couches des RPCs.

Comparativement au modèle OSI, les protocoles de communication RPC se décomposent en six couches (Figure 46). La définition des trois premières couches est la même que celle présentée pour les protocoles de découverte de services. On rappelle brièvement que le *support physique* indique le type de réseau sur lequel les données sont transmises, la couche *réseau* spécifie comment les données sont transférées d'une entité à une autre et la couche transport détermine le mode de délivrance des messages. En revanche, si les couches OSI *session* et *présentation* n'avaient pas véritablement d'existence dans le contexte des SDPs *multicast* étant donné leur simplicité, ce n'est plus le cas pour les protocoles de communication RPC. Ils possèdent deux nouvelles couches, la couche *invocation* et *présentation*. La première, similaire à la couche OSI *session*, gère l'établissement de sessions entre différentes entités communicantes, tandis que la seconde, analogue à la couche OSI *présentation*, encode les messages transmis selon un format d'encodage spécifique. Enfin, on retrouve de nouveau la couche *application* qui fournit aux applications une *interface* permettant d'invoquer un service distant.

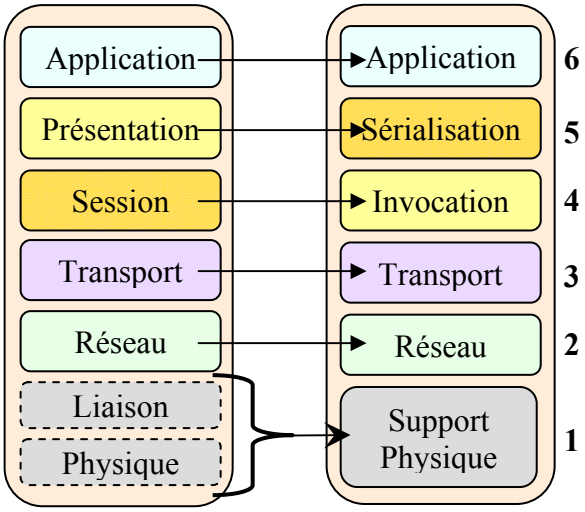


Figure 46. Décomposition en couches des protocoles de communication RPC

En théorie, la traduction dynamique d'une pile d'un protocole réseau est réalisée *via* la composition d'autant de connecteurs C-UNIV que de couches. Dans la pratique, les protocoles de communication RPC, tout comme pour les SDPs, sont composés d'un certain nombre de couches identiques minimisant le nombre de connecteurs C-UNIV et par conséquent le nombre d'unités que le système SySTDyP doit composer. Comme introduit dans la section 5.2.1, les protocoles RPC diffèrent au niveau de leurs fonctions d'invocation et de sérialisation, et diffèrent au niveau de leur couche de même nom.

Dans la Figure 47 on illustre l'hétérogénéité des piles de protocoles de deux périphériques qui souhaitent communiquer. On distingue deux périphériques A et B, le premier embarquant une pile de protocoles RMI et le deuxième une pile de

protocoles de services Web. Bien que le design de leur pile de protocoles soit similaire, leurs applications ne peuvent communiquer. Un client de A ne peut pas invoquer un service de B étant donné l'hétérogénéité des couches 4 et 5 de leur pile respective. En ce qui concerne la pile de protocoles RMI, la couche 5 (sérialisation) offre des fonctions d'encodage/décodage des données en provenance de la couche application dans un format binaire selon le protocole de sérialisation d'objets Java (JOSSP pour *Java Object Serialization Stream Protocol*). Comparativement, cette même couche, issue de la pile de protocoles de services Web, encode/décode les données dans un format XML selon le protocole SOAP.

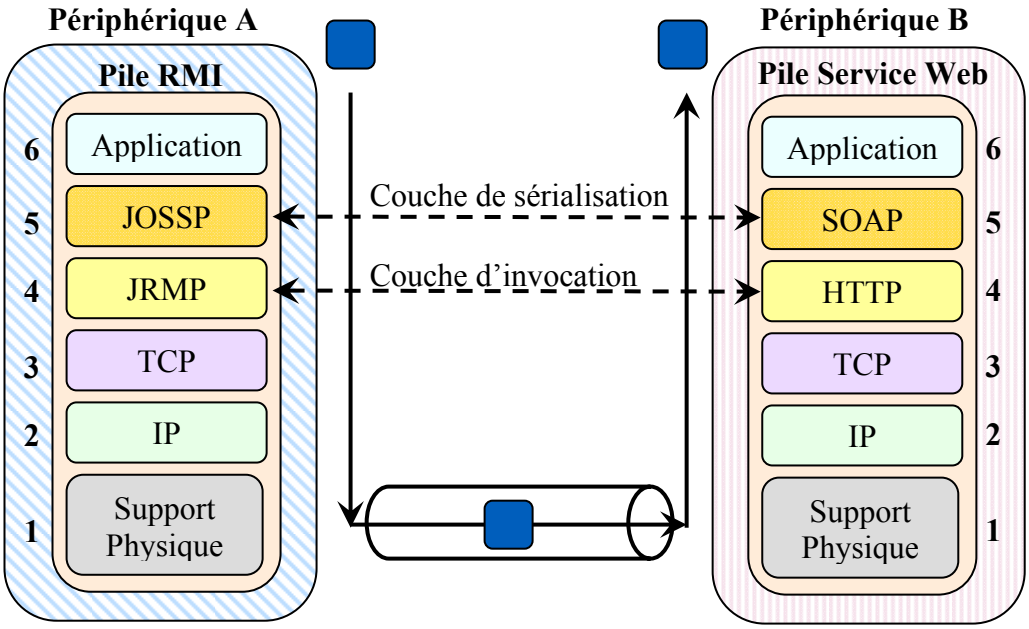


Figure 47. Hétérogénéités des piles des protocoles de communications RPC

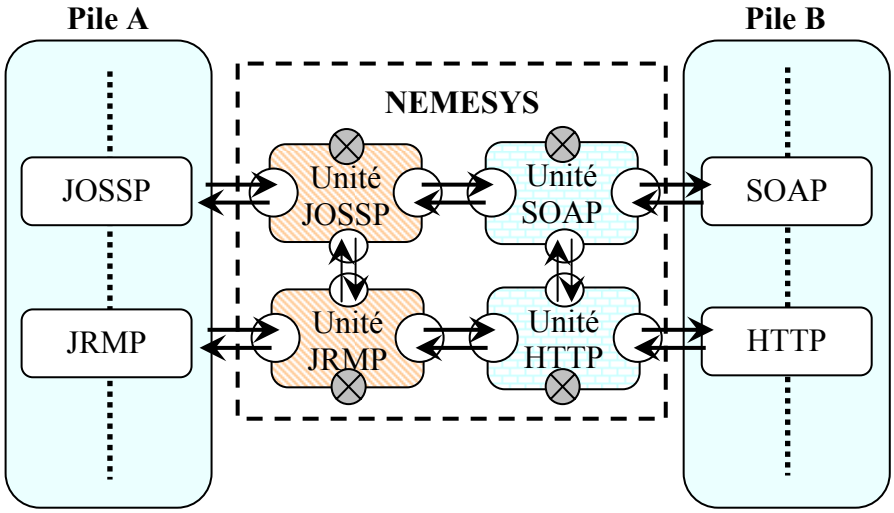


Figure 48. Composition verticale et horizontale d'unités de NEMESYS

Ces couches de sérialisation issues de piles de protocoles hétérogènes ne diffèrent pas en termes de fonctionnalités mais dans la façon de représenter/transformer les données. Il en est de même pour la couche 6 (invocation) : les applications dépendant de la pile de protocoles RMI transmettent des messages à travers le réseau dans un format binaire selon la spécification JRMP (*Java Remote Method Protocol*) tandis que celles qui dépendent de la pile protocole de services Web emploient HTTP comme protocole. Quelle que soit la pile de protocoles considérée, la couche d'invocation offre toujours les mêmes fonctions mais diffère dans la façon dont les messages sont transmis à travers le réseau.

Enfin, la majorité des protocoles de communication RPC, étant basée sur TCP/IP, les couches 1 à 3 sont identiques, et le connecteur C-UNIV-Réseau s'applique uniquement de la couche 4 à 5. Ainsi le système SySTDyP résout les incompatibilités des protocoles de ces deux couches *via* une composition dynamique d'unités à la fois verticalement et horizontalement, comme illustré dans Figure 48, et suivant les principes de fonctionnement décrits dans le chapitre 4.

Formellement, le connecteur C-UNIV-Réseau pour les RPCs résulte de la composition de deux connecteurs C-UNIV : l'un pour la couche sérialisation, l'autre pour la couche invocation. Ainsi, le *protocole image* commun aux RPCs, défini ci-après, se décompose en deux sous-protocoles images communs.

Protocole image commun aux RPCs.

Conformément à la spécification de C-UNIV-Réseau et au fait que l'on s'intéresse uniquement aux couches des niveaux 4 et 5 de la pile de protocoles des protocoles RPC (Figure 47), formellement C-UNIV-Réseau se note :

$$\text{C-UNIV-Réseau} = \text{C-UNIV-4} \parallel \text{C-UNIV-5}$$

Définissons, par conséquent, dans un premier temps, le connecteur C-UNIV-4 permettant de résoudre les incompatibilités entre les différents protocoles de la couche d'invocation. Cette dernière assure une fonction dite d'invocation *via* un protocole de type requête/réponse synchrone comme indiqué précédemment dans la section 5.2.1. On distingue donc deux types de messages : des requêtes et des réponses, chacune d'elles se décompose en deux parties : l'entête et des données considérées à ce niveau de la pile comme des *données brutes*[14] (5-PDU) traitées par la couche immédiatement supérieure (de niveau 5), c'est-à-dire la couche de sérialisation. Les informations contenues dans l'entête des messages correspondent aux informations de contrôle du protocole d'invocation (4-PCI) qui incluent :

1. Certaines propriétés du protocole de transport sous-jacent tel que le destinataire et l'expéditeur du message

[14] Non interprétable par la couche traversée

2. Le type du message parmi les suivants : requête ou réponse.

Chacune de ces informations est projetée sous forme d'évènements *via* la fonction de projection du connecteur C-UNIV-4 afin de faire abstraction de l'hétérogénéité de leur format, quant aux *données brutes*, elles sont transférées vers le connecteur C-UNIV-5. On définit en conséquence un ensemble d'évènements S_{evt} qui se décompose en trois sous ensembles. Soit :

S_{evt} = "RPC_Control_Events" ∪ "RPC_NetworksEvents"
∪ "RPC_Invocation_Events"

Avec,

1. "SDP_Control_Events" : les évènements de cet ensemble permettent de marquer le début et la fin d'un flux d'évènements extrait à partir d'une réponse ou d'une requête reçue.

2. "SDP_NetworksEvents" : cet ensemble regroupe, comme pour INDISS, les évènements qui décrivent les propriétés du protocole de transport sous-jacent. Ces évènements indiquent notamment l'adresse source et l'adresse de destination.

3. "SDP_InvocationEvents" : chacun de ces évènements définit les types des messages qui transitent à travers C-UNIV parmi {requête, réponse, erreur}.

Pour chaque ensemble, on définit une série d'*évènements images* obligatoires que la fonction de projection de chaque RPCs, pris en charge par C-UNIV-4, doit générer pour assurer une *traduction minimale* (voir Tableau 3).

Ensemble d'évènements	Evènements	Sémantique
RPC Control Events	rpc.c.start rpc.c.stop	Reception d'un message Fin de reception
RPC Network Events	rpc.net.source.addr rpc.net.dest.addr	Adresse de l'emetteur Adresse du destinataire
RPC Invocation Events	rpc.request rpc.response rpc.error	Requête (invocation) Réponse (valeur de retour) Erreur

Tableau 3. Evènements obligatoires du protocole commun C-UNIV-4

On définit dorénavant le protocole commun C-UNIV-4, c'est-à-dire le comportement minimal que la projection de la composition des *glues* des protocoles d'invocation

incompatibles, doit respecter. On s'assure essentiellement que le paradigme du protocole d'invocation est bien du type requête/réponse synchrone.

Q1 = rpc.c.start → request,
request = (rpc.net.source.addr | rpc.net.dest.addr) → request
| rpc.request → rpc.c.stop → response,
response= (rpc.net.source.addr | rpc.net.dest.addr) → response
| rpc.response → rpc.c.stop → Q1

Enfin, le contenu des messages en provenance ou en direction de la couche de niveau 4 transite à travers le connecteur C-UNIV-5. Le rôle de ce dernier est de résoudre les incompatibilités des différents protocoles de la couche de sérialisation. Le protocole de cette dernière consiste précisément à définir les règles de formatage du contenu des données qui transite par son intermédiaire. Cette couche assure deux fonctions :

1. Une fonction de sérialisation qui encode correctement (i) les données nécessaires à l'invocation d'une méthode distante, ou (ii) la valeur de retour correspondant à une invocation de méthode reçue précédemment.

2. Une fonction de dé-sérialisation qui effectue l'opération inverse, c'est à dire qui extrait les données des messages en provenance de la couche d'invocation en permettant de déterminer : (i) la signature de la méthode invoquée, ou (ii) la valeur de retour d'une méthode précédemment invoquée.

On définit deux nouveaux sous-ensembles d'évènements spécifiques à C-UNIV-5 qui viennent compléter l'ensemble S_{evt} défini par C-UNIV-4 afin de représenter le contenu attendu d'une requête d'invocation (Tableau 4). On note:

$S_{evt} = S_{evt} \cup$ "Serialization_Control_Events" $\cup$ "Serialization_Methods_Events"

Avec,

1. "Serialization_Control_Events" : cet ensemble d'évènements a un rôle similaire à celui de "RPC_Control_Events": il regroupe les évènements précisant le début et la fin d'un flux d'évènements extraits à partir des données provenant de la couche sérialisation.

2. "Serialization_Methods_Events" : cet ensemble regroupe les évènements grâce auxquels il est possible de décrire le contenu d'une requête d'invocation. Comme indiqué dans la section 5.2.1, ce contenu prend la forme suivante :

   NomMéthode ((type1,valeur1), (type2, valeur2), (type3, valeur3),…, (typeN, valeurN)) : (typeRetour)

Par conséquent, cet ensemble est défini par trois types d'évènements : (i) *ser.meth.name* représentant le nom de la méthode, (ii) *ser.meth.arg* correspondant aux arguments de la méthode, (iii) *ser.meth.return* représentant la valeur de retour d'une méthode.

Ensemble d'évènements	Evènements	Sémantique
Serialization Control Events	ser.start	Début de la serialisation
	ser.stop	Fin de la sérialisation
Serialization Methods Events	ser.meth.name	Nom de la méthode
	ser.meth.arg	Arguments de la méthode
	ser.meth.return	Valeur de retour de la méthode

Tableau 4. Evènements obligatoires du protocole commun C-UNIV-5

Il est intéressant de noter qu'il n'y a pas d'évènements représentant les fonctions de sérialisation et de dé-sérialisation, ou indiquant le moment où ces dernières sont appliquées. En effet, la fonction de sérialisation s'applique automatiquement sur toutes les données en provenance de la couche immédiatement supérieure à la couche de sérialisation, c'est-à-dire la couche de niveau 6, et à destination de la couche d'invocation (de niveau 4). A l'inverse, la fonction de dé-sérialisation s'applique automatiquement à toutes les données (c'est à dire au contenu des requêtes et des réponses) en provenance de la couche de niveau 4 à destination de la couche de niveau 6. Plus précisément, le connecteur C-UNIV-5 se synchronise avec C-UNIV-4 et *vice versa via* les évènements *rpc.request* et *rpc.response*. En d'autres termes, cela signifie que C-UNIV-5 est capable :

1. De générer, à partir des évènements *rpc.request* et *rpc.response*, les évènements requis pour décrire la méthode invoquée, ou décrire la valeur de retour d'une méthode précédemment invoquée afin de produire des données (6-PDUs) à destination de la couche de niveau 6.

2. De générer des évènements *rpc.request* et *rpc.response* (ou des 4-PDUs) à partir des évènements, décrivant la méthode distante à invoquer ou la valeur de retour à envoyer, obtenus à partir des données (6-PDUs) en provenance de la couche de niveau 6.

De ce qui précède, on déduit le *protocole image* commun de C-UNIV-5 suivant :

Q2 = ser.c.start → (rpc.request | rpc.response) → *toDeserialize*
| *toSerialize*,
toDeserialize = ser.meth.return → ser.c.stop → Q2
| ser.meth.name → args1,
args1 = ser.meth.arg → *args1*
| ser.meth.return → ser.c.stop → Q2
toSerialize = ser.meth.return → rpc.response → ser.c.stop → Q2
| ser.meth.name → *args2*,
args2 = ser.meth.arg → *args2*
| ser.meth.return → rpc.request → ser.c.stop → Q2,

Au niveau du système, l'automate des unités des protocoles correspondant à la couche d'invocation (resp. de sérialisation) doit inclure l'automate obtenu à partir du processus FSP Q1 (resp. Q2) pour garantir une *traduction minimale*.

L'architecture du système SySTDyP adaptée pour les RPCs est facilement extensible. Il est possible d'introduire de nouveaux évènements *a posteriori* pour augmenter la qualité de la traduction dynamique, c'est-à-dire pour prendre en charge de nouvelles fonctionnalités, ou des particularités que partagent seulement certains protocoles RPC. L'adjonction de nouveaux évènements ne peut qu'améliorer la *traduction minimale* et cela sans altérer le fonctionnement du système. Dans la section suivante, nous présentons une implémentation du raffinement de l'architecture logicielle de SySTDyP pour les RPCs, nommé NEMESYS.

5.2.3 Le système NEMESYS

NEMESYS est un système qui se compose d'un ensemble d'unités de protocoles assemblées au cours de son fonctionnement. Etant donné le nombre d'unités impliquées dans la traduction d'un protocole de communication, leur chaînage dynamique et bidirectionnel (vertical et horizontal) représente un coût non négligeable au niveau de la consommation de ressources (processeur, mémoire, espace disque). Cette consommation se répercute implicitement sur le temps nécessaire pour traduire un protocole. Afin d'optimiser le fonctionnement du système, le chaînage vertical des unités, pour chaque pile de protocoles, peut être réalisé à l'avance de façon statique. En effet, dans le cas général, la structure d'une pile de protocoles d'un protocole de communication RPC est connue à l'avance et est figée. Ainsi, NEMESYS n'a plus qu'à composer, en fonction de son contexte d'exécution, des chaînes verticales d'unités pré-assemblées correspondant aux protocoles réseaux identifiés.

Comme pour INDISS, l'architecture logicielle du système NEMESYS est décrite dans un fichier de configuration à partir duquel le système est généré. Bien que ce fichier de configuration s'organise toujours en deux parties, la première diffère

légèrement. Elle définit, en plus des unités prises en charge par le système (pour les couches d'invocation et de sérialisation dans le contexte de NEMESYS), quelles sont les chaînes verticales d'unités statiquement composées correspondant aux piles de protocoles gérées par NEMESYS (Figure 49). La deuxième partie du fichier de configuration est strictement similaire à celle d'INDISS : elle définit la structure interne de chacune des unités du système.

Configuration du système

```
System NEMESYS = {
//Description de l'architecture du système
Component Unit JRMP;
Component Unit JOSSP;
Component Unit MOBILECODE;
Component Unit HTTP;
Component Unit SOAP;
Component Unit IIOP;
Component Unit CDR;
Unit Chain SOAP = {HTTP, SOAP}
Unit Chain RMI_1 = {JRMP, JOSSP}
Unit Chain RMI_2 = {HTTP, MOBILECODE}

//Description des unités
……}
```

Figure 49. Fichier de configuration du système NEMESYS

Une fois le système NEMESYS instancié à partir de son fichier de configuration, la *traduction dynamique* de RPCs devient fonctionnelle. La traduction est réalisée à travers la composition dynamique des chaînes d'unités verticales (Figure 50A). La composition de ces chaînes verticales entraine la composition horizontale de chacune de leurs unités. Par ailleurs, bien que le fichier de configuration définisse des chaînes verticales d'unités pré-assemblées, NEMESYS peut toujours en créer/instancier de nouvelles ou reconfigurer celles déjà existantes en ajoutant, supprimant ou en changeant une unité de protocole par une autre en fonction de son environnement d'exécution (Figure 50C). Le fait de pouvoir instancier à la volée des unités et de les imbriquer les unes avec les autres dans différents sens, à la fois horizontalement et verticalement, procure au système une certaine habilité à s'adapter à son contexte d'exécution qui évolue, au cours de son fonctionnement, en fonction des protocoles de communication utilisés dans l'environnement. Par ailleurs, les unités ne sont pas nécessairement spécifiques à un protocole de communication, il est possible de les assembler ou de les chaîner de différentes façons. Par exemple, la chaîne nommée RMI_2 de la Figure 50, dédiée à la gestion du code mobile des clients/services RMI, dépend d'une unité HTTP qui est aussi utilisée par la chaîne SOAP.

Enfin, la détection des protocoles de communication RPC utilisés par les clients et les services distants est réalisée *via* le composant moniteur (Figure 50B) qui comme indiqué dans la section 5.2.2 utilise INDISS comme implémentation de sa stratégie de détection des protocoles RPCs. Le système NEMESYS étant dorénavant défini, nous présentons dans la section suivante son fonctionnement dans des scénarios d'utilisation concrets, dans le contexte d'un *environnement ubiquitaire*.

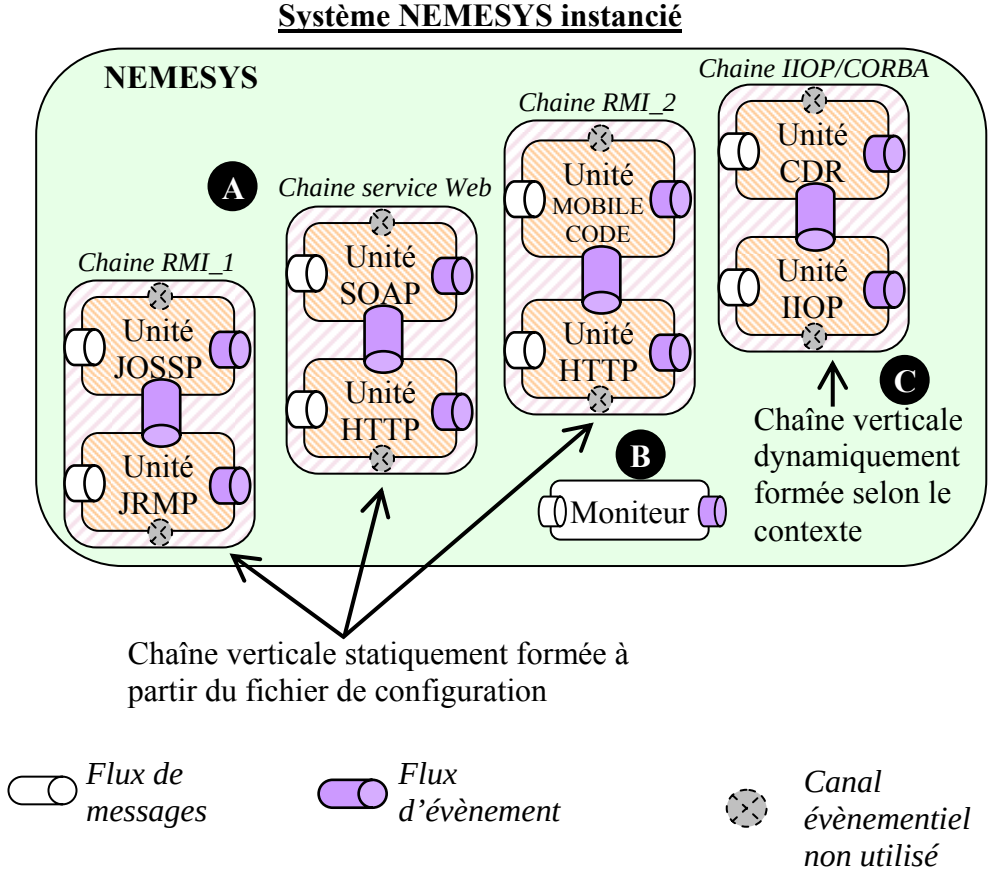


Figure 50. Instanciation du système NEMESYS

5.2.4 Scénario d'utilisation

Dans une *architecture de services ubiquitaires*, les clients et les services sont capables d'interagir entre eux, indépendamment de la spécificité de leur intergiciel, s'ils sont capables de faire abstraction des incompatibilités de leur protocole de communication et de découverte de services. En combinant le système NEMESYS et le système INDISS, ces incompatibilités de protocoles sont en mesure d'être dynamiquement résolues de façon transparente, c'est-à-dire sans avoir à altérer les clients, les services et leur intergiciel respectif. Etant donné qu'INDISS et NEMESYS partagent une même architecture évènementielle, leur coopération est directe et se révèle même nécessaire car pour communiquer avec des services

distants, les clients doivent dans un premier temps les découvrir *via* l'utilisation de SDPs afin d'obtenir suffisamment d'informations pour amorcer par la suite une communication. Le couple INDISS-NEMESYS peut être déployé aussi bien sur l'hôte des applications clientes que sur celui hébergeant les services. Le déploiement du système sur une passerelle est également envisageable.

Enfin, dans la section 5.1.1, nous avons distingué deux types de SDPs : les *centralisés* et les *décentralisés* suivant qu'ils requièrent l'utilisation ou non d'un annuaire. En conséquence, nous présentons deux scénarios d'utilisations du couple INDISS-NEMESYS.

Découverte de services et communication sans annuaire.

A partir des SDPs utilisés dans l'environnement ambiant, INDISS connaît les protocoles de communication employés à la fois par des clients et des services. Par exemple, lorsque le service nommé S1, déployé sur le périphérique B, annonce sa présence *via* JESA (Figure 51, étape❶), INDISS suppose que le protocole de communication utilisé pour interagir avec S1 est RMI, car par défaut, l'utilisation de JESA pour découvrir/annoncer un service implique que le client/service utilise RMI pour communiquer. De façon analogue, lorsque le client du périphérique A envoie une requête UPnP pour trouver un service (Figure 51, étape❷), INDISS détecte que le protocole de communication utilisé par le client est basé sur SOAP (UPnP implique toujours SOAP). Si l'on considère que le service S1 correspond au service recherché par le client de A, alors INDISS répond à la requête du client en indiquant l'adresse IP de NEMESYS comme étant celle du service recherché. Simultanément, INDISS transfert ses connaissances sur les protocoles de communication utilisés à NEMESYS (Figure 51, étape❸). Ce dernier apprend alors qu'un client UPnP, identifié par l'adresse IP du périphérique A, souhaite interagir avec un service RMI, identifié par l'adresse IP de B. Les piles de protocoles UPnP et RMI étant similaires, NEMESYS est capable de reconfigurer ses unités de façon à composer verticalement la chaine d'unités UPnP avec celle de RMI. Au final, les messages UPnP en provenance de A sont prêts à être traduits en messages RMI à destination de B, et *vice versa* (Figure 51, étape❹).

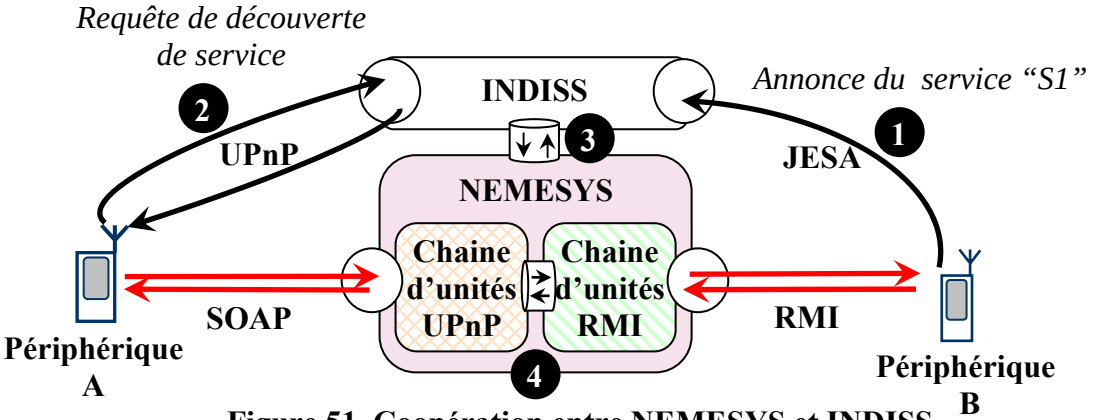


Figure 51. Coopération entre NEMESYS et INDISS

Il existe des SDPs, pour lesquels INDISS est incapable de prédire le protocole de communication utilisé par les clients et/ou les services ou de faire une mauvaise prédiction. Par exemple, des SDPs, tel que SLP, ne sont pas associés par défaut à un protocole de communication particulier. Dans tous les cas, NEMESYS est toujours capable de détecter la structure du protocole de communication en analysant les en-têtes des messages et en assemblant si nécessaire dynamiquement les unités pertinentes requises pour assurer la traduction. Considérons maintenant le cas où la découverte de services s'effectue *via* un annuaire.

Découverte de services et communication avec annuaires.

Dans ce scénario, les services doivent dans un premier temps s'enregistrer auprès d'un annuaire afin d'être découverts ensuite par leurs clients potentiels. L'enregistrement permet aux services d'exporter leurs *interfaces* et leurs références. Pour rappel (Voir chapitre 2), une *interface* est un ensemble de méthodes qui décrit les invocations à distance qu'il est possible d'effectuer sur un service tandis qu'une référence indique comment localiser l'instance du service. Les services exportent ces données sous forme d'un *stub* auprès de l'annuaire. En consultant ce dernier, les clients découvrent les services disponibles, et en rapatriant leurs *stubs* correspondants, ils sont capables de communiquer avec les services recherchés. Un des avantages d'utiliser un annuaire est d'obtenir dynamiquement les *stubs* des services recherchés. Toutefois, l'obtention dynamique de *stubs via* un annuaire n'est pas toujours obligatoire. Certains clients utilisent des *stubs* obtenus statiquement (c'est-à-dire lors de leur conception), tandis que d'autres sont capables de les générer à la volée dès lors qu'ils connaissent la localisation du service distant qu'ils souhaitent invoquer. C'est en particulier le cas lorsque la découverte de services et la communication s'effectuent sans l'utilisation d'annuaire comme dans le cas précédent.

La communication entre un client et un service distant *via* un annuaire se fait au terme d'une procédure de découverte de services en trois étapes :

1. Découverte de la localisation de l'annuaire par le client et le service *via* un SDP *multicast*.

2. Exportation par le service de son *stub* auprès de l'annuaire.

3. Consultation de l'annuaire et rapatriement du *stub* du service recherché par le client.

Si grâce aux SDPs *multicast*, les clients et les services sont capables de découvrir les annuaires présents dans leur environnement ambiant, encore faut-il qu'ils soient capables d'exporter/rapatrier des *stubs* et ou de consulter l'annuaire. Ce dernier est en fait spécifique à un protocole de communication RPC et les *stubs* enregistrés sont

dédiés à l'intergiciel des services qui les ont exportés. Ainsi, l'exportation et l'importation de *stubs* ne sont possibles que si l'annuaire est respectivement compatible avec l'intergiciel du service et avec celui du client. Une solution pour résoudre ce problème consiste à utiliser le couple INDISS-NEMESYS comme un annuaire universel. NEMESYS fournit aux clients et aux services des fonctions élémentaires communes à tous les annuaires (c'est-à-dire l'exportation, la consultation et l'importation) tandis qu'INDISS leur fournit la possibilité de découvrir NEMESYS. Ce dernier assure les fonctions d'un annuaire indépendamment des protocoles de communications à travers son mécanisme de multi-chaînage d'unités.

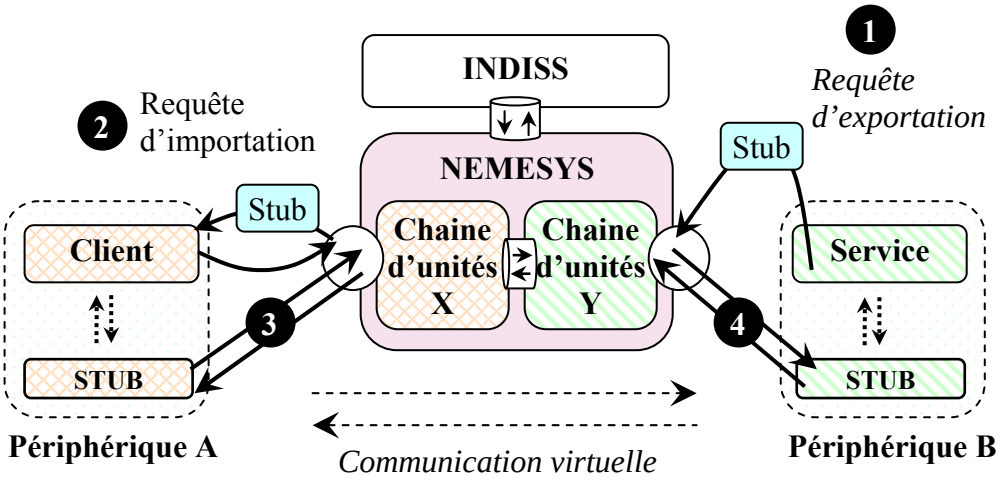


Figure 52. Annuaire universel

La Figure 52 décrit comment s'effectue la coopération entre INDISS et NEMESYS. Grâce à INDISS, d'une part, les clients et les services sont capables de découvrir le service d'annuaire fourni par NEMESYS, d'autre part, NEMESYS connaît les protocoles de communication utilisés par les clients et les services. Une fois la localisation de l'annuaire connue, les services sont en mesure d'exporter leur *stub* (Figure 52, étape❶) tandis que les clients sont capables de consulter et d'importer le *stub* du service recherché (Figure 52, étape❷). Il est important de préciser, que le *stub* exporté, spécifique à l'intergiciel des services, est transformé *via* les chaines d'unités de NEMESYS en un ensemble d'évènements afin de générer un nouveau *stub* compatible avec l'intergiciel du client lors de sa requête d'importation. Une fois l'échange et la conversion de *stub* effectués, NEMESYS prend en charge l'interopérabilité de la communication entre le client et le service (Figure 52, étapes ❸ et ❹).

Dans la section suivante, nous présentons l'implémentation du prototype de notre système NEMESYS et nous évaluons les performances de la *traduction dynamique de protocoles* de communication RPC, dans le contexte du second scénario, c'est-à-

dire lorsque le client découvre un service distant *via* un annuaire afin de mettre en valeur la capacité de NEMESYS à agir comme un annuaire.

5.2.5 Implémentation du prototype et évaluation

Dans cette section, nous justifions tout d'abord les choix d'implémentation du prototype NEMESYS, nous analysons par la suite la taille mémoire requise du système pour enfin évaluer le coût en temps de la *traduction dynamique de protocoles*.

Choix d'implémentation

Le prototype NEMESYS inclut les unités nécessaires pour traduire le protocole de communication d'un intergiciel RMI en un protocole de communication compatible avec un intergiciel dédié aux services Web et *vice versa*. Les intergiciels dédiés aux services Web, utilisent le langage XML pour décrire le contenu des messages échangés lors de la découverte ou de l'invocation des services distants. De ce fait, le protocole de communication étant un protocole *orienté texte*, son interprétation peut se faire indépendamment des caractéristiques de la plateforme locale, c'est-à-dire de l'OS et du matériel de l'hôte. A l'inverse, les intergiciels RMI utilisent un protocole *orienté binaire*. Le protocole étant binaire, il ne peut être interprété que s'il existe une homogénéité des plateformes des périphériques souhaitant interagir. Cette homogénéité est assurée uniquement si tous les périphériques embarquent une machine virtuelle Java (JVM) qui abstrait leur hétérogénéité matérielle respective. L'interprétation des protocoles binaires tels que JOSSP (*Java Object Serialization Specification Protocol*) et JRMP (*Java Remote Method Protocol*) utilisés par RMI pour invoquer les services distants, est dépendante de la présence locale ou non d'une JVM [21]. Par conséquent, l'interopérabilité entre les intergiciels RMI et non-RMI est *a priori* réalisable uniquement s'ils embarquent tous deux une JVM.

Nous avons donc choisi d'implémenter, pour ce prototype, les unités correspondant à une pile de protocoles RMI et à celles des services Web, de façon à mettre en perspective la capacité de NEMESYS à traduire un protocole *orienté texte* en un protocole *orienté binaire* et *vice versa,* tout en étant indépendant des caractéristiques de la plateforme et de l'intergiciel de son hôte. Une des meilleures méthodes pour démontrer cette indépendance de NEMESYS est d'implémenter le système dans un langage de programmation différent de Java de façon à illustrer son indépendance totale vis-à-vis d'une JVM.

Comme illustré dans la Figure 53, NEMESYS assure l'interopérabilité de la communication avec des intergiciels RMI (embarquant une JVM) tout en étant lui-même embarqué sur un hôte dépourvu de JVM. Plus précisément, en déployant NEMESYS sur un hôte B équipé d'un intergiciel dédié aux services Web, on démontre la capacité de NEMESYS à donner l'illusion que les services de B, basés sur SOAP, sont des services RMI auprès de clients RMI distants (Figure 53A) ou que

les clients de B, basés sur SOAP, sont des clients RMI auprès de services RMI distants (Figure 53B).

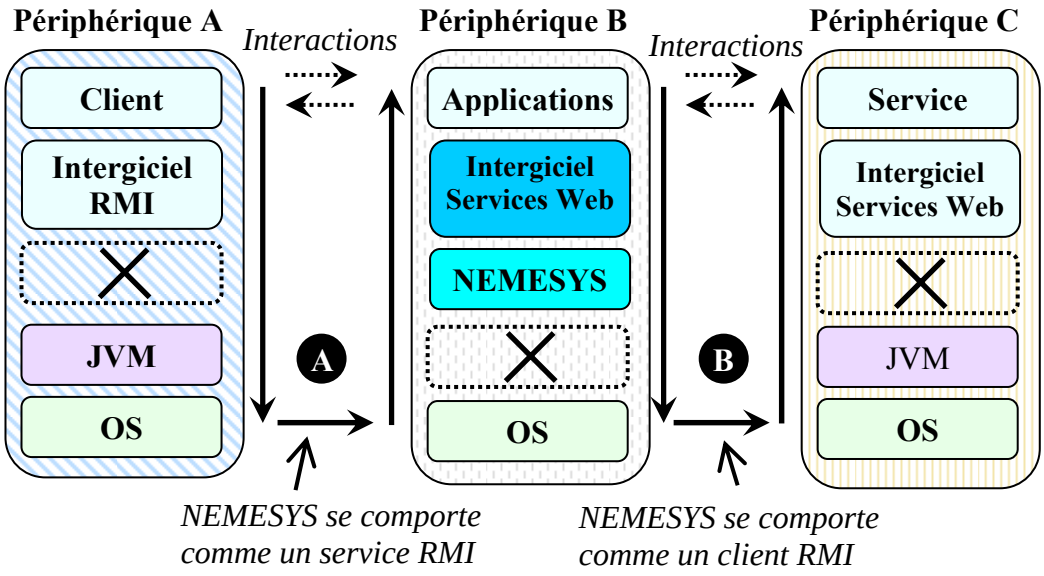


Figure 53. Indépendance de NEMESYS vis-à-vis de la plateforme locale

Afin de démontrer la capacité de NEMESYS à traduire le protocole RMI sans avoir recours à l'utilisation d'une JVM, le prototype du système ne profite pas de ce qui a été déjà implémenté pour INDISS. En effet, ce dernier étant codé en Java, il requiert une JVM pour fonctionner. NEMESYS est implémenté en ANSI-C. Le langage de programmation C a été choisi car il permet d'optimiser la vitesse de fonctionnement du système et de raccourcir les temps de traduction. Toutefois, NEMESYS peut être développé dans n'importe quel autre langage de programmation et/ou être dédié à une plateforme spécifique afin d'augmenter davantage son efficacité.

Analyse de la taille mémoire requise par NEMESYS

Pour interpréter la pile de protocoles RMI, NEMESYS utilise deux chaînes d'unités différentes, développées en C, parmi les suivantes :

- L'unité JRMP, qui correspond au protocole de la couche d'invocation de RMI.
- L'unité JOSSP, qui permet d'interpréter le protocole de la de sérialisation de RMI.

- Les unités HTTP et Java Mobile Code, qui correspondent aux protocoles permettant aux clients et aux services d'exporter et d'importer des *stubs/proxys* auprès d'un annuaire.

NEMESYS					SUN JVM	
Units		**Size (Ko)**	**Chaîne RMI 1**	**Chaîne RMI 2**	**JRE**	**J2ME**
Code Mobile	*Parser* *Composer*	140	**-**	**X**		
JOSSP	*Parser* *Composer*	56	**X**	**-**		
JRMP	*Parser* *Composer*	40	**X**	**-**		
http	*Parser* *Composer*	164	**-**	**X**		
IO abstraction		36	**X**	**X**		
Event Manager		200	**X**	**X**		
TOTAL in Ko		**636**	**332**	**540**	**45000**	**3000**

Tableau 5. Taille des chaînes RMI comparativement à une pile RMI de Sun

Comme indiqué dans le Tableau 5, NEMESYS interprète la pile de protocoles RMI *via* une chaîne RMI dont la taille n'excède pas 636 Ko. Officiellement, le protocole RMI requiert l'utilisation d'un *package* logiciel incluant une JVM et une librairie RMI. Sun met a disposition deux packages différents : l'un adapté aux périphériques contraints en ressources (*package* J2ME), l'autre adapté aux stations de travail (*package* J2RE). Ainsi, les 636 Ko de NEMESYS sont à comparer aux 3 Mo du *package* J2ME et aux 45 Mo du *package* J2RE. Par ailleurs, le *package* J2ME étant une version restreinte du *package* J2RE, il permet uniquement d'instancier des applications RMI ayant un comportement de clients. Les services RMI nécessitent, quant à eux, impérativement l'utilisation d'un *package* J2RE pour fonctionner afin notamment de générer dynamiquement leur *proxy*/*stub* Java à la volée dans l'objectif de s'enregistrer auprès d'un annuaire. En ce qui concerne NEMESYS, *via* une reconfiguration adéquate de ses unités RMI, il peut se comporter indifféremment comme un client et/ou un service RMI. De ce fait, NEMESYS réduit drastiquement l'espace mémoire requis pour prendre en charge l'ensemble des fonctionnalités de la spécification RMI, étant donné qu'il n'a besoin ni d'une JVM, ni des librairies Java pour traduire le protocole RMI.

Les unités de NEMESYS relatives au protocole de communication des services Web, et en particulier les unités SOAP et HTTP, sont réalisées à l'aide de librairies SOAP déjà existantes développées en C. A notre connaissance, il n'existe pas actuellement de librairie SOAP disponible qui soit simultanément *open source* et adaptée aux périphériques limités en ressources. Par conséquent nous utilisons la librairie CSOAP[15] bien qu'elle requiert un espace mémoire conséquent, comme indiqué dans le Tableau 6. Par ailleurs, nous avons également pris en considération l'existence de GSOAP[16] qui est une librairie reconnue pour économiser les ressources. Mais nous l'avons écartée car malheureusement, elle n'offre pas la possibilité de créer dynamiquement des appels SOAP à la volée. Enfin, il est intéressant de noter que certaines versions commerciales de SOAP requièrent uniquement 150 Ko contre 1524 Ko pour CSOAP. En conséquence, cela est très prometteur pour les prochains prototypes de NEMESYS en termes de coût mémoire. Néanmoins, bien que le prototype NEMESYS actuel soit partiellement optimisé, sa taille est déjà moins importante que celle des *packages* J2ME et J2RE, tout en fournissant l'interopérabilité aux clients et aux services.

NEMESYS			
Units		**Size**	**Web services Stack**
SOAP Unit	*Parser* *Composer*	1360	**X**
HTTP Unit	*Parser* *Composer*	164	**X**
Event Manager		200	X
TOTAL in Ko		**1724**	**1724**

Tableau 6. Taille de la chaîne verticale correspondant à une pile SOAP

Evaluation de la traduction dynamique de protocoles

Pour évaluer les performances de la *traduction dynamique de protocoles* réalisées par NEMESYS, nous mesurons le temps écoulé (le temps de latence) pour qu'un client réceptionne, à la suite de l'émission d'une requête, une réponse d'un service distant. Plus précisément, notre expérimentation consiste à mesurer et à comparer plusieurs temps de latence selon que le protocole de communication RPC utilisé par le client et le service sont identiques (communication native) ou hétérogène (communication traduite *via* NEMESYS).

[15] http://csoap.sourceforge.net/
[16] http://gsoap2.sourceforge.net/

Notre expérimentation étant focalisée sur la comparaison des temps de latences selon que la communication soit native ou traduite, la nature du matériel sous-jacent importe peu. Nous avons donc réalisé les tests et les mesures sur une station de travail équipée de 256 Mo sur un processeur Intel PIV cadencé à 1.8 GHz doté d'un Linux RedHat Fedora Core 2 comme système d'exploitation (OS). Le service invoqué pour réaliser les mesures est un service *echo* qui renvoie l'écho de la chaîne de caractères que le client a donné préalablement comme argument à sa requête. Nous avons implémenté différentes versions d'un service *echo* (resp. d'un client). Chacune des versions utilise un protocole de communication RPC différent (ou est dépendant d'un intergiciel différent (de type RMI ou SOAP)).

On distingue alors :

- Un premier couple client/service dédié aux technologies des services Web, communicant *via* le protocole SOAP. Nous avons développé deux versions : l'une en C avec la librairie CSOAP, l'autre en Java avec la librairie Apache Axis[17].

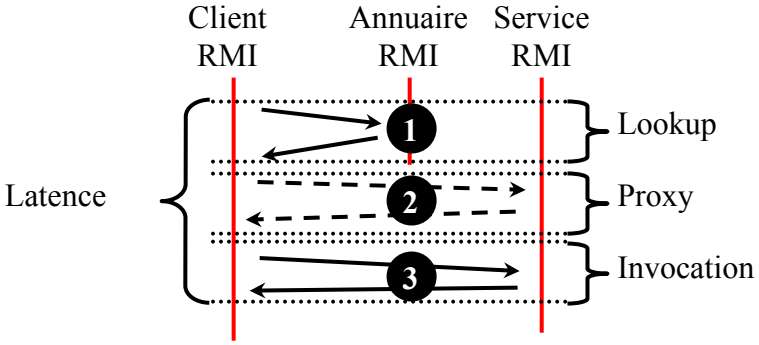
- Un deuxième couple client/service communicant *via* le protocole RMI. Ce protocole requiert des clients et des services qu'ils soient développés en Java (et donc dépendant d'une JVM). Par conséquent, contrairement au cas précédent, il n'y a qu'une seule version de ce couple, qui est développé à l'aide du JDK 1.4.2 de Sun.

Les différents résultats sont mesurés en *ms* et correspondent à la médiane de trente tests consécutifs pour éviter une moyenne biaisée par une valeur anormalement élevée ou basse. Par ailleurs, étant donné que l'on souhaite mesurer explicitement les performances de NEMESYS, tous les tests sont exécutés sur un unique hôte pour éviter les délais réseau. En effet, NEMESYS garantit l'interopérabilité entre protocoles existants, sans générer de trafic additionnel et donc sans augmenter la consommation de la bande passante.

La Figure 54 décrit une communication RMI native entre un client et un service *echo* RMI. Par défaut, avant d'invoquer le service *echo*, le client RMI consulte un annuaire RMI afin de découvrir la localisation du service distant en rapatriant son *stub* (Figure 54, étape ❶). D'autre part, lorsqu'une communication entre un client et un service est du type RMI, cela signifie normalement qu'il existe une JVM de part et d'autre de la communication. Ainsi, cela offre la possibilité aux clients RMI de télécharger dynamiquement, auprès du service distant, le code Java du *proxy* pour invoquer les méthodes du service (Figure 54, étape ❷) s'il ne le possède pas déjà. Dans la terminologie Java, il ne faut pas confondre *stub* et *proxy*. Un *stub* permet de configurer le *proxy* d'un client afin que ce dernier puisse invoquer les méthodes d'une instance spécifique d'un service préalablement découvert *via* l'annuaire d'un SDP. Dans le contexte de notre expérimentation décrite dans la Figure 54, le client

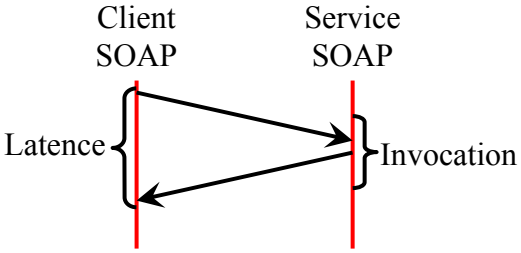
[17] http://ws.apache.org/axis/

possède déjà le *proxy* du service *echo*. Ainsi, le temps de latence total, incluant à la fois la requête d'importation du *stub* et la requête d'invocation, est de 201ms. Cependant, si l'on considère exclusivement l'invocation, le temps de latence d'une requête/réponse est réduite à 1 *ms* contre 8.08 *ms* et 20 *ms* pour une invocation similaire selon le protocole SOAP entre un client et un service SOAP développés respectivement en C et en Java (Figure 55).



	Temps écoulé
❶ ***Lookup RMI***	200
❸ **Requête/réponse RMI**	1
Total with proxy	201

Figure 54. Communication RMI native

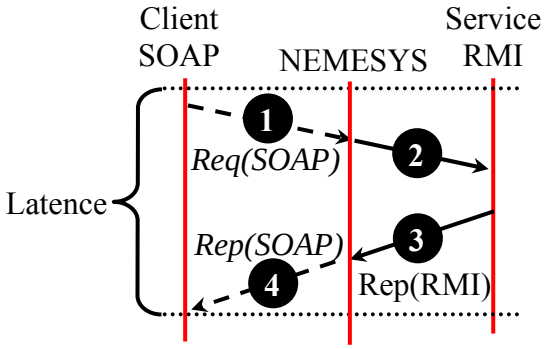


	Temps écoulé (ms)	
	CSOAP	***AXIS***
Latence	8.08	20

Figure 55. Communication SOAP native en C et Java

Le protocole RMI étant *orienté binaire*, les invocations sont naturellement plus rapides que celles réalisées en SOAP. Par ailleurs, la différence de temps de latence observée entre les versions C et Java, de l'invocation SOAP, met en perspective l'impact positif sur les performances d'une implémentation en C par rapport à celle réalisée en Java.

Maintenant, si l'on considère le scénario où le client et le service communiquent *via* NEMESYS, avec des protocoles de communication RPC hétérogènes, on distingue deux cas :



	Latence (ms)
❶ **Parser SOAP**	5
❷ **Composer RMI**	0.3
❸ **Parser RMI**	0.3
❹ **Composer SOAP**	3,4
Total	**9**

Figure 56. Invocation interopérable entre un client SOAP et un service RMI

- Cas 1 : (voir Figure 56). Lorsque le client est SOAP et le service est RMI, le client obtient une réponse à son invocation en *9 ms*. Comparativement à un appel SOAP natif à destination d'un service SOAP développé en C (*8.08 ms*), le temps de latence a augmenté globalement de *1 ms*. Ce laps de temps additionnel correspond à l'interaction RMI (Figure 56, étape ❷, ❸) que NEMESYS a initié à la réception de la requête SOAP. En d'autres termes, le coût en temps de l'interopérabilité entre un client SOAP et un service RMI correspond à l'addition du temps de latence d'une invocation SOAP et de celui d'une invocation RMI. Si l'on compare les *9 ms* au *20 ms* requises pour une interaction SOAP à destination d'un service SOAP développé en Java, cela met en perspective les bonnes performances de NEMESYS qui réduit de façon non négligeable le temps de latence nécessaire. En revanche, comparativement, à une invocation RMI native (*1 ms*), les performances de NEMESYS sont médiocres, mais sont inhérentes au protocole SOAP, qui requiert nativement dans le meilleur des cas *8.08 ms*.

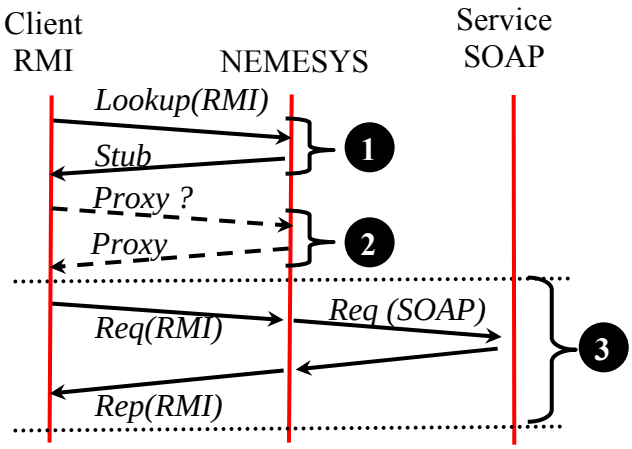
- Cas 2 (voir Figure 57). Lorsque le client est RMI et le service est SOAP, NEMESYS se comporte à la fois comme un annuaire et un service RMI. Selon la spécification RMI [21], le client interagit avec un service en trois étapes :

    1. Le client interroge NEMESYS comme si ce dernier était un véritable annuaire RMI afin d'obtenir le *stub* du service distant. L'obtention du *stub* par le client auprès de NEMESYS (voir Figure 57, étape❶) se fait en *0.30 ms* contre *200 ms* lorsqu'un annuaire RMI standard (inclus dans le jdk1.4.2 de Sun) est utilisé. Cette différence de temps, surprenante et conséquente, s'explique par le fait que NEMESYS

accède directement et nativement aux fonctions réseaux du système d'exploitation sans passer par un intermédiaire, c'est-à-dire une JVM.

2. Le client doit posséder le code Java du *proxy* correspondant au *stub* acquis lors de l'étape 1 afin d'invoquer une des méthodes du service distant. Dans le contexte où le client ne trouve pas ce proxy dans l'environnement de sa JVM, il doit l'obtenir directement auprès du service distant. Ce dernier étant représenté *via* NEMESYS comme un service RMI, NEMESYS génère directement à la volée le code Java du proxy à partir de l'*interface* qui a été exportée préalablement par le service SOAP lors de son enregistrement auprès de NEMESYS. L'obtention dynamique du proxy par le client requiert *0.85 ms* (voir Figure 57, étape❷).

3. Une fois que le client est en possession du *stub* et du *proxy* du service distant avec lequel il souhaite communiquer, il peut enfin interagir avec ce dernier. Le client obtient alors une réponse à son invocation en *9 ms.* Ce résultat est similaire à celui du cas précédent. Le coût en temps de la traduction reste constant indépendamment du fait que le client et le service sont respectivement RMI et SOAP ou inversement.



	Latency (ms)
❶ **Generation du *Stub***	0.30
❷ **Génération du code mobile**	0.85
❸ **Invocation**	9
Total avec proxy	**9.30**
Total sans proxy	**10.15**

Figure 57. Invocation interopérable entre un client RMI et un service SOAP

Le temps de latence totale maximum est égal à la somme des temps de latence des étapes ❶,❷ et ❸, soient *10,15 ms* tandis que le temps de latence minimum est de *9 ms*. Dans tous les cas, il est clair qu'une interaction interopérable entre une application RMI et SOAP ne peut pas être inférieure à la somme des temps de latence requis pour une interaction RMI native (*1ms*) et une interaction SOAP native (*8ms*). Par conséquent, le surcoût en temps produit par l'utilisation de NEMESYS peut être considéré comme négligeable.

Les prototypes des systèmes INDISS et NEMESYS étant complémentaires, nous les avons, dans le cadre du projet européen Amigo [127], fusionnés en un unique système, dévoloppés en C et nommés INMIDIO[18] (INteroperable MIddleware for service Discovery and service InteractiOn) dans le but de concevoir un système interopérable pour l'informatique diffuse résolvant à la fois l'incompatibilité des protocoles de découverte de services et de communication.

Nous présentons dans la section suivante, la façon nous avons intégré le système INMIDIO dans le projet Amigo pour déployer une *architecture de services ubiquitaires* dans l'habitat, intégrant de façon transparente dans un *réseau domestique* (*home network*) les *objets communicants* (c'est-à-dire les appareils électroménagers, numériques, mobiles, domotiques) du domicile de l'utilisateur.

5.3 Un système interopérable pour l'informatique diffuse

Le système INMIDIO est conçu pour s'intercaler entre le système d'exploitation et les intergiciels des *objets communicants*. Par conséquent, INMIDIO n'est pas un nouvel intergiciel, il s'agit d'un système de traduction qui permet aux intergiciels existants de devenir interopérables sans être modifiés. L'avantage d'INMIDIO est de résoudre de façon transparente l'incompatibilité des protocoles d'interaction des intergiciels hétérogènes. Par contre, son inconvénient est de ne pas pouvoir résoudre les incompatibilités qui peuvent survenir au niveau de la couche application et donc de ne pas gérer la *compatibilité sémantique* entre les services fournis et les services requis par les différents *objets communicants*. Dans le cadre du projet européen Amigo, dans la section 5.3.1, nous présentons la manière dont, les travaux des auteurs de [173] et de [177], sur la *compatibilité sémantique* des services d'une *architecture de services ubiquitaires*, complète le système INMIDIO pour former un nouvel intergiciel ubiquitaire interoperable pour les *réseaux domestiques*. Enfin, dans la section 5.3.2, nous évaluons ce nouvel intergiciel par rapport aux intergiciels ubiquitaires existants.

5.3.1 Intergiciel ubiquitaire interopérable

[18] http://www-rocq.inria.fr/arles/download/inmidio/index.html

L'architecture de l'intergiciel Amigo se décompose en deux sous-modules (Figure 58) [174]. Le premier, appelé *interoperable core middleware*, intègre le système INMIDIO afin de résoudre l'hétérogénéité des protocoles d'interactions des services ubiquitaires présents dans l'habitat (Figure 58A). Le second, nommé *Amigo base middleware*, vient se greffer au dessus du sous-module précédent afin de former l'intergiciel ubiquitaire Amigo. Ce dernier fournit des fonctions avancées permettant de résoudre l'hétérogénéité des services de l'architecture de services ubiquitaires au niveau de la couche application. Ce sous-module fournit également de nombreuses autres fonctions comme la gestion du contexte, de la sécurité, de la confidentialité, et la composition automatique des services en fonction des besoins de l'utilisateur et du contexte [176] (Figure 58B). Pour profiter de ces fonctionnalités avancées, les services de l'*environnement ubiquitaire* doivent être spécifiquement développés pour l'intergiciel Amigo (Figure 58D). Cependant ces services restent interopérables (*via* le sous-système *interoperable core middleware)* avec les services de l'habitat non équipés de l'intergiciel Amigo. Par ailleurs, les deux sous-modules sont indépendants l'un de l'autre. Par conséquent, il existe trois types de services susceptibles d'être déployés dans l'habitat selon leur dépendance *vis-à-vis* de l'intergiciel Amigo : les *services standards*, les *services interopérables*, et les *services Amigo*.

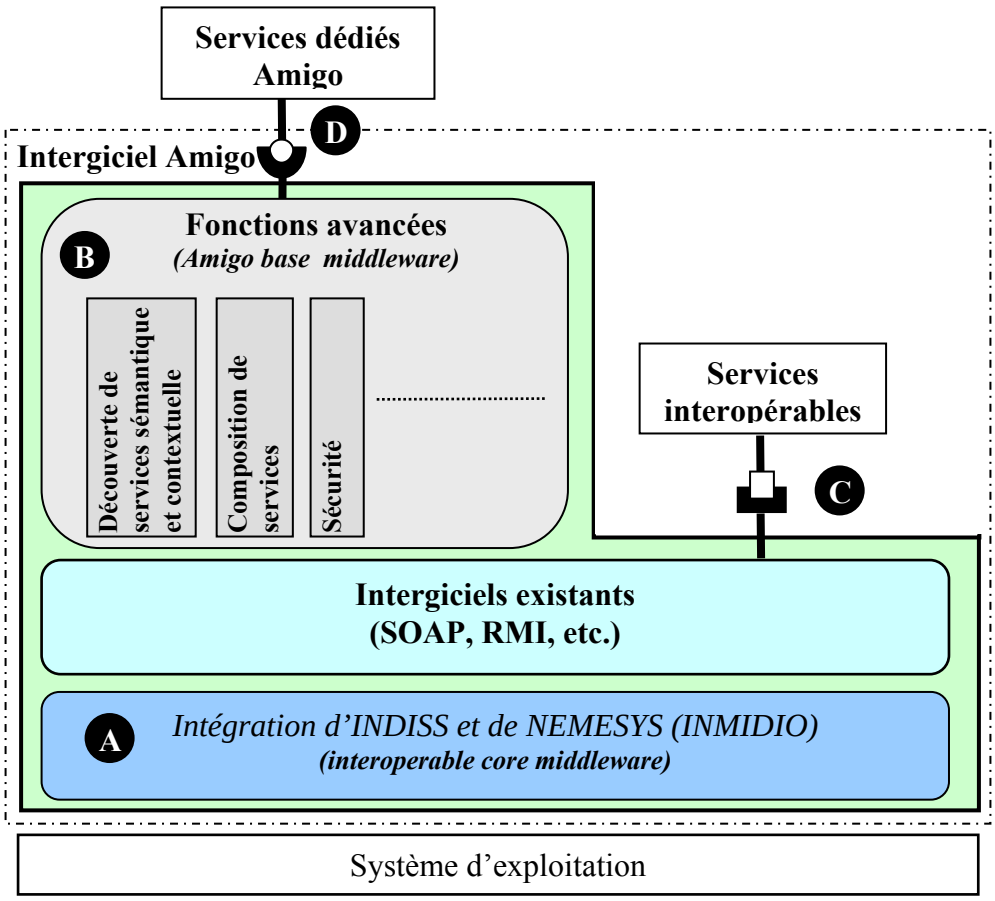


Figure 58. Architecture d'un interigicielle pour l'intelligence ambiante

Les *services standards* sont des services conçus à partir d'intergiciels hétérogènes déployés sur des *objets communicants*. Les *services interopérables* sont des *services standards* dont leur hôte est équipé du sous-module *interoperable middleware core* mais qui reste indépendant de l'intergiciel Amigo (Figure 58D). Enfin, les *services Amigo* sont des *services interopérables* qui exploitent l'ensemble des fonctionnalités de l'intergiciel Amigo (qui deviennent ainsi dépendantes de ce dernier), et en particulier des fonctions avancées permettant de résoudre l'hétérogénéité, au niveau applicatif, entre les services distants (Figure 58D, E).

Ainsi, grâce à l'architecture de l'intergiciel Amigo, trois niveaux d'interopérabilités différents existent entre les clients et les services de l'habitat (Figure 59) :

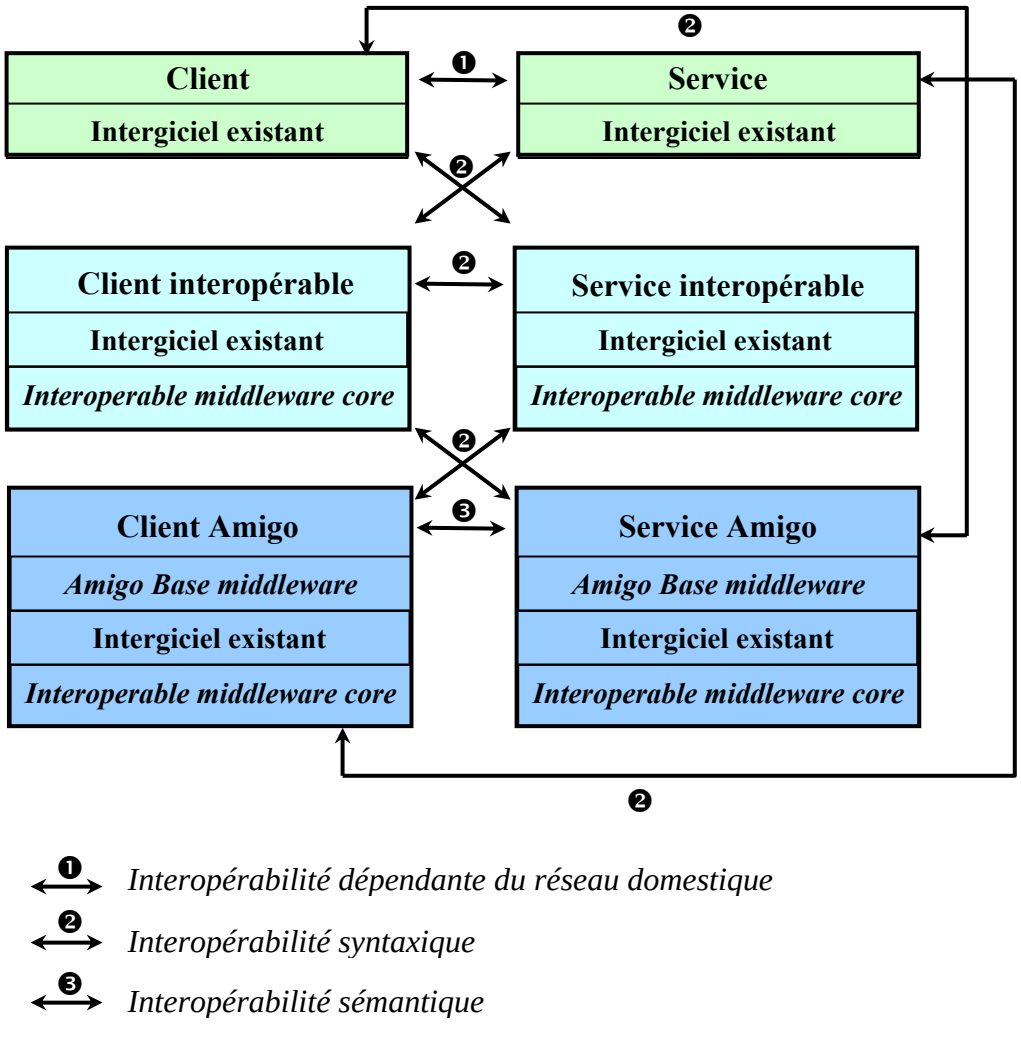


Figure 59. Différents niveaux d'interopérabilité

1. **Une interopérabilité dépendante du *réseau domestique***. Ce niveau d'interopérabilité apparaît lorsque les clients et les services de l'environnement ne sont ni l'un ni l'autre équipés du sous-module

interoperable core middleware. Dans ce cas, ils ne peuvent être interopérables que si l'une des deux conditions suivantes est vérifiée : soit ils sont implémentés suivant un même intergiciel, soit il existe une passerelle déployée dans le réseau domestique embarquant le sous-module *interoperable core middleware*. Ainsi, la passerelle agit comme un intermédiaire capable de traduire à la volée les protocoles d'interactions des différents services souhaitant interagir. L'interopérabilité est alors syntaxique, comme défini ci-après.

2. **Une *interopérabilité syntaxique***. Les *services standards,* développés à l'aide d'intergiciels hétérogènes, deviennent interopérables de façon transparente *via* le module *interoperable core middleware*. Cependant, ce dernier résout uniquement les incompatibilités des protocoles réseaux utilisés par les différents intergiciels hétérogènes. En d'autres termes, il ne permet pas de résoudre les incompatibilités sémantiques au niveau de la couche application. En effet, l'interopérabilité entre services n'est possible que si ces derniers utilisent tous des *interfaces* de descriptions communes, qui peuvent toutefois différer dans le langage de description utilisé à condition que des *interfaces* syntaxiquement compatibles impliquent obligatoirement une compatibilité sémantique.

3. **Une *interopérabilité sémantique***. Les *services Amigo* déployés dans l'habitat peuvent interagir et être composés avec d'autres *services Amigo* automatiquement grâce au système Amigo *via* la description sémantique de leur comportement. Les *services Amigo*, étant spécifiquement conçus pour le système Amigo, le développeur doit, pour les déployer, fournir systématiquement une description sémantique de leur *interface* et de leur intergiciel sous jacent.

5.3.2 Evaluation par rapport aux intergiciels ubiquitaires existants

Afin d'intégrer spontanément et dynamiquement les *objets communicants* de l'utilisateur dans son environnement immédiat, indépendamment de leur caractéristique, les intergiciels ubiquitaires doivent principalement [44] : (i) pouvoir être déployés sur des *objets communicants* limités ou non en ressources, (ii) gérer de façon transparente la mobilité des *objets communicants* dans un réseau *ad hoc* et/ou dans une infrastructure réseau sans fil, (iii) être interoperables. Nous comparons notre solution et les intergiciels ubiquitaires existants sur ces trois critères.

On distingue une premiere catégorie d'intergiciels ubiquitaires qui a la particularité de requérir une infrastructure réseau et logicielle préinstallée dans les différents environnements (comme une pièce, un bâtiment, une maison) traversés par l'utilisateur [166], [167], [43] pour intégrer de façon tranparente les *objets communicants*. Ces intergiciels sont fondés suivant le principe du *cyber foraging*

[44] : ils permettent aux *objets communicants*, sans fil et limités en ressources, de s'intégrer à leur environnement en profitant des services de son infrastructure réseau. Toutefois, par exemple, si l'on considère Gaia [167], cette intégration est possible à la seule et unique condition que tous les *objets communicants* soient équipés de l'intergiciel Gaia qui fournit une interface de programmation (*API pour Application Programming Interface*) faisant abstraction des protocoles de découverte de services et de communication qui lui sont spécifiques. Contrairement à notre solution, ces intergiciels ubiquitaires disposent de trois inconvénients : (i) tous les services doivent être développés spécifiquement pour un intergiciel particulier, (ii) sans infrastructure les objets communicants ne peuvent plus interagir (c'est-à-dire dès lors que leur modèle d'interconnexion est *ad hoc*), (iii) les interactions avec des services développés avec des intergiciels hétérogènes ne sont pas possibles.

Le recourt à une infrastucture réseau n'est pas une obligation pour que l'ensemble des *objets communicants* d'un environnement interagisse. L'intergiciel WSAMI [165] permet de découvrir et de composer les services deployés dans un environnement quel que soit le modèle d'interconnexion (*ad hoc* ou infrastructure) de l'objet communicant, limité ou non en ressources, sur lequel il est déployé. Cet intergiciel est conçu suivant des technologies standards telles que SLP pour la découverte de services, WSDL pour langage de description de service, et SOAP comme protocole de communication, permettant une interopérabilité avec les services distants, pas forcément dévoloppés avec l'intergiciel WSAMI, mais utilisant les mêmes protocoles d'interaction. Selon une démarche similaire, la solution présentée dans la référence [185] introduit quant à elle un intergiciel utilisant UPnP pour la découverte (plus précisément UPnP/SSDP), la communication (*via* le protocole SOAP) et la description des services (*via* des documents XML). L'utilisation, par exemple, d'un protocole standard comme SOAP pour la communication, permet l'utilisation de ponts afin de garantir l'interopérabilité vers d'autres intergiciels (comme présenté dans le chapitre 3), mais cela reste une solution statique qui ne prend pas en compte l'aspect dynamique d'un environnement ubiquitaire.

Pour résoudre dynamiquement l'hétérogénéité des intergiciels, beaucoup d'efforts ont été réalisés, ces dernieres années, dans la conception d'intergiciels ubiquitaires reconfigurables tel que Runes [187], Plugin-ORB [186], Base [184], Wings [188]. Ces intergiciels implémentent le principe de la *substitution de protocoles* suivant : (i) différentes technologies de composants logicielles et/ou (ii) des différentes contraintes matérielles des objets communicants sur lesquels ils sont destinés à être déployés. Dans tous les cas, chacun de ces intergiciels ubiquitaires fournit une API, qui leur est spécifique, pour le développement de leurs applications respectives. Par conséquent contrairement à notre solution, l'interopérabilité n'est pas transparente.

5.4 Synthèse

Dans ce chapitre nous avons proposé deux systèmes : INDISS et NEMESYS. Le premier résout les incompatibilités entre les protocoles de découverte de services tandis que le second résout celles des protocoles de communication RPC. Ces deux systèmes valident l'architecture logicielle déduite de la spécification formelle de la *traduction dynamique de protocoles* que nous avons proposée dans le chapitre 4. INDISS et NEMESYS étant conçus selon le même modèle, ils partagent un certain nombre de caractéristiques communes dont les suivantes :

- Ils se déploient et opèrent au dessus du système d'exploitation et en dessous de l'intergiciel de l'hôte sur lequel ils sont déployés

- Ils interceptent de façon transparente les messages en provenance du réseau *via* le système d'exploitation, les traduit et les redirige vers l'intergiciel.

- Ils s'adaptent dynamiquement à leur contexte et effectuent une *traduction dynamique de protocoles* en fonction des protocoles utilisés dans l'*environnement ubiquitaire* et de ceux utilisés par l'hôte sur lequel ils sont déployés.

- Ils peuvent être déployés indifféremment sur l'hôte d'un consommateur ou d'un fournisseur de services que sur une passerelle.

Il ressort de ces caractéristiques que les incompatibilités de protocoles entre les clients et les services d'une *architecture de services ubiquitaires* sont résolues de façon transparente sans requérir de modifications de leur intergiciel respectif. Enfin, comme démontré par les prototypes d'INDISS et de NEMESYS, les performances de la *traduction dynamique de protocoles* sont encourageantes. A partir des différentes expérimentations, il apparaît que le coût en temps de la *traduction de protocoles* se répercute peu sur le temps de latence nécessaire pour qu'un client obtienne une réponse d'un service distant.

6 Conclusion

La vision de l'informatique diffuse (ID) telle qu'introduite par Weiser [58] devient une réalité grâce aux progrès réalisés ces dernières années dans le domaine de la miniaturisation de composants électroniques, au faible coût des *objets communicants* et à l'émergence et l'expansion des technologies réseaux sans fil basées ou non sur des infrastructures (WLAN, Bluetooth, GSM, GPRS, UMTS) [55], [56].

L'objectif de l'ID est de permettre aux utilisateurs d'accéder à des ressources et/ou à des services n'importe où et à tout instant à travers divers objets communicants. Pour réaliser cette vision, ces objets doivent être en mesure de découvrir les services de leur environnement immédiat et d'interagir avec eux indépendamment de l'hétérogénéité matérielle et logicielle de leur hôte. Les intergiciels ont été justement introduits en ayant pour objectif de concevoir des services indépendamment, de ces hétérogénéités [2], [3]. Cependant, l'émergence d'intergiciels différents, pour prendre en compte les spécificités/contraintes de certains domaines d'applications et/ou d'infrastructures réseaux particulières, aboutit inévitablement à un nouveau problème d'hétérogénéité de plus haut niveau [136], [120], [134]. Cette hétérogénéité constitue un obstacle majeur à l'établissement d'une *architecture de services ubiquitaires* où les intergiciels dont dépendent les services ne sont pas connus à l'avance. En d'autres termes, en fonction de son environnement immédiat, un *objet communicant* qui n'est pas équipé de l'intergiciel adéquat se retrouve isolé. Par la suite, nous résumons la nature de nos démarches à la fois pour identifier, raisonner et proposer différentes solutions permettant de résoudre cette problématique, pour enfin introduire une ouverture sur les perspectives futures de nos travaux.

6.1 Contributions

Afin de déterminer les différentes solutions possibles pour résoudre l'hétérogénéité des intergiciels dans un *environnement ubiquitaire*, ce dernier est envisagé comme un système distribué de systèmes hétérogènes. Puis, nous l'avons modélisé sous la forme d'un graphe de connecteurs, représentant les intergiciels, et de composants, représentants les clients/services. Ainsi, déterminer si un client dépendant d'un intergiciel A est en mesure d'interagir avec le service d'un intergiciel B revient à raisonner à un niveau plus abstrait sur l'incompatibilité de couplage entre composants et connecteurs, et à raisonner sur des problèmes d'incompatibilités de protocoles que nous formalisons grâce à l'algèbre de processus. L'objectif de cette modélisation/formalisation est de présenter les différentes solutions existantes indépendamment des contraintes et des caractéristiques des technologies les mettant en œuvre. Nous avons en particulier exposé et détaillé deux spécifications formelles correspondant à la *substitution* et à la *traduction de protocoles* qui nous semblent être deux approches particulièrement représentatives des solutions existantes. Nous avons par ailleurs illustré la validité de nos spécifications formelles en présentant plusieurs technologies provenant tant du domaine industriel qu'académique, les mettant en

œuvre, comme par exemple, les intergiciels réflexifs, les ponts logiciels, les services Web, les EAIs, et les ESBs. En particulier, la *substitution de protocoles* requiert l'implémentation de nouveaux intergiciels, et donc de nouveaux clients/services, afin qu'ils s'adaptent dynamiquement à leur environnement d'exécution. Le concept de l'adaptation dynamique est un avantage certain dans le domaine de l'ID. En revanche le fait de concevoir de nouveaux intergiciels et de devoir ré-implémenter en conséquence les clients/services existants signifie que :

1. L'interopérabilité est non transparente
2. Une nouvelle source d'hétérogénéité émerge entre ces nouveaux intergiciels.

Ces deux inconvénients constituent un handicap majeur pour l'ID. Comparativement, la mise en œuvre du principe de la *traduction de protocoles* ne requiert ni la conception de nouveaux intergiciels, ni la modification des clients/services existants : l'interopérabilité est transparente. Toutefois, elle est statique, et doit être planifiée à l'avance.

Ainsi, il apparaît que la *substitution* et la *traduction de protocoles* sont deux approches complémentaires. Nous avons *fusionné* ces approches afin de définir les principes de *la traduction dynamique de protocoles* qui consistent à résoudre dynamiquement les incompatibilités de protocoles entre des intergiciels hétérogènes existants. Notre solution a l'avantage d'être à la fois dynamique et transparente : son principe est d'intercepter à la volée le trafic entre l'intergiciel d'un client et celui d'un service afin de résoudre leurs incompatibilités de protocole. Une des contributions de ce document est la spécification formelle de *la traduction dynamique de protocoles* qui permet de résoudre l'hétérogénéité des intergiciels quelles que soient les spécificités de leurs caractéristiques, de leurs protocoles et de leurs technologies. Enfin, nous avons démontré qu'il était possible de déduire à partir de notre formalisation une architecture logicielle ouvrant la voie à l'implémentation d'un système correspondant. Une autre contribution de cette thèse, est la mise en œuvre des systèmes INDISS et NEMESYS spécifiquement conçus pour résoudre les incompatibilités de protocoles de découverte de services et de communication de type RPC entre les clients et les services d'une *architecture de services ubiquitaires*. L'évaluation des prototypes de ces systèmes a révélé que le coût en temps de la *traduction de protocoles* se répercutait peu sur le temps de latence usuellement observé lors d'interactions natives (c'est-à-dire sans traduction) entre un client et un service. Cependant la *traduction dynamique de protocoles* (modélisée par le connecteur C-UNIV) n'est pas sans contrainte.

En effet, une de ses contraintes inhérente au concept même de la traduction, est de pouvoir traduire uniquement les protocoles partageant un minimum de similitudes. Si l'on considère divers protocoles réseaux, cela revient à garantir l'*interopérabilité* entre les fonctionnalités que les uns et les autres partagent, c'est-à-dire à assurer l'*interopérabilité* de l'intersection des fonctionnalités offertes par ces derniers. Ainsi, le principe de C-UNIV, dans certaines conditions, peut paraître trop restrictif, voire

inapplicable dans certains domaines. Une autre approche pertinente, à l'opposé de la nôtre, est de garantir le fonctionnement de l'union des fonctionnalités de différents protocoles *via* la conception d'un nouveau protocole regroupant l'ensemble de leurs fonctionnalités [168]. L'inconvénient de cette démarche est qu'elle est non transparente, c'est-à-dire qu'elle requiert le développement de nouveaux clients/services. Par ailleurs, cette approche nécessite l'utilisation d'une plateforme réalisant, à un moment donné ou à un autre, *la substitution de protocoles* afin de garantir l'interopérabilité avec les services existants [169]. Toutefois, notre solution est valide dans la mesure où elle permet d'intégrer les services dans un réseau domestique sans aucunes modifications ou développements supplémentaires. En effet, les concepts de C-UNIV, ainsi que les prototypes INDISS et NEMESYS ont été, avec succès, intégrés dans le projet Européen Amigo [127] afin de déployer une *architecture de services ubiquitaires* dans l'habitat.

6.2 Perspectives

Le sytème INDISS est un système de *traduction dynamique de protocoles* conçu pour résoudre les incompatibilités des protocoles de découverte de services d'une infrastructure réseau sans fil. L'émergence de nombreux protocoles de découverte de services, incompatibles entre eux, dédiés aux réseaux mobiles *ad-hoc* (MANETs pour Mobile *Ad hoc* Network) tels que par exemple, *Scalable Service Discovery* (SSD) [189], *Group Service Discovery* (GSD) [191], [192], ALLIA [190], SLP-based [193], et leur utilisation par un nombre croissant d'intergiciels, crée une nouvelle source d'hétérogénéité parmi ces derniers [194]. Le système INDISS doit être étendu, par l'adjonction de nouvelles unités pour résoudre également les incompatibilités de ces protocoles émergents. De la même façon, NEMESYS est une implémentation particulière de la *traduction dynamique de protocoles* pour les protocoles RPC. NEMESYS doit être étendu pour prendre en charge des protocoles de communication suivant un modèle d'interaction autre que synchrone, c'est-à-dire des protocoles asynchrones tels que le *message queuing*, le *message passing* ou le *publish-subscribe*.

Bibliographie

[1] ISO 7498-1.(1994). *Information processing systems – Open Systems Interconnection – Basic Reference Model: The Basic Model*. Technical report, International Standards Organisation.

[2] S. E. Mullender. (1993). *Distributed Systems*. Addison-Wesley, 2nd Edition.

[3] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and design*. Addison-Wesley, 3rd Edition, 2000.

[4] W. Emmerich. (2000). *Engineering Distributed Objects*. John Wiley & Sons, Apr.

[5] W. Emmerich. (2000). *Sofware Engineering and Middleware: A Roadmap*.

[6] J. Gray. (1978). *Notes on Database Operating Systems*. Operating systems: An advanced course. Volume 60 of *Lecture Notes in Computer Science,* pages 393--481. Springer.

[7] J. Gray, A. Reuter. *Transaction Processing: Concepts and techniques*. (1993). Morgan Kaufman Publishers.

[8] G. Vossen, G. Weikum. (2001). *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control*. Morgan Kaufman Publishers.

[9] S. B. Davidson, H. Garcia-Molina, D. Skeen. (1985). *Consistency in Partitioned Networks*. ACM Comp. Surveys, vol. 17, no. 3.

[10] K. P. Birman. (1997). *Building Secure and Reliable Network Applications.* Manning Publishing.

[11] N. Davies, H.-W. Gellersen. (2002). *Beyond Prototypes: Challenges in Deploying Ubiquitous Systems*. IEEE Pervasive computing.

[12] D. A. Menascé. (2005). *MOM vs. RPC: Communication Models for Distributed Applications*.

[13] J.E. White. (1976). *High-level Framework for Network-based Resource Sharing*. RFC 707; www.ietf.org/rfc/rfc707.txt.

[14] A.D. Birrell, B.J. Nelson. (1984). *Implementing Remote Procedure Calls*. *ACM Trans. Computer Systems*, vol. 2, no. 1, pp. 39–59.

[15] E. Gamma, R. Helm, R. Johnson, et J. Vlissides (1995). *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[16] Object Management Group. *The Common Object Request Broker: Architecture and Specification.* http://www.omg.org.

[17] Object Management Group. *CORBAservices: Common Object Services Specification.* http://www.omg.org.

[18] Object Management Group. *Event Service Specification.* http://www.omg.org.

[19] Object Management Group. *Notification Service.* http://www.omg.org.

[20] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, S. Weerawarana. (2002). *Unraveling the Web Services Web. An introduction to SOAP, WSDL, and UDDI.* Internet Computing. Vol. 6, Nr. 2. IEEE 2002. pp. 86-93.

[21] W. Grosso. (2001). *Java RMI*. O'Reilly & Associates.

[22] A. Tanenbaum. (1995). *Distributed Operating Systems*. Prentice Hall, Inc.

[23] OASIS. (2002). *Universal Discovery, Description, and Integration (UDDI), Version 3.0.*

[24] W3C. *Extensible Markup Language (XML).* http://w3.org/

[25] W3C. *SOAP*. http://w3.org/

[26] W3C. (2003). *Web Services Description Language (WSDL) Version 1.2*. W3C working draft. www.w3.org/TR/wsdl12/.

[27] L. Opyrchal, M. Astley, J. Auerbach, G. Banavar, R. Strom, D. Sturman. (2000). *Exploiting IP Multicast in Content-Based Publish-Subscribe Systems*. Proceedings of IFIP/ACM International Conference on Distributed Processing (Middleware 2000). New York, USA, pp. 185-207. Springer- Verlag.

[28] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, D. Sturman. (1999). *An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems*. Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS'99), Austin, TX, USA, pp. 262-272.

[29] P.A. Bernstein. (1996) .*Middleware : A model for distributed system services*. Communications of the ACM 39, no. 2, 86–98.

[30] J. Bacon, K. Moody, J. Bates, R. Hayton, C. Ma, A. McNeil, O. Seidel, M. Spiteri. (2000). *Generic Support for Distributed Applications*. IEEE Computer, 33, 68-76.

[31] A. Carzaniga, D. S. Rosenblum, A. L. Wolf. (2001). *Design and Evaluation of a Wide-Area Event Notification Service.* ACM Transactions on Computer Systems, Vol. 19, No. 3, Pages 332–383.

[32] G. Banavar, T. Chandra, R. Strom, D. Sturman. (1999). *A Case for Message Oriented Middleware*. In LNCS 1693. Springer Verlag, pages 1-18.

[33] L. Gilman and R. Schreiber. (1996). *Distributed Computing with IBM MQSeries.* Wiley.

[34] M. Hapner, R. Burridge, and R. Sharma. *Java Message Service Specification*. http://java.sun.com/products/jms.

[35] Microsoft. *Microsoft Message Queuing (MSMQ).* http://www.microsoft.com/

[36] D. S. Rosenblum, A. L. Wolf. (1997). *A design framework for Internet-scale event observation and notification*. Proceedings of the Sixth European Software Engineering Conference. Lecture Notes in Computer Science, no. 1301, pp. 344–360, Springer–Verlag.

[37] G. Cugola, E. Di Nitto, A. Fuggetta. (1998). Exploiting an event-based infrastructure to develop complex distributed systems. In *Proceedings of the 20th International Conference on Software Engineering (ICSE '98)* (Kyoto, Japan), pp. 261–270.

[38] Object Management Group. *Minimum CORBA Specification*. **http://www.omg.org.**

[39] M. Román, A. Singhai, D. Carvalho, C. Hess, and R.H. Campbell. (1999). *Integrating PDAs into Distributed Systems: 2K and PalmORB. International Symposium on Handheld and Ubiquitous Computing (HUC'99)*, Karlsruhe, Germany.

[40] M. Romàn, F. Kon, R. Campbell. (2001). *Reflective Middleware: From Your Desk to Your Hand*. IEEE Distributed Systems Online, 2(5).

[41] G. S. Blair, G. Coulson, P. Robin, M. Papathomas. (2000). *An Architecture for Next Generation Middleware*. Proceedings of Middleware 2000, Lake District, UK.

[42] G. S. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R. Moreira, N. Parlavantzas, and K. Saikoski. (2001). *The Design and Implementation of Open ORB version 2*. IEEE Distributed Systems Online Journal, vol. 2, no. 6.

[43] M. Addlesee, R. Curwen, S. Hodges, J. Newman, P. Steggles, A. Ward, A. Hopper. (2001). *Implementing a Sentient Computing System*. *IEEE Computer Magazine*, vol. 34, no. 8, pp. 50-56.

[44] M. Satyanarayanan. (2001). *Pervasive Computing: Vision and challenges*. IEEE Personal Communications.

[45] G. Cugola, E. Di Nitto, and A. Fuggetta. (2001). *The JEDI event-based infrastructure and its application to the development of the OPSS WFMS*. *IEEE Trans. On Software Engineering*, vol. 27, pp. 827–850.

[46] B. Segall *et al.* (2000). *Content Based Routing with Elvin4*. *Proceeding of AUUG2K*, (Canberra, Australia).

[47] R. Meier, V. Cahill. (2002). *Taxonomy of Distributed Event Based Programming Systems*. Proceedings of the International Conference on Distributed Computing Systems (ICDCS/DEBS'02), Vienna.

[48] *M. P. Papazoglou, D. Georgakopoulos.* (2003). Service-Oriented Computing. Communications of the ACM, 46(10).

[49] *M. N. Huhns, M. P. Singh.* (2005).Service-oriented computing: key concepts and principles. IEEE Internet Computing, 9(1):75-81

[50] E. Guttman, C. Perkins, J. Veizades, and M. Day. (1999). *Service Location Protocol, Version 2*. IETF RFC 2608, Network Working Group.

[51] C. Bettstetter and C. Renner. (2000). *A Comparison of Service Discovery Protocols and Implementation of the Service Location Protocol*. Proceedings of the 6th EUNICE Open European Summer School: Innovative Internet Applications.

[52] R. E. Schantz and D. C. Schmidt. (2001). *Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications*. Encyclopedia of Software Engineering (J. Marciniak and G. Telecki, eds.), New York: Wiley & Sons.

[53] C. Mascolo, L. Capra, W. Emmerich. (2002). *Middleware for Mobile Computing*. Advanced lectures on networking, vol 2497. Lecture Notes in Computer Science, Springer, Berlin Heidelberg New York, pp 20–58.

[54] M. Satyanarayanan. (1996). *Fundamental Challenges in Mobile Computing*. Proceedings of Symposium on Principle of Distributed Computing, (PODC).

[55] R. Want, T. Pering. (2005). *System Challenges for Ubiquitous & Pervasing Computing*. Proceedings of the 27th international conference on Software engineering, (ICSE'05), pp 9-14, NEW York, NY, USA. ACM Press.

[56] D. Saha, A. Mukherjee.(2003). *Pervasive Computing: A Paradigm for the 21st Century*. IEEE Computer Society, 36(2):25-31.

[57] M. Addlesee, R. Curwen, S. Hodges, J. Newman, P. Steggles, A. Ward, A. Hopper. (2001). *Implementing a sentient computing system*. IEEE Computer 34:50-56.

[58] M. Weiser. (1991). *The computer for the 21st century*. Scientific American, vol. 265, no. 3, pp. 94-104.

[59] B. P. Crow, I. Widjaja, J. G. Kim, P. T. Sakai.(1997). *IEEE 802.11 Wireless Local Area Networks*. IEEE Communications Magazine.

[60] M. Musolesi, C. Mascolo, S. Hailes. (2006). *EMMA: Epidemic Messaging Middleware for Ad hoc networks*. Personal and Ubiquitous Computing. Springer. 10(1), pp. 28-36.

[61] R. Meier, V. Cahill. (2002). *STEAM: Event-Based Middleware for Wireless Ad Hoc Networks*. Proceedings of the International Workshop on Distributed Event-Based Systems (ICDCS/DEBS'02), Vienna, Austria.

[62] A. Zeidler, L. Fiege. (2003). *Mobility Support with REBECCA*. Proceedings of the 23rd international Conference on Distributed Computing Workshops (ICDCSW'03).

[63] T.S. Rappaport. (2002). *Wireless Communications: Principles and Practice*. Prentice Hall.

[64] M. Boulkenafed, V. Issarny. (2003). *A Middleware for Mobile Ad Hoc Data Sharing, Enhancing Data Availability*. In proceedings of the 4th ACM/IFIP/USENIX International Middleware Conference.

[65] G. H. Forman, J. Zahorjan. (1994). *The Challenges of Mobiles Computing*. IEEE Computer, 27(4), pp. 38-47.

[66] R. Klefstad, S. Rao, D. Schimidt. (2003). *Design and Performance of a Dynamically Configurable Messaging Protocols Framework for Real-time CORBA*. Proceedings of Distributed Object and Component-based Software Systems part of Software Technology Track at the 36th Annual Hawaii International Conference on System Sciences.

[67] M. Haahr, R. Cunningham, V. Cahill. (1999). *Supporting CORBA in a Mobile Environment (ALICE)*. Proceedings of Mobile Computing and Networking (MobiCom). ACM Press, pp 36-47.

[68] P. Reynolds, R. Brangeon. *Service Machine Development for an Open Longterm Mobile and Fixed Network Environment*. DOLMEN Consortium.

[69] A. D. Joseph, J. A. Tauber, M. F. Kaashoek. (1997). *Mobile Computing with Rover Toolkit*. IEEE Transactions on Computers: Special issue on Mobile Computing, 46, 3, pp 337-352.

[70] F. Kon, T. Yamane, C. Hess, R. Campbell, and M. D. Mickunas. (2001). *Dynamic Resource Management and Automatic Configuration of Distributed Component Systems*. 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'2001), San Antonio, Texas.

[71] G. Mühl, L. Fiege, F. Gärtner, A. Buchmann. (2002). *Evaluating Advanced Routing Algorithms for Content-Based Publish/Subscribe Systems*. Proceeding of 10th IEEE International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunications Systems, pp 167-176, Fort Worth, Texas.

[72] L. Fiege, F. Gärtner, O. Kasten, A. Zeidler. (2003). *Supporting Mobility in Content Based Publish/Subscribe Middleware*. Proceedings of Middleware 2003, Rio de Janeiro, Brazil.

[73] P. Sutton, R. Arkins, B. Segall. (2001). *Supporting Disconnectedness – Transparent Information Delivery for Mobile and Invisible Computing*. IEEE International Symposium on Cluster Computing and the Grid, Brisbane, Australia.

[74] T. Wall, V. Cahill. (2001). *Mobile RMI : Supporting Remote Acess to Java Server Objects on Mobile Hosts*. Proceedings of the international Symposium on Distributed Objects and Applications (DOA 2001).

[75] G. Biegel, V. Cahill, M. Haahr. (2002). *A Dynamic Proxy-Based Architecture to Support Distributed Java Objects in Mobile Environments*. Proceedings of the international Symposium on Distributed Objects and Applications (DOA 2002).

[76] Object Management Group. (2003). *Wireless Access and Terminal Mobility in CORBA, v1.0*. http://www.omg.org.

[77] A. L. Murphy, G. P. Picco, G.-C. Roman. (2006). *LIME: A Coordination model and middleware supporting mobility of hosts and agents*. ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 15, no. 3, pp. 279-328.

[78] A. L. Murphy, G. P. Picco, G.-C. Roman. (2001). Lime: A Middleware for Physical and Logical Mobility. In Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21).

[79] M. Mutka and D. Zhu. (2004). *Promoting Cooperation Among Strangers to Access Internet Services from an Ad Hoc Network*. Proceedings of IEEE International Conference on Pervasive Computing and Communications, (PerCom'04)

[80] B. Wang, J. Bodily, and S. K. S. Gupta. (2004). *Supporting Persistent Social Groups in Ubiquitous Computing Environments Using Context-Aware Ephemeral Group Service*. Proceedings of IEEE International Conference on Pervasive Computing and Communications (PerCom'04).

[81] M. Beigl, T. Zimmer, A. Krohn, C. Decker, P. Robinson. (2004) *Creating Ad-Hoc Pervasive Computing Environments*. Proceedings of the Second International Conference on Pervasive Computing, Linz/Vienna, Austria.

[82] S. Chetan, J. Al-Muhtadi, R. Campbell, M. D. Mickunas. (2005). Mobile Gaia : A Middleware for Ad-hoc Pervasive Computing. IEEE Consumer Communications & Networking Conference (CCNC 2005), Las Vegas.

[83] M. Mamei, F. Zambonelli. (2004). *Programming Pervasive and Mobile Computing Applications with TOTA Middleware*. Proceeding of Second IEEE International Conference on Pervasive Computing and Communications (PerCom'04).

[84] M. O. Killijian, R. Cunningham, R. Meier, L. Mazare, V. Cahill. (2001). *Towards group communication for mobile participants*. *POMC01*, pages 75–82. Newport, RI, USA, 2001.

[85] J.D. Day, H. Zimmermann. *The OSI Reference Model*. Proc. IEEE, Vol. 71,No. 12, pp. 1334-1340.

[86] G. J. Holzmann. (1991). *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[87] R. Allen, D. Garlan.(1997). *A Formal Basis for Architectural Connection*. ACM Transactions on Software Engineering and Methodology.

[88] N. Medvidovic, R. N. Taylor. (2000). *A Classification and Comparison Framework for Software Architecture Description Languages.* IEEE Transactions on Software Engineering, vol. 26, no. 1, pp. 70-93.

[89] V. Issarny, F. Tartanoglu, J. Liu, F. Sailhan,. (2004). *Software Architecture for Mobile Distributed Computing*. Proceedings of the 4th IEEE/IFIP Conference on Software Architecture (WICSA).

[90] V. Issarny. (1997). *Configuration-Based Programming Systems*. Proceedings of SOFSEM'97: Theory and Practice of informatics, pages 183-200. Springer-Verlag.

[91] D. Garlan.(2003). *Formal Modelling and Analysis of Software Architecture: Components, Connectors, and Events*. Third International School on Formal Methods for the Design of Computer, Communication and Software systems: Software Architectures. LNCS 2804, pp. 1-24.

[92] R. Allen. (1997). *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science. Issued as CMU Technical Report CMU-CS-97-144.

[93] R. J. Allen, R. Douence, D. Garlan. (1998). *Specifying and Analyzing Dynamic Software Architectures*. Proceedings of the 1998 Conference on Fundamental Approaches to Software Engineering (FASE '98).

[94] D. Garlan, R. T. Monroe, D. Wile. (1997). *Acme: An Architecture Description Interchange Language*. Proceedings of CASCON'97, Pages 169-183.

[95] D. Garlan, R. T. Monroe, D. Wile. (2000). *Acme: Architectural Description of Component-Based Systems.* Foundations of Component-Based Systems, Gary T. Leavens and Murali Sitaraman (eds), Cambridge University Press, pp. 47-68.

[96] D. Garlan, R. Allen, J. Ockerbloom. (1995). *Architectural Mismatch: Why Resuse is So Hard*. IEEE Software, 12(6):17-26.

[97] C. Hoare.(1986). *Communicating Sequential Processes.* Prentice-Hall.

[98] R. Milner.(1989). *Communication and concurrency.* Prentice-Hall.

[99] M. Shaw, D. Garlan (1996). *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall.

[100] N. R. Mehta, N. Medvidovic, S. Phadke. (2000). *Towards a taxonomy of software connectors*. Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000), pp. 295-304.

[101] M. Shaw, P. Clements. (1997). *A field guide to boxology: Preliminary classification of architectural styles for software systems*. Proceedings of 21st International Computer Software and Applications Conference (COMPSAC97), pp. 6-13.

[102] N. Medvidovic, D.S. Rosenblum, R.N. Taylor. (1999). *A Language and Environment for Architecture-Based Software Development and Evolution*. Proceedings of 21st International Conference Software Engineering (ICSE '99), pp. 44-53.

[103] P. Oreizy, N. Medvidovic, R.N. Taylor. (1998). *Architecture-Based Runtime Software Evolution*. Proceedings of 21st International Conference Software Engineering (ICSE '98), pp. 177-186.

[104] J. Magee, J. Kramer. (1996). *Dynamic Structure in Software Architectures*. Proceedings of ACM SIGSOFT '96: Fourth Symposium Foundations of Software Eng. (FSE4), pp. 3-14.

[105] J. Magee, J. Kramer. (1999). *Concurrency: State Models and Java Programs*. John Wiley and Sons.

[106] B. Spitznagel, D. Garlan. (2003). *A Compositional Formalization of ConnectorWrappers*. Proceedings of the 2003 International Conference on Software Engineering (ICSE'03), Portland, Oregon, USA.

[107] B. Spitznagel. (2004). *Compositional Transformation of Software Connectors*. PhD thesis, Carnegie Mellon, School of Computer Science.

[108] M. Shaw. (1995). *Architectural issues in software reuse: it's not the functionality, it's the packaging*. Proceedings of the Symposium on Software Reuse (SSR'95).

[109] P. Grace, G. S. Blair, S. Samuel. (2005). *A Reflective Framework for Discovery and Interaction in Heterogeneous Mobile Environments*. ACM SIGMOBILE Mobile Computing and Communications Review, 9(1), pp 2-14, special section on Discovery and Interaction of Mobile Services.

[110] C. Szyperski. (1998). *Component Sofware: Beyond Object-Oriented Programming*. Addison-Wesley/ACM Press.

[111] T. Meijler et O. Nierstrasz. (1997) *Beyond objects: Components*. *Cooperative Information Systems: Current Trends and Directions*. Academic Press.

[112] Kiczales, G., J. des Rivières, D.G. Bobrow. (1991). *The Art of the Metaobject Protocol*. MIT Press.

[113] F. Kon, F. Costa, G.S. Blair, R. Campbell. (2002). *The Case for Reflective Middleware: Building Middleware that is Flexible, Reconfigurable, and yet simple to Use*. CACM, Vol. 45, No. 6.

[114] F. Kon, M. Roman, P. Liu, J. Mao, T. Yamane, L. Magalhaes, R. Campbell. (2000). *Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB*. Proceedings of Middleware 2000, ACM/IFIP.

[115] T. Ledoux. (1999). *OpenCorba: a Reflective Open Broker.* 2nd International Conference on Reflection and Meta-level Architectures, St. Malo, France.

[116] G. S. Blair, G. Coulson, P. Grace.(2004) . *Research Directions in Reflective Middleware : the Lancaster Experience.*

[117] OSGI Alliance. (2003). Open Service Gateway Initiative: *The OSGi Service Platform Specification, Release 3*. http://www.osgi.org

[118] D. A. Chappell. (2004). *Enterprise Service Bus*. O'Reilly

[119] H. Cervantes, R. Hall. (2004). *Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model*. Proceedings of the 26th International Conference of Software Engineering (ICSE 2004). Edinburgh, Scotland: ACM Press, pp. 614–623.

[120] P. Grace. (2004). *Overcoming Middleware Heterogeneity in Mobile Computing Applications*. PhD Thesis, Lancaster University, March 2004.

[121] Microsoft. *COM Spécification*. http://www.microsoft.com/com.

[122] Sun. *Enterprise Java Bean*. http://java.sun.com/products/ejb.

[123] Object Management Group. *Corba Component Model.* http://www.omg.org.

[124] ObjectWeb. *Fractal Framework*. http://fractal.objectweb.org.

[125] Microsoft. *.Net*. http://www.microsoft.com/net.

[126] G. Coulson, G. S. Blair, P. Grace, A. Joolia, K. Lee, and J. Ueyama. (2004). *A Component Model for Building Systems Software*. Proceedings of IASTED Software Engineering and Applications (SEA'04), Cambridge MA, USA.

[127] AMIGO. http://www.hitech-projects.com/europrojects/amigo/

[128] H. Cervantes. (2004). Vers un Modèle à Composants Orienté Services pour Supporter la Disponibilité Dynamique. Thèse, université Joseph Fourier.

[129] SOAP2CORBA. **http://soap2corba.sourceforge.net**

[130] Iona Technologies. *OrbixCOMet*. http://www.iona.com/support/whitepapers/ocomet-wp.pdf

[131] RMI-IIOP. http://java.sun.com/products/rmi-iiop

[132] IIOP-.NET. http://iiop-net.sourceforge.net/index.html

[133] A. Slominski, M. Govindaraju, D. Gannon, R. Bramley. (2001). *Design of an XML based Interoperable RMI System: SoapRMI C++/Java 1.1.* Proceedings of Parallel and Distributed Processing Techniques and Applications Conference.

[134] L. Pautet. (2001). *Intergiciels schizophrènes : une solution à l'interopérabilité entre modèles de répartition*. Technical report, Université Pierre et Marie Curie - Paris VI.

[135] G. Alonso, F. Casati, H. Kuno, V. Machiraju. (2004). *Web Services. Concepts, Architectures and Applications.* Springer-Verlag.

[136] S. Vinoski.(2003). *Integration with Web Services*. IEEE Internet Computing.

[137] D. A. Chappell. (2004). *Enterprise Service Bus*. O'Reilly.

[138] M. Duftler, N. Mukhi, A. Slominski and S. Weerawarana. (2001). *Web Services Invocation Framework (WSIF).* Proceedings of OOPSLA 2001 Workshop on Object Oriented Web Services, Tampa, Florida.

[139] IONA. *Artix.* http://iona.com/products/middlewareint.htm

[140] A. Tanenbaum. (2003). *Computer Networks*. Fourth Edition. Pearson Education, Inc.

[141] T. Bolognesi et E. Brinksma. (1987). *Introduction to the ISO Specification Language LOTOS*. Computer Networks and ISDN Systems, vol. 14, no. 1, pp.25-59.

[142] S. S. Lam. *Protocol conversion-Correctness problems*. (1986). Proceedings of *ACM SIGCOMM.*

[143] S. S. Lam. *Protocol conversion.* (1988). IEEE Trans. Software Eng., vol. 14,

[144] D. S. Linthicum. (1999). *Enterprise Application Integration*. Addison-Wesley.

[145] S. C. Cheung, J. Kramer. (1999). *Checking Safety Properties Using Compositional Reachability Analysis*. ACM Transactions on Software Engineering and Methodology. Vol. 8, No. 1, pp49-78.

[146] D. Giannakopoulou, J. Magee, J. Kramer. (1999). *Checking Progress with Action Priority : Is it Fair ?*. 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'99), Toulouse, France, 1687, pp. 511-527.

[147] G. Pujolle. (2000). *Les réseaux*. Eyrolles.

[148] N. D. Ryan, A. L. Wolf. (2004). *Using Event-Based Translation to Support Dynamic Protocol Evolution*. ICSE 2004: 408-417

[149] J. Waldo. (1999). *Jini Architecture for network centric computing*. Communication of the ACM**,** 42(7).

[150] R. Gupta, S. Talwar, D. P. Agrawal. (2002)**.** *Jini Home Networking : A Step toward Pervasive Computing***.** IEEE Computer Society, 35(8): 34-40.

[151] UPnP Forum. (2003). UPnP™ Device Architecture 1.0. http://www.upnp.org/resources/documents.asp, Version 1.0.1.

[152] Salutation Consortium, Salutation Architecture Specification. http://www.salutation.org.

[153] Y. Goland, T. Cai, P. Leach, Y. Gu, S. Albright. (1999). *Simple Service Discovery Protocol 1.0*. http://www.upnp.org/download/draft_cai_ssdp_v1_03.txt, Internet Draft.

[154] Y. D. Bromberg, V. Issarny. (2004). *Service Discovery Protocol Interoperability in the Mobile Environment*. Proceedings of the International Workshop Software Engineering and Middleware (SEM'04).

[155] Y. D. Bromberg, V. Issarny. (2005). *INDISS: Interoperable Discovery System for Networked Services*. Proceedings of ACM/IFIP/USENIX 6th International Middleware Conference (Middleware'2005).

[156] Z. Albanna, K. Almeroth, D. Meyer, M. Schipper. *IANA Guidelines for IPv4 Multicast Address Assignments*. IETF RFC 3171, Network Working Group.

[157] Y.Y. Gloand, T. Cai, P. Leach. (2000). *SSDP: Simple service discovery protocol*. IETF Draft, http://ietf.org.

[158] E. Guttman, C. Perkins, J. Kempf. (1999). *Service Templates and Service: Schemes*. IETF RFC 2609, Network Working Group.

[159] S. Preuss. (2002). *JESA Service Discovery Protocol*. Proceedings of Networking pp. 1196-1201, Pisa, Italy.

[160] Apple Computer. *Bonjour*. **http://developer.apple.com/networking/bonjour/index.html**

[161] D. Steinberg, S. Cheshire. (2005). *ZeroConfiguration Networking: The Definitive Guide*. O'Reilly.

[162] E. Guttman. (2001). *Autoconfiguration for IP Networking : Enabling Local Communication*. IEEE Internet Computing**.**

[163] S. Helal, C. Lee. (2002). *Protocols for Service Discovery in Dynamic and Mobile Networks*. International Journal of Computing Research, vol.22, no.1, pp.1-12.

[164] N. Georgantas, S. Ben Mokhtar, Y.-D. Bromberg, V. Issarny, J. Kalaoja, J. Kantarovitch, A. Gérodolle, R. Mevissen. (2005).*The Amigo Service Architecture for the Open Networked Home Environment.* Proceedings of 5th Working IEEE/IFIP Conference on Software Architecture (WICSA).

[165] V. Issarny, D. Sacchetti, F. Tartanoglu, F. Sailhan, R. Chibout, N. Levy, A. Talamona. (2005). *Developing Ambient Intelligence Systems: A Solution based on Web Services*. Journal of Automated Software Engineering. Vol 12.

[166] D. Garlan, D. Siewiorek, A. Smailagic, and P. Steenkiste. (2002). *Project Aura: Toward distraction-free pervasive computing*. IEEE Pervasive Computing, 1(2):22-31.

[167] M. Román, C. K. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, K. Nahrstedt. (2002). *Gaia: A Middleware Infrastructure to Enable Active Spaces*. IEEE Pervasive Computing, pp. 74-83.

[168] P.-G. Raverdy, V. Issarny, R. Chibout, A. La Chapelle. (2006). *A Multi-Protocol Approach to Service Discovery and Access in Pervasive Environments*. Proceedings of MOBIQUITOUS - The 3rd Annual International Conference on Mobile and Ubiquitous Systems: Networks and Services.

[169] Y.-D. Bromberg, V. Issarny, P.-G. Raverdy. (2006). *Interoperability of Service Discovery Protocols: Transparent versus Explicit Approaches*. Proceedings of the 15th IST Mobile & Wireless Communications Summit. Myconos, Greece.

[170] B. Neuman. (1994). *Scale in Distributed Systems*. Readings in Distributed Computing Systems, IEEE Computer Society Press, pp. 69-78.

[171] Charles E. Perkins. (2001). *Ad Hoc Networking*. Addion Wesley Professional.

[172] RUNES Consortium. (2005). *Survey of Middleware for Networked Embedded Systems*. Deliverable D5.1. http://www.ist-runes.org/docs/deliverables/D5_01.pdf

[173] N. Georgantas, S. Ben Mokhtar, F. Tartanoglu, V. Issarny. (2005). *Semantic-aware Services for the Mobile Computing Environment*. Architecting Dependable Systems III, LNCS 3549.

[174] Amigo Consortium. (2005). *Deliverable D3.1: Detailed Design of the Amigo Middleware Core*. http://www.hitech-projects.com/europrojects/amigo/

[175] N. Georgantas, S. Ben Mokhtar, Y.-D. Bromberg, V. Issarny, J. Kalaoja, J. Kantarovitch, A. Gérodolle, R. Mevissen. (2005). *The Amigo Service Architecture for the Open Networked Home Environment*. Proceedings of the 5th IEEE/IFIP Working Conference on Software Architecture.

[176] S. Ben Mokhtar, N. Georgantas, V. Issarny. (2006). *COCOA: COnversation-based Service COmposition in PervAsive Computing Environments*. Proceedings of the IEEE International Conference on Pervasive Services (ICPS'06).

[177] S. Ben Mokhtar, A. Kaul, N. Georgantas, V. Issarny. (2006). *Towards Efficient Matching of Semantic Web Service Capabilities*. Proceedings of the international workshop on Web Services Modeling and Testing (WS-MATE'06).

[178] *C. Dabrowski, K. Mills, S. Quirolgico.* (2005). *A Modelbased Analysis of First-Generation Service Discovery Systems.* Special Publication 500-260, National Institute of Standards and Technology.

[179] R.S. Marin-Perianu, P.H. Hartel, J. Scholten. (2005). *A Classification of Service Discovery Protocols.* Technical Report TR-CTIT-05-25 Centre for Telematics and Information Technology, University of Twente, Enschede. ISSN 1381-3625

[180] A. Friday, N. Davies, E. Catterall. (2001). *Supporting Service Discovery, Querying and Interaction in Ubiquitous Computing Environments*. Proceedings of MobiDE.

[181] C. Bettstetter, C. Renner. (2000). *A Comparison Of Service Discovery Protocols and Implementation of The Service Location*. Proceedings of 6th EUNICE Open European Summer School: Innovative Internet Applications (EUNICE).

[182] S. Ben Mokhtar, A. Kaul, N. Georgantas, V. Issarny. *Efficient Semantic Service Discovery in Pervasive Computing Environments*. Proceedings of the ACM/IFIP/USENIX 7th International Middleware Conference (Middleware'06).

[183] L. Arnstein, R. Grimm, C.-Y. Hung, J. H. Kang, A. LaMarca, S. B. Sigurdsson, J. Su, and G. Borriello. (2002). *Systems Support for Ubiquitous Computing: A Case Study of Two Implementations of Labscape*. Proceedings of IEEE International Conference on Pervasive Computing and Communications, (PerCom'02).

[184] C. Becker, G. Schiele, H. Gubbels, K. Rothermel. (2003). *BASE: A Micro-Broker-Based Middleware for Pervasive Computing*. Proceedings of IEEE PerCom'03, pp 443-451.

[185] T. Nakajima, I. Satoh. (2004). *Personal Home Server: Enabling Personalized and Seamless Ubiquitous Computing Environments*. Proceedings of IEEE PerCom'04, 341-345

[186] A. Acierno, G. De Pietro, A. Coronato, G. Gugliara. (2005). *Plugin-Orb for Applications in a Pervasive Computing Environment*. Proceedings of the 2005 International Conference on Pervasive Systems and Computing.

[187] P. Costa, G. Coulson, C. Mascolo, G. P. Picco, S. Zachariadis. (2005). *The RUNES Middleware: A Reconfigurable Component-based Approach to Networked Embedded Systems*. Proceedings of the IEEE International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC'05).

[188] E. Loureiro, F. Bublitz, N. Barbosa, A. Perkusich, H. Oliveira de Almeida, G. Ferreira. (2006). *A Flexible Middleware for Service Provision Over Heterogeneous Pervasive Networks*. Proceedings of the IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM 2006).

[189] F. Sailhan, V. Issarny. *Scalable Service Discovery for MANET*. (2005). *Proceedings of the 3rd IEEE PerCom*, Kauai Island, USA.

[190] O. Ratsimor, D. Chakraborty, A. Joshi, T. Finin. (2002). *Allia: Alliance based service discovery for ad-hoc environments*. Proceedings of the 2nd international Workshop on Mobile Commerce, Atlanta, Georgia, USA.

[191] D. Chakraborty, A. Joshi, T. Finin, Y. Yesha. (2002). *GSD: A Novel Group-based Service Discovery Protocol for MANETs*. Proceedings of the 4th IEEE MWCN, Stockholm, Sweden, September 2002.

[192] D. Chakraborty, A. Joshi, Y. Yesha, T. Finin. (2006). *Toward distributed service discovery in pervasive computing environments*. IEEE Transactions on Mobile Computing, 5(2):97-112.

[193] S. Penz. (2005). *SLP-based Service Management for Dynamic Ad-hoc Networks. Proceedings of the 3rd International Workshop on MPAC, Grenoble, France*.

[194] C. Cortes, G. Blair, P. Grace. (2006). *A Multi-protocol Framework for Ad-hoc Service Discovery***.** Proceedings of the 4th International Workshop on on Middleware for Pervasive and Ad-Hoc Computing (MPAC '06), co-located with Middleware 2006, Melbourne, Australia.