

Une Architecture pour les Services Télécom

Arnaud Fontaine

▶ **To cite this version:**

Arnaud Fontaine. Une Architecture pour les Services Télécom. domain_other. Télécom ParisTech, 2004. English. NNT : . pastel-00001427

HAL Id: pastel-00001427
https://pastel.hal.science/pastel-00001427

Submitted on 15 Nov 2005

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Thèse

présentée pour obtenir le grade de docteur

de l'Ecole Nationale Supérieure des Télécommunications

Spécialité : Informatique et Réseaux

Arnaud Fontaine

Une Architecture basée composants pour les Services Télécom

soutenue le 21 juin 2004 devant le jury composé de :

Guy Bernard	INT	Président
Marie-Pierre Gervais	Lip6	Rapporteur
Richard Castanet	Labri	Rapporteur
Sylvie Vignes	ENST	Examinateur
Michel Ruffin	Alcatel	Examinateur
Elie Najm	ENST	Directeur de Thèse

Ecole Nationale Supérieure des Télécommunications

PhD Thesis

Ecole Nationale Supérieure des Télécommunications

Computer Science and Network Department

Arnaud Fontaine

A Component based Architecture for telecom services

Defence date : 21 june 2004

Committee in charge :

Guy Bernard	INT	Chairman
Marie-Pierre Gervais	Lip6	Reporter
Richard Castanet	Labri	Reporter
Sylvie Vignes	ENST	Examiner
Michel Ruffin	Alcatel	Examiner
Elie Najm	ENST	Advisor

Ecole Nationale Supérieure des Télécommunications

A mon directeur de thèse Elie Najm, qui réussit à me transmettre sa passion,
A Michel Ruffin (Alcatel) qui demeura un soutien inébranlable, malgré mes multiples investigations et changements de cap,
A Caroline mon épouse qui sut trouver les mots et l'attention, sans lesquels ce manuscrit n'aurait peut-être jamais vu le jour.

TABLE DES MATIÈRES

1 INTRODUCTION 14

2 NOTRE MODÈLE MÉTIER 18

2.1 INTRODUCTION 18

2.2 LES HYPOTHÈSES FONDATRICES 18

2.2.1 LA TÉLÉPHONIE SUR IP ET LES SERVICES TÉLÉCOMS 18
2.2.2 L'UTILISATEUR FINAL 19
2.2.3 SESSIONS ET ACCÈS 20
2.2.4 LA MOBILITÉ 22
2.2.5 LA FACTURATION 23
2.2.6 LA PERSONNALISATION DES SERVICES 23
2.2.7 LA COMPOSITION DE SERVICES 24
2.2.8 UNE CONCEPTION ORIENTÉE MODÈLES 24

2.3 LES RÔLES MÉTIER 25

2.3.1 NOTRE MÉTHODOLOGIE 25
2.3.2 LE FOURNISSEUR D'ACCÈS 25
2.3.2.1 Patrimoine 25
2.3.2.2 Offre de services 26
2.3.2.3 Dépendances 26
2.3.3 LE DÉTAILLANT 27
2.3.3.1 Patrimoine 27
2.3.3.2 Offre de services 27
2.3.3.3 Dépendances 27
2.3.4 LE FOURNISSEUR DE SERVICES TIERS 27
2.3.4.1 Patrimoine 27
2.3.4.2 Offre de services 28
2.3.4.3 Dépendances 28
2.3.5 LE FOURNISSEUR DE CONNECTIVITÉ 28
2.3.5.1 Patrimoine 28
2.3.5.2 Offre de services 28
2.3.5.3 Dépendances 28
2.3.6 LE GESTIONNAIRE DE FACTURATION 28
2.3.6.1 Patrimoine 28
2.3.6.2 Offre de services 29
2.3.6.3 Dépendances 29

2.4 EXEMPLE : LE SERVICE DE MISE EN RELATION 29

2.5 CONCLUSION 31

3 LA CRÉATION DU SERVICE 32

3.1 INTRODUCTION 32

3.2 ETAT DE L'ART SUR L'UTILISATION DE COMPOSANTS TÉLÉCOM 32
3.2.1 PRÉSENTATION 32
3.2.2 L'EXISTANT 32
3.2.3 UN BESOIN DE MODULARISER LES SERVICES 35
3.2.4 QU'EST CE QU'UN COMPOSANT 36
3.2.5 LA RELATION RESSOURCES – COMPOSANTS 37
3.3 LE CYCLE DE CRÉATION DE NOS COMPOSANTS 38
3.4 L'ANALYSE DU SERVICE 39
3.4.1 LE DIAGRAMME DE « USE CASE » 39
3.4.1.1 Présentation 39
3.4.1.2 Etude de cas 41
3.4.2 LES DIAGRAMMES D'ACTIVITÉ 43
3.4.2.1 Présentation 43
3.4.2.2 La transition Use Case vers Diagramme d'activité 44
3.4.2.3 Extension du diagramme d'activité 48
3.4.2.4 Etude de cas 51
3.4.3 LE DIAGRAMME DE COMPOSANTS 56
3.4.3.1 présentation 56
3.4.3.2 L'algorithme de transformation 57
3.4.3.3 Le formalisme de description de l'activité des composants 63
3.5 LE DÉVELOPPEMENT DU SERVICE 76
3.5.1 PRÉSENTATION 76
3.5.2 LA GÉNÉRATION DU CODE 76
3.5.3 FACETTES ET RÉCEPTACLES 78
3.5.4 LE BROKER 80
3.5.5 LES COMPOSANTS SESSION / UTILISATEUR 83
3.5.5.1 Un besoin de spécialisation 83
3.5.5.2 Le composant Session 83
3.5.5.3 Le composant Utilisateur 84
3.5.5.4 Un exemple : le tableau blanc 84
3.6 CONCLUSION 86
4 L'INTÉGRATION DU SERVICE 87
4.1 INTRODUCTION 87
4.2 PRÉSENTATION 87
4.3 LES COMPOSANTS MÉTIER 87
4.4 LES FACETTES ET RÉCEPTACLES MÉTIER 89
4.4.1 LES FACETTES MÉTIER 89
4.4.2 LES RÉCEPTACLES MÉTIER 92
4.5 LES FACETTES ET RÉCEPTACLES DE COLLABORATION 92
4.5.1 PRÉSENTATION 92

4.5.2 LA DÉFINITION DES FACETTES ET RÉCEPTACLES DE COLLABORATION 92
4.5.3 L'INTERACTION ENTRE RÉCEPTACLES ET FACETTES DE COLLABORATION 94

4.6 L'ARCHITECTURE D'EXÉCUTION 96

4.7 CONCLUSION 101

5 L'EXPLOITATION DU SERVICE 102

5.1 INTRODUCTION 102

5.2 PRÉSENTATION 102

5.3 L'UTILISATION DU SERVICE DE PRÉSENCE 102

5.4 SIP 105

5.4.1 LE CHOIX DE SIP 105
5.4.2 L'ARCHITECTURE AUTOUR DE SIP 105
5.4.3 LE PRINCIPE DE FONCTIONNEMENT DE SIP 106
5.4.4 L'ACCÈS AUX SERVICES 109
5.4.5 UNE ORGANISATION AUTOUR DE SERVEURS 111
5.4.6 SIP COMME PROTOCOLE DE PRÉSENCE 112
5.4.7 SIP ET NOS COMPOSANTS 113

5.5 LA PA-ARCHITECTURE 116

5.5.1 VUE D'ENSEMBLE 116
5.5.2 PRÉSENTATION DÉTAILLÉE DE L'ARCHITECTURE 117
5.5.2.1 La relation application-AUA 117
5.5.2.2 La relation AUA-PA 126
5.5.2.3 La relation AUA-SA 129
5.5.2.4 La relation entre Services et PA/SA 133
5.5.3 EXEMPLE DE SESSION DE SERVICE 135

5.6 LA GESTION DU COMPORTEMENT 139

5.6.1 LES BASES DU COMPORTEMENT 139
5.6.1.1 Etat de présence – Statut de l'application 139
5.6.1.2 CPL 140
5.6.1.3 La programmation par contrat 140
5.6.2 LA DÉFINITION DES CONTRATS 141
5.6.2.1 Les contrats 141
5.6.2.2 Les règles de coordination 142
5.6.2.3 Les dépendances entre contrats 146
5.6.3 LA PRISE EN CHARGE DES CONTRATS 147
5.6.4 L'UTILISATION DES CONTRATS DE COORDINATION 148
5.6.4.1 La composition de services 148
5.6.4.2 La personnalisation du comportement des services 153

6 CONCLUSION ET PERSPECTIVES 157

7 BIBLIOGRAPHIE 158

8 ANNEXES 160

8.1 ANNEXE 1 : FICHIER WSDL DU COMPOSANT SERVICE1 160

8.2 ANNEXE 2 : FLOT DE MESSAGES SIP, POUR UNE CONNEXION UTILISANT DEUX PROXY. 162

8.3 ANNEXE 3 : FLOT DE MESSAGES SIP, POUR L'UTILISATION D'UN LOCATION SERVEUR. 169

TABLE DES ILLUSTRATIONS

Figure 1 La structure en oignon de nos services *16*
Figure 2 La relation entre accès au réseau et accès aux services *20*
Figure 3 L'accès au réseau *21*
Figure 4 Les rôles métier en lien direct avec l'utilisateur *25*
Figure 5 Les interactions métiers associées au service de messagerie instantanée *30*
Figure 6 Une implémentation du service de mise en relation par carte *33*
Figure 7 La Pattern sur laquelle repose nos composants *37*
Figure 8 Le cycle de création d'un composant *38*
Figure 9 Un Use Case *40*
Figure 10 L'utilisation de la relation stéréotypée "include" *40*
Figure 11 Cas d'utilisation du service de Paiement *41*
Figure 12 Raffinement du cas d'utilisation "Payer" *42*
Figure 13 Un exemple de diagramme de Use Case *45*
Figure 14 Le diagramme d'activité issu de la transformation *47*
Figure 15 Utilisation de gardes sur les transitions *48*
Figure 16 L'utilisation du stéréotype "présentation" *49*
Figure 17 Utilisation du stéréotype "Resource" *50*
Figure 18 Utilisation du stéréotype "Exception" *51*
Figure 19 Le diagramme d'activité généré automatiquement *52*
Figure 20 Le diagramme d'activité avec les stéréotypes "Presentation" *53*
Figure 21 Le diagramme d'activité avec les stéréotypes "Resource" *54*
Figure 22 Le diagramme d'activité intégrant les chemins alternatifs *55*
Figure 23 Réceptacles et Facettes *56*
Figure 24 Les points d'activité *57*
Figure 25 La fusion de réceptacles *60*
Figure 26 L'activité à transformer *60*
Figure 27 Les composants générés automatiquement *61*
Figure 28 Le diagramme de composants généré automatiquement *62*
Figure 29 La décomposition du chemin reliant deux points d'activité *63*
Figure 30 La relation entre une facette et le cœur du composant *63*
Figure 31 Un exemple de point d'activité aux retours multiples *66*
Figure 32 La relation entre points d'activité locaux et distants *67*

Figure 33 Le développement du point d'activité local L2 ..... *68*
Figure 34 La connexion au flux de retour *69*
Figure 35 La connexion au flux d'erreur *70*
Figure 36 La connexion au point d'activité principal *71*
Figure 37 La prise en charge du retour des points d'activité secondaires *72*
Figure 38 Le composant dans son environnement d'exécution *73*
Figure 39 Le composant BlackList statique *73*
Figure 40 Le composant BlackList dynamique *73*
Figure 41 Le diagramme de composants associé à notre service de Paiement *75*
Figure 42 Le document WSDL de description d'une facette *78*
Figure 43 La définition des facettes et réceptacles techniques *79*
Figure 44 La définition d'une relation entre classes de composants *81*
Figure 45 La spécialisation d'une relation entre instances de composants *82*
Figure 46 Les composants associés au service "WhiteBoard" *85*
Figure 47 La description d'une facette métier *91*
Figure 48 La description d'une facette de collaboration *94*
Figure 49 Liens entre facettes et réceptacles *95*
Figure 50 L'incarnation d'un réceptacle de collaboration *96*
Figure 51 Schéma de construction d'une pile d'intercepteurs *98*
Figure 52 L'architecture d'exécution de la méthode m1() *100*
Figure 53 Le Service de présence *104*
Figure 54 Le Processus de notification d'un changement d'état *105*
Figure 55 Etablissement d'une connexion RTP en utilisant SIP *108*
Figure 56 La demande d'accès au service *109*
Figure 57 La page d'accès au service *109*
Figure 58 L'accès au Service *110*
Figure 59 L'utilisation d'un serveur de localisation *112*
Figure 60 L'utilisation de l'API Jain SIP Lite *114*
Figure 61 Notre architecture de présence *116*
Figure 62 La relation entre applications et événements à destination des services *118*
Figure 63 La prise en charge des événements au niveau de l'AUA *121*
Figure 64 L'envoi du message SIP par le Stub *121*
Figure 65 La relation entre AUA et PA *123*
Figure 66 L'initialisation de la relation AUA-SA *130*
Figure 67 Les relations entre AUA, SA et Services *131*

Figure 68 La structure d'un message SIP NOTIFY de corps SOAP..... 132
Figure 69 Le message SIP NOTIFY associé à l'appel de la méthode addELement(Element) .. 133
Figure 70 Ouverture de l'application utilisateur..... 136
Figure 71 Ouverture d'une session de service..... 137
Figure 72 L'invitation d'un utilisateur..... 137
Figure 73 Le flux d'information entre applications utilisateur..... 138
Figure 74 Le retrait d'un utilisateur..... 139
Figure 75 Un exemple de contrat..... 141
Figure 76 La spécification d'une règle de coordination..... 143
Figure 77 L'héritage de contrats..... 147
Figure 78 Le gestionnaire de contrats dans notre architecture d'exécution..... 148
Figure 79 La répartition des contrats sur les agents de service..... 154

1 Introduction

L'ambition initiale de notre démarche était de mettre en exergue les mécanismes intervenant dans les problématiques de composition et de personnalisation de services télécoms, pour être en mesure de proposer dans un second temps une solution générique de création de ces services. Après avoir investigué les différentes solutions existantes (Réseau Intelligent, TINA, H323, CCM, WebServices, …) pendant les trois premières années de ma thèse, et compte tenu des orientations technologiques actuelles du monde des services que j'ai pu percevoir au cours de mes années de conseil aux entreprises, il m'est apparu qu'une solution à base de composants logiciels apportait la souplesse recherchée pour la construction d'une architecture de services télécoms. Notre volonté n'était cependant pas de réutiliser dans l'état telle ou telle autre technologie de composants, mais de proposer un processus complet de création de services basés sur des composants aux interfaces collaboratives novatrices. Si mon travail se caractérise par une forte partie abstraite, de réflexion sur la conception de services, il a été néanmoins accompagné par de nombreuses expérimentations sur les serveurs d'applications du marché (à composante Java/Corba ou .NET/SOAP) partiellement évoquées au long de ce manuscrit.

Il est à noter qu'un certain nombre des solutions proposées dans ce manuscrit peuvent être retranscrites dans des domaines fonctionnels éloignés du monde des télécoms. En effet, si la physionomie de nos composants a été commandée par le contexte particulier des télécoms (personnalisation, contraintes de qualité de service, contrepartie et interaction de services) la démarche en elle-même de construction des services peut être réutilisable pour des composants d'une autre nature et dans des domaines des plus variés.

Le premier constat que nous pourrions dresser serait la trop grande distorsion entre les modèles théoriques très prometteurs qu'ont apporté TINA ou ODP (où des solutions techniques étaient apportées à des problématiques métier), et les standards que sont SIP ou MEGACO, dont les études publiques restent bien souvent confinées au simple aspect technique, sans se soucier de leur impact dans les métiers des acteurs des télécoms. Ensuite, les travaux existants sur la création de service ont tendance à isoler les deux processus que sont le développement du service en tant que tel et l'insertion du service dans son environnement. Eprouvée et combattue dans le monde informatique, cette approche ne permet pas de traiter convenablement les contraintes d'intégration dans, et d'interaction avec, le système d'information existant. En effet ce fonctionnement engendre presque toujours des services théoriquement performants, mais non moins souvent mal adaptés à leur utilisation réelle, et dont l'adaptation en fin du processus de création de service peut s'avérer fort onéreuse. L'arrivée du développement basé modèle, et notamment le Model Driven Architecture de l'OMG, est de nature à faciliter le développement des services. Mais les différents types de modèles intervenants dans les chacune des phases du développement ne sont pas encore bien identifiés, et certains, notamment ceux permettant de décrire le comportement interne des composants, font défaut. Enfin, la réflexion de personnalisation du comportement des services se porte d'avantage au niveau du service, qu'au niveau de l'architecture. Ce manque de flexibilité dans les architectures ainsi générées, de par l'absence d'interfaces standard de communication entre services, contrarie les approches de collaboration de services, au profit de l'utilisateur final.

Sur ces bases, dans cette thèse nous proposons un processus novateur de création de service basé sur l'utilisation successive de modèles de composants logiciels, de la phase de conception jusqu'au développement et assemblage des composants constitutifs des services. Le fil conducteur de notre travail a été d'identifier, formaliser et expérimenter des solutions techniques applicables aux composants, pour faciliter leur utilisation dans un environnement d'exploitation ouvert. Les modèles de composants permettent de raffiner progressivement les traitements interne du service, et ses communications avec son environnement. Les notations utilisées sont basées sur UML, en profitant de ses possibilités d'extensions par le biais de stéréotypes, ou d'une nouvelle notation dans le cadre du comportement interne des composants, car aucune notation existante ne convenait totalement à cet usage. Pour que cette définition de modèles ne se limite pas à une simple énumération de propriétés abstraites, nous avons eu le souci de proposer des modèles allant jusqu'à la mise en oeuvre des services dans une architecture basée sur la signalisation SIP existante. Avec ce dernier modèle nous démontrons comment le fonctionnement des services peut tirer parti des moyens de collaborations évolués définis entre les composants. Enfin, notre contribution veut clairement se situer au niveau de l'architecture des services, en proposant un modèle flexible d'exécution des services, articulé autour d'un service de présence. Cette propriété permettra ainsi de rapprocher l'exécution des services de l'état des utilisateurs, pour faciliter leur intégration dans le mode de vie de l'abonné. Comme prolongement de cette malléabilité de notre architecture de services, nous réutilisons le concept de la programmation par contrats, pour la matérialisation des prérogatives de l'utilisateur final sur l'exécution de ses services.

Notre premier axe de réflexion (Chapitre 2) s'est orienté vers l'édification d'un modèle métier rassemblant les différents acteurs des télécoms. Pour chaque rôle identifié, nous avons mis en évidence un ensemble de fonctionnalités caractéristiques, pour préciser dans un second temps leurs relations de dépendance. Ce modèle introduit ainsi la notion de Détaillant, comme acteur principal de la relation avec l'utilisateur final, capable de centraliser et rendre disponible des services proposés par un fournisseur de services tiers. Un Gestionnaire de Facturation peut proposer à nos deux acteurs un ensemble de composants capables d'assumer des fonctionnalités liées à la facturation, tandis qu'un fournisseur de connectivité s'intéresse aux caractéristiques de la connexion de l'utilisateur, prisent en compte au moment de l'évaluation des contraintes de qualité de service. La définition de ces différents acteurs a motivé notre choix de concevoir des composants aux interfaces reconfigurables. En effet, dans une architecture multi-fournisseurs comme la nôtre, un service offert à l'utilisateur final par le Détaillant, en tant qu'unité fondamentale, peut en réalité être le résultat d'une collaboration entre composants issus d'une multitude de fournisseurs. Or, les relations entre fournisseurs étant sujettes à des évolutions continuelles, les services ciblés par notre architecture doivent reposer sur un environnement aisément modelable, afin de préserver le contrat établi entre le Détaillant et l'utilisateur final et ce malgré les vicissitudes des relations entre fournisseurs.

Un élément important de notre travail a été ensuite de revisiter le processus de création de service (Chapitre 3), en considérant l'usage de composants. Nous avons fait le choix d'utiliser le formalisme d'UML (Unified Modeling Language) pour préciser au fur et à mesure des phases d'analyse du service, les composants mobilisés par le service en cours de création. Si

cette démarche se limite dans un premier temps à une réutilisation de diagrammes UML existants, avec l'introductions d'algorithmes de générations de diagrammes assurant la transition d'un diagramme général à un diagramme plus fin, notre contribution s'est étendue à la définition d'un nouveau type de diagramme, inspiré de la philosophie graphique d'UML, que nous avons baptisé diagramme de composants. Si la démarche classique, rendue possible grâce à l'existence du méta modèle d'UML, est la définition de profile UML pour étendre les diagrammes UML classiques (tel [Ref] pour les CCM), notre type de diagramme ne trouvant pas d'équivalent dans les diagrammes UML actuels, nous avons entrepris de définir une nouvelle classe de diagramme. Ce type de diagramme est capable de préciser le comportement interne d'un composant (traitements locaux, interactions avec des composants distants), afin de réduire l'espace séparant les phases d'analyse et de développement du service. Dans notre processus de création de service, la phase d'analyse permet la génération du code d'implémentation du composant. Aussi, pour permettre une composabilité et une configurabilité optimale de nos composants, nous avons conçu leur structure telles les enveloppes d'un oignon.

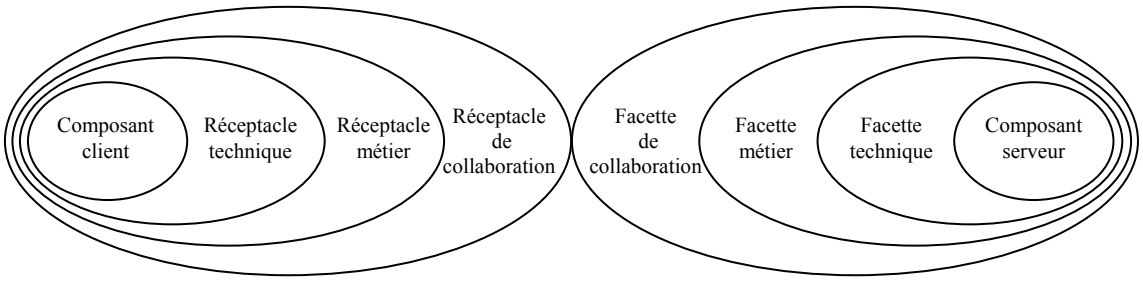


Figure 1 La structure en oignon de nos services

Avec cette construction présentée dans le chapitre 4, le code d'implémentation classique représente l'enveloppe interne du composant, enrobée de membranes technique, métier et de collaboration. Ces membranes permettent de doter le composant en capacités de communication, une facette précisant les opérations offertes par un composant serveur, et les réceptacles les besoins d'un composant client en terme de composants serveur. La couche technique commence par préciser les interfaces (coté client et serveur) proposées par le composant, les paramètres d'instance configurables et les méthodes nécessitant l'utilisation de composants tiers. Si cette couche ne s'enquière pas encore des problèmes de qualité de service et des valeurs à attribuer aux paramètres de configuration, elle permet en revanche de rassembler les contraintes de dépendance entre composants, en vue de l'instauration de contrats de collaboration entre fournisseurs de composants. La couche métier est utilisée ensuite par le fournisseur du composant, pour préciser un ensemble de contraintes libres de qualité de service et de contreparties (pouvant faire appel à des services de facturation). Cette couche, qui reste encore indépendante d'une relation particulière entre composants, permet à un constructeur de services en recherche de composants, de visualiser les contraintes de qualité de service pour lesquelles le fournisseur distant est disposé à s'engager, et les contreparties demandées en cas de respect ou non de ces contraintes. Un contrat de collaboration, générant de part et d'autre de la relation des réceptacles et facettes de collaboration, vient sceller un accord entre deux fournisseurs, sur la base des interfaces métier de leurs composants respectifs. Cette dernière couche de nos composants précise les valeurs à attribuer aux contraintes libres de qualité de service et de contrepartie.

Le dernier grand chantier de notre architecture (Chapitre 5) a été la prise en compte de la variabilité de l'attente de l'utilisateur envers ses services. Sur cet aspect nous sommes partis du constat que les standards actuels comme le réseau intelligent ou les architectures plus récentes autour de H323 proposent un modèle statique de composition de services, le plus souvent dédié à la composition de services particuliers, mais faisant peu de cas de ce qu'il est commun d'appeler l'état de l'utilisateur. Aussi, pour conditionner l'exécution de services dépendant de l'utilisateur, nous avons développé trois types d'agents gravitant autour d'un service de présence. Un premier agent, déployé sur le terminal de l'utilisateur, supervise les évolutions d'une application pour le cas échéant notifier son statut de présence à l'agent de présence de l'utilisateur, déployé dans le domaine d'un Détaillant. Pour permettre aux services du Détaillant d'entrer en relation avec l'utilisateur interfacé au moyen de ses applications, nous avons introduit des agents de services, capables de maintenir les données des services exclusives à l'utilisateur, et de d'adapter le comportement du service selon les souhaits exprimés par l'utilisateur. Enfin nous avons utilisé SIP (Session Initiation Protocol), comme protocole d'échange entre des agents, chargé de véhiculer la signalisation associée à l'état de présence des applications de l'utilisateur, et la signalisation relative au fonctionnement des services.

2 Notre Modèle Métier

2.1 Introduction

Sans avoir la prétention de proposer un modèle métier abouti et donc utilisable dans l'état pour une mise en œuvre immédiate, nous allons évoquer les différentes caractéristiques que devra assumer notre architecture cible, et le modèle métier dans lequel évolueront nos services, pour nous aider à situer l'implication de chacun des acteurs métier dans le processus de création de service. Si notre architecture s'inspire fortement de TINA [TIN97], nous avons choisi de développer un modèle métier spécifique afin de prendre en compte des fonctionnalités non encore appréhendées dans ce modèle. Dans le contexte de libéralisation du monde des télécoms, la distinction accès réseau / accès services nous a amené à définir deux acteurs pouvant dans la pratique être assumés par un même intervenant. Les contraintes de mobilité, personnalisation et composition de services gouvernant notre architecture nous demandent de préciser plus que dans toute autre architecture le spectre des compétences des fournisseurs de services. Enfin, pour prendre en charge nativement la gestion de la qualité de service et de facturation qui lui est associée, et ne pas aboutir à une architecture où les questions de facturation seront "bricolées au cas par cas", nous avons défini un gestionnaire de facturation, utilisé comme médiateur impartial dans les relations entre acteurs métier.

2.2 Les hypothèses fondatrices

2.2.1 La Téléphonie sur IP et les services télécoms

Pour concevoir nos services télécoms, nous nous sommes basés sur le principe de la téléphonie sur IP. Ce terme générique désigne les nombreuses possibilités de transmission de signaux vocaux, de télécopie, etc., sur des réseaux IP à commutation par paquets. L'utilisation croissante des réseaux fondés sur le protocole Internet (IP) pour les services de communication, y compris pour les applications telles que la téléphonie, est devenue une question cruciale pour l'industrie des télécommunications dans le monde entier. La possibilité d'acheminer du trafic vocal sur des réseaux IP, avec les problèmes de qualité de service inhérents, mais aussi un formidable potentiel d'ouverture, notamment au niveau de l'intégration voix/données, contribue activement à la convergence de deux technologies qui ont vu le jour dans des contextes politiques et réglementaires très différents:

- le réseau téléphonique public commuté (RTPC), fondé sur la technologie de la commutation de circuits et qui, jusqu'à ces dernières années, était fortement réglementé dans la plupart des pays;

- l'Internet, fondé sur la technologie de la commutation par paquets, et qui s'est transformé en réseau de données est peu, voire pas du tout, réglementé.

La transmission téléphonique sur des réseaux IP peut être subdivisée en deux grandes catégories, à savoir la téléphonie IP et la téléphonie sur Internet, qui se différencient par la nature du réseau IP sous-jacent: le premier cas utilise des réseaux IP privés interconnectés, tandis que le second repose essentiellement sur l'Internet public. Plusieurs des principaux opérateurs historiques ont consenti des investissements lourds pour faire passer l'ensemble de leur trafic international par des plates-formes IP. Cette migration opérée en l'espace de quelques

années s'explique, entre autres, par le faible coût de la migration du trafic vers des réseaux IP; ainsi les opérateurs estiment que cette technologie leur permettra d'acheminer le trafic pour un coût représentant 25% du coût d'un réseau classique à commutation de circuits. La libéralisation des marchés contribue également à favoriser cette migration vers les réseaux IP. A la fin de l'année 2000, plus des trois quarts du trafic international avait pour origine des pays dans lesquels la téléphonie IP est libéralisée. Même si les avis divergent concernant la vitesse à laquelle la téléphonie IP se développera au cours des prochaines années, il est généralement admis que tout ira assez vite, en effet à l'échelle mondiale, le volume de trafic acheminé par des réseaux IP est déjà de loin supérieur au trafic téléphonique acheminé par le RTPC. Pour les consommateurs, la téléphonie IP peut permettre de réduire fortement le coût des appels téléphoniques longue distance et internationaux par rapport à l'utilisation d'une ligne fixe à commutation de circuits ou d'un réseau mobile, même si cette économie s'accompagne pour le moment d'une perte de qualité du service.

Par ailleurs, la téléphonie IP permet d'offrir aux consommateurs des services évolués intégrant la voix et les données, à l'exemple des services de présence, de messagerie ou d'échange de fichiers. Ce concept de service, qui avait été introduit dans le RTPC avec le réseau intelligent, a été considérablement étendu et ouvert avec les architectures applicatives TINA et plus récemment avec l'utilisation des protocoles H323 et SIP au dessus d'IP. Toutes architectures ont en commun de permettre à l'abonné de dépasser le simple appel de base du réseau téléphonique commuté, en accédant à des entités fonctionnelles autonomes appelées services, constituées de briques fonctionnelles élémentaires ou éléments de services.

2.2.2 L'utilisateur final

Alors que la plupart des Modèles sont orientés autours des métiers des différents acteurs du monde des télécoms, nous avons voulu concevoir une architecture articulée autour d'un utilisateur global (dont l'identité, l'état et les actions ne sont pas propres à un acteur particulier). Dans notre architecture, l'utilisateur est doté d'un comportement caractérisé par des actions menant à l'exécution de services offerts par le réseau. Pour formaliser ces actions, et fournir un environnement d'exécution aux services, l'utilisateur dispose de terminaux jouant le rôle d'interface avec le réseau.

Dans les architectures existantes, comme le réseau intelligent, l'utilisateur est intimement lié à son terminal. Ainsi un appel entrant sur un terminal pourra éventuellement déclencher l'exécution d'un service, en tenant compte de l'état du terminal, cet état étant assimilé à l'état de l'utilisateur. Cependant une telle représentation de l'état de l'utilisateur entraîne une inconsistance de son état global, dès lors qu'il peut être associé à plusieurs terminaux (ie l'utilisateur qui serait associé à un terminal occupé et à un terminal libre). Ainsi nous avons fait l'hypothèse que l'utilisateur disposait d'un service de présence, capable de suivre son activité, conditionnant l'exécution de services, l'état des terminaux en lien avec l'utilisateur n'étant que des composantes de l'état de l'utilisateur.

Dans ce contexte, nous avons conçu une architecture où l'utilisateur dispose de terminaux multimédia (capables d'enregistrer et jouer une grande variété de flux audio/vidéo), équipés d'une interface réseau. Notre utilisateur a ainsi la liberté de se déplacer avec ses terminaux, le réseau ayant à sa charge d'assurer une exécution de ses services conforme à son état global.

2.2.3 Sessions et Accès

Lorsqu'un utilisateur dispose sur son terminal d'un élément capable d'établir une session avec une ressource distante, on dira qu'il a accès à cette ressource. Pour ouvrir une session deux entités vont commencer par s'échanger leurs identités selon une méthode qu'elles considèreront fiable, avant d'éventuellement accepter de communiquer.

D'après ces définitions nous pouvons d'ores et déjà distinguer deux catégories d'accès, l'accès au réseau et l'accès aux services. Chacun de ces accès repose sur un ensemble imbriqué de sessions, l'accès au réseau étant un prérequis à l'accès aux services.

La session réseau représente la relation existant entre le terminal de l'utilisateur et le réseau. L'existence d'une telle session assure que l'utilisateur a la possibilité de communiquer avec le réseau, ce qui lui permet le cas échéant d'ouvrir une session de services.

Une session de services est une relation entre un utilisateur (résidant dans le contexte d'un terminal) et un fournisseur de services sur le réseau, qui repose sur une session d'accès. Cette session permet de véhiculer la signalisation à destination des services. Une session de services peut déboucher, dans le cadre de l'exécution de certains services, à l'établissement d'une session d'information au dessus de la session de services, correspondant à un flux d'information entre une source d'information et un récepteur. C'est par exemple le cas pour des services à base de transfert de fichiers (FTP).

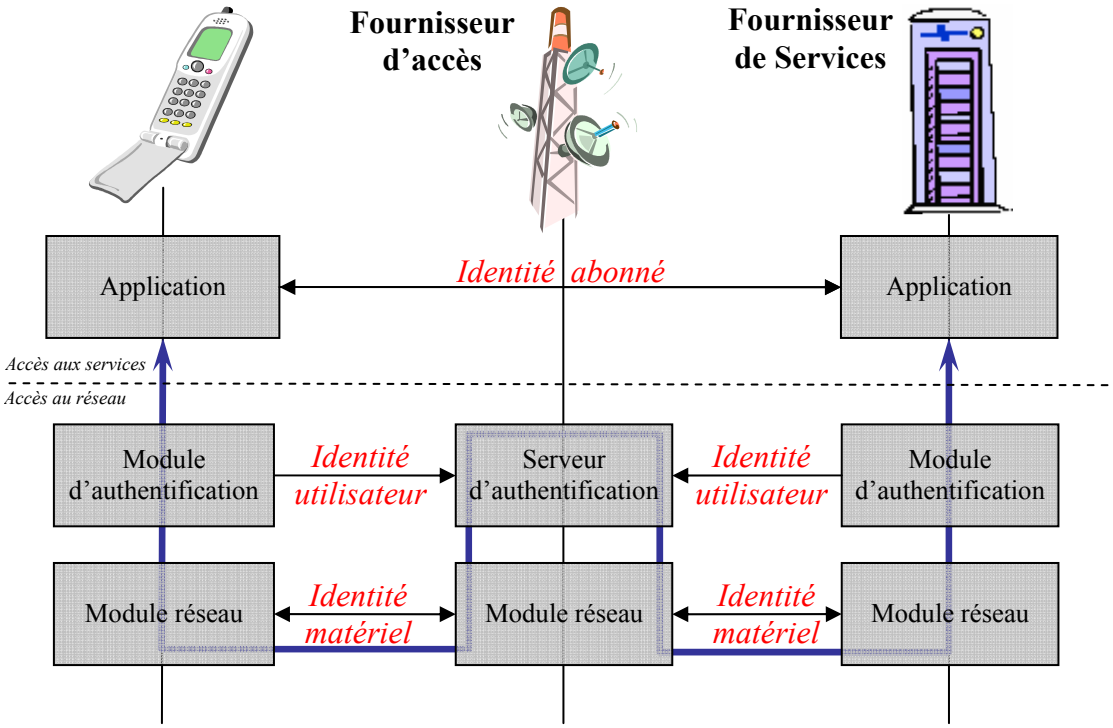


Figure 2 La relation entre accès au réseau et accès aux services

Un utilisateur désireux d'entrer en relation avec un service hébergé sur le réseau devra commencer par établir une session avec le réseau, pour enchaîner sur une phase d'accès au service. Compte tenu de la séparation des terminaux et des utilisateurs, cette première phase d'accès au réseau devra être décomposée en deux étapes : l'accès du terminal au réseau et l'accès de l'utilisateur au réseau.

L'accès du terminal au réseau, ou plus généralement l'accès au niveau matériel, est un processus passif (qui ne requière pas l'intervention de l'utilisateur). Dans le cadre des réseaux sans fil, un certain nombre de longueur d'onde seront réservées au processus d'authentification des terminaux, chaque fournisseur d'accès disposant d'un sous-ensemble de ces longueurs d'onde, utilisées par son parc de stations de base. Au cours de ce processus, les modules réseau vont s'échanger leurs identités matérielles, le fournisseur d'accès vérifiant la légitimité du terminal, avant de communiquer au terminal une seconde longueur d'onde qu'il devra utiliser pour la seconde second niveau d'accès au réseau : l'authentification de l'utilisateur. Avec ce premier niveau de l'accès au réseau, il est possible d'appliquer une politique d'accès allant de l'ouverture totale (ce que nous rencontrons dans un réseau 802.11b), à la fermeture rencontrée sur le réseau Palladium imaginé par Microsoft, qui ne permettrait qu'à des machines Palladium d'entrer en relation. Dans notre monde d'opérateurs télécom, un terminal commencerait par émettre des messages de demande d'accès sur les longueurs d'onde des opérateurs qu'il connaît, ceux-ci retournant un message d'acquittement en cas de bonne réception du message, afin de préciser au terminal la disponibilité du réseau. Le fournisseur d'accès a derrière ce mécanisme la possibilité de définir un système de contrôle basé sur des informations de plus haut niveau, comme les utilisateurs associés aux terminaux, pour limiter les références des terminaux envers lesquels il acceptera d'établir un accès de niveau 1.

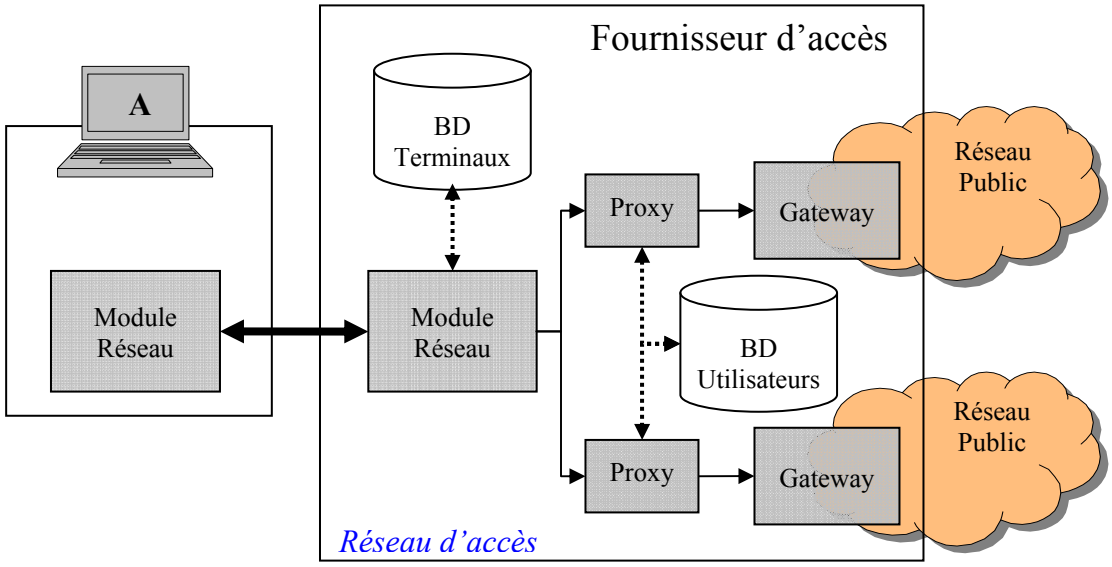


Figure 3 L'accès au réseau

L'accès au réseau de niveau 2, dit l'accès de l'utilisateur au réseau, demande cette fois une première authentification de l'utilisateur vis-à-vis du réseau. Cette authentification, réalisée par le terminal pour le compte du fournisseur d'accès, permet à ce dernier de légitimer

l'utilisateur du terminal, et ainsi de lui donner un accès au réseau conforme à ses souscriptions. La gestion de l'accès, prise en charge par un Fournisseur d'accès, passe par l'utilisation de Gateway, assurant la communication entre le réseau d'accès et le réseau public que cherche à attendre l'utilisateur (cf. Figure 3). Ce type d'architecture permet à le Fournisseur d'accès d'associer un Proxy filtrant à ses Gateway, afin de filtrer les prises en charge de certains protocoles applicatifs comme HTTP, SIP, … Le Fournisseur d'accès est ainsi une agrégation de « protocol providers » [GAL00][TIN97], chacun de ces protocoles pouvant faire l'objet d'une souscription de la part de l'utilisateur final.

Sur la base d'une session d'accès au réseau, l'utilisateur a la possibilité d'initier une session d'accès aux services. Le Détaillant de Services, également appelé Détaillant dans certaines architectures [TIN97], doit pour ce faire définir un mécanisme d'authentification de l'utilisateur, afin de garantir l'identité du client à tout fournisseur de service tiers. En effet c'est le Détaillant qui sera dans un second temps chargé de maintenir l'ensemble des informations propres à l'utilisateur, afin de les rendre disponible dans le cadre de l'exécution des services.

2.2.4 La Mobilité

La mobilité ou « roaming » est une fonctionnalité permettant à un abonné de bénéficier des services auxquels il a souscrit, en dehors du réseau géré par son opérateur. Dans le réseau GSM classique, la possibilité de bénéficier du mobilité dépend de deux paramètres : d'une part l'existence d'un accord entre l'opérateur de l'utilisateur et l'opérateur où se situe présentement l'utilisateur, d'autre part la capacité du terminal de l'utilisateur à utiliser les fréquences de communication et les protocoles d'accès imposés par l'opérateur visité. Une fois ces deux conditions remplies, l'utilisateur utilise les services de l'opérateur visité, selon une tarification négociée entre les deux opérateurs. Un utilisateur Français se rendant en Finlande effectuera donc des appels locaux (en Finlande) selon la tarification locale en vigueur, et pourra même éventuellement initier des appels vers son pays d'origine selon une tarification préférentielle.

Le mobilité s'applique dans la pratique tant à la l'accès au réseau qu'à l'accès aux services. Avec le réseau GSM où les services se limitent le plus souvent au seul appel de base, les notions d'accès au réseau et d'accès aux services sont confondues, dans la mesure où c'est le même opérateur qui assume l'exécution du service de bout en bout. Cependant, dans une architecture telle que la notre où les fournisseurs d'accès et fournisseurs de services sont deux entités distinctes, différencier les deux types de mobilité a un sens.

La mobilité d'accès au réseau s'applique quand l'utilisateur souhaite établir une session réseau avec un Fournisseur d'accès autre que son opérateur habituel. Il permet comme dans le réseau GSM d'autoriser l'utilisateur à accéder au réseau, moyennant l'application de tarifications spéciales.

La mobilité d'accès aux services, partiellement abordée avec l'architecture Camel, s'applique quand l'utilisateur a déjà établi une session de services avec un Détaillant. Il permet dans ce cas de faire bénéficier de services auxquels il aurait souscrit auprès d'autres Détaillants, en commandant leur exécution depuis le Détaillant actuellement utilisé.

Nous veillerons durant ce manuscrit à trouver un mécanisme qui permette à l'utilisateur de préciser le contexte d'exécution de ces services bénéficiant de la mobilité d'accès aux services.

2.2.5 La Facturation

La relation entre un service et son client peut être matérialisée par une entité appelée « contrat de qualité de service ». Ainsi la souscription à un service de la part d'un client mènera à l'établissement d'un contrat précisant les termes de l'utilisation du service par le client. Par ce contrat le fournisseur du service d'une part s'engage sur la mise à disposition du service, en respectant un ensemble de contraintes de qualité de service (temps de réponse, débit sur les flux, taux de perte,…), le client d'autre part s'engageant à une compensation financière pour l'utilisation du service. Cette notion de contrat se retrouve à tous les niveaux de l'architecture, tant entre les différents acteurs participant à la mise à disposition des services finaux, qu'entre les acteurs et les utilisateurs finaux.

Nous voyons de ce fait que la facturation est un problème central d'une architecture de services. [TIM98] a souligné l'importance des mécanismes de facturation, pour une juste rétribution des différentes étapes utiles à l'exécution d'un service. En effet si du point de vue du client un service est perçu comme une entité clairement délimitée, délivrée par un unique opérateur ; il est en réalité le plus souvent pris en charge par une multitude d'intervenants, s'appuyant eux aussi sur des contrats de qualité de service leur permettant de revendiquer une subdivision du capital perçu par le fournisseur de service identifié par le client.

La facturation est ainsi basée sur une relation unitaire, directement liée à l'application d'un contrat de qualité de service. Cette facturation peut être linéaire ou variable en fonction du degré de satisfaction des contraintes de qualité de service. Autour de ce service commun de facturation, notre architecture doit prévoir des mécanismes permettant la corrélation de facturations, pour permettre d'un coté la ventilation des facturations pour un même service, et simplifier de l'autre la facture, en réalisant une agrégation des différents flux comptables entre deux acteurs.

2.2.6 La Personnalisation des services

Les réseaux de seconde génération ont offert à l'utilisateur la possibilité de choisir la mélodie de son terminal, le message d'accueil de son service de messagerie ou les numéros contenus dans sa liste noire. Toutes ces nouveautés convergent vers l'appropriation des services par les utilisateurs, en leur permettant de personnaliser eux-mêmes les services auxquels ils ont souscrit.

Cette personnalisation des services, initialement prise en compte pour des services déployés directement au sein des terminaux (mélodie, filtrage des appels entrant et sortant, messagerie), se retrouve au fur et à mesure pour des services présent sur le réseau. Pour permettre une généralisation de la personnalisation de services, notre architecture doit permettre d'exhiber les paramètres personnalisables de chaque service, et doit maintenir les

valeurs définies par chaque abonné pour chacun de ces paramètres, afin de permettre une exécution des services conforme aux attentes des utilisateurs.

2.2.7 La Composition de services

La principale raison d'être de notre architecture est de proposer un modèle permettant une composition des services, de la propre initiative de l'utilisateur final. Avec cette motivation notre architecture a été pensée pour aboutir à un système apportant une facilité de composition « à la SIB ». En effet, la grande force du réseau intelligent fut de proposer un plan fonctionnel global permettant de concevoir des services par enchaînement des boites qu'étaient les SIBs. Pour ce faire, nous avons donc commencé par une réflexion sur les services eux-mêmes, avant de proposer un mécanisme permettant à l'utilisateur de définir de vrais scénarios d'enchaînements de services.

Nous verrons plus loin dans ce manuscrit que les mécanismes mis en œuvre pour appréhender la composition de services pourront être réutilisés pour traiter du problème extrêmement plus complexe qu'est l'interaction de services.

2.2.8 Une conception orientée Modèles

Avec l'arrivée des langages de modélisation, dont le plus célèbre est UML [UML04], les phases d'analyse et de conception ont pu se doter d'outils à base de modèles, capables d'automatiser la production du code d'exécution des composants logiciels. Ces langages à forte composante graphique permettent de spécifier, assembler voire même tester les différents éléments d'un ensemble complexe de composants, pour finalement générer le squelette du service en cours d'élaboration.

Face à la profusion de ces langages de modélisation, et pour en rationaliser leur utilisation, l'OMG a introduit récemment une nouvelle approche de la spécification d'applications nommée MDA (Model Driven Architecture). Cette approche prône l'utilisation de modèles indépendants de toute plateforme (PIM – Plateform Independant Model) pouvant être projetés vers un modèle spécifique à une plateforme (PSM – Plateform Specific Model). Ce découpage permet de distinguer, et ce dès l'analyse du service, la partie métier des parties purement liées à la plateforme technique cible. D'autre part, le concept d'indépendance de la plateforme permet de tutoyer la notion de pérennité de l'application, des règles de transformation d'un PIM pouvant être créées pour générer un model relatif à une plateforme qui n'existe pas encore aujourd'hui.

En référence à cette stratégie de développement MDA, notre approche de la conception de services a été d'utiliser une succession de modèles indépendants de toute plateforme, écrits en XML, en précisant néanmoins le cas échéant les règles de transformation permettant la génération d'un modèle spécifique à une plateforme Java expérimentale développées dans le cadre de ma thèse à partir du serveur d'applications JBoss.

2.3 Les Rôles métier

2.3.1 Notre méthodologie

Afin de tenir compte des spécificités des différents métiers des acteurs actuels et à venir du monde des télécoms, la construction de notre architecture a débuté par la définition d'un schéma fonctionnel, permettant d'exhiber un certain nombre de rôles et relations fonctionnels pouvant être assumés par ces acteurs. Ce schéma sans avoir la prétention d'être exhaustif sur le plan de la variété des acteurs, a été pensé pour s'articuler autour de l'utilisateur final. Dans notre architecture un rôle fonctionnel est caractérisé par trois composantes : un patrimoine architectural, une offre de services et un ensemble de dépendances.

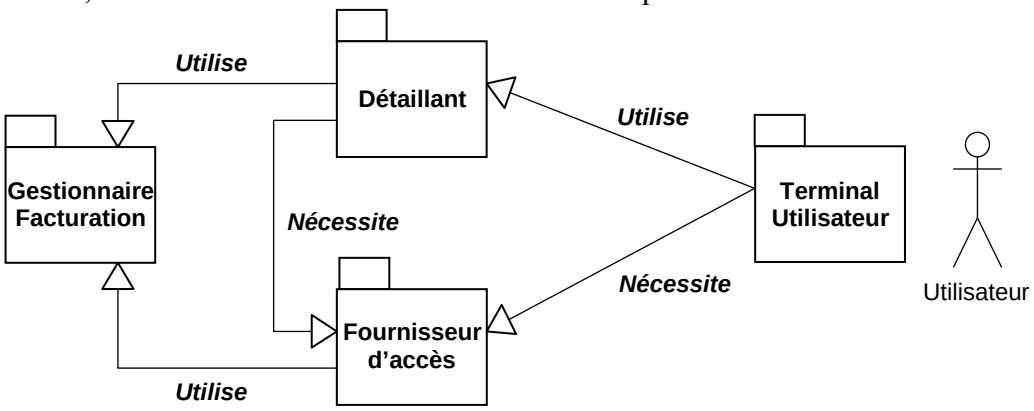


Figure 4 Les rôles métier en lien direct avec l'utilisateur

Le patrimoine est l'infrastructure matérielle et logicielle sur laquelle repose l'offre de services du fournisseur. Cette infrastructure peut être interne ou externe, ce qui occasionne dans ce dernier cas une dépendance envers un fournisseur tiers.

Une offre de services est la valeur ajoutée du fournisseur par rapport à l'utilisateur final. Cette offre occasionne une relation contractuelle avec un ensemble de fournisseurs, définissant la qualité avec laquelle les services doivent être rendus, afin d'établir les termes des dépendances entre fournisseurs.

Une dépendance résulte d'un besoin externe en vue de satisfaire une offre de services. De cette assomption il résulte qu'une dépendance non satisfaite implique une offre de services non satisfaisable.

A partir de ces définitions, nous avons dégagé un ensemble de rôles, chacun étant caractérisé par un patrimoine, une offre et des dépendances.

2.3.2 Le Fournisseur d'accès

2.3.2.1 Patrimoine

Un Fournisseur d'accès dispose d'un parc de stations de raccordement, pour permettre à l'utilisateur final de se connecter au réseau. Ces stations peuvent être filaires, pour un

raccordement par boucle locale, ou hertzienne, pour offrir une connexion par terminal mobile. Ce parc de stations de raccordement peut dans la pratique être la possession exclusive ou mutualisée du Fournisseur d'accès, ou la location d'un matériel possédé par un autre acteur (le plus souvent le propriétaire du réseau cœur).

Le patrimoine du Fournisseur d'accès s'arrête à ces stations, parfois également appelées routeur d'extrémité ou « edge router », le réseau cœur n'étant pas de son ressort.

Un Fournisseur d'accès possède un certain nombre de serveurs, pouvant être subdivisés en deux classes, les policy servers, qui sont chargés de vérifier les autorisations et d'appliquer les contraintes de qualité de service en fonction des souscriptions des utilisateurs, et les serveurs de localisation, qui sont spécialisés dans le suivi des terminaux des utilisateurs.

2.3.2.2 Offre de services

Classiquement un Fournisseur d'accès est chargé de l'attribution d'une adresse réseau à tout matériel mis en place sur le réseau, pour ensuite prendre en charge le transport de paquets vers les ports ouverts par les terminaux. Cette activité de transport de l'information s'accompagne le plus souvent d'une activité de conversion de protocoles, pour palier à l'hétérogénéité des réseaux d'accès gérés par les différents Fournisseur d'accès. Si ces activités qualifiées de bas niveau ne constituent pas des services à part entière, elles permettent en revanche de préciser les contraintes de qualité de service qui influeront sur l'exécution de services de plus haut niveau. Aussi nous pouvons faire l'hypothèse que les souscriptions de l'utilisateur auprès de son accès provider sont matérialisables sous la forme de composants, permettant à un service de s'adapter aux contraintes imposées par l'accès de l'utilisateur au réseau.

Le Fournisseur d'accès centralise également la localisation des terminaux des utilisateurs, cette information étant mise ensuite à disposition des fournisseurs de services. Cette séparation de l'accès des services permet par exemple à un fournisseur de services d'entrer en relation avec l'ensemble des Fournisseur d'accès d'un utilisateur donné, pour choisir l'accès optimal et exécuter le cas échéant un reroutage ou « handover » au niveau du Fournisseur d'accès en cas de déplacement de l'utilisateur.

2.3.2.3 Dépendances

Tout acteur de notre architecture est en lien avec au moins un Fournisseur d'accès pour pouvoir accéder au réseau. De même tout Fournisseur d'accès a besoin d'au moins un fournisseur de cœur de réseau pour satisfaire la demande d'accès au réseau de ses utilisateurs. En effet, si sa compétence se borne à l'acheminement de l'information, de l'utilisateur au réseau, il doit faire appel à un opérateur tiers pour véhiculer ce contenu à une autre extrémité du réseau.

2.3.3 Le Détaillant

2.3.3.1 Patrimoine

Un Détaillant est un fournisseur de services au détail. Il dispose pour ce faire d'informations sur ses clients, relatives à leurs profils et leurs souscriptions, et des services en eux-mêmes, qui peuvent être souscrits et exécutés directement par l'utilisateur final.

Typiquement les informations sur les utilisateurs seront stockées au sein d'une base de données, alors que les services seront hébergés sur des serveurs selon une structure multi tiers.

2.3.3.2 Offre de services

Le Détaillant est l'acteur qui propose les services perçus comme tels par l'utilisateur final. Il joue le rôle de concentrateur des services déployés sur l'ensemble du réseau. C'est par lui que l'utilisateur passera pour souscrire et exécuter un service, qu'il soit proposé directement par le Détaillant ou supporté par un fournisseur de services tiers.

Un Détaillant propose à l'utilisateur final un mécanisme de composition et de personnalisation de ses services. Il dispose d'un service de présence, modélisant l'activité de l'utilisateur final, permettant d'adapter son offre de services à l'état de ses abonnés.

2.3.3.3 Dépendances

Un Détaillant propose une offre de services à l'utilisateur final. Ces services peuvent être partiellement ou totalement externalisés vers des fournisseurs de service tiers, des interfaces typées définissant les besoins et offres exprimés de part et d'autre de cette relation Détaillant – fournisseur de services tiers.

Un Détaillant dépend également du Fournisseur d'accès associé à l'utilisateur final. Il passe par le Fournisseur d'accès pour connaître les contraintes relatives à la connexion de l'utilisateur, de manière à dimensionner efficacement les flux multimédia échangés lors de l'exécution de services.

2.3.4 Le Fournisseur de Services Tiers

2.3.4.1 Patrimoine

Cet acteur est spécialisé dans la fourniture de services à destination exclusive du Détaillant. En effet un Fournisseur de Services Tiers n'a pas vocation à proposer ses services directement à l'utilisateur final, ce qui l'affranchi des considérations relatives aux souscriptions des utilisateurs. Le patrimoine d'un Fournisseur de Services Tiers se limitera donc à un ensemble de serveurs mettant à disposition les composants utilisés par le Détaillant.

2.3.4.2 Offre de services

Un Fournisseur de Services Tiers peut proposer des composants comme briques de base à la construction par le Détaillant de services plus complexes, ou des services prêts à l'emploi.

2.3.4.3 Dépendances

Un Fournisseur de Services Tiers est en relation avec un ensemble de Détaillants, chaque service ou composant appartenant au Fournisseur de Services Tiers faisant l'objet d'un contrat d'utilisation par le Détaillant. A cet effet notre architecture intègrera un mécanisme de prise en charge des contrats de qualité de service au niveau du conteneur de nos composants, pour par exemple rétribuer l'utilisation d'un composant en gérant une relation avec le Gestionnaire de Facturation

2.3.5 Le Fournisseur de Connectivité

2.3.5.1 Patrimoine

Il gère un réseau de transport dont dépendront les Fournisseur d'accès. Le réseau d'un Fournisseur de Connectivité ou « Connectivity Provider » est constitué de policy servers chargés de déterminer les termes de l'utilisation du réseau par un Fournisseur d'accès donné, et de routeurs capables de véhiculer les informations échangées entre utilisateurs du réseau.

2.3.5.2 Offre de services

Le Fournisseur de Connectivité n'offre pas de services tels que nous l'entendons du point de vue de l'utilisateur. En effet son action n'est pas perceptible depuis l'utilisateur final, le seul acteur ayant une relation avec lui étant le Fournisseur d'accès. Dans la pratique un Fournisseur de Connectivité loue l'utilisation de son réseau de transport aux Fournisseur d'accès, les policy servers assurant la qualité de service induite par le contrat de liant les deux parties.

2.3.5.3 Dépendances

Un Fournisseur de Connectivité ne dépend d'aucun autre acteur si ce n'est le Gestionnaire de Facturation qui permet de contractualiser les relations établies avec des Fournisseur d'accès.

2.3.6 Le Gestionnaire de Facturation

2.3.6.1 Patrimoine

Dans une architecture délibérément orientée vers l'interopérabilité entre acteurs, la réussite de l'ouverture des services entre fournisseurs repose sur un système de facturation rationnel et clairement identifiable par tous. C'est dans ce contexte que nous avons fait apparaître ce nouvel acteur : le Gestionnaire de Facturation, typiquement assumé par un organisme financier, capable de gérer des comptes bancaires au service des utilisateurs finaux, ou dans le cadre des relations entre acteurs de notre modèle métier.

Un Gestionnaire de Facturation possède une architecture articulée autour d'une base de données hébergeant les comptes bancaires, supportant en entrée la saisie de flux financiers entre différents comptes, ce qui permet de rétribuer en direct les différentes opérations effectuées sur le réseau.

2.3.6.2 Offre de services

Un Gestionnaire de Facturation propose un ensemble de services directement lié à son métier de gestion de ressources bancaires. Il proposera des moyens de paiement, des stéréotypes de transactions financières applicables à ses relations client, et du fait de son impartialité il pourra proposer optionnellement des services de mesure de la qualité de services, pouvant donner lieu à des actions de facturation.

Ainsi si la facturation d'un service est une relation entre le(s) prestataire(s) du service et son utilisateur, le règlement de cette facture est une relation entre (s) prestataire(s) du service, l'utilisateur final et le Gestionnaire de Facturation.

2.3.6.3 Dépendances

Aucune, si ce n'est la réglementation sur les opérations bancaires.

2.4 Exemple : Le service de mise en relation

Pour préciser comment les différents acteurs définis concourent à mise à dispositions de services aux utilisateurs de notre architecture, étudions comme exemple un service d'envoi de messages instantanés entre deux utilisateurs. Admettons que l'utilisateur A souhaite expédier un message à la destination d'un utilisateur B.

Durant une phase initiale (0) les terminaux de nos deux utilisateurs devront entrer en relation avec un Fournisseur d'accès pour qu'une session d'accès au réseau permette le cas échéant un accès aux services. Nos deux utilisateurs peuvent accéder au réseau en passant respectivement par les Fournisseur d'accès APA et APB. Dans ce contexte, chaque terminal émet périodiquement un message d'identification, capté par une station de base du Fournisseur d'accès, permettant de mettre à jour le serveur de localisation des utilisateurs du Fournisseur d'accès.

Pour accéder à son service d'envoi de messages instantanés (1), notre utilisateur A va établir une session de services avec son Détaillant RA. L'utilisateur A@RA dispose alors d'une page, chargée de collecter les données utiles à l'exécution du service, dans laquelle il précise le contenu du message qu'il souhaite envoyer à l'utilisateur B@RB.

L'exécution proprement dite du service (2) commence par une phase de vérification des règles de collaboration susceptibles de s'appliquer dans le contexte actuel d'exécution du service. Ces règles qui peuvent être spécifiques à l'utilisateur ou définies par le Détaillant lui-même à destination de l'ensemble de ses abonnés, permettent de préciser les relations de composition et de personnalisation des services. Le Détaillant peut par exemple préciser à ce niveau les contraintes de facturation du service, et ainsi interagir avec un Gestionnaire de

Facturation (3). Les règles de collaboration appliquées, le service va chercher à entrer en relation avec le Détaillant RB de B, en tenant compte cette fois des contraintes liées à l'exécution du service du coté de B. Cette démarche permet ainsi de considérer des politiques de facturation personnalisées, en utilisant à bon escient les règles de collaboration dans les domaines de RA et RB. Si le Détaillant de B prend la décision de contacter B pour lui faire parvenir le message envoyé par l'utilisateur A, sa première initiative sera de contacter l'Fournisseur d'accès avec lequel B a pu négocier une session d'accès au réseau (4). Implicitement nous pouvons préciser qu'au moment de l'ouverture de la session d'accès au réseau, et tout au long de l'évolution de la connexion de l'utilisateur au réseau, les messages d'identification envoyés périodiquement par l'utilisateur doivent contenir les informations permettant à le Fournisseur d'accès de contacter le Détaillant, afin de mettre à jour l'état et la localisation de l'utilisateur en terme d'Fournisseur d'accès.

Finalement (5), en utilisant les informations retournées par le Fournisseur d'accès, le Détaillant RB présentera le message au terminal de B, une application déployée sur le terminal se chargeant des considérations relatives à sa présentation.

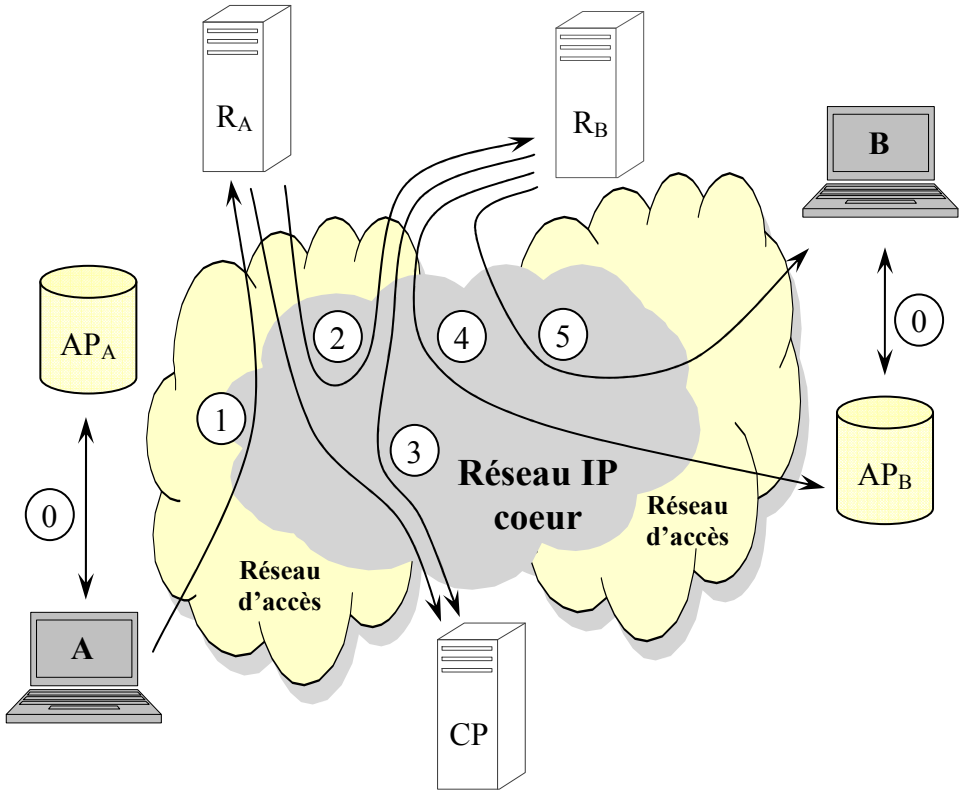


Figure 5 Les interactions métiers associées au service de messagerie instantanée

Cet exemple autour d'un simple service de messagerie instantanée nous a permis de mettre en relief les interactions entre les différents rôles précédemment introduits. Dans la suite de notre travail, nous allons plus particulièrement nous intéresser aux étapes 1, 2 et 3 de notre exemple. En d'autres termes nous allons pénétrer au cœur du domaine du Détaillant pour préciser les mécanismes d'accès aux services, les briques de base de construction de nos

services, et les fameuses règles de collaboration qui permettent à la fois la personnalisation et la composition des services.

2.5 Conclusion

En exhibant les différents acteurs gravitant autour de la sphère des services télécoms, avec en particulier l'introduction d'un Gestionnaire de Facturation, contrairement aux modèles existants du type TINA ou PARLAY notre modèle métier nous permet d'appréhender les problématiques de composition, de gestion de la qualité de services et de leur facturation. Nous allons ainsi présenter dans la suite de ce manuscrit la manière de construire des services à base de composants, dans notre modèle métier.

3 La création du service

3.1 Introduction

Cette partie va commencer par une présentation du contexte d'apparition des composants logiciels, en le comparant aux approches par blocs procéduraux, au cœur des systèmes à base de MEGACO.

Nous présenterons ensuite un cycle original d'analyse et de conception de service, basé sur l'utilisation de modèles écrits suivant la notation UML. Le caractère novateur de cette partie réside dans l'extension faite à UML en commençant une formalisation des diagrammes de cas d'utilisation (Use Case), en vue d'automatiser la génération de diagrammes d'Activité eux-même enrichis par le biais de stéréotypes, pour prendre en compte les conraintes de composition et d'interaction des composants dès les premières phases d'analyse du service. Cette analyse du service pourra ensuite aboutir sur l'utilisation d'un nouveau type de diagramme, que nous avons baptisé diagramme de composants, capable d'expliciter graphiquement les traitements internes au composant, son but étant d'automatiser la production d'un code de programmation du composant et de fichiers de description de ses interfaces.

3.2 Etat de l'art sur l'utilisation de composants télécom

3.2.1 Présentation

Avant d'aborder les considérations de composition et d'interaction de services, arrêtons nous sur la structure interne des services, car de la nature individuelle des services, dépendra la facilité que nous aurons ensuite à les composer. Notre architecture permet l'ajout de services télécom avancés, déployés au sein d'un serveur d'applications. Ces services tels la messagerie instantanée, la conversion de flux, l'interaction avec des services Internet ou la facturation par carte de crédit, font appel le plus souvent à une multitude de fonctionnalités : transposition de texte en voix, reconnaissance de caractères digitaux, interaction avec une base de données… ([MCJ02] donne un bon aperçu des fonctionnalités envisageables). Sans mécanisme de décomposition, la construction de services intégrant un tel foisonnement de caractéristiques s'avèrerait extrêmement ardue.

3.2.2 L'existant

Le mécanisme de décomposition le plus répandu consiste à concentrer d'un coté les aspects de signalisation et de contrôle des données, que nous appèlerons la partie applicative du service (AS), et de l'autre les considérations liées au flux temps réel au sein cette fois de la partie média du service (notée MS). Face à un tel découpage, les constructeurs de services disposent d'interfaces de contrôle, comme MGCP [ADE99], [GRR00] ou MEGACO [CGH00], permettant à la partie applicative du service de piloter la partie media.

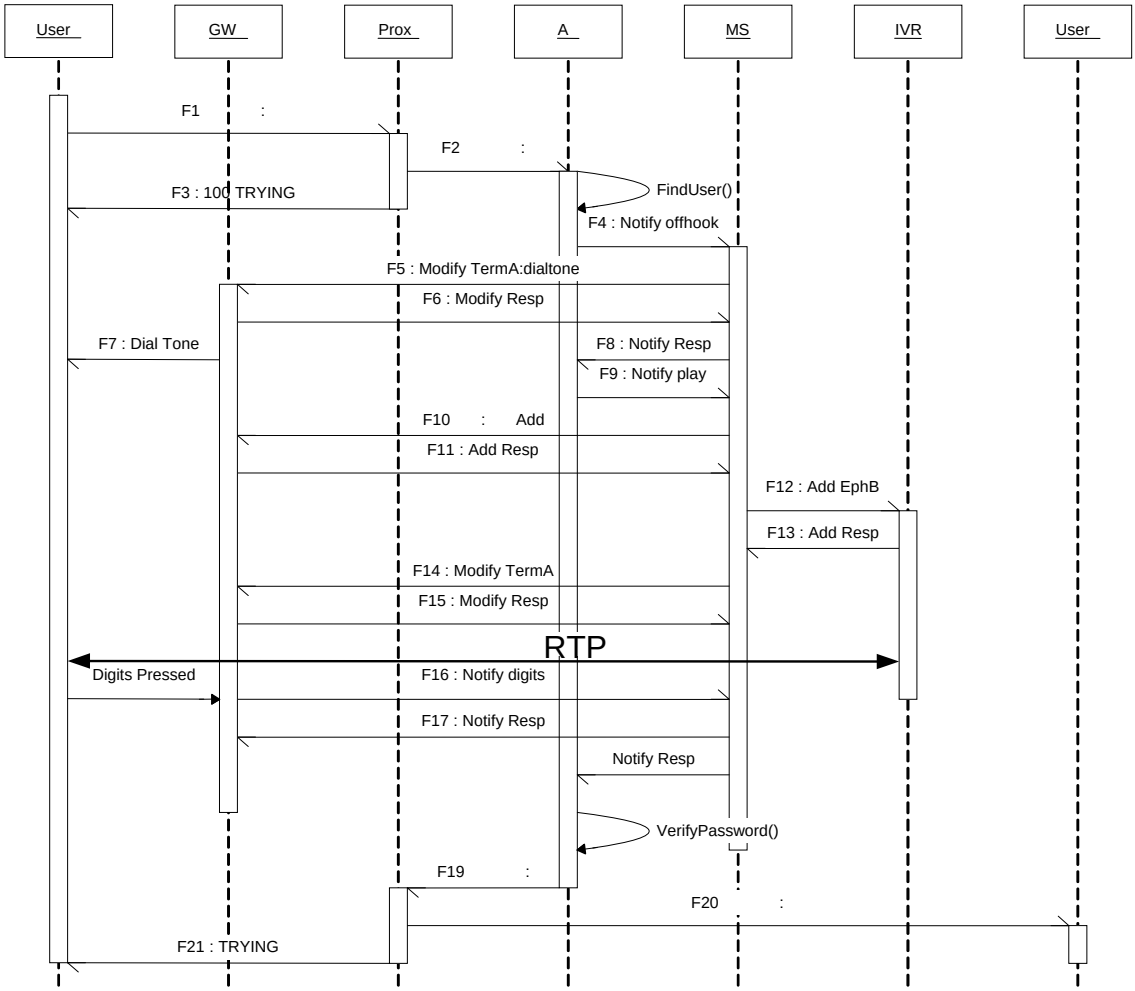


Figure 6 Une implémentation du service de mise en relation par carte

Ce découpage, dans la lignée du réseau intelligent, dans la mesure où nous retrouvons d'un coté l'équivalent du contrôleur de services (le SCP) et de l'autre le(s) gestionnaire(s) de ressources matérielles (IR et SSP), permet aux opérateurs de migrer au fil de l'eau leurs architectures procédurales et fermées de services vers des implémentations objets plus ouvertes. Cependant cette manière de procéder, critiquée dans [RMS01], présente bon nombre d'inconvénients, que nous allons passer en revue en étudiant plus en profondeur le service de mise en relation par carte, autour du protocole SIP, respectant le découpage AS/MS.

Admettons que la logique de notre service se décompose avec les étapes suivantes :

- Vérifier dans une base de données que le code de carte reçu existe.
- Récupérer les données d'authentification associées à la carte, à partir de cette base de données.
- Vérifier les données d'authentification fournies par l'utilisateur.
- Poursuite de la mise en relation.

Une implémentation possible de ce service (cf. diagramme de séquence donné en Figure 6) pourrait être de demander à l'utilisateur de composer un code confidentiel d'authentification sur son terminal.

La première contrainte émanant de cette architecture est la nécessité pour le développeur de l'AS de connaître en détails le modèle sous-jacent du MS. En effet, demander à l'utilisateur de saisir un mot de passe sur son terminal (message F14) impose au développeur de l'AS de commander successivement au MS un changement de tonalité sur le terminal de l'utilisateur, puis un message l'invitant à entrer son mot de passe, une collecte des touches pressées et enfin un retour de l'interprétation de ces touches pressées. Dans la pratique cet enchaînement peut être réalisé de deux façons.

- La sémantique des évènements gérés par le MS peut être enrichie, de manière à créer un évènement duquel découlera l'enchaînement désiré. L'AS invoquera à ce moment le MS en faisant référence à cet identifiant d'événement, et en accompagnant cet appel des données utiles aux opérations prises en charge par le MS. Par exemple, on pourra créer l'évènement d'identifiant 1111 et transmettre au MS l'adresse et le port sur lequel ouvrir une connexion RTP (données disponibles au niveau de l'AS car véhiculées initialement dans le message SIP INVITE) pour permettre à l'IVR de jouer les annonces sur le terminal de l'utilisateur.

  ```
  F4 : AS -> MS

  MEGACO/1 [as_enst.example.com]: 8000
  Transaction = 1525 {
       Context = - {
            Notify = 100.101.102.103:49172 {
                 ObservedEvents = 1111 {
                      20020923T10461350:al/of
                 }
            }
       }
  }
  ```

  Ce message notifie au MS que dans le contexte du terminal 100.101.102.103 l'événement 1111 (correspondant à un décrochage) a eu lieu à 10H46 le 23 septembre 2002. Sur réception de F4, le MS va expédier le message F5 à destination de la Gateway du coté de l'appelant pour modifier le contexte de l'appel. La gateway va commencer par jouer sur TermA la tonalité correspondant à la mise en attente (Signals(cg/dt), pour ensuite charger la table d'analyse des touches pressées par l'appelant Vmap{ (UserA) }, et enfin armer l'événement 1112 correspondant à l'arrivée d'un flux.

  ```
  F5 : MS -> GWA

  MEGACO/1 [ms_enst.example.com]: 9000
  Transaction = 3000 {
       Context = - {
  ```

```
            Modify = TermA {
                 Signals {cg/dt},
                 VoiceMap= Vmap {(UserA)}
                 Events = 1112 {
                      al/on, dd/ce {VoiceMap=Vmap}
                 }
            }
       }
}
```

De la même manière, quand l'événement 1112 surviendra au niveau de la Gateway, cette dernière retournera le flux au MS pour que l'information remonte jusqu'à l'AS.

- L'AS gère directement les utilisations des organes de bas niveau que sont l'IVR et les Gateways. Cette solution permet de conserver une sémantique standard au niveau du MS, l'AS utilisant les directives normalisées dans MEGACO pour commander au MS l'utilisation des ressources Media. Cependant cette méthode présente l'inconvénient de mêler la logique du service au niveau de l'AS, à la logique d'implémentation liée aux interactions avec le MS, ce qui abouti rapidement à un service difficilement lisible quand les interactions avec le MS augmentent.

De manière générale, lier si fortement la partie Application du service avec sa partie Média présente de grands inconvénients, essentiellement au niveau de la lisibilité de la logique de nos services. Cette contrainte est d'autant plus pénalisante que des Media Server émanant de différents constructeurs n'auront pas nécessairement la même logique interne, les développeurs de services devant se réadapter dans ce cas à la modélisation propre à chaque Media Server.

Faire dépendre ainsi l'AS du MS présente également un inconvénient dommageable, en cas de modification apportée au niveau du MS. En effet si le constructeur du MS désirait faire évoluer son matériel, c'est l'ensemble des services basés sur ce MS qui se verrait par transitivité impactés, en ayant dans la plupart des cas une bonne partie des services à réécrire.

3.2.3 Un besoin de modulariser les services

La construction de services sous la forme d'un bloc applicatif (AS) associé à un bloc média (MS) implique un certain nombre d'inconvénients majeurs. Tout d'abord la notion de bloc ne permet pas une bonne réutilisation des éléments de services, ceux-ci n'étant pas clairement délimités. Ensuite la maintenance des services peut s'avérer extrêmement complexe, car modifier un traitement dans un bloc applicatif peut, par effet de bord, altérer un autre traitement du même bloc applicatif. Enfin cette structure de bloc n'est pas adaptée aux mécanismes de gestion du trafic ou de tolérance aux pannes. En effet, si le service repose sur un bloc MS, la panne ou la surcharge d'un élément du MS (par exemple l'accès à un IVR) pénalise l'ensemble du service sans possibilité de trouver une parade à cette déficience, dans la mesure où l'appel à l'élément figure dans la logique du service.

Nous avons vu que le modèle AS/MS était un bon modèle pour la construction d'applications prenant en charge le protocole SIP, tout en nécessitant l'accès à un ensemble de ressources média. Cependant la construction de services sur ce modèle ne peut se concevoir sans une modularisation de leurs fonctionnalités, pouvant correspondre à une interaction entre AS et MS. Modulariser les services permettra d'une part de mieux réutiliser les éléments des services antérieurs, et d'autre part de faciliter la maintenance des services, en permettant qu'une intervention sur un aspect d'un service n'impacte pas l'ensemble du service. En outre cette modularisation n'est pas finie, dans ce sens où une fonctionnalité d'un service peut elle même être décomposée en d'autres modules.

3.2.4 Qu'est ce qu'un composant

Les langages C++, Java et maintenant C# ont démocratisé le concept d'objet. Avec ce modèle le constructeur d'application dispose de classes, permettant de masquer les mécanismes internes de traitement des données, et de rendre visible certains éléments depuis l'extérieur. Une classe est constituée d'attributs, permettant d'accéder à une valeur prise par un paramètre, et de méthodes donnant accès à un traitement métier. L'exécution d'application Java ou C# se fait au sein d'une machine virtuelle disposant d'une « factory », chargée de créer des objets, correspondant à une instance d'une classe. Un objet est doté d'une identité, maintenue par la machine virtuelle, et d'un état, correspondant aux valeurs prisent par ses paramètres ; il est de plus capable de communiquer avec d'autres objets en utilisant des mécanismes offerts par la machine virtuelle.

Le modèle objet s'est surtout imposé ces dernières années pour ses grandes capacités d'adaptation. De fait, si les anciennes applications en C étaient très performantes pour des communications horizontales de type Client-Serveur, elles sont mal adaptées dans le monde d'aujourd'hui où la communication est à la fois horizontale et verticale. En effet, une application ne peut plus se permettre aujourd'hui de rester isolée, elle doit offrir des interfaces de communications avec ses applications environnantes, ce que la modélisation objet a su permettre avec simplicité.

Les applications type Client-Serveur ont eu tendance ces dernières années à se déséquilibrer, pour aboutir à la notion de client léger (sous la forme d'un simple navigateur) et de serveur d'applications. Dans ce nouveau rapport de force, le serveur apporte d'emblée aux applications un environnement enrichi d'exécution, avec la prise en charge de mécanismes complexes comme la répartition de la charge, les transactions, l'authentification de l'utilisateur, … Avec la complexification sans cesse grandissante de ces serveurs d'applications, offrant un nombre pléthorique d'interfaces d'accès aux services communs, la modélisation objet a inéluctablement évoluée vers la modélisation de composants. Dans ce concept, le serveur d'applications présente un conteneur, jouant le rôle d'interface globale entre les applications et les services communs que sont l'authentification, l'activation ou le service d'annuaire. Les composants, qui constituent les applications, sont en réalité un regroupement d'objets et d'éléments de configuration, qui permettent d'expliciter la relation entre le composant et le serveur d'applications.

3.2.5 La relation Ressources – Composants

Avant de s'intéresser au comportement de nos composants, notre réflexion s'est portée sur la pattern qui permettrait de rendre nos composants capables d'interagir avec tout type de ressource. La Figure 7 présente la structure sur laquelle repose la construction de nos composants. Le premier quart, représenté le plus à droite est constitué des ressources physiques : les bases de données, les périphériques multimédia…

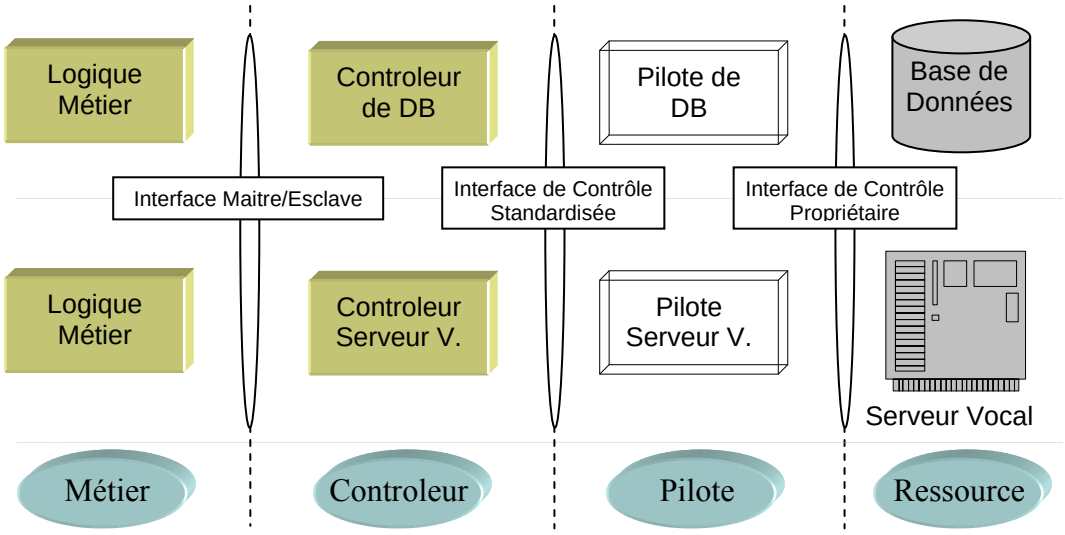


Figure 7 La Pattern sur laquelle repose nos composants

Comme nous l'avons vu dans le modèle AS/MS, l'enjeu de notre construction est de concevoir des composants indépendants des ressources physiques sous-jacentes. Pour ce faire nous avons introduit un second quart "driver", permettant de faire la transition d'une interface de contrôle Propriétaire vers une interface de contrôle Standardisée. Ce driver est le seul élément de la construction qui doit connaître la logique interne du matériel en aval, son interface standard le rendant ensuite utilisable en amont par les composants. Les architectures telles que JINI ou PARLAY [PAR00] sont totalement orientées autour de ces drivers ou contrôleurs de ressource, en allant jusqu'à porter la réflexion sur la notion de ressource JINI ou PARLAY. De fait, dans la pratique un driver peut être pris en charge nativement par la ressource, en l'intégrant directement sur le hardware (ce que font par exemple les imprimantes JINI), soit en gérant un driver logiciel, mis à la disposition des applications utilisatrices (ce qui est par exemple le cas des drivers de base de données). A partir des interfaces de contrôle standardisées il nous est possible de construire nos composants que nous subdiviserons en deux catégories : les composants "contrôleur" et les composants "métier".

Un composant contrôleur délivre un service intermédiaire aux composants métier, sur la base des drivers à sa disposition. Ce type de composant est conçu pour simplifier la relation entre le composant métier et les ressources. En effet, si l'on prend le cas d'un composant contrôleur de bases de données, un composant contrôleur conservera le résultat de l'appel à une procédure stockée, pour permettre une réutilisation des données sans nécessiter systématiquement l'accès à la ressource.

Le composant métier est le service final qui sera présenté à l'utilisateur. Ce type de composant peut faire appel à d'autres composants dans le cadre de l'accomplissement de son service, ou il peut avoir besoin d'accéder à des ressources, en passant dans ce cas par des composants contrôleurs.

3.3 Le cycle de création de nos composants

La création de nos composants, tant contrôleurs que métiers, suit un cycle composé de cinq états : les phases d'Analyse, de Développement, de Conditionnement, de Déploiement et d'Exécution. Ces cinq étapes, qui ne sont pas nécessairement assumées par la même personne, permettent de raffiner au fur et à mesure le composant, en partant d'une vue de haut niveau, liée aux fonctionnalités attendues par le service à créer, pour arriver à un ensemble de classes et de fichiers de configuration précisant le contexte d'exécution du composant. Ce cycle présenté dans la Figure 8 peut être divisé en deux périodes : la période fonctionnelle, et la période relationnelle.

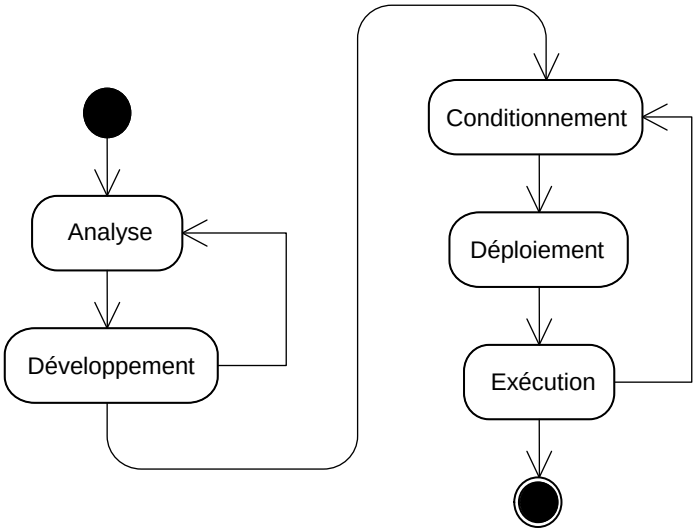


Figure 8 Le cycle de création d'un composant

La période fonctionnelle, constituée des phases d'analyse et de développement, se concentre sur le comportement à associer au composant.

L'analyse est traditionnellement réalisée par un architecte logiciel, qui se basera sur le cahier des charges du service à créer, pour établir la liste des composants à utiliser. Le but de cette étape est de dégager une première représentation du service en terme de composants. Pour mener à bien cette phase, nous utilisons une représentation graphique des composants, permettant de rendre compte d'une part des fonctionnalités apportées par chaque composant, et précisant d'autre part les interactions entre les différents composants.

Le développement est l'étape qui voit naître le cœur du composant. Cette phase réalisée par un développeur objet commence par l'écriture des interfaces issues de l'analyse, puis s'enchaîne avec la définition de leurs classes d'implémentation. Le développement du

composant doit être en ligne avec la phase d'analyse, sans se focaliser pour le moment sur les contraintes d'exécution qui enrichiront le composant durant la période relationnelle.

A l'issue de la période fonctionnelle, le développeur est capable de livrer un composant compilé, achevé sur le plan fonctionnel. De ce fait, les différentes étapes composant la période relationnelle n'affecteront pas le comportement interne du composant, mais permettront de le doter des classes et fichiers de configurations nécessaires à son fonctionnement et son utilisation commerciale.

La phase de conditionnement permet de préciser les termes de l'utilisation des facettes (interface précisant les services rendus par le composant) et réceptacles (interface indiquant les besoins en interaction du composant) définis et implémentés durant les phases d'analyse et de développement. Cette précision porte à la fois sur la qualité avec laquelle le service doit être rendu, sur la relation du composant avec les différents services internes au conteneur, et sur la rétribution à accorder à la délivrance du service.

Alors que la phase de conditionnement ne fait que définir des fichiers de configuration précisant le contexte d'utilisation du composant, la phase de déploiement permet d'appliquer ces prérogatives en créant les classes utiles au conteneur du composant. La phase de déploiement permet également d'accéder au second niveau de notre architecture : la composition de services, en utilisant des contrats de collaboration comme "liant" entre les composants.
La phase d'exécution du composant considère la relation entre le composant et les services du conteneur. C'est également en étudiant cette phase que nous verrons comment aborder le problème de l'interaction de services.

Nous allons maintenant revenir plus en détail sur ces différentes phases, en étudiant en fil rouge la création d'un service de paiement de péages d'autoroutes, via un téléphone mobile.

3.4 L'analyse du service

Réalisée par un architecte, l'analyse se base sur le cahier des charges détaillant la valeur ajoutée du service à créer. Son but est d'aboutir à une première spécification du service mettant en exergue les composants mobilisés. Pour raffiner cette spécification, et ainsi passer d'un cahier des charges informel à une représentation formelle de composants nous utiliserons successivement trois types de diagrammes dérivés d'UML:

- Les diagrammes de Use Cases
- Les diagrammes d'activité
- Les diagrammes de composants télécoms.

3.4.1 Le diagramme de « Use Case »

3.4.1.1 Présentation

Ce premier type de diagramme, directement issu de la lecture du cahier des charges, est utilisé pour décomposer le comportement du service, en insistant sur la perception qu'en auront ses

utilisateurs. A cet effet, le service sera segmenté en transactions (Use Case) ayant un sens pour une catégorie donnée d'utilisateurs (Actor).

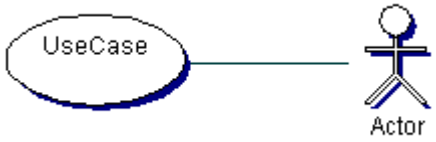


Figure 9 Un Use Case

Notons que cette représentation simpliste du service n'intègre pas à ce niveau l'ensemble des informations contenues dans le cahier des charges. Aussi pour être exploitable un diagramme de Use Case doit être enrichi selon deux procédés :

- la définition de sous-transactions
- la description détaillée des Use Case

Le premier procédé consiste à décomposer un cas d'utilisation en cas plus fins. Cette opération classique dans les diagrammes de Use Case est basée sur l'utilisation de la relation stéréotypée "include" précisant que toute activité associée à un sous-cas d'utilisation sera également associée au cas d'utilisation à la base de la relation "include".

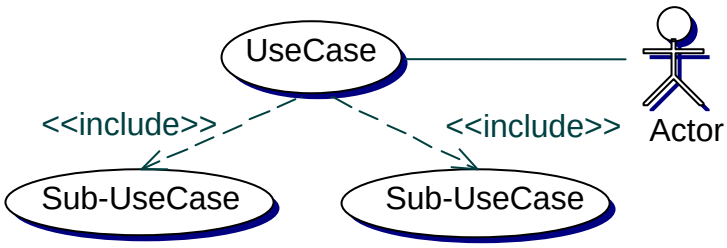


Figure 10 L'utilisation de la relation stéréotypée "include"

Le deuxième élément d'enrichissement des diagrammes de Use Case est la définition d'une description détaillée, en langage naturel, du futur traitement que devra assumer le service pour le cas d'utilisation donné. Cette description, directement issue du cahier des charges comprend quatre entrées :

- Les Acteurs concernés par le Use Case
- Les préconditions permettant de préciser l'état du système au moment où le cas d'utilisation intervient.
- L'enchaînement courant du cas d'utilisation, décrivant les différentes étapes par lesquelles le système doit passer.
- Les enchaînements alternatifs, qui permettent au système de s'écarter de l'enchaînement courant en cas d'anomalie.

Cette description qui reste pour le moment informelle, permettra de préciser les interactions avec l'utilisateur du service, les flux et la gestion des erreurs durant la suite de la phase

d'analyse. Cette description détaillée peut être d'autant plus indispensable, que les différentes étapes d'analyse peuvent ne pas être prises en charge par la même personne.

3.4.1.2 Etude de cas

Nous allons à travers une étude de cas, portant sur la construction d'un service de paiement d'autoroute, étudier l'intégralité du processus de création de service. Comme tout processus de création de service, notre étude commence par la réalisation du cahier des charges, qui donne un premier aperçu des fonctionnalités attendues :

Cahier des Charges

La Société Roul-Patrovit souhaite offrir la possibilité à ses usagers de régler leurs frais de péage en utilisant les capacités de leur véhicule communicant, capable de prendre en charge le protocole SIP.

Description :
Ce service devra permettre d'assurer une facturation dépendant des caractéristiques des véhicules, avec la possibilité d'instaurer un système de pénalité en cas de dépassement de la vitesse ou de la charge du véhicule autorisée. Pour assurer un temps d'exécution acceptable du service, il ne sera pas demandé à l'usager de confirmer son identité par un moyen d'authentification. La sécurité des moyens de paiement de l'usager sera assurée en n'imputant pas directement la facturation sur son compte bancaire, mais en utilisant un service de paiement volatil, mis à la disposition de l'usager par la société d'autoroute.

Les Business Actors mobilisés par ce service seront :
- le Fournisseur d'accès de la société d'autoroute, qui assurera la communication d'une part entre les usagers et les bornes d'accès de la société d'autoroute, et d'autre part entre la société d'autoroute et le réseau public.
- Le Détaillant de la société d'autoroute, hébergeant le service.
- Le Gestionnaire de Facturation de la société d'autoroute, mettant à disposition les moyens de paiement.
- Le Gestionnaire de Facturation de l'usager, gérant le compte bancaire de l'usager à partir duquel le compte volatil pourra être crédité.

La lecture de ce cahier des charges peut nous amener à la définition d'un premier diagramme de Use Case donné ci-dessous :

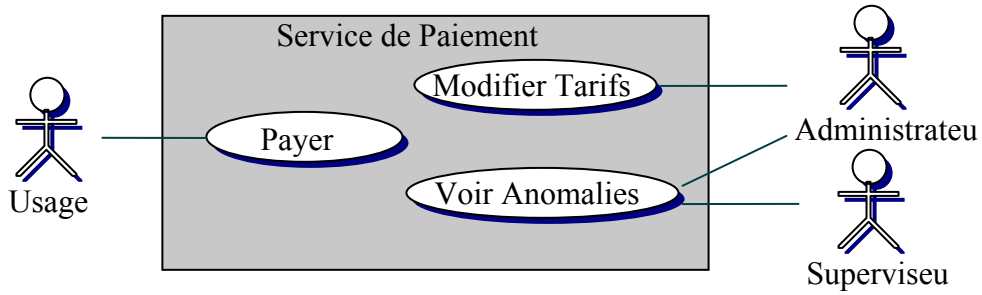


Figure 11 Cas d'utilisation du service de Paiement

L'étude de cet exemple n'ayant pas la prétention d'être exhaustive, nous nous limiterons au cas d'utilisation "Payer" lié à l'usager, les administrations et autres supervisions du service étant trop longues à détailler dans ce manuscrit. Un travail de raffinement de notre Use Case pourrait nous mener au diagramme suivant :

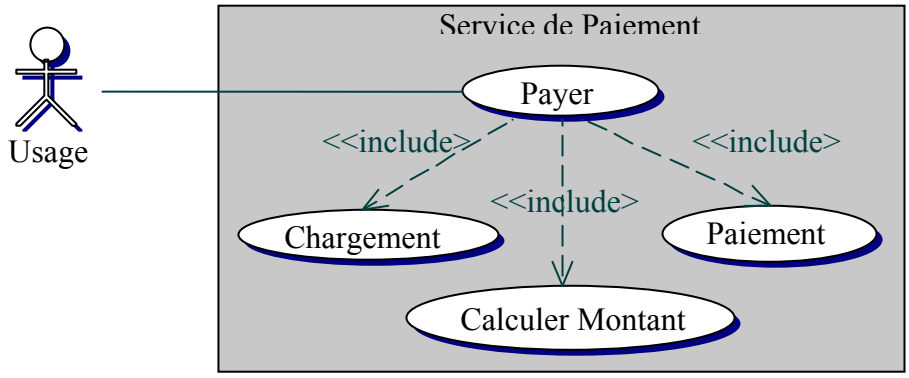


Figure 12 Raffinement du cas d'utilisation "Payer"

La dernière phase autour du diagramme de Use Case va être l'adaptation du cahier des charges aux cas d'utilisation définis dans le diagramme. Le cas d'utilisation "Payer" peut être décrit comme suit :

Use Case : Payer

Acteur : Usager

Préconditions : l'utilisateur a souscrit au service, en conséquent :
- Ses caractéristiques personnelles sont connues (langue, monnaie, …)
- Il possède un compte volatil
- Ses coordonnées bancaires sont accessibles

Enchaînement courant :
1. L'usager fournit les données utiles à son identification et à sa facturation. (Chargement)
2. Le service charge les informations liées à l'usager.
3. Le service évalue le montant de la somme à imputer à l'usager. (Calculer Montant)
4. Le montant est présenté à l'usager, en vue de confirmer son règlement.
5. Le montant est débité du compte volatil de l'usager. (Paiement)

Enchaînements alternatifs :
a. (1-2-3-4-5) L'usager peut annuler l'exécution du service.
b. (4) L'usager peut refuser le montant à régler.
c. (5) Si le montant à régler est supérieur au solde de l'usager sur son compte volatil, il lui sera demandé de créditer son compte à partir d'un de ses comptes bancaires connus.
d. (5) Si la procédure de crédit du compte volatil échoue, le service de paiement échouera à son tour.

Dans l'enchaînement courant du traitement nous avons précisé les traitements couverts par les sous-cas d'utilisation que sont "Chargement", "Calculer Montant", "Paiement". Après avoir précisé de manière encore informelle le comportement du service, la phase d'analyse va se

poursuivre en définissant un ensemble de diagrammes d'activité, en tirant parti des définitions des Use cases.

3.4.2 Les Diagrammes d'activité

3.4.2.1 Présentation

Pour faciliter la mise en évidence des fonctions assumées par le système, et leur répartition en terme de composants, nous utilisons le diagramme d'activité. Ce type de diagramme permet de préciser les enchaînements d'activités, sur la base du diagramme de Use Case, l'architecte ayant ensuite l'initiative de regrouper un sous-ensemble du diagramme au sein d'une même structure, qui permettra le cas échéant de faire le rapprochement avec les composants existants, ou la définition de nouveaux composants. Le diagramme d'activité s'appuie sur les éléments suivants :

- L'activité, qui constitue l'unité fondamentale du diagramme du même nom. Une activité est dessinée sous la forme d'une ellipse contenant son nom. Elle peut être accompagnée d'un descriptif utilisable au moment de son implémentation.
- L'état d'activité, qui introduit la notion d'enchaînement d'activités. Dans la pratique toute activité peut être raffinée et explicitée sous forme d'état d'activité, pour aller plus profondément dans l'analyse du service à créer. La difficulté de l'analyse réside précisément dans les limites de la description du service, ou corollairement dans la décision de représenter une fonctionnalité plutôt par un état d'activité ou une simple activité. Ainsi un use case peut dans sa plus simple expression être décrit sous la forme d'une unique activité, pouvant le cas échéant être décomposé sous la forme d'un état contenant lui-même d'autres sous activités. Les états d'activité permettent en outre de hiérarchiser les activités, un état pouvant contenir ou mener à un autre état, ce qui donne une première abstraction des futures structures de composants à implémenter. Un état d'activité est représenté par un cartouche à deux niveaux ; le niveau supérieur du cartouche précise le nom de l'état, alors que son niveau inférieur encapsule l'enchaînement d'activités effectué lorsque l'état est activé.
- L'état initial d'un état d'activité, représenté sous la forme d'un disque plein, marquant le point de départ d'un enchaînement d'activités. Cet élément est unique au sein d'un état d'activité.
- L'état final, représenté sous la forme d'un disque plein contenu dans un cercle, précise le terme d'un enchaînement d'activités. Lorsqu'un état final est activé, une transition sortant de l'état d'activité qui le contient est franchie.
- Les transitions, matérialisées sous la forme d'une flèche, elles relient les différents éléments composant un diagramme d'activité. Elles sont franchies suite à la survenue d'un événement dans l'élément à son origine. Quand une transition est franchie, on dit qu'elle active l'entité qui est à son extrémité. Dans un diagramme d'activité, toute entité (excepté les états initiaux et finaux) doit posséder au moins une transition entrante, pour être atteignable, et une transition sortante, car seuls les états finaux peuvent être des puits. Dans

le cas où une activité admettrait plusieurs transitions en sortie, les transitions peuvent être assujetties à des gardes, permettant de conditionner leur franchissement.

La notion d'utilisateur du système n'apparaît plus dans le diagramme d'activité. Si cette information était utile au niveau du diagramme de Use Case pour délimiter les différents cas d'utilisation du service à créer, le diagramme d'activité se focalisant sur un cas d'utilisation en particulier, la notion d'utilisateur n'a plus lieu d'être. En revanche nous verrons que le diagramme d'activité peut être enrichi au moyen de stéréotypes, pour décrire la manière dont le service peut interagir avec l'utilisateur.

3.4.2.2 La transition Use Case vers Diagramme d'activité

L'utilisation de diagrammes d'activité commence par une phase de génération automatique, qui part du diagramme de Use Case pour aboutir au diagramme d'activité. Préciser cette phase de génération nous demande de formaliser quelque peu nos deux types de diagrammes.

Admettons que ' U → C1 ' décrive un cas d'utilisation.
Admettons que ' C1 ↓ C2 ' décrive une relation "include" entre deux cas d'utilisation.

La grammaire hors contexte ci-dessous :

```
G1 : U → C
U  : User
C  : D | D ↓ ( C F* )
D  : Use Case
F  : , C
```

Formalise et défini les contraintes sur les diagrammes de Use Case utilisés dans le cadre de notre problématique de création de services. Pour préciser ces contraintes, déroulons l'analyse de cette grammaire, de façon à caractériser le langage qui lui est associé, en fonction des symboles non terminaux U et D.

1 : U → C
2 : U → D | D ↓ (C F*)
3a: U → D
3b: U → D ↓ (C F*)
4 : U → D ↓ (D | D ↓ (C F*) F*)
5a: U → D ↓ (D F*)
6a: U → D ↓ (D , C , C)
7a: U → D ↓ (D , D , D)
5b: U → D ↓ (D ↓ (C F*) F*)
6b: U → D ↓ (D ↓ (D F*) F*)
7b: U → D ↓ (D ↓ (D , C) F*)
8b: U → D ↓ (D ↓ (D , D) , C)
9b: U → D ↓ (D ↓ (D , D) , D)

En d'autres termes, les mots dérivés de l'axiome G1 correspondent à un ensemble de diagrammes où un Use Case peut être à l'origine et la destination de plusieurs relations "include", cette relation étant transitive, antisymétrique et sans cycle. D'autre part, nous avons fait le choix d'utiliser une relation "include" parenthésée et exhaustive. Cette dernière propriété suppose que la représentation d'un Use Case en un mot dérivé de G1 doit tenir compte de l'ensemble des relations "include" accessibles depuis le Use Case à représenter. Ainsi la représentation du Use Case C1 associé à l'utilisateur U dans le digramme de la Figure 13 ne se limitera pas à l'expression :

$U \rightarrow C1 \downarrow (C2 , C3 , C4)$

Mais prendra plutôt la forme exhaustive:

$U \rightarrow C1 \downarrow (C2 , C3 \downarrow (C5 \downarrow (C6 , C7)) , C4 \downarrow (C5 \downarrow (C6 , C7)))$

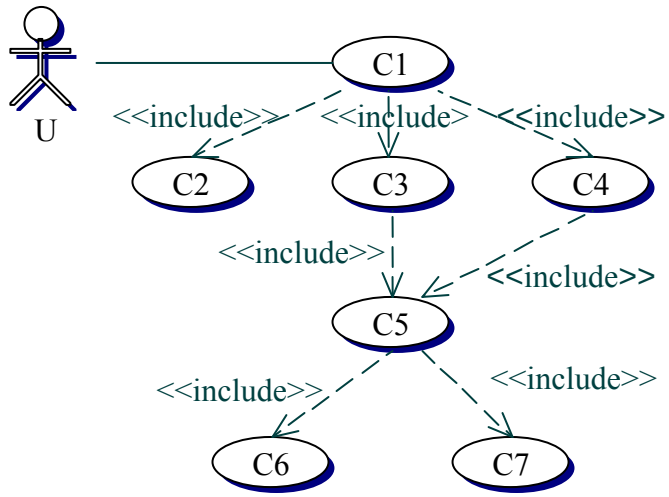


Figure 13 Un exemple de diagramme de Use Case

Cette propriété d'exhaustivité est commandée par le fait que l'expression de notre Use Case est destinée à produire la représentation d'un diagramme d'activité, à l'origine de la production de nos composants. De fait si nous prenons un Use Case (C5) intervenant dans la résolution de deux cas d'utilisations distincts (C3 et C4). Si cette double utilisation de C5 n'a pas lieu d'être différenciée au niveau d'un diagramme de Use Case, la différencier au niveau d'un diagramme d'activité permet de désolidariser les cas d'utilisation C3 et C4, et avoir ainsi une approche modulaire de l'application dès la phase de construction du diagramme d'activité.

Sur le même principe que le diagramme de Use Case, nos diagrammes d'activité peuvent être représentés en utilisant la grammaire suivante :

G2 : [E]
E : I C ► T
I : Initial State
T : Terminal State
C : ► A ► C | ► [E] ► C | ε

A : Activity

Les mots du langage associé à G2, exprimé en fonction des littéraux I, T et A, sont une représentation de diagrammes d'activités linéaires, composés d'enchaînements d'activités et d'états d'activité, chaque état pouvant lui-même contenir un d'enchaînement d'activités et d'états d'activité. Cette restriction quant à la linéarité du diagramme est imposée par notre problématique de génération automatique du diagramme d'activité à partir d'un diagramme de use case, dont la fonction de transformation φ respecte les propriétés suivantes :

φ(D ↓ (E1))	= ► [I φ(E1) ► T]	(i)
φ(E1 , E2)	= φ(E1) φ(E2)	(ii)
φ(D)	= ► A	(iii)

(i) précise que l'ensemble des cas d'utilisation ayant une relation "include" avec le cas d'utilisation D, auront leur transformation contenue dans l'état d'activité associé à D.
(ii) séquentialise les différents cas d'utilisation en relation avec un même cas d'utilisation. Cette séquentialisation est due au fait qu'un état d'activité ne peut disposer que d'un unique état initial, la non séquentialité des activités impliquant la définition d'états initiaux et d'états de reprise intermédiaires, difficilement gérables en terme de génération automatique.
(iii) défini que tout cas d'utilisation terminal (ne faisant l'objet d'aucune relation "include" sortante) devient une activité.

Comme application de cette fonction de transformation, nous pouvons reprendre la représentation du cas d'utilisation de la Figure 13 :

φ(C1↓(C2,C3↓(C5↓(C6,C7)),C4↓(C5↓(C6,C7))) =
(i) ►[I φ(C2,C3↓(C5↓(C6,C7)),C4↓(C5↓(C6,C7))) ►T] =
(ii) ►[I φ(C2) φ(C3↓(C5↓(C6,C7))) φ(C4↓(C5↓(C6,C7))) ►T] =
(iii) ►[I ►A2 φ(C3↓(C5↓(C6,C7))) φ(C4↓(C5↓(C6,C7))) ►T] =
(i) ►[I ►A2 ►[I φ(C5↓(C6,C7)) ►T] φ(C4↓(C5↓(C6,C7))) ►T] =
(i) ►[I ►A2 ►[I ►[I φ(C6,C7) ►T] ►T] φ(C4↓(C5↓(C6,C7))) ►T] =
(ii) ►[I ►A2 ►[I ►[I φ(C6) φ(C7) ►T] ►T] φ(C4↓(C5↓(C6,C7))) ►T] =
(iii) ►[I ►A2 ►[I ►[I ►A6 ►A7 ►T] ►T] φ(C4↓(C5↓(C6,C7))) ►T] =
(i) ►[I ►A2 ►[I ►[I ►A6 ►A7 ►T] ►T] ►[I φ(C5↓(C6,C7)) ►T] ►T] =
(i) ►[I ►A2 ►[I ►[I ►A6 ►A7 ►T] ►T] ►[I ►[I φ(C6,C7) ►T] ►T] ►T] =
(ii) ►[I ►A2 ►[I ►[I ►A6 ►A7 ►T] ►T] ►[I ►[I φ(C6) φ(C7) ►T] ►T] ►T] =
(iii) ►[I ►A2 ►[I ►[I ►A6 ►A7 ►T] ►T] ►[I ►[I ►A6 ►A7 ►T] ►T] ►T]

En appliquant les différentes propriétés de notre fonction φ, nous aboutissons à une expression conforme à la grammaire G2, ayant la représentation graphique suivante :

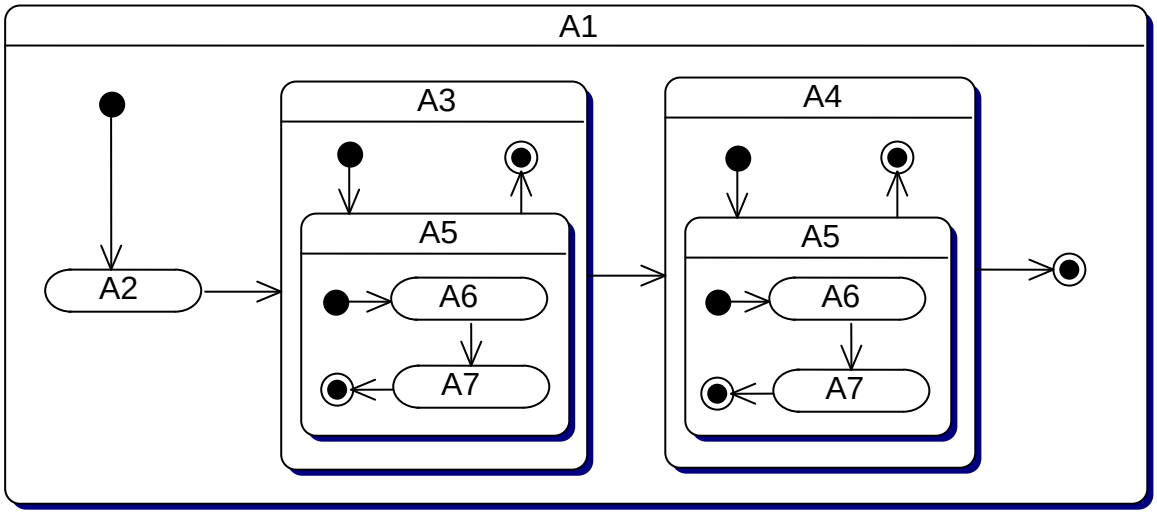


Figure 14 Le diagramme d'activité issu de la transformation

Le diagramme d'activité généré automatiquement à partir d'un diagramme de Use Case (cf. Figure 14), peut déjà permettre à l'architecte d'appréhender la problématique de réutilisation d'applications existantes. En effet, il est possible à ce niveau de substituer une activité encore incomplètement définie, par une activité ou un état d'activité issu de composants présents au sein d'une bibliothèque de composants. D'autre part, les transactions, arbitrairement définies entre les différentes activités composant un état d'activité, peuvent être redéfinies pour préciser l'ordre d'enchaînement des différents éléments.

Pour atteindre un niveau de pertinence satisfaisant notre diagramme d'activités doit permettre de décrire les données en entrée et sortie des activités, de manière à anticiper la notion d'interaction entre activités et la relation avec des mécanismes plus élaborés, faisant appel à de nouveaux stéréotypes d'activités. Les données manipulées par les activités peuvent être représentées sous la forme de triplets :

- Orientation, pouvant prendre les valeurs "in", "out" ou "inout".
- Type, correspondant à un type de données présent dans le référentiel de types du système.
- Nom, identifiant le nom de la donnée au sein de l'activité.

La signature ainsi associée à chaque activité permettra de préciser les données nécessaires à l'accomplissement de l'activité, et celles qui seront accessibles après son achèvement. La notion de signature est associative au fur et à mesure de la composition des activités, ainsi la signature d'un enchaînement d'activités correspondra à la réunion des données en entrée et sortie des activités composant cet enchaînement.

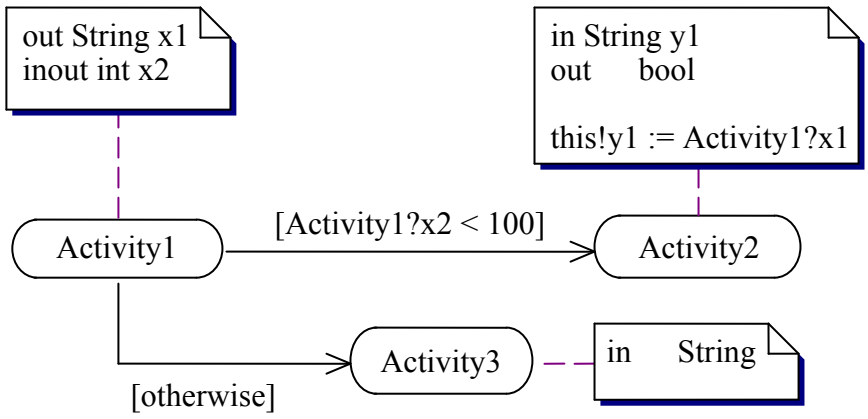


Figure 15 Utilisation de gardes sur les transitions

Parallèlement à la définition de données au niveau des activités, nous devons établir une syntaxe permettant de lier la valeur prise par une variable en entrée d'une activité, avec par exemple une donnée en sortie d'une activité la précédant. Pour établir une telle syntaxe, nous avons noté Activité!nom_donnée la valeur prise par la donnée "nom_donnée" en entrée de l'activité "Activité", par opposition à Activité?nom_donnée représentant la valeur prise par cette même variable, en sortie de l'activité. Cette notation nous permet d'exprimer des gardes portant sur les transitions entre activités, et des contraintes sur l'initialisation des données en entrée d'une activité.

L'exemple de la Figure 15 nous présente une transition dont le franchissement est conditionné par la valeur prise par une donnée x2 en sortie de l'activité Activity1. Une activité Activity2 y est également définie, à laquelle est associée une contrainte précisant qu'à sa variable y1 sera affectée la valeur de x1 en sortie d'Activity1.

3.4.2.3 Extension du diagramme d'activité

Partant d'un diagramme d'activité généré automatiquement, auquel nous avons ajouté la possibilité de définir des contraintes sur les données manipulées, nous avons utilisé le caractère extensif d'UML en définissant trois stéréotypes dérivés de l'activité, pour faciliter la génération du modèle de composants de l'application en cours de création.

Presentation

Partant de l'intuition que le processus de création de service ne devait pas s'enfermer dans une approche par composants, mais s'ouvrir sur une vue générale du service, nous avons introduit le stéréotype "presentation". Au delà des simples considérations d'interface homme machine qui ne se posent pas encore à ce stade de la construction du service, s'attarder lors de l'analyse sur des questions de présentation permet une conception harmonieuse des composants, avec des interfaces adaptées aux besoins. En outre la définition de ce stéréotype permet de raffiner l'interaction entre activités de présentation et activités métier, et précise ainsi les points d'entrée de nos composants. Pour contourner la limitation relative à l'unicité

de l'état initial à l'intérieur d'un état d'activité, nous avons fait appel aux émetteurs et récepteurs de signaux, pour permettre une communication entre stéréotype présentation et état d'activité. De ce fait, une activité de présentation émettra un signal, capté par un récepteur associé à une activité métier. D'après ce principe le diagramme de la Figure 14 devient :

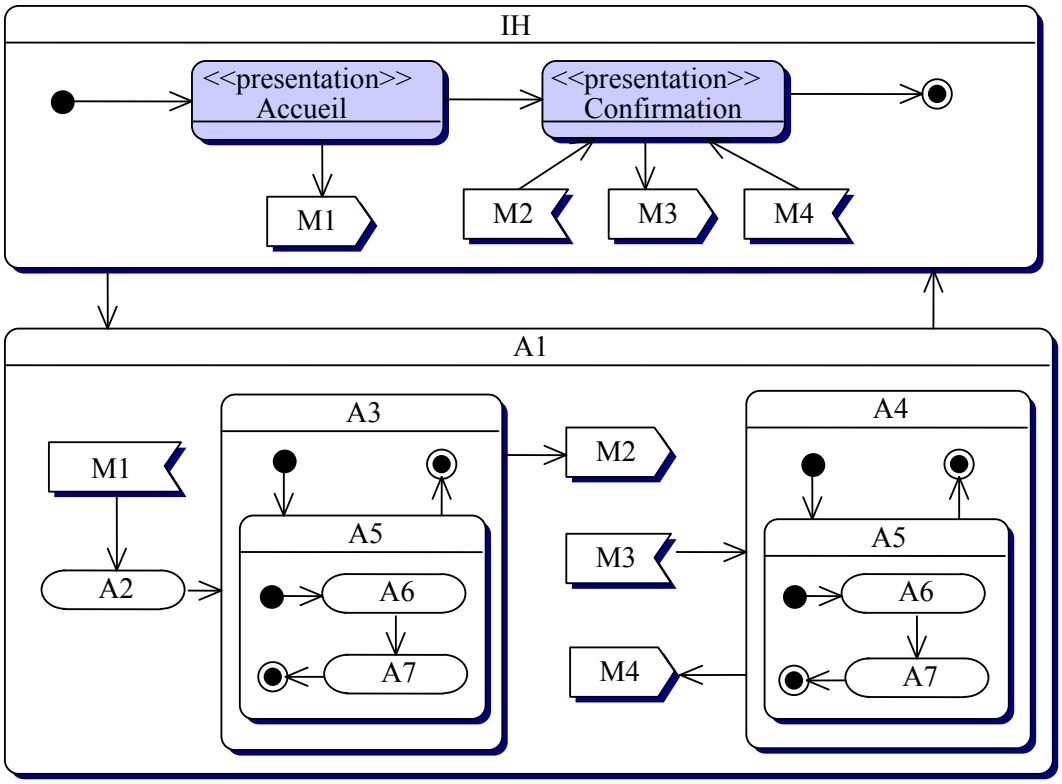


Figure 16 L'utilisation du stéréotype "présentation"

Du diagramme d'activité Figure 16 nous pouvons remarquer que notre état d'activité A1 a été décomposé en deux enchaînements disjoints d'activités : l'un consécutif à la réception du message M1, l'autre au message M3. Ainsi si l'introduction du stéréotype de présentation n'a pas vocation à préciser la nature de l'IHM du service, elle se justifie par l'information qu'il apporte quant aux stimuli en entrée des états d'activité destinés à devenir nos composants.

Préciser les interfaces de nos futurs composants demande une spécification des messages échangés entre stéréotypes presentation et activités. Pour reprendre notre formalisme de description des données, un message est constitué d'un ensemble de couples (Type, Nom), tandis que les émetteurs et récepteurs du message assurent la traduction entre les informations contenues dans le message et les données internes à l'activité. Cette transcription, directement effectuée au niveau d'un émetteur ou récepteur, nous permet de lier une composante du message à respectivement une donnée en sortie des activités précédant le récepteur ou en entrée d'activités succédant à l'émetteur. Pour reprendre l'exemple de la Figure 16, une implémentation de M1 pourrait être :

```
M1 : String  login
     String  password
     double  kilometers
     int     category
```

Un exemple d'émetteur et récepteur pourrait à ce moment être :

Emetteur :	Récepteur :
`M1!login = Accueil?id`	`M1?login = A2!login`
`M1!password = Accueil?auth`	`M1?password = A2!pass`
`M1!kilometers = Accueil?kilos`	`M1?!kilometers = A3!length`
`M1!category = Accueil?cat`	`M1?category = A3!base`

Nous pouvons remarquer que d'une part la définition de l'émetteur permet de préciser les données attendues dans la partie présentation du service, alors que le récepteur, future méthode d'un composant métier, reprend les données du message qui seront manipulées dans l'enchaînement d'activités.

Resource

Ce stéréotype également dérivé de l'activité permet la définition d'activités dont la portée va au-delà de notre modèle de composants. L'architecte a ainsi la possibilité d'utiliser ce stéréotype pour préciser que la résolution d'une activité passe par l'utilisation d'une ressource telle la consultation d'une base de données, l'envoi d'un message dans un Middleware Orienté Messages ou l'utilisation d'un moteur Transactionnel. La gestion de ce stéréotype peut passer par l'utilisation d'une bibliothèque de ressources, chaque élément référençant les opérations exécutables sur la ressource, et les données utiles à la résolution de chacune de ces opérations.

L'utilisation de ce stéréotype dans un diagramme d'activité demande donc de préciser à la fois le nom du stéréotype, correspondant à une entrée dans le référentiel des ressources connues, et le nom de la méthode utilisée sur cette ressource. Admettons que dans notre exemple de la Figure 16, l'architecte souhaite exécuter l'activité A2 dans le cadre d'une transaction. Cette activité pourrait alors être raffinée suivant l'état d'activité :

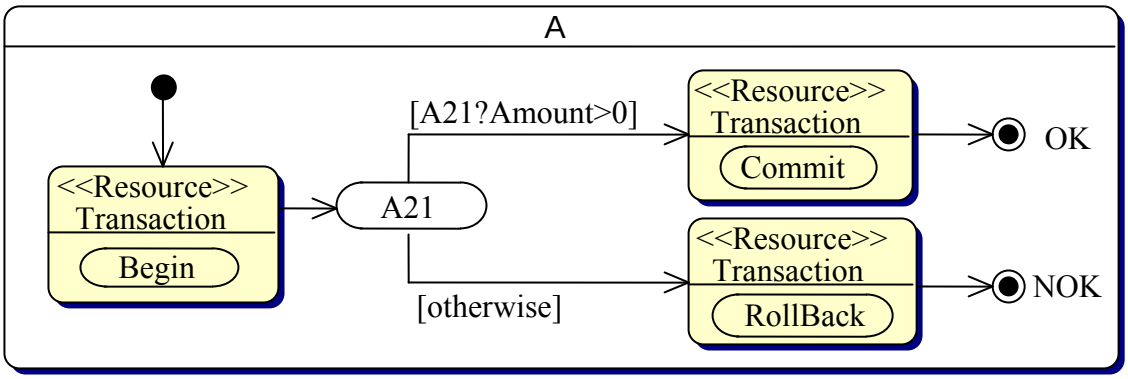


Figure 17 Utilisation du stéréotype "Resource"

Cet état d'activité fait appel à trois instances du stéréotype Resource, chacune d'entre elles faisant référence à la même ressource "Transaction", pour des opérations néanmoins différentes. Le premier stéréotype délimite le début de la transaction, dont la portée est déterminée par le positionnement de stéréotypes dédiés aux opérations "Commit" ou "RollBack" de la même ressource Transaction.

Conformément au schéma de la Figure 7, l'utilisation d'une ressource se fait à travers l'utilisation d'un composant de type Controller. Aussi l'utilisation d'un stéréotype Resource, permettra la génération automatique d'un composant controller attaché à une ressource particulière (un service Transactionnel dans notre exemple ci-dessus), utilisable par le composant issu de l'état d'activité hébergeant le stéréotype (ici l'état d'activité A2).

Exception

Le stéréotype d'Exception a été introduit pour préciser les comportements anormaux d'une activité. La Figure 18 nous présente une activité A1 normalement suivie de l'activité A2. Cependant, si l'activité A1 s'achevait avec une variable val en dehors de l'intervalle [0;100], l'exception OutOfBoundsException serait levée avec l'exécution de l'activité A3.

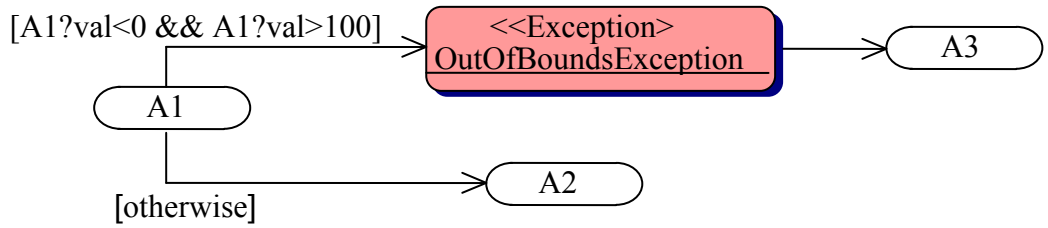


Figure 18 Utilisation du stéréotype "Exception"

Introduire le stéréotype Exception dans un diagramme d'activités permet de compléter le diagramme en intégrant les scénarios alternatifs définis dans le diagramme de cas d'utilisation. Le stéréotype apparaît comme un point d'initialisation d'un enchaînement classique d'activités, sa survenue étant conditionnée par la garde portant sur la transition la précédant.

3.4.2.4 Etude de cas

Pour développer l'analyse de notre service de paiement d'autoroute, nous allons maintenant construire le diagramme d'activité en rapport avec le cas d'utilisation "Payer". La définition de ce diagramme débute par une phase de génération automatique, tenant compte du diagramme de Use Case donné en Figure 12.

L'application de la transformation φ à notre diagramme de Use Case mène à la création du diagramme d'activité suivant :

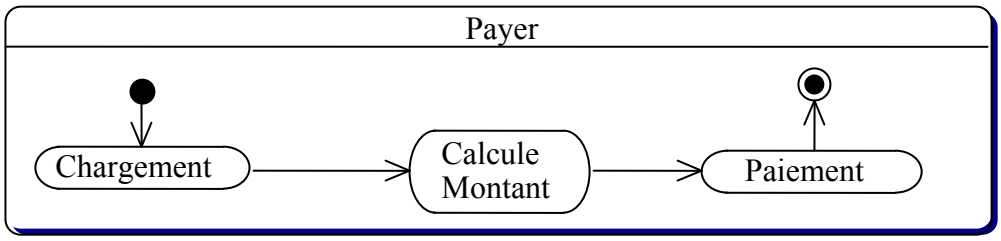


Figure 19 Le diagramme d'activité généré automatiquement

Cette génération produit un premier état d'activité (Payer) contenant les activités assumées par les futurs composants du service. Pour préciser les points d'interaction entre les composants et l'extérieur du service, l'architecte a la possibilité à ce moment d'utiliser les stéréotypes de présentation. Un exemple d'évolution du diagramme d'activité pourrait alors être celui donné en Figure 20. Ce dernier diagramme permet de préciser la nature des données en entrée et en sortie des différents enchaînements d'activités constituant l'état d'activité "Payer". Le premier enchaînement fait suite à la réception d'un message UserData, construit à partir des données caractérisant l'utilisateur du service. La définition de ce message demande une première introspection des activités "Chargement"et "Calculer Montant", afin de déterminer les données utiles à leur accomplissement. L'existence de ce message permettra d'une part de générer automatique l'interface du composant en entrée du service en cours de création, et d'autre part de préciser la structure du formulaire de présentation du service. De la même manière, l'enchaînement d'activités se termine par l'envoi d'un message "Amount" relatant le montant du paiement demandant au préalable une validation de l'utilisateur. Nous admettrons que le choix de l'architecte a été de retourner un booléen au service, signifiant la validation ou non de la transaction à l'utilisateur, son exécution générant un identifiant de transaction affiché à l'utilisateur pour un archivage de sa part.

```
UserData : String  login       Amount :      double  price
           String  password                   vector  accounts
           double  kilometers
           int     category

Execute :  bool    validated   Terminated :  long    transactionId
           Long    account
```

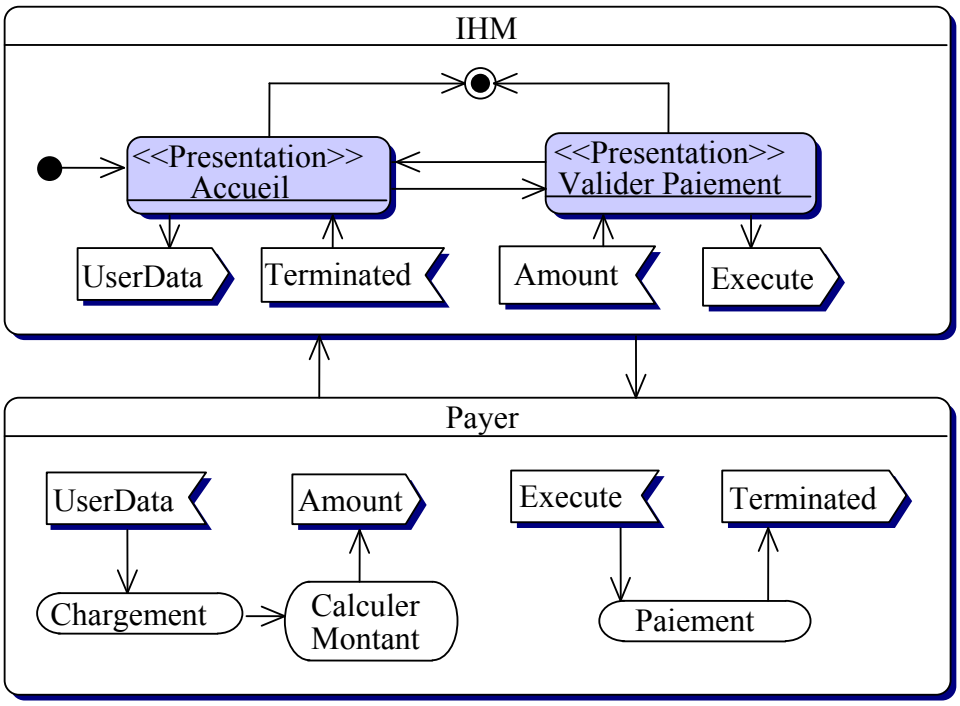


Figure 20 Le diagramme d'activité avec les stéréotypes "Presentation"

Les éléments de présentation définis, l'affinage du diagramme d'activité peut se poursuivre avec la définition des ressources utilisées par le service. L'activité "Chargement" se décomposera ainsi en une requête effectuée sur une base de données, suivie d'un paramétrage du service. La requête demande l'utilisation d'un stéréotype "Resource" Database, sur lequel nous utiliserons l'opération select. L'analyse de l'activité peut aller jusqu'à la définition de la requête SQL, utilisée comme argument de l'opération select, et la sélection des données utiles à l'utilisation de la ressource sous-jacente, qui sera ici la base de données. La phase de paramétrage du service utilise le résultat de la requête effectuée sur la base de données, pour préciser les contraintes liées à la tarification. A cet effet, la signature de l'activité pourrait être :

```
inout   vector   Accounts
inout   int      UserCategory
```

La relation entre les données d'entrée de l'activité et les données de sortie du stéréotype Resource est assurée par les égalités:

```
Paramétrage!UserCategory  = DataBase.Select?Subscription
Paramétrage!Accounts      = DataBase.Select?Accounts.Id[]
```

Nous admettrons également que l'activité "Paiement" sera exécutée dans le cadre d'une transaction, de manière à permettre un retour en arrière en cas de problème.

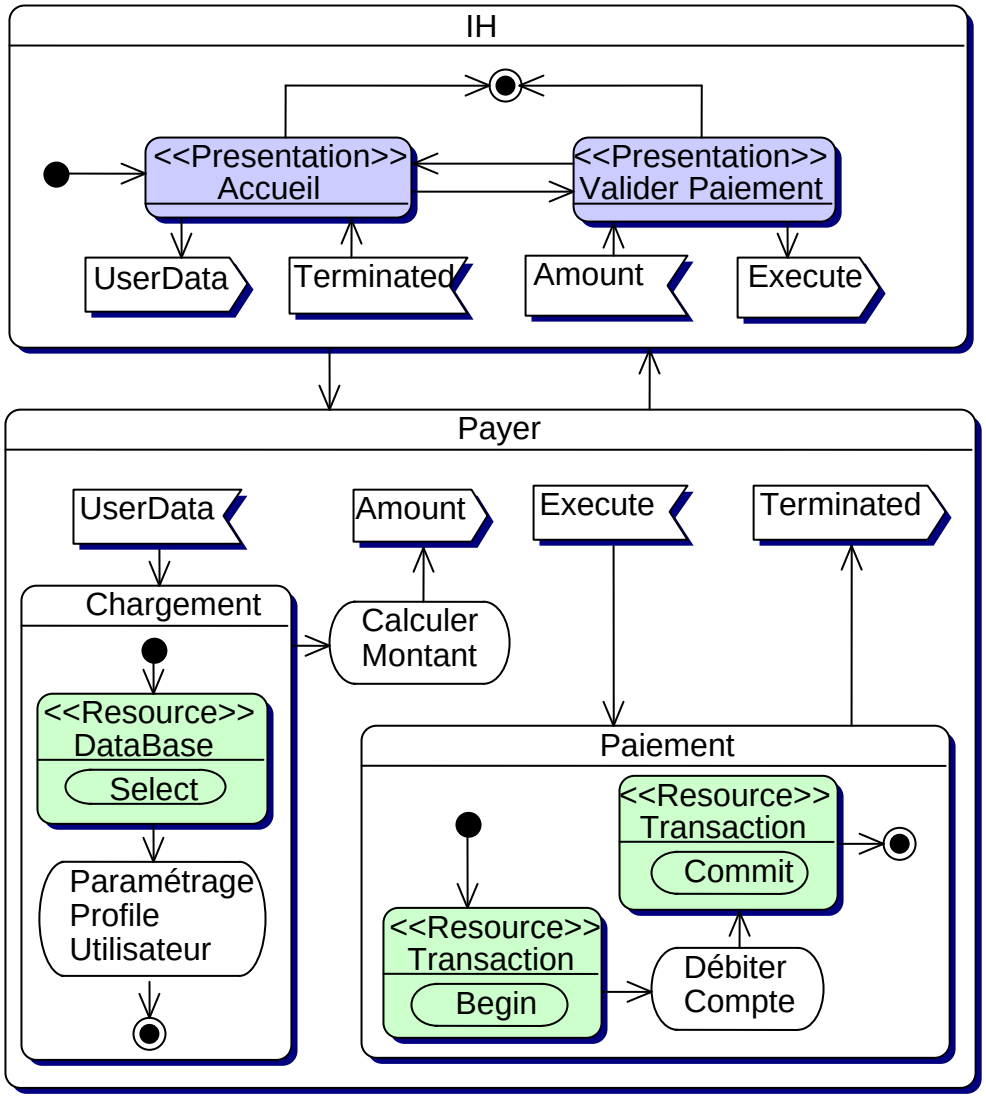


Figure 21 Le diagramme d'activité avec les stéréotypes "Resource"

Après avoir transposé l'exécution normale du service en diagramme d'activités, nous devons enrichir cette représentation en intégrant les enchaînements alternatifs décris dans le cas d'utilisation. Cet enrichissement est rendu possible par la définition de gardes sur les transitions et l'utilisation de nos stéréotypes Exception. Les gardes sont utilisées pour déterminer la transition à franchir parmi l'ensemble des transitions issues de l'activité ou de l'état d'activité qui vient de s'achever. Dans ce dernier cas, la liaison entre un état terminal d'un état d'activité et une transition en sortie de cet état d'activités, est assurée par le libellé de l'état terminal, réutilisé pour la garde de la transition.

L'introduction des exceptions s'est accompagnée de la définition d'un stéréotype de présentation spécialisé dans la gestion des erreurs.

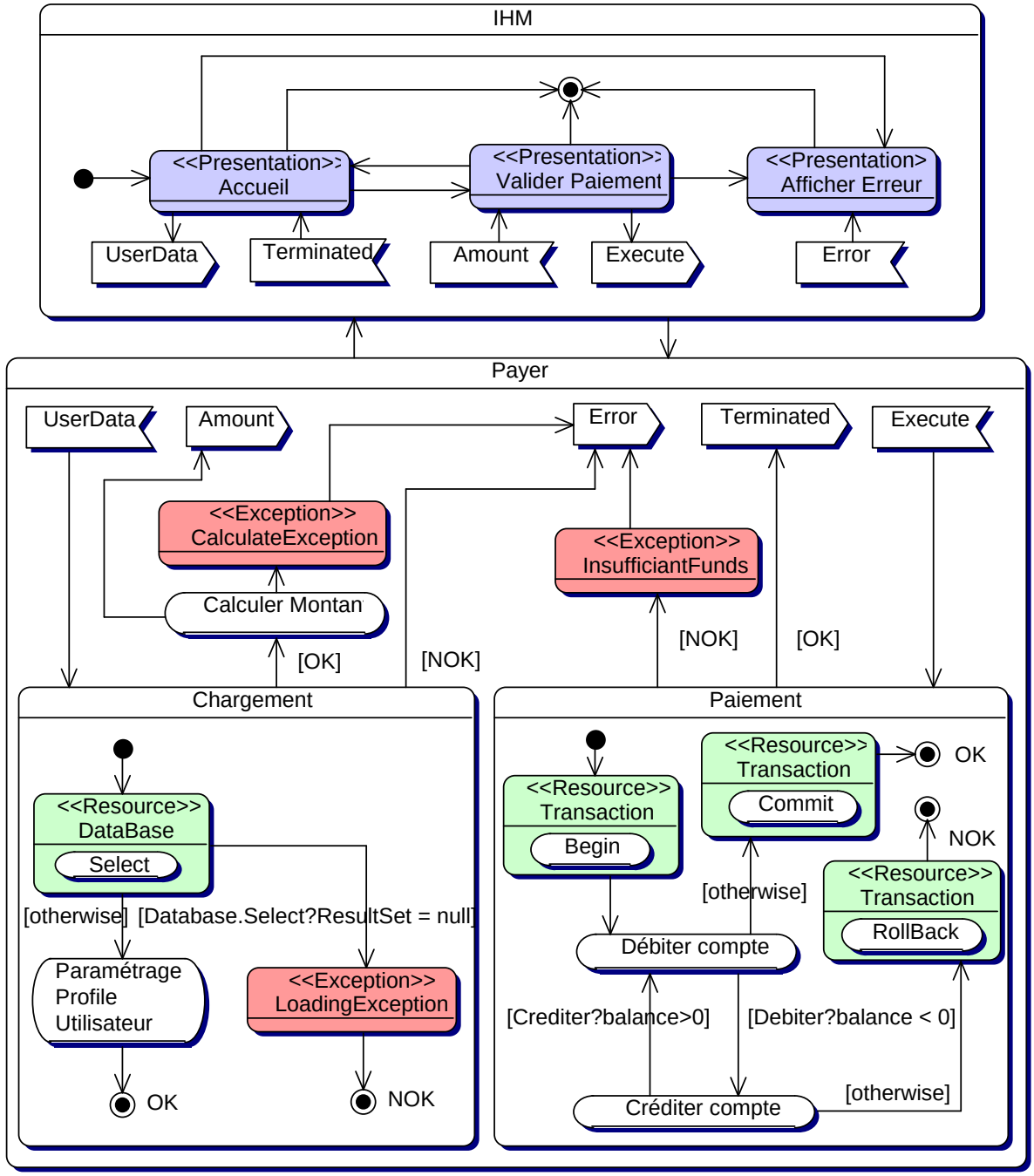


Figure 22 Le diagramme d'activité intégrant les chemins alternatifs

3.4.3 Le diagramme de composants

3.4.3.1 présentation

Avec ce troisième type de diagramme nous commençons à répartir les fonctionnalités du service sur des unités logicielles abstraites appelées composants. Du point de vue extérieur, nos composants peuvent être représentés sous la forme de boites munies de facettes et réceptacles, précisant leurs apports et dépendances envers leur environnement.

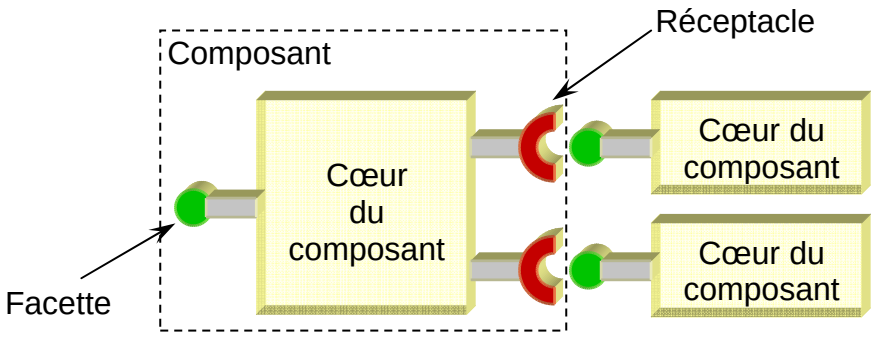


Figure 23 Réceptacles et Facettes

La notion de facette et de réceptacle avait été introduite avec les Composants Corba (CCM) [MME01]. Une facette expose un ensemble de méthodes mises à disposition des clients du composant. A l'opposé un réceptacle précise au serveur d'application, qu'un service rendu par ce composant ne peut être accompli sans l'aide d'un composant tiers. Nous verrons par la suite que nous extrapolerons ces notions de facette et réceptacle pour aller jusqu'à y préciser des contraintes de qualité de service, et contreparties permettant entre autre d'appréhender les questions de facturation liée à l'utilisation des composants.

Générer un diagramme de composants pertinent pour la phase de développement, demande la définition d'un formalisme, permettant de synthétiser le comportement du cœur de nos composants, sur la base des informations figurant dans le diagramme d'activité. Pour rester dans l'esprit d'UML, nous avons voulu que ce formalisme intègre une forte composante graphique, aussi nous avons pris le cœur du composant comme cadre de l'expression graphique de son activité. Pour représenter le comportement d'un composant, nous avons introduit la notion de point d'activité. Un point d'activité matérialise un ensemble de traitements internes au composant. En effet, si les réceptacles représentent un besoin du composant en interaction avec un autre composant, il est un certain nombre d'actions que le composant peut réaliser en mobilisant les classes qui le compose. Cette activité, qui n'apparaît pas au niveau des communications entre composants, peut néanmoins être détaillée au niveau des points d'activité. Ce détail comprend pour chaque traitement du point d'activité : l'état de l'environnement (l'ensemble des informations connues), l'emplacement et l'action réalisée. Notre formalisme représente un point d'activité sous la forme d'un disque plein, auquel se rattachera textuellement l'information le détaillant.

A partir de ces points d'activité, nous pouvons définir le comportement d'un composant comme étant un chemin soumis à un certain nombre de contraintes, reliant un ensemble de

points d'activités. Comme l'utilisation d'un composant demande toujours de passer par l'une de ses interfaces, nous avons fait correspondre à nos facettes un point d'acticité au sein du cœur du composant, initiant le chemin de l'implémentation de l'interface. De la même manière, pour intégrer les réceptacles au chemin décrivant le comportement du composant, nous avons fait correspondre à nos réceptacles un point d'activité au sein du cœur du composant. Cependant, un réceptacle faisant par nature référence à une activité accomplie par un autre composant, nous parlerons ici de point d'activité distant, par opposition aux points d'activité locaux du composant. Ces points d'activité distants seront représentés sous la forme d'un disque grisé, comme l'indique la Figure 24.

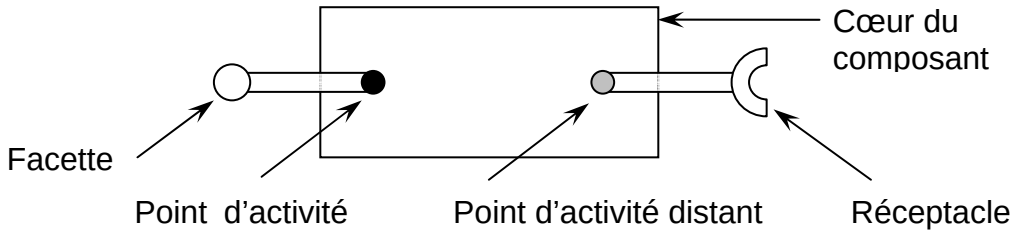
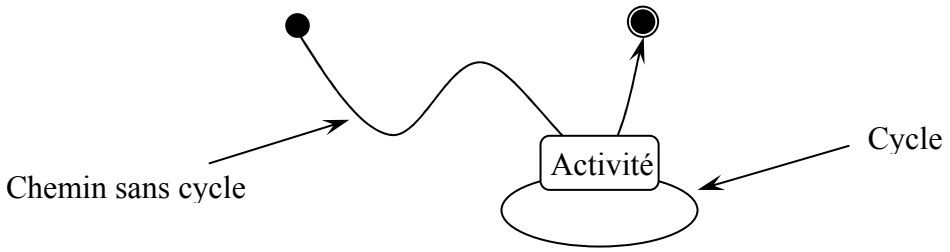


Figure 24 Les points d'activité

3.4.3.2 L'algorithme de transformation

Pour assurer une certaine continuité de la phase d'analyse d'une application, nous nous devons de proposer un algorithme qui permette le passage d'un diagramme d'activité à un diagramme de composants. En préambule nous noterons qu'un diagramme de composants étant plus exhaustif qu'un simple diagramme d'activité, il ne peut exister d'algorithme empirique permettant de générer automatique un diagramme de composants complet (ie. décrivant entièrement les facettes et réceptacles de nos composants) sur la base d'un diagramme d'activités. Néanmoins, nous allons introduire un algorithme, qui permet la génération partielle de notre diagramme de composants, en laissant à l'architecte le soin d'apporter l'information supplémentaire, qui pourra être exploitée pendant la phase d'implémentation du composant.

La génération de notre diagramme d'activité part du constat que la définition de tout service réside au sein d'un état d'activité contenant d'autres états d'activités, de simples activités (cf. Figure 22) et des stéréotypes de présentation, ressource et exception. Les transitions reliant ces différents éléments constituent ainsi des chemins d'exécution du composant, que notre algorithme de transformation devra parcourir itérativement pour générer un digramme de composant. Par chemin nous entendons des enchaînements sans cycle d'éléments d'activité à l'intérieur d'un état. En effet, nous avons adopté la convention que toute activité sur un chemin sans cycle, mais étant à la base d'un cycle (cf. figure ci-dessous), représentait un composant à part entière, en charge de la gestion du cycle, qui sera rattaché au chemin sans cycle via un réceptacle au moment de construire le diagramme de composants.



La construction du diagramme de composants nous demande de définir les ensembles suivants :

- Soit $(\Phi_i)_0^n$ l'ensemble des chemins sans cycle reliant l'état initial de l'état (ou l'un de ses récepteurs de messages), à l'un de ses états terminaux (ou émetteur de messages).

- Soit $(A_{i_j})_0^{p_i}$ la suite d'éléments d'activité présents sur le chemin Φ_i tel que pour $\forall i$, A_{i_0} est l'état initial de E ou un récepteur de message et $A_{i_{p_i}}$ l'un de ses états terminaux ou un émetteur de messages.

Appliqué à l'état principal du diagramme d'activité l'algorithme de transformation aura le comportement suivant :

```
Construire(Etat E)
  Créer un composant C.
  Créer une facette F sur le composant C.
  Pour tout i allant de 0 à n faire

  Créer une méthode mi sur F, reprenant le nom de E
    Construire signature de mi tel que :
     A_i0 correspond aux paramètres d'entrée de mi
     A_ipi correspond au type retourné par mi
    créer p un point d'activité local
    lier p à Construire( A_i1 )
    implémentation de mi ← p
  Fin pour tout
  Retourner C
Fin construire
```

Cette première partie de l'algorithme de transformation permet de passer d'un chemin d'activité, délimité d'un coté par un récepteur de message R_1 et de l'autre par un expéditeur S_1, à une méthode m_1 sur notre composant à la signature S_1 : $m_1(R_1)$. A l'implémentation de cette méthode correspondra ensuite la transformation du chemin reliant récepteur et expéditeur du message.

Appliqué à une simple activité, l'algorithme de transformation devra avoir le comportement.

```
Construire(Activité A)
  Créer un Réceptacle R.
  Créer un composant C.
  Créer une facette F sur le composant C.
  Créer une méthode m_i (nom typée pour le moment) reprenant le nom de A.
  Attacher m_i à la facette F.
  Soit (T_k)_{k=0}^{q_A} l'ensemble des transitions sortantes de A.
  Soit (Δ_k)_{k=0}^{q_A} l'ensemble des gardes portant sur les transitions de (T_k)_{k=0}^{q_A}.
  Pour k allant de 0 à q_A faire
    Soit Elmt l'élément d'activité à l'extrémité de la transition T_k
    Si type de Elmt = "Exception"
      Ajout l'exception Elmt sur F et sur R.
      Lier le retour en exception de R avec le résultat de la
      transformation du successeur de Elmt.
    Sinon
      lier via une condition Δ_k, le retour du réceptacle R au résultat de
      la transformation construire(Elmt)
    Fin Si
  Lier R à F.
  Retourner R
Fin construire
```

Cette deuxième partie de l'algorithme de transformation permet d'éclater notre application en de multiples composants, reliés entre eux par un mécanisme de facette et réceptacle. Le traitement d'une activité commence par la construction d'un composant vide, modélisant l'Activité en cours de transformation. Nous greffons sur ce composant une facette (et le point d'activité qui lui est associé au cœur du composant) initialement vide, à l'intérieur de laquelle nous plaçons une méthode reprenant le libellé de l'activité. La méthode à ce niveau n'est pas encore comparable à l'idée que nous nous en faisons dans un langage de programmation, mais elle constitue l'embryon du futur service rendu par le composant. En effet, il n'est pas question à ce niveau de parler de signature de la méthode ni de type retourné, ces informations n'étant pas présentent dans le diagramme d'activités. En revanche, une fois la phase de génération automatique du diagramme de composants, un opérateur pourra

manuellement préciser les définitions de ces méthodes, pour raffiner les interactions entre composants. La lecture de cet algorithme permettra de remarquer que nous manipulons à ce niveau des composants, facettes et réceptacles dénués de toute identité. Ce manque apparent de rigueur vient du fait que ces entités ne sont pour le moment vues que comme des cartouches abstraites, dans lesquelles nous injectons au fur et à mesure des méthodes, contraintes de retour, comme autant d'éléments de typage qui permettront de typer nos facettes, réceptacles et par transitivités nos composants, au terme de la construction de notre diagramme de composants.

D'après notre algorithme, nous générons des facettes et par la suite des réceptacles ne contenant qu'une unique méthode. Cette contrainte, imposée du fait de la non interprétation des activités par notre algorithme, tombera dès l'intervention d'un opérateur humain, qui pourra fusionner les facettes et réceptacles unitaires, le cœur du composant gérant autant de point d'activités liés à la facette ou au réceptacle, que de méthodes qu'ils hébergent. Par exemple en Figure 25, notre algorithme n'est pas capable d'identifier que les trois réceptacles `begin`, `rollback` et `commit` sont dans la pratique associés au même composant (Transaction).

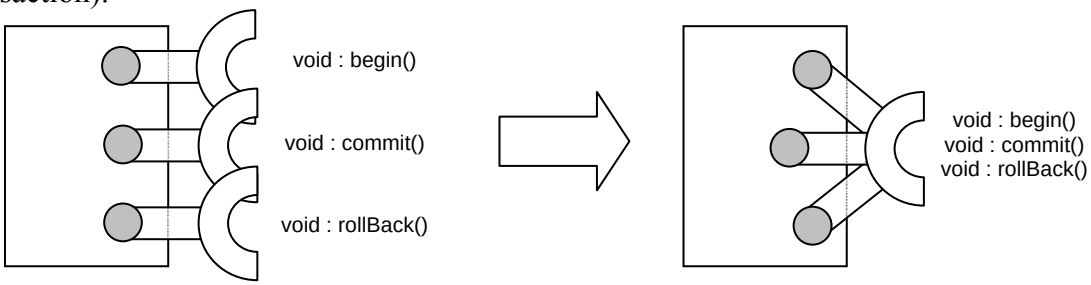


Figure 25 La fusion de réceptacles

L'algorithme de transformation ayant vocation à produire un diagramme de composants, il ne sera pas amené à considérer les états hébergeant des stéréotypes de présentation. En revanche, il devra considérer les stéréotypes `Resource`, pour établir les relations du composant avec les éléments extérieurs au serveur d'application. Comme nous le présentions en partie 3.2.5, l'utilisation de ressource fait appel à un composant contrôleur, interfaçant le composant client de la ressource et son pilote. Ainsi, sur le modèle des algorithmes précédemment exprimés, en appliquant la transformation au schéma suivant :

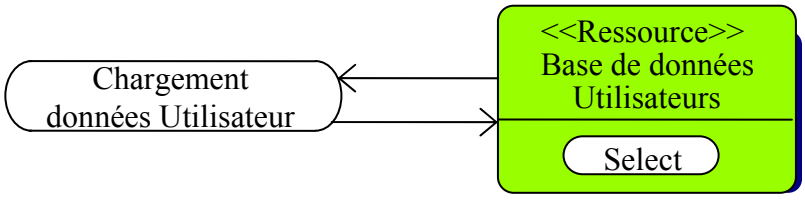


Figure 26 L'activité à transformer

Nous obtenons un diagramme de composants correspondant à un composant classique et un composant contrôleur.

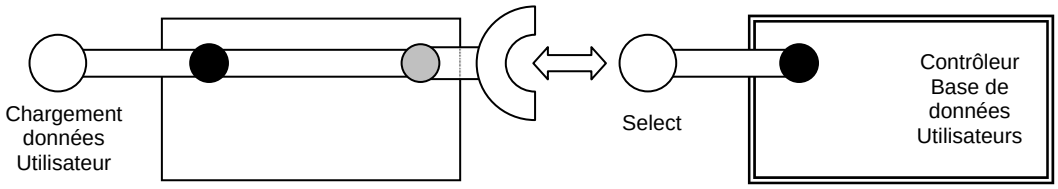


Figure 27 Les composants générés automatiquement

Appliqué au diagramme d'activité de notre exemple de la Figure 22, l'algorithme de transformation produira un diagramme de composants donné en Figure 28. Nous avons représenté en partie gauche les différents éléments de présentation, hors du système mais accédant néanmoins au système, reliés aux facettes du composant frontal issu de la transformation de l'état d'activité principal (`Payer`) du diagramme d'activité. chaque activité ou état d'activité a ensuite produit un composant autonome, que l'architecte pourra regrouper ultérieurement en composants plus généraux, reliés entre eux par des facettes et réceptacles.

Si ce diagramme donne une vue fonctionnelle des composants à construire, il est encore notoirement insuffisant dans son expression des activités assumées par ces derniers, avec l'objectif de faciliter la phase de développement en automatisant une partie de la génération du code du composant. De ce fait nous allons introduire un formalisme essentiellement graphique au sein du diagramme de composants, capable de préciser les traitement internes effectués.

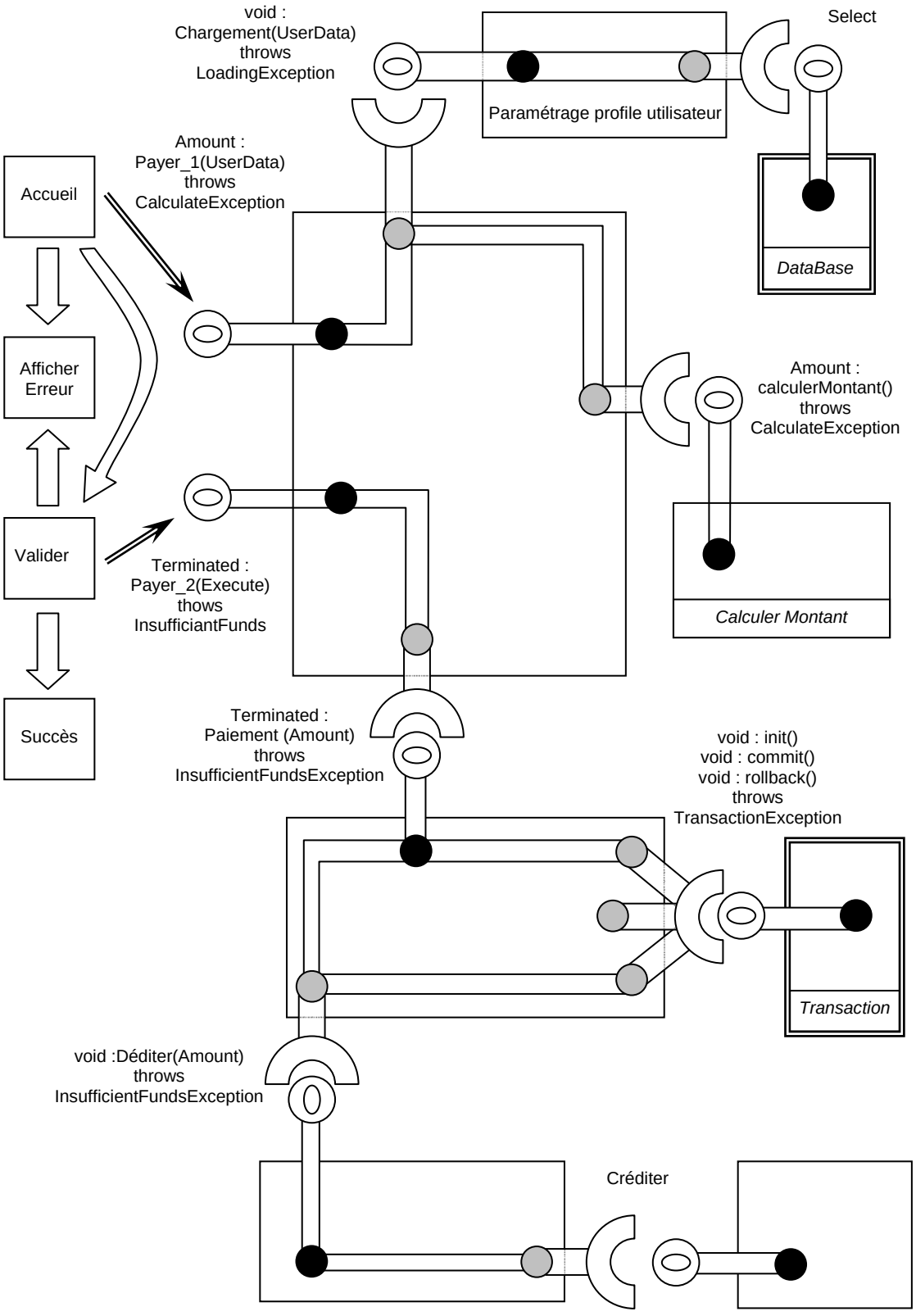


Figure 28 Le diagramme de composants généré automatiquement

3.4.3.3 Le formalisme de description de l'activité des composants

Comme base de l'expression de notre notation, nous avons décomposé le chemin bidirectionnel reliant deux points d'activité (représenté en Figure 29) sous la forme de trois flux, donnant une représentation intermédiaire entre les transitions évènementielles des diagrammes d'activité et les méthodes des diagrammes de classes utilisés lors de l'implémentation. Le premier flux : le flux Aller est représenté sous la forme d'une flèche discontinue, assurant l'orientation du chemin. Ce flux peut être associé à un ensemble de contraintes visant à préciser les informations présentées en entrée du point d'activité destinataire du flux. Les deux autres flux, représentés sous la forme d'une ligne continue, matérialisent la sortie normale et la sortie en erreur du point d'activité invoqué lors du flux aller.

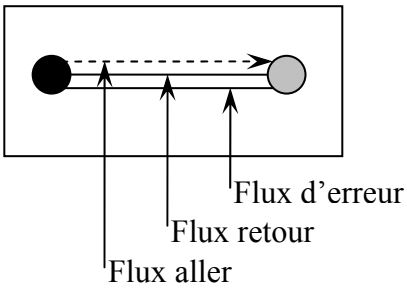


Figure 29 La décomposition du chemin reliant deux points d'activité

Cette décomposition des chemins nous permet de préciser la relation entre les facettes et le cœur du composant, tel qu'exprimé en Figure 30. La facette est connectée à un port d'entré du corps du composant, symbolisant la présentation de la méthode au code d'implémentation du composant. La définition de ce port (`I1` dans notre exemple ci-dessous) nous sert de support à l'introduction au niveau du conteneur, de mécanismes d'interception des données envoyées par le client. Un premier chemin joint à ce moment la facette au port d'entrée du cœur du composant, le port étant ensuite relié unidirectionnellement à un point d'activité local. En sortie de ce point d'activité, nous retrouvons une transition reliant le point d'activité à point de retour implanté sur un flux de sortie du chemin entre la facette et le corps du composant. Nous verrons par la suite qu'un point de retour appliqué d'un filtre peut permettre de lier le retour d'un point d'activité à l'exécution d'un autre point d'activité. Pour le moment la construction de la transition de sortie du point d'activité impose la définition d'un port terminal du corps du composant (`T1` dans notre exemple), matérialisant l'interception du retour du composant par le conteneur, avant son envoi proprement dit au client. D'après ces définitions nous avons introduit en Figure 30 un composant présentant la méthode triviale `m1`, dont l'implémentation reste locale.

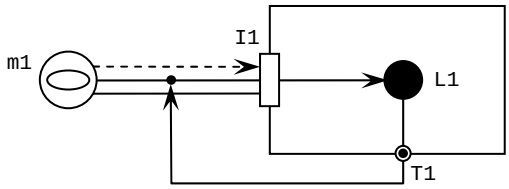


Figure 30 La relation entre une facette et le cœur du composant

A la représentation graphique ci-dessus nous pouvons associer une notation plus formelle, capable d'assumer la génération automatique de code. Au préalable nous introduirons les définitions suivantes :

- Nous noterons $L1^R$ le flux de sortie standard du point d'activité L1.
- Nous noterons $L1^E$ le flux de sortie en erreur du point d'activité L1.
- Nous considèrerons par la notation $I1_<$ la structure de données présentée en entrée du port I1, avant son éventuel traitement par le conteneur, et par $I1^>$ la structure de données présentée en sortie de I1.
- Nous considèrerons par la notation $T1_<$ la structure de données présentée à T1, et par $T1^>$ la structure de données retournée par le port T1, après son éventuel traitement par le conteneur.
- Nous noterons <m1> l'implémentation de la méthode m1, correspondant à une collection de chemins d'exécution reliant un port d'entrée du composant à un port de sortie du composant.

D'après ces définitions nous pouvons en déduire les propriétés suivantes :

- $I1_<$ correspond au type du message reçu en entrée de l'état d'activité à l'origine de la création du composant.
- $T1^>$ correspond au type du message retourné en sortie de l'état d'activité à l'origine de la création du composant.
- m1 n'étant reliée au cœur du composant que par un unique chemin, $I1_<$ représentera les paramètres d'entrée de m1.
- m1 pouvant être atteint depuis le cœur du composant par une multitude de transition passant par des ports Ti, les données en sortie de m1 correspondront à la réunion des structures $Ti^>$. Les transitions présentent la contrainte que tout les Ti reliés au flux de sortie standard de la facette doivent être de même type.

D'après l'exemple de la Figure 30 nous pouvons déduire :

- <m1> = {T1∘L1∘I1}
- signature de m1 = $T1^>$: m1($I1_<$)

Note : en notation fonctionnelle, T1∘L1∘I1 signifie que l'action de m1 correspond à l'action de T1 appliquée au retour de l'action de L1, lui-même issu de l'action de I1.

La signature d'une méthode est obtenue en considérant les deux extrémités des expressions figurant dans l'ensemble d'implémentation de la méthode. En l'occurrence la seule expression référencée dans notre exemple mentionne les extrémités I1 et T1. La signature de la méthode déterminée, les différentes expressions sont rapprochées pour générer proprement dit un squelette du futur code du cœur du composant. Or la recherche des expressions d'implémentation est donnée comme ci-dessus en notation fonctionnelle, car une expression modélisant un chemin reliant le port d'entrée à un port de sortie du composant, sa construction demande cette fois un parcours du diagramme d'un port de sortie vers le port d'entrée du composant. Cependant, le rapprochement entre les différentes expression sera plus aisé avec une représentation inverse des expressions, correspondant à l'enchaînement des

traitements du composant. A partir de cette inversion, nous écartons des expressions leurs extrémités, pour ne retenir que l'enchaînement des points d'activités constituant le code d'implémentation de la méthode. Ce code sera néanmoins dépendant des paramètres d'entrée de la méthode, et devra retourner à son terme une structure de données conforme à la signature de la méthode. Pour notre exemple, l'expression privée de ses extrémités se limitant à un simple `L1`, le code de la méthode générée prendra la forme :

```
public T1> m1(I1<)
{
  L1(I1>◁Env_init)
  return T1<
}
```

La génération du code s'intéresse au point d'activité `L1`, qui s'exécute dans l'environnement initial du composant enrichi par les données reçues sur le port d'entrée du composant, retraitées par le conteneur. Si nous fixons les paramètres de notre exemple avec le cas d'utilisation :

- `I1 : int`
- `T1 : int`
- `L1 : int j = i+1; return j;`

Le code généré automatiquement ressemblera à :

```
public int m1(int i)
{
  int j = i+1;
  return j;
}
```

Généraliser la génération du code associé à un point d'activité nous demande de voir chaque transition sortant du point d'activité comme un chemin alternatif dans le code, dont le type du retour correspondra à l'étiquette de la transition. L'implémentation de la méthode `m1` exposée dans la Figure 31 possède ainsi trois expressions d'implémentation :

- `<m1> = {T1∘L1∘I1, T2∘L1∘I1, T3∘L1∘I1}`

La signature associée ressemblera à

- `T1> : m1(I1<) : T2>, T3>`

Ou en d'autres termes :

- `R : m1(I1<) : E2, E3`

Lors du rapprochement des expressions d'implémentation privées de leurs extrémités, nous n'aurons qu'une expression se limitant à L1, avec cependant des conditions de sortie qui diffèrent, occasionnant autant de branchements au terme de l'exécution de L1. Le code associé à notre exemple de la Figure 31 deviendra ainsi :

```
public T1> m1(I1<) throws T2>, T3>
{
  L1(I1>)
  <branchement1>
    return T1<
  <branchement2>
    return T2<
  <branchement3>
    return T3<
}
```

Notre formalisme ne va pas jusqu'à exprimer graphiquement le test effectué dans chaque branchement (ce qui rendrait le schéma rapidement illisible), cependant il est concevable d'accompagner le point d'activité d'informations textuelles permettant la définition automatique de ces tests dans le code d'implémentation. Dans la pratique un branchement peut être assumé par une construction `if/then/else` et selon certains langages de programmation par une structure `switch`. Un exemple d'implémentation de m1 pourra ainsi être en Java:

```
private static final int low = 0;
private static final int up = 10;

public int m1(int i) throws LowerException, UpperException
{
  int j = i+1;
  if ((i>low)&&(i<up))
    return j;
  else if (i>low)
    throw new UpperException();
  else
    throw new LowerException();
}
```

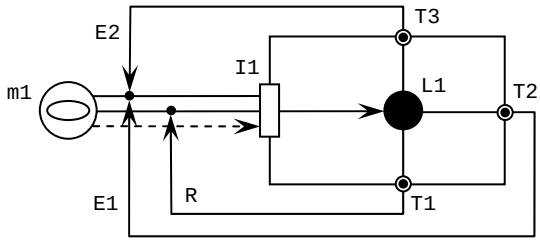


Figure 31 Un exemple de point d'activité aux retours multiples

Notre formalisme peut ensuite être utilisé pour exprimer les relations entre plusieurs points d'activité. Si nous considérons un composant délivrant une méthode `m2` (cf. Figure 32), nécessitant l'utilisation de la méthode `m1` vue précédemment, le point d'activité `L2` implémentant `m2` va devoir entrer en relation avec un point d'activité distant `D2`, pour accéder à la facette du composant distant via un réceptacle.

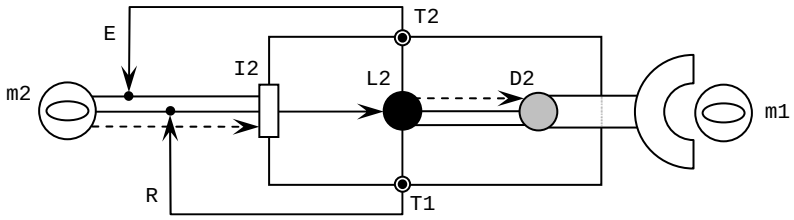


Figure 32 La relation entre points d'activité locaux et distants

Un point d'activité distant dans notre formalisme nous donne une abstraction de l'utilisation de la classe générée à partir de l'importation du descripteur du composant distant (cf. paragraphe 3.5.2), jouant le rôle de proxy de la facette distante. Cet élément est utilisé par le composant pour permettre une construction locale des appels de méthode, qui seront ensuite distribués vers le composant distant. L'implémentation de `L2` dans l'exemple de la Figure 32 s'articulera ainsi :

```
Code local
Recherche du proxy assurant l'accès à la facette connectée
                          au réceptacle
Appel d'une méthode sur le proxy/réceptacle
Retour
```

Le code de la classe d'implémentation de notre composant s'enrichira alors du mécanisme de recherche de la facette connectée au réceptacle, et de l'appel à la méthode distante. Cependant notre formalisme s'avère encore insuffisant pour préciser la synchronisation entre l'exécution du code local `L2`, l'appel au point d'activité distant `D2` et le retour vers les ports terminaux `T1` et `T2`. En effet, compte tenu des outils jusqu'ici introduit, notre exemple de la Figure 32 peut se décliner en une multitude de développements dont deux instances sont données ci-dessous :

```
public T1> m2(I1<) throws T2>
{
  L2(I1>)
  try
  {
    D2(L2(I1>)◁I1>◁Env_init)
    return T1<
  }
  catch (E1)
  {
    return T2<
  }
  catch (E1)
  {
    return T2<
  }
}
```

```
public T1> m2(I1<) throws T2>
{
  try
  {
    D2(L2(I1>)◁I1>◁Env_init)
    L2(I1>)
    return T1<
  }
  catch (E1)
  {
    return T1<
  }
  catch (E1)
  {
    return T2<
  }
}
```

Pour faire tomber cet aspect non déterministe de notre formalisme, nous avons défini un deuxième niveau de représentation des points d'activité, correspondant à un zoom par point d'activité sur les interactions entre ports d'entrée/sortie du composant et les points d'activités

du composant qu'il utilise. Ainsi L2 pourra être déterminé par le schéma ci-dessous. Initialement présenté comme une unité atomique, le point d'activité local L2 est subdivisé en trois points d'activité L2a, L2b et L2c, en association avec les différentes interactions liées à l'utilisation du point d'activité D2. Le point d'activité L2a permet d'exécuter du code local avant l'appel à D2, le point d'activité L2b s'exécute en cas de retour standard de D2, alors que L2c sera mobilisé suite à la levée d'une exception E1 ou E2.

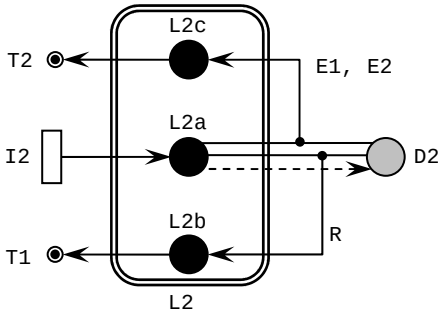


Figure 33 Le développement du point d'activité local L2

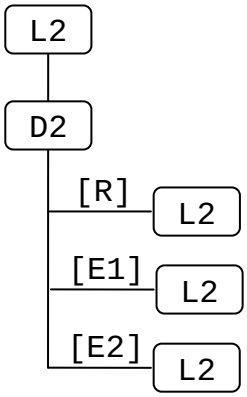
Si la détermination de la signature de la méthode m2 ne nécessite pas la prise en charge de ce second niveau de détail, la construction de l'ensemble des expressions d'implémentation de m2 devra considérer les deux plans de représentations de L2 pour une production déterministe du code du composant.
L'ensemble des expressions d'implémentation de m2 sera ainsi :

<m2> = {T1∘L2b∘D2[R]∘L2a∘I1, T2∘L2c∘D2[E1]∘L2a∘I1, T2∘L2c∘D2[E2]∘L2a∘I1}

Si nous détaillons cette fois le processus de génération du code du composant, les expressions contenues dans <m2> expurgées de leurs extrémités et inversées devient :

{L2a∘D2[R]∘L2b, L2a∘D2[E1]∘L2c, L2a∘D2[E2]∘L2c}

Le rapprochement de ces trois expressions donnera un code dont la représentation sera :



Les trois expressions ayant le même préfixe L2a, le code généré commencera par un tronc commun correspondant à un premier bloc de code local. L'appel au point distant figurant ensuite dans chaque expression avec néanmoins des filtres de retour distincts, l'appel sera inséré dans une construction try/catch, afin de satisfaire chaque branche de notre représentation ci-dessus. Nous arrivons ainsi au code intermédiaire ci-dessous, ne nécessitant plus qu'une spécialisation des différents points d'activité L2a, L2b, L2c et D2 et des types de données manipulés I1, T1, T2, R, E1 et E2 :

```
public T1> m2(I1<) throws T2>
{
  L2a(I1>◁Env_init)
  try
  {
     D2(L2a(I1>)◁I1>◁Env_init)
     L2b(R◁L2a(I1>)◁I1>◁Env_init)
     return T1<
  }
  catch (E1)
  {
     L2c(L2a(I1>)◁I1>◁Env_init)
     return T2<
  }
  catch (E1)
  {
     L2c(L2a(I1>)◁I1>◁Env_init)
     return T2<
  }
}
```

La construction de composants complexes faisant appel à des méthodes locales au composants (aussi appelées privées) ou parfois même à des classes induites, notre formalisme doit permettre à un point d'activité de s'exécuter en sortie d'un précédent point, en utilisant pour ce faire le mécanisme d'écoute des deux flux de sortie d'un point d'activité. Quand le nombre de points d'activité devient important, il peut être nécessaire d'utiliser des filtres sur les flux de retour, pour préciser les conditions de composition des points d'activité. Cependant, compte tenu du traitement particulier des erreurs dans les langages de programmation, nous avons distingué la manière d'appréhender ces deux flux. Alors que le flux de sortie standard (cf. Figure 34) permet une écoute conditionnée par le contenu, l'écoute du flux d'erreur (cf. Figure 35) portera sur le contenant, à savoir le type de l'erreur levée.

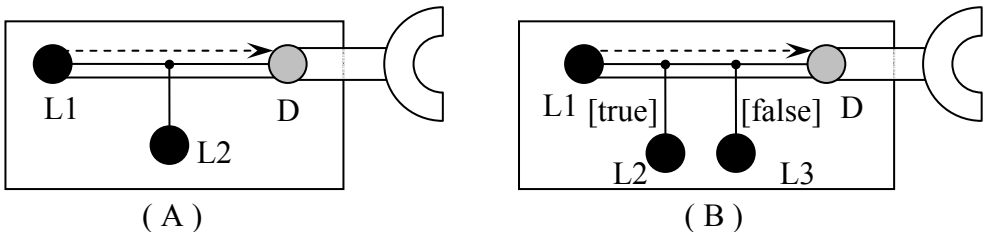


Figure 34 La connexion au flux de retour

En Figure 34-A, nous voyons qu'un point d'activité `L2` peut s'exécuter consécutivement à l'exécution d'un point d'activité distant `D1`, sans préciser la moindre condition. En revanche, la présence de deux points d'activité (Figure 34-B), pour un point `D1` retournant un booléen, demande la définition de gardes (identifiées par l'usage de crochets ouvrants et fermants) sur les liens reliant le flux de sortie standard de `D1` et les points d'activité `L2` et `L3`. Un exemple d'implémentation du schéma de la Figure 34-B peut être :

```
public T1>,T2> m1(I1<)
{
  L1(I1>◁Env_init)
  if(D1)
  {
    return m2()    // méthode dans laquelle est accessible L2
  }
  else
```

```
    {
      return C1.m3() // méthode dans laquelle est accessible L3
    }
}
private T1< m2()
{
  L2
  return T1<
}
private class C1
{
  private T2< m3()
  {
     L3
     return T2<
  }
}
```

En ce qui concerne la gestion du flux d'erreur, tous les langages de programmations objet (Java, C++, C#) qui seront à priori utilisé pour l'implémentation de nos composant, proposent un mécanisme de capture des exceptions. Dans la Figure 35, nous avons représenté un point d'activité capable de générer des erreurs du type "Inconnu" et "Paiement Refusé". Si ces deux erreurs s'avéraient totalement disjointes (sans relation hiérarchique entre elles), ces deux points peuvent s'enregistrer en parallèle sans ambiguïté (Figure 35-A). En revanche, si une relation hiérarchique existait entre nos deux exceptions (cf. Figure 35-B), la définition d'une relation d'héritage permet de définir une priorité envers l'erreur la plus fine (erreur "Inconnu" dans notre exemple), celle-ci devant être mentionnée en premier dans la construction de capture des exceptions.

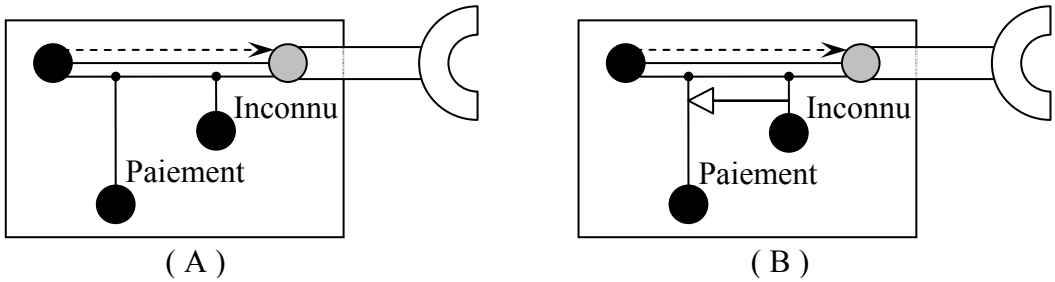


Figure 35 La connexion au flux d'erreur

Un exemple d'implémentation du schéma de la Figure 35-B peut être :

```
public T1> m1(I1<)
{
  L1(I1>◁Env_init)
  try
  {
      D1
      return T1<
  }
  catch(Inconnu i)
  {
    return m2()
  }
  catch(PaiementRefusé p)
  {
```

```
      return m3()
    }
  }
  private T1< m2()
  {
    L2
    return T1<
  }
  private T1< m3()
  {
    L3
    return T1<
  }
```

Si l'utilisation de points d'activité, que nous pourrons appeler secondaires, permet de simplifier le code du composant, elle ne doit pas nous faire oublier que c'est le point d'activité immédiatement associé à la facette qui centralise le code d'implémentation de la méthode, et donc le mécanisme de retour vers le client. Si nous complétons à cet effet le schéma de la Figure 34-B, nous obtenons un schéma tel que défini dans la Figure 36.

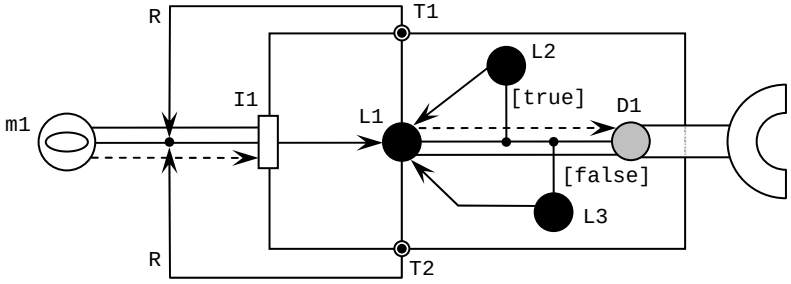


Figure 36 La connexion au point d'activité principal

Ce premier niveau de représentation du composant permet de visualiser les échanges entre points d'activité. Il permet de préciser que le thread retourne à `L1`, suite à l'achèvement des traitements `L2` ou `L3`, pour assumer le passage par les transitions `T1` ou `T2`. Cependant notre schéma demande l'utilisation du second niveau de représentation, pour préciser les interactions entre points d'activité (`L2` et `L3`) et ports de sortie du composant (`T1` et `T2`) au niveau du point d'activité `L1`. La Figure 37 se focalise ainsi sur `L1`, en relatant d'une part la relation entre le port d'entrée du composant `I1` et le point d'activité distant `D1`. D'autre part nous retrouvons la prise en charge du retour des points d'activité `L2` et `L3`, mis en relation avec les ports de sortie du composant. Ce second niveau de représentation nous apporte une grande finesse d'interconnexion entre points d'activité et ports du composant, pour modeler le comportement du composant, selon les actions particulières de ses points d'activité.

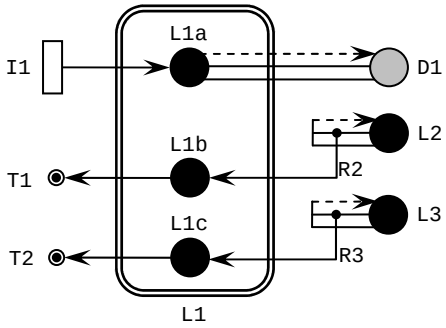
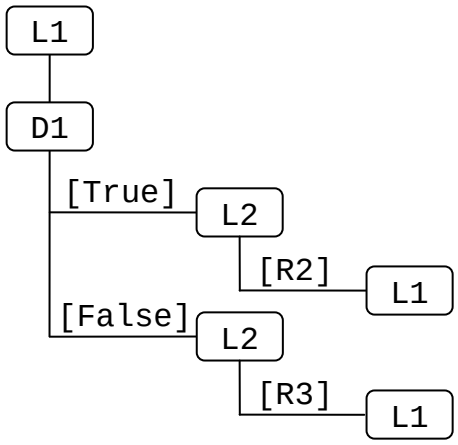


Figure 37 La prise en charge du retour des points d'activité secondaires

Compte tenu des deux diagrammes précédents, l'ensemble des expressions d'implémentation de la méthode m1 s'articule comme il suit :

```
<m1> = {T1oL1boL2[R2]oD1[true]oL1aoI1, T2oL1coL3[R3]oD1[false]oL1aoI1}
```

Le rapprochement de ces deux expressions mène à la représentation graphique ci-dessous, utilisée pour la génération d'un code proche de celui donné en page précédente.



Par l'étude d'un composant de service célèbre : la liste noire, nous allons montrer que notre formalisme offre une grande souplesse dans le degré de paramétrisation à apporter à un composant. Admettons qu'un composant dédié à un abonné (`Subscriber`) présente une facette où réside une méthode `check(User)`, ne retournant rien en cas de succès, et levant une exception `Unauthorized Error` en cas de problème, et ce quelle que soit son origine, de manière à faire échouer la demande d'exécution d'un service. Ce composant utilise une connexion à une base de données pour vérifier la présence de l'utilisateur dans la liste de personnes indésirables. Sur la base du comportement attendu, le flux de retour du réceptacle vers le contrôleur de base de données doit comporter trois points d'écoute :

- Un point regardant si l'utilisateur est dans la liste noire.
- Un point regardant si l'utilisateur n'est pas dans la liste noire.
- Un point regardant si une erreur est intervenue au moment d'entrer en relation avec le composant contrôleur de la base de données connecté au réceptacle.

La définition de ces trois points d'écoute effectuée, il nous reste à aborder le problème de la transmission du résultat de l'exécution du composant retourné à la facette. Pour comprendre la manière d'influer sur ce mécanisme, et ainsi accéder au paramétrage du composant, nous allons faire une courte incursion dans le futur contexte d'exécution du composant, donné en Figure 38. Ce schéma nous montre la répartition physique des différentes entités à la base du composant au sein d'un serveur d'application. Un conteneur de composants agira comme un agent du composant, en gérant ses différentes interfaces (facettes et réceptacles) pour faire appel au cœur du composant le moment venu. Si le composant est vue comme une boite noire, le conteneur à en revanche la possibilité d'intercepter le résultat de l'exécution du composant, pour éventuellement l'altérer avant de le présenter au composant client de la facette.

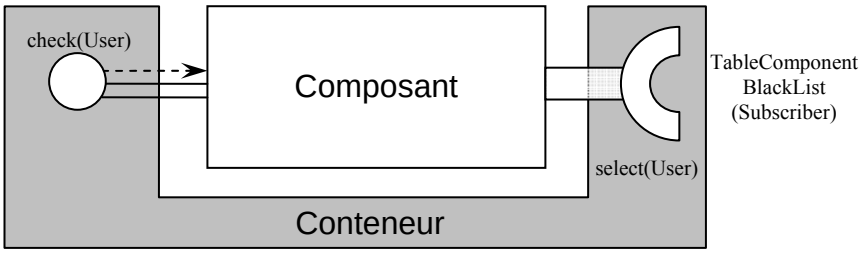


Figure 38 Le composant dans son environnement d'exécution

De ce fait, quand nous évoquons la paramétrisation au niveau de notre diagramme de composants, nous entendons par là la possibilité d'intervenir au niveau du conteneur du composant pour paramétrer le mécanisme de retour de la facette. Le formalisme que nous avons introduit ne permet pas de contrôler les points d'activité interne du composant depuis le conteneur, ce qui ne représente pas une réelle limitation dans la mesure où la propriété d'encapsulation du composant empêche ce genre d'opération. Cependant, en introduisant des ports terminaux du cœur du composant, notre formalisme permet d'effectuer la jointure entre le retour du cœur du composant et le retour de la facette du composant, comme première instruction du traitement à effectuer par le conteneur sur le retour de son composant.

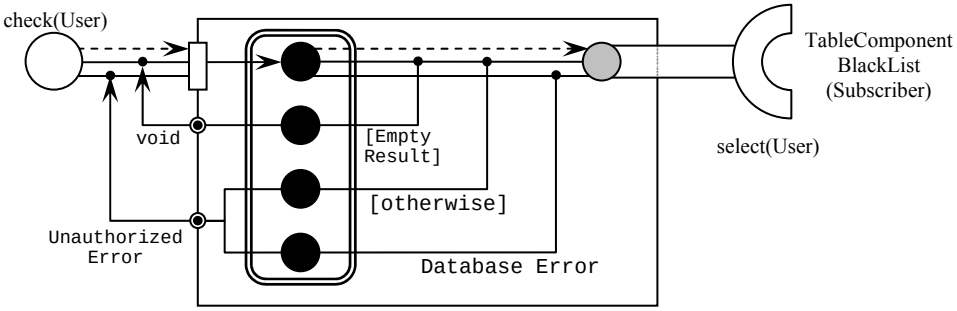


Figure 39 Le composant BlackList statique

Si nous revenons à notre exemple de composant `BlackList`, construire un composant ne permettant pas de personnaliser le retour du composant demande l'utilisation de points d'activité intermédiaires entre le flux de retour du réceptacle et les différents états terminaux (voir Figure 39). Dans cette configuration un point d'activité est abonné à un type particulier de retour, et relaye sa décision à un état terminal. Cette définition statique du composant `BlackList` permet d'aboutir à la définition d'un composant ayant un comportement invariable, à retour du réceptacle égal.

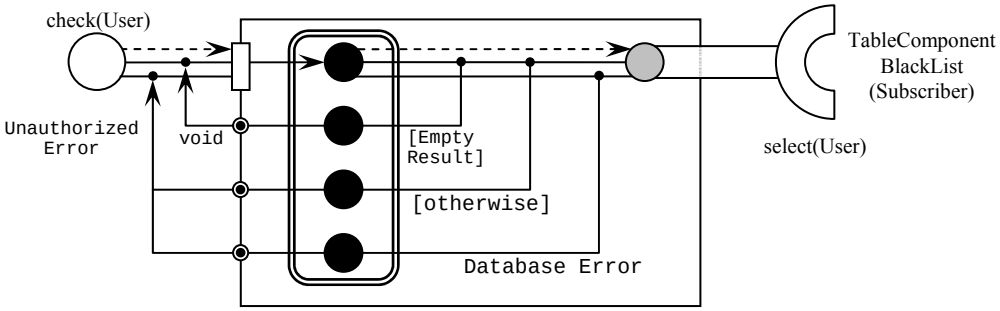


Figure 40 Le composant BlackList dynamique

Nous voyons en Figure 39, qu'en sortie du cœur du composant un processus extérieur, en l'occurrence le conteneur du composant, n'aura pas la possibilité de déterminer si l'utilisation du port terminal d'erreur est consécutive ou non à une "`Database Error`", les chemins d'activité ne gardant pas en mémoire leurs prédécesseurs. Ce type d'attente pourrait s'avérer si d'aventure le créateur du service voulait habiliter le conteneur à moduler les retours de la facette du composant. Par exemple, un fournisseur de services souhaiterait en cas de panne grave de sa base de données, « débrayer » son service de liste noire en précisant au niveau de son conteneur qu'une "`Database Error`" ne mène pas temporairement à une "`Unauthorized Error`". Permettre une telle paramétrisation du composant demande de lier directement les états terminaux, localisés sur la frontière du cœur du composant, aux retours du réceptacle (cf. Figure 40). Avec cette configuration le composant laisse la liberté au conteneur d'assurer la jointure entre les ports terminaux du cœur du composant (liés aux retours du réceptacle) et les retours attendus en sortie de la facette.

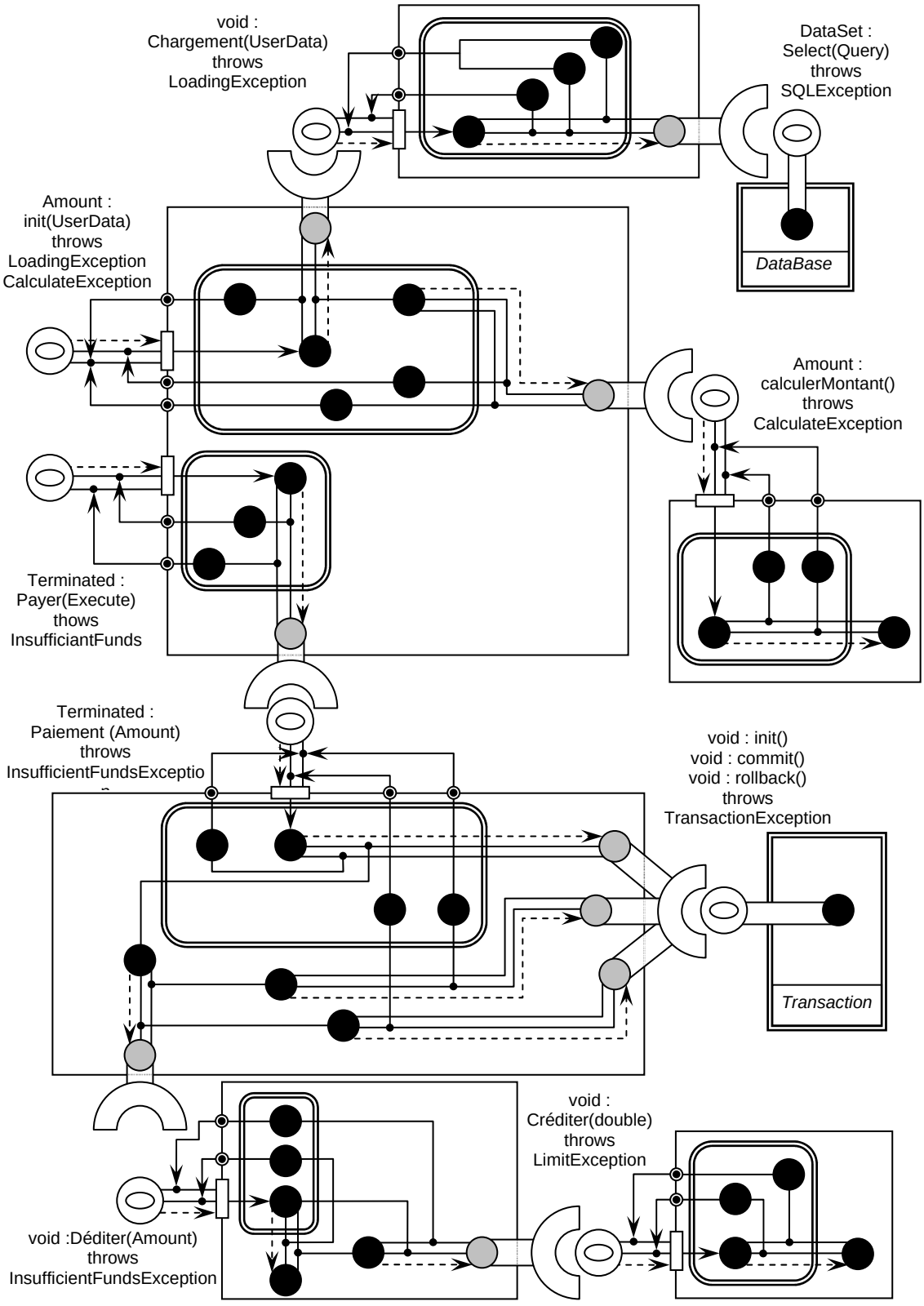


Figure 41 Le diagramme de composants associé à notre service de Paiement

Le schéma ci-dessus, dont l'interprétation sera laissée à la discrétion du lecteur, reprend le diagramme de composants de la Figure 28, enrichi au moyen de notre formalisme de description de l'activité des composants. Par commodité nous avons reproduit à l'intérieur

d'un même diagramme les deux niveaux de représentation d'un composant. Ce diagramme permet de conclure la phase d'analyse du service à créer, avec une vue exposant relativement clairement les interactions et traitements locaux des composants, tout en offrant l'avantage de générer un code de base du service, à partir duquel pourra commencer la phase de développement.

3.5 Le développement du service

3.5.1 Présentation

Si la phase d'analyse a permis d'appréhender le service dans son ensemble, sa construction va se dérouler en plusieurs étapes, conformément à la structure en oignon représentée en Figure 1, le développement se limitant à la mise en place des briques de base, que constituent les composants. Ces composants seront ensuite enrichis de capacités d'interaction au cours des phases ultérieures de conditionnement et de déploiement.

Outre la génération de code que permet déjà le diagramme de classe UML, notre diagramme de composant nous apporte une information précieuse quant au fil d'exécution des méthodes, mettant en évidence à la fois les articulations logiques dans le code du composant à générer et les dépendances entre facettes et réceptacles. Cette double information véhiculée par notre diagramme de composant va occasionner deux types d'opérations durant le développement du composant. D'une part la génération du code incarnant le composant, en s'appuyant sur un ou plusieurs langage(s) de programmation. D'autre part, la production de descripteurs du composant, explicitant les services offerts, les possibilités de personnalisation et les dépendances du composant.

Pour faciliter la gestion des services communs, et fournir un environnement modulaire d'exécution à nos composants, nous avons adopté un modèle de programmation basé sur la notion de conteneur, perçu au cours du développement comme une entité abstraite, utilisée pour unifier les ressources accessibles aux composants. Un conteneur permet par exemple de déléguer la gestion de la persistance des données d'un composant ou l'authentification lors de l'accès. L'utilisation d'un conteneur permet d'épurer le code des composants d'une multitude de traitements récurrents sans grande valeur fonctionnelle, qui seront ajoutés de façon transparente au moment de la compilation, pour faire du couple composant/conteneur une entité indissociable.

3.5.2 La génération du code

La première phase de développement du composant utilise le diagramme de composant pour générer automatiquement le squelette du code du service en cours de création. Le processus de génération automatique du code va être appliqué successivement à toutes les facettes du composant, pour traiter l'ensemble des fils d'exécution contenus dans le diagramme de composant.

Définir une architecture basée sur la notion de conteneur nous demande d'établir une certaine généricité entre nos composants, afin de les doter de propriétés communes, maintenues par le conteneur. En effet, comme dans les architectures à base d'EJB, l'instanciation, l'activation, la mise en cache de l'état d'un composant passent par l'utilisation de classes génériques dont devront hériter nos composants. Notre travail n'ayant pas vocation à définir un tel système de classes particulier demandant l'élaboration d'un véritable environnement d'exécution de nos

composants, nous nous contenterons des dénominations `ComponentInterface`, `ComponentImpl` et `ComponentStub` comme superclasses respectivement de l'interface du composant, sa classe d'implémentation et le stub utilisé par nos réceptacles pour accéder aux facettes distantes. Pour chacun de nos composants nous aurons ainsi une interface de la forme :

```
package fr.enst.autoroute;
public interface IPaiementAutoroute extends ComponentInterface { }
```

Une classe implémentant cette interface :

```
package fr.enst.autoroute;
public class PaiementAutoroute extends ComponentImpl { }
```

Et une classe souche d'accès à une facette distante :

```
package fr.enst.autoroute;
public class CalculerMontant extends ComponentStub { }
```

Si le chemin d'activité d'un composant emprunte un point d'activité distant (à la base d'un réceptacle), la génération de code, en s'inspirant du processus de développement basé WebServices, demande une phase d'importation des caractéristiques de l'interface distante. Cette opération, réalisable sur des composants déjà déployés au sein d'un serveur d'application, passe par le chargement d'un document WSDL [CCM01] construit de toute pièce par introspection du composant serveur, et accessible dans un contexte Web maintenu par le serveur d'application, dont un extrait est donné ci-dessous (cf. Annexe 1 pour le document complet),.

```
<?xml version="1.0"?>
<definitions name="Component1"
             targetNamespace="http://www.enst.fr/component1.wsdl"
             xmlns:tns="http://www.enst.fr/component1.wsdl"
             xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <schema targetNamespace="http://www.enst.fr/component1.xsd"
            xmlns="http://www.w3.org/2000/10/XMLSchema">
      <element name="in">
        <complexType>
          <element name="p1" type="Type_1"/>
          ...
          <element name="pm" type="Type_m"/>
        </complexType>
      </element>
      <element name="out">
        <complexType>
          <element name="result" type="ResultType"/>
        </complexType>
      </element>
      <element name="error">
        <complexType>
          <restriction base="Exception">
            <enumeration value="ExceptionType1"/>
            <enumeration value="ExceptionType2"/>
          </restriction>
        </complexType>
      </element>
    </schema>
  </types>
```

```
  <message name="m1SIPExecuteIn">
    <part name="body" element="tns:in" />
  </message>
  <message name="m1SIPExecuteOut">
    <part name="body" element="tns:out" />
  </message>
  <message name="m1SIPExecuteError">
    <part name="body" element="tns:error" />
  </message>
  <portType name="Facet1SIPExecute">
    <operation name="m1">
      <input message="tns:m1SIPExecuteIn" />
      <output message="tns:m1SIPExecuteOut" />
      <error message="tns:m1SIPExecuteError" />
    </operation>
  </portType>
</definitions>
```

Figure 42 Le document WSDL de description d'une facette

Le document WSDL liste l'ensemble des ports du composant, un port étant caractérisé par un couple (facette, protocole d'accès). Nous avons représenté ici le port `Facet1SIPExecute` associé au protocole SIP, lié à la facette Facet1 hébergeant une unique méthode `m1` matérialisée par le tag `<operation>`. Nous avons réutilisé la possibilité offerte par XML Schéma d'étendre la sémantique d'un port, en introduisant le tag `<error>`. Le document WSDL liste ainsi les types d'entrée (`tns:in`) et de sortie (`tns:out`) des méthodes présentes sur la facette, et les exceptions susceptibles d'être levées (`tns:error`). Il est à noter que les éléments de base des types complexes sont par nature de type élémentaire (String, Long, Boolean,…). Aussi présenter des méthode utilisant des types dérivés ou des exceptions personnalisées suppose que notre architecture contienne un référentiel de types communs, dans lequel chaque fournisseur pourra puiser les informations nécessaires à l'interprétation des signatures des méthodes présentes sur les facettes. Outre la possibilité de tester le composant serveur à chaud, la prise en charge de ce document permet la création des classes utiles à la construction des requêtes du coté client, à l'image du contrat IDL dans Corba. Comme dans les architectures réparties du type Corba/RMI ou dotNet Remoting, notre client devra passer par une classe projetée du document WSDL pour initier ses requêtes. Ces classes seront ensuite aidées de stub, chargés de la construction, l'envoi et la réception du retour de l'appel à la méthode distante.

3.5.3 Facettes et réceptacles

Utiliser des composants nous permet de distinguer le code d'implémentation, par définition statique, des considérations de facette et de réceptacle, capables de s'adapter aux contraintes d'interactions entre composants, dépendantes de l'environnement d'exécution offert par le serveur d'application. Le document WSDL généré à partir de l'interface de programmation du composant nous a permis de prendre en charge la partie typée statique de l'interaction de service. En revanche, la seule déclaration des ports et types attendues sur l'interface du composant distant, ne suffiront pas à appréhender les problématiques de composition, d'interaction ou de personnalisation de service. Aussi comme extension du fichier WSDL du composant, nous introduisons en XML schéma un document de description de nos facettes et réceptacles techniques, par opposition aux facettes et réceptacles métier manipulés au moment de l'intégration du composant. Ce document commence par un encart déclaratif

référençant le document WSDL du composant auquel les facettes et réceptacles se rapportent ainsi que les documents WSDL des composants utilisés pour le typage des réceptacles.

```
<tech-recfac name="MyComponent"
             targetNamespace="http://www.enst.fr/myComponent.xsd"
             xmlns:ref1="http://www.enst.fr/component1.wsdl"
             xmlns:myComp="http://www.enst.fr/myComponent.wsdl"
             xmlns="http://schemas.xmlsoap.org/wsdl/">
```

Le document se poursuit par l'énumération des réceptacles et facettes externalisés par le composant. Chaque réceptacle est caractérisé par un ensemble de méthodes distantes, en précisant l'identifiant d'une opération (dans la dénomination WSDL, une opération correspond à une méthode sur un port du composant) importée des références WSDL en entête de notre document. La déclaration des facettes collecte ensuite des méthodes de l'interface de programmation réutilisées pour la conception d'interfaces techniques. Pour chaque méthode présente sur nos facettes nous précisons ses paramètres de personnalisation (tag `<param>`) et ses dépendances en terme de réceptacle. La définition de paramètres de personnalisation permettra aux futurs interfaces métier du composant d'intervenir sur le cours de l'exécution du code du composant, ce dernier allant collecter dans son contexte d'exécution la valeur du paramètre à considérer. Un service basé sur un algorithme de compression audio, réglable selon un taux de compression, pourra ainsi externaliser ce paramètre pour construire des interfaces métiers où sa valeur sera prédéfinie. Enfin, préciser les réceptacles dont dépendent les méthodes d'une facette anticipe les questions de composition de service au niveau métier.

```
<?xml version="1.0"?>
<tech-recfac name="MyComponent"
             targetNamespace="http://www.enst.fr/myComponent.tec"
             xmlns:ref1="http://www.enst.fr/component1.wsdl"
             xmlns:myComp="http://www.enst.fr/myComponent.wsdl"
             xmlns="http://schemas.xmlsoap.org/wsdl/">
  <receptacles>
    <receptacle receptacle-id="rec1">
      <method operation-name="ref1:m1"/>
    </receptacle>
  </receptacles>
  <facets>
    <facet facet-id="fac1">
      <method operation-name="myComp:myMethod">
        <receptacle receptacle-id="rec1"/>
        <param type=”String” name=”custParam”/>
      </method>
    </facet>
  </facets>
</tech-recfac>
```

Figure 43 La définition des facettes et réceptacles techniques

Basé sur l'interface de programmation du composant, notre schéma XML définit une série de facettes construites à partir d'un sous ensemble de cette interface. L'objectif des étapes de conditionnement et de déploiement du composant sera alors de construire de réelles interfaces métiers à partir de ces facettes essentiellement techniques. Dans une optique de séparation des métiers de développement et de conditionnement des composants, utiliser des facettes techniques permet à un opérateur de conditionnement d'altérer la topologie des interfaces du composant, sans reconstruction de son code interne.

3.5.4 Le broker

La question du référencement des composants revêt une dimension toute particulière dans notre architecture où des services peuvent combiner des composants issus de plusieurs intervenants métier. Comme nous l'avons vu dans la partie 3.5.3, la question du réceptacle au niveau de l'implémentation du composant se limite au type de la facette attendue pour l'accomplissement du service. Cependant pour comprendre le fonctionnement de l'interaction entre composants nous allons nous placer plus en avant dans la phase d'intégration du composant, pour comprendre plus en détail notre mécanisme de mise en relation des facettes et réceptacles.
A l'image des architectures réparties que sont Corba/RMI ou .NET Remoting, un composant client (relié à un réceptacle) va devoir passer par un stub pour entrer en relation avec le squelette d'un composant serveur (offrant la facette). Cependant dans notre architecture la construction de ce stub n'est pas assujettie à un fichier IDL statique, mais prise en charge par un broker, pour tenir compte des relations entre facette et réceptacle. Un broker pourra par exemple présenter un stub au client qui exécutera en parallèle la requête sur plusieurs facettes de plusieurs composants. L'utilisation de ce stub demande au composant client d'utiliser le code suivant :

```
Broker broker = this.getContext().getInitialReferences("Broker");
Object ref = broker.resolve("receptacle1");
Facet1 fac = Facet1Helper.narrow(ref);
fac.m1(p1,...,pm);

// fac correspond au stub construit par le broker.
// m1() est appelée localement sur le stub puis relayée au
composant serveur.
```

La récupération d'une référence vers le broker se fait en passant par le contexte d'exécution du composant, obtenu par la méthode `getContext()`, héritée de la superclasse `ComponentImpl`. Sur le modèle d'un service de nommage, le composant peut utiliser cette instance de broker pour importer la référence du stub maintenu par le broker, utilisé pour accéder à la ou les facette(s) connectée(s) à son réceptacle. Le composant utilise enfin la classe Helper associée à la facette, pour typer le stub avant l'appel à la méthode distante.

Au déploiement d'un composant sur un serveur d'application, un opérateur va relier chaque réceptacle du composant à au moins une facette compatible avec le type attendu par le composant, puis cette relation sera référencée par le broker, pour permettre au composant client de s'approprier les facettes identifiées par l'opérateur. Ainsi les informations maintenues par le broker dans le cadre de la relation réceptacle /facette sont :

- Le service de nommage ou annuaire à partir duquel il est possible de localiser le composant.
- Le nom sous lequel le composant peut être localisé au sein du service de nommage.
- Le nom de la facette utilisée.

Par exemple :

```
ns       = infres.enst.fr:3075/NameService
ns-name  = infres.services.WhiteBoard
facet    = MainFacet
```

Ceci permet à un broker de retourner à un composant client la référence d'un stub associé à une facette connectée à son réceptacle. Dans l'exemple ci-dessus (inspiré de CORBA), le broker va contacter le service locator présent sur le serveur `infres.enst.fr`, joignable à

travers le port `3075`, pour atteindre le service de nommage, référencé sous le nom de processus corba "`NameService`". Le broker va devoir ensuite récupérer une référence à la facette "`MainFacet`" sur le composant indexé dans ce service de nommage sous le nom `WhiteBoard`, à l'intérieur du contexte de nommage `infres.services`.

Cependant ces informations ne permettent que la construction d'un lien statique entre classes de composants, une relation entre instances demandant la définition de contraintes visant à préciser la facette d'une instance de composant à connecter au réceptacle d'une autre instance de composant. De fait la relation entre réceptacle et facette doit être enrichie par des paramètres caractérisant les instances de composants serveur, utilisés par le broker pour localiser une instance particulière de composant. Ces paramètres doivent tenir compte des moyens à disposition du broker pour une telle opération, ici en l'occurrence les opérations offertes par la home du composant. Nos contraintes seront donc fonction des paramètres que peuvent accepter les méthodes create() et find() des composants serveur.

Comme première étape de l'enregistrement d'une relation entre composants, l'intégration d'un service produira un fichier XML comme ci-dessous, manipulé exclusivement par le broker.

```
<relation relation-id="relation1">
  <receptacle name="Users"
              ns="infres.enst.fr:3075/NameService"
              ns-name="infres.services.WhiteBoard"/>
  <facet facet-id="infres"
         ns="infres.enst.fr:3075/NameService"
         ns-name="infres.sua.WhiteBoard"
         name="MainFacet">
    <find>
      <find-arg name="login"
                value="inviteUser.login"
    </find>
  </facet>
</relation>
```

Figure 44 La définition d'une relation entre classes de composants

Le tag <find> précise au broker le critère de rapprochement ente instances de composant client et instances de composants serveur, avec l'utilisation de la méthode findByLogin() sur sa home, munie du champ login reçu par le composant client sur sa méthode inviteUser(). Notons qu'avec ce mécanisme la méthode initialement appelée sur le composant client est vue comme une structure, à partir de laquelle il est possible de collecter des données pour la construction des contraintes. En outre cette manière de procéder implique deux hypothèses sur la relation broker / composant client. D'une part la définition des réceptacles du composant client doit préciser les méthodes dépendant de la satisfaction de ces réceptacles, pour le contrôle de cohérence des contraintes. D'autre part le broker doit pouvoir accéder à l'environnement d'exécution du composant client (donc être situé au sein du même serveur d'application) pour être capable de prélever les données manipulées par le composant. En effet pour être pleinement réutilisable, notre architecture facette/réceptacle ne doit pas demander au composant client de préciser explicitement dans son code ses contraintes d'interaction, c'est le broker qui doit assumer cette tache de manière autonome.

La structure `<relation>` qui reste encore globale au composant ne permet pas de lier concrètement deux instances de composants, car cette opération doit être réalisée au cours de

la vie du composant. C'est dans cette optique que nous avons conçu sur nos composants une facette spécifique, où l'on trouve les méthodes inviteUser(), inviteSession, byeUser() et byeSession(). Ces méthodes invoquées par le serveur SIP à réception d'un message INVITE ou BYE, permettent d'ajouter ou retirer un composant utilisateur à une session de service, ou de faire collaborer deux composants session. Le code de ces méthodes contiendra ainsi les instructions :

```
broker.addConstraint(this.Users, "najm@enst.fr");
broker.removeConstraint(this.Users, "najm@enst.fr");
```

Le broker maintiendra ainsi les contraintes des relations facette/réceptacle en utilisant le format suivant :

```
<constraint relation-id="relation1">
  <receptacle instance="WhiteBoard@123456789"/>
  <facet facet-id="infres">
    <find>
      <find-arg name="login"
                value="najm@enst.fr"/>
    </find>
  </facet>
</constraint>
```

Figure 45 La spécialisation d'une relation entre instances de composants

Une fois les facettes connectées au réceptacle, le broker doit fournir notre fameux stub au composant client pour la construction des appels distribués. Le broker peut pour ce faire préparer un ensemble de stubs correspondant aux différentes configurations de facettes connectées au réceptacle, en fournissant une méthode `resolve()` évoluée, permettant au composant client de choisir dynamiquement le stub à utiliser. A cet effet nous avons prévu l'utilisation de trois types de méthodes `resolve` :

- `resolve(nom_receptacle)`
- `resolve(nom_receptacle, etiquette_facette)`
- `resolve(nom_receptacle, type_filtre, filtre)`

La première méthode se contente de retourner au composant client une référence à un stub capable de distribuer sa requête locale sur l'ensemble des facettes connectées au réceptacle. La seconde requête limite le stub à une unique facette en utilisant une étiquette, comme cela peut être le cas dans un receptacle multiplex. La dernière méthode permet de préciser des filtres qualifiant ou disqualifiant, pour par exemple exclure un composant lors de la distribution de l'appel.

```
static final int QUALIFY=0 ;
static final int DISQUALIFY=1 ;

Object ref = broker.resolve("Users", QUALIFY, "najm@enst.fr");
Facet1 fac = Facet1Helper.narrow(ref);
fac.m1(p1,...,pm);
```

3.5.5 Les composants Session / Utilisateur

3.5.5.1 Un besoin de spécialisation

Une étude d'un certain nombre de services télécom désormais couramment utilisés (messagerie instantanée, tableau blanc, vidéoconférence, …) nous a permis de mettre en évidence une design pattern récurrente, nécessitant la distinction de deux types de composants. D'un coté nous avons les composants dits de session, assurant la partie métier du service et indépendants de toute donnée utilisateur. De l'autre nous avons les composants utilisateur, chargés de centraliser les données propres à un utilisateur dans le contexte d'un service.

3.5.5.2 Le composant Session

Un composant session matérialise la partie métier d'un service, l'existence de ses instances étant limitées à la durée de vie de la session de service des utilisateurs participants. Il est directement accessible à un serveur SIP (le protocole de signalisation que nous utilisation au niveau de nos services), pour répondre à une demande d'exécution de service. Si un composant session est accédé par le serveur SIP au moment du démarrage de la session de service et à chaque modification de la configuration de cette session, il n'est en revanche accédé que par l'intermédiaire de composants utilisateur pendant le déroulement de la session de service.

Dans ce contexte un composant session est équipé de trois catégories de facettes et de deux catégories de réceptacles :

- Une facette privée et unique permettant à la home du composant (la factory utilisée par le serveur SIP pour accéder aux services) de gérer le cycle de vie des instances du composant, à travers des opérations du type create(), find(sessionId), remove(sessionId).
- Une facette unique utilisée par le serveur SIP à réception de messages SIP INVITE ou BYE, pour permettre à un utilisateur de se joindre ou de quitter la session de service, en connectant ou déconnectant son composant utilisateur au composant session, via les méthodes inviteUser(UserId) et byeUser(UserId). Cette facette est également utilisée pour modifier les propriétés fonctionnelles de la session (nombre max d'utilisateurs, QOS, …) grâce à une méthode prenant en argument le corps du message SIP INVITE réceptionné par le serveur SIP.
- Un ensemble de facettes aussi bien utilisées par des composants utilisateur que par d'autres composants session, sur lesquelles seront déployées les méthodes métier du composant. Une même méthode peut apparaître sur plusieurs facettes, compte tenu des angles de vues définis pour le composant session.
- Un ensemble de réceptacles vers des composants session tiers, nécessaires pour l'accomplissement des méthodes supportées par le composant.
- Un ensemble de réceptacles à destination de composants utilisateurs, permettant au composant session de jouer le rôle de medium de communication entre composants utilisateur.

3.5.5.3 Le composant Utilisateur

Un composant utilisateur a une existence qui survit à la session de service. Dédié à un service, il joue le rôle d'intermédiaire entre les applications déployées sur le terminal de l'utilisateur et les services offerts par le réseau. Créé lors de la souscription de l'utilisateur au service, son état est réexaminé lors de chacune de ses utilisations au sein d'une session de service, pour s'adapter au contexte de l'utilisateur. Un composant utilisateur peut être également équipé de trois catégories de facettes et de deux catégories de réceptacles :

- Une facette privée et unique permettant à la home du composant de gérer le cycle de vie des instances du composant, à travers des opérations du type create(), find(userId), remove(userId).
- Un ensemble de facettes utilisées par le composant session. Les méthodes présentes sur ce type de facette permettent d'appliquer à l'utilisateur des actions effectuées au niveau du service.
- Un ensemble de facettes utilisées par les applications de l'utilisateur. A l'opposé ces facettes permettent aux applications du terminal d'entrer en relation avec le service.
- Un ensemble de réceptacles utilisés par le composant utilisateur pour appliquer aux applications du terminal les traitements demandés par le composant session.
- Un ensemble de réceptacles utilisés par le composant utilisateur pour remonter au service les actions demandées au niveau des applications terminal client.

3.5.5.4 Un exemple : le tableau blanc

Comme exemple d'utilisation de nos composants Session et Utilisateur, nous allons nous intéresser à un service de tableau blanc, que nous étudierons plus en avant dans la suite de ce manuscrit. Pour le moment nous pouvons voir le service comme un assemblage de trois composants : un composant Session et deux composants Utilisateur. A l'état initial du système, nous faisons l'hypothèse que les futurs utilisateurs mobilisés par la session de services ont déjà souscrit au service de tableau blanc, et par conséquent qu'ils disposent chacun d'une instance de composant `InputWhiteBoard` et `OutputWhiteBoard` dans le domaine de leur Détaillant propre. Admettons que l'utilisateur afontaine@wanadoo.fr souhaite entrer en relation, via notre service de tableau blanc, avec l'utilisateur najm@enst.fr. Il commencera par faire appel à la méthode createSession() sur le composant WhiteBoard, afin d'initier une nouvelle session de service. Notre utilisateur appellera ensuite la méthode inviteUser(afontaine@wanadoo.fr) pour se rattacher à cette session, avant d'inviter l'utilisateur najm@enst.fr à faire de même. Si l'utilisateur invité accepte la demande de notre utilisateur appelant, il passera lui aussi par la méthode inviteUser(najm@enst.fr) pour attacher son composant utilisateur à la session.

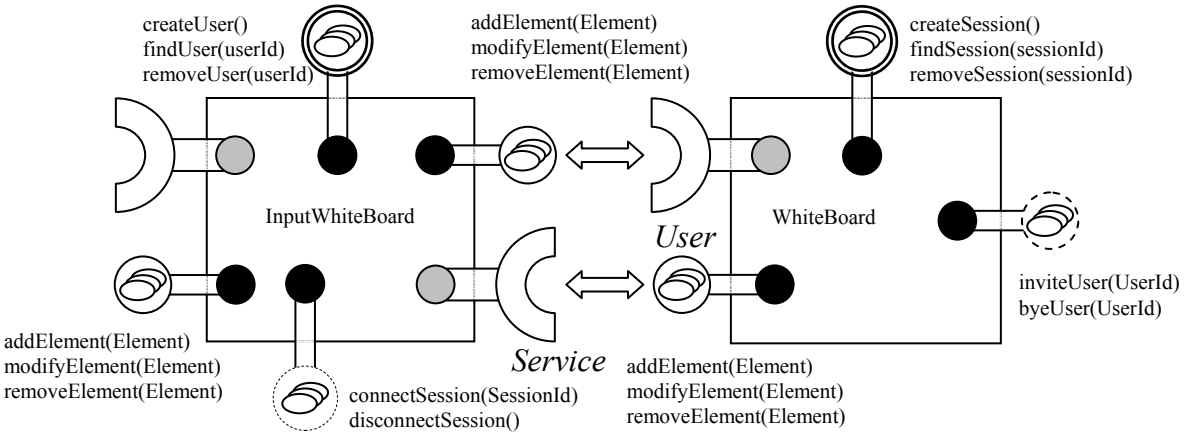


Figure 46 Les composants associés au service "WhiteBoard"

A ce moment précis de la session de service, arrêtons nous sur l'état de la relation entre les réceptacles et facettes de nos composants. Avant même l'instanciation du composant session, le réceptacle "Users" est référencé par notre broker avec la configuration ci-dessous :

```
<relation relation-id="relation1">
  <receptacle name="Users"
              ns="infres.enst.fr:3075/NameService"
              ns-name="infres.services.WhiteBoard"/>
  <facet facet-id="infres"
         ns="infres.enst.fr:3075/NameService"
         ns-name="infres.sua.WhiteBoard"
         name="MainFacet">
    <find>
      <find-arg name="login" value="inviteUser.login"
    </find>
  </facet>
  <facet facet-id="wanadoo"
         ns="frontal.wanadoo.fr:3075/NameService"
         ns-name="wanadoo.sua.WhiteBoard"
         name="MainFacet">
    <find>
      <find-arg name="login" value="inviteUser.login"
    </find>
  </facet>
</relation>
```

D'après cette déclaration, notre composant session est capable d'interagir avec deux types de composants Utilisateur, appartenant à deux Détaillants différents. Fort de cette configuration, à chaque appel de sa méthode `inviteUser(userId@domain)`, le composant session entre en relation avec le broker pour ajouter une référence vers une instance de composant Utilisateur à son réceptacle `"Users"`. A ce moment le broker va commencer par entrer en contact avec le service de nommage du domaine de l'utilisateur, pour récupérer une référence vers la home du composant utilisateur, et localiser l'instance demandée en utilisant la méthode find(userId) conformément au tag <find> présent dans la configuration. Sur cette référence il va ensuite appeler la méthode `connectSession(sessionId)` pour connecter le réceptacle du composant Utilisateur au composant session. Le composant utilisateur entrera alors en relation avec son propre broker, en se limitant cette fois à la mise à jour de l'état de son réceptacle. Enfin le broker du composant Session ajoute une contrainte permettant de matérialiser la liaison entre le composant le réceptacle "Users" et le composant Utilisateur.

Après l'appel de la méthode `(userId@domain)` par chacun des utilisateurs, les contraintes associées au réceptacle "Users" seront de la forme :

```
<constraint relation-id="relation1">
  <receptacle instance="WhiteBoard@123456789"/>
  <facet facet-id="infres">
    <find>
      <find-arg name="login"
                value="najm@enst.fr"/>
    </find>
  </facet>
  <facet facet-id="wanadoo">
    <find>
      <find-arg name="login"
                value="afontaine@wanadoo.fr"/>
    </find>
  </facet>
</constraint>
```

Les contraintes arrêtées, les composants sont connectés entre eux, et prêts à transmettre les informations liées à l'exécution du service. Pour synchroniser l'ajout d'un éléments aux tableaux blancs des utilisateurs membres de la session, un utilisateur passe par les méthodes add/modify/removeElement disponibles sur son composant utilisateur. Le composant utilisateur invoque alors la méthode `resolve("Service")` sur le broker, afin de récupérer le stub lui permettant d'entrer en relation avec le composant connecté à son réceptacle.

Pour dispatcher l'ajout de l'élément à l'ensemble des utilisateurs, le composant session va devoir sélectionner le stub pertinent à chaque appel, parmi les combinaisons possibles de composants utilisateurs, en excluant l'utilisateur à l'origine de l'opération sur la tableau blanc, pour ne pas lui notifier une modification qu'il a lui même apporté. Pour ce faire le composant session va récupérer une référence vers son stub en appelant la méthode `resolve("Users", disqualify, "login=afontaine@wanadoo.fr")`.

Enfin, au moment de se retirer de la session de service, un utilisateur va appeler la méthode byeUser(userId) sur le composant session, pour ensuite retirer la référence de son composant utilisateur du réceptacle Users.

3.6 Conclusion

En introduisant un nouveau type de diagramme, nous avons appréhendé la production du code générique des composants, à l'origine des futurs services. Contrairement aux approches EJB et CCM, qui reposent sur des composants aux interfaces statiques sous l'hégémonie d'un simple contrat IDL, les composants ici définis sont constitués d'un cœur programmable entouré de facettes et réceptacles dynamiques (dont l'état dépend des connexions effectives entre composants) maintenus par un broker.

4 L'intégration du service

4.1 Introduction

Si notre étude de l'analyse d'un composant nous a permis de produire un composant muni de facettes et réceptacles, ce composant n'a néanmoins pas encore de capacités réellement nouvelles de communiquer avec son environnement. Or l'intégration du service est précisément l'étape qui permet de spécifier les termes des échanges entre composants. Pour formaliser et intégrer un certain nombre de traitements connexes mais encore hors du domaine des composants (comme la facturation ou la gestion de la qualité de service), nous avons introduit le concept totalement novateur d'interfaces métier puis de collaboration.

Une interface métier (facette ou réceptacle) s'appuie sur une interface technique définie lors de la construction du composant, pour préciser un certain nombre de contraintes pouvant être garanties pendant l'exécution du composant. En correspondance à ces contraintes, les facettes et réceptacles métier introduisent le principe de contrepartie, comme un moyen d'appliquer par exemple un service de facturation, suite au respect ou non des contraintes exprimées.

Comme spécialisation des interfaces métier qui restent générales, notre architecture introduit les interfaces de collaboration, qui permettent de préciser des valeurs à appliquer aux contraintes et contreparties des interfaces métier, dans le cadre d'une collaboration entre deux composants bien identifiés.

4.2 Présentation

Conformément à notre schéma de la Figure 8, la phase d'intégration du service passe par un premier stade de conditionnement puis un second de déploiement. L'objectif de cette phase est d'une part de doter le service et plus particulièrement ses composants, des informations nécessaires à son exploitation par le serveur d'application, et d'autre part de préciser les aspects statiques d'interaction de services, comme la gestion de la facturation ou de la qualité de service.

Les phases d'analyse et de développement ont convergé vers la définition de composants techniques, en se focalisant sur la partie interne des composants, qui ne pourra subir aucune modification dans la suite du processus de création du service. Cette définition interne des composant a été néanmoins accompagnée par l'écriture de facettes et réceptacles techniques, comme autant de points de contacts entre le composant et le monde extérieur. Si au sortir de la phase de développement, nos composants ne considèrent pas encore les questions d'utilisation de services communs comme la gestion de la facturation, de l'authentification ou de la qualité de service, ces facettes et réceptacles techniques vont nous servir de support pour la promotion de nos composants au rang de composant métier.

4.3 Les composants métier

Dans un environnement multi fournisseur, les services doivent être capable de s'accorder sur l'utilisation de services commun, ne serait-ce que pour assurer une facturation cohérente, telle que pratiquée dans les architectures actuelles de e-procurement. Dans ce système collaboratif, les services peuvent facturer aux composants client l'exécution de chaque méthode, compte tenu du niveau de qualité de service garanti par le serveur et attendu par le client. En généralisant ce principe, notre architecture doit permettre à nos composants métier de définir :

- Des facettes métier proposant des méthodes à contraintes garanties, et contrepartie demandée.
- Des réceptacles métier nécessitant des méthodes à contraintes demandées, et contrepartie garantie.

Pour étayer notre propos, prenons l'exemple suivant :

Soit R un composant muni d'un réceptacle. R nécessite une facette proposant la méthode m(), capable de lui garantir un temps d'exécution inférieur à T ms. Pour ce service, le Détaillant propriétaire de R est disposé à payer le montant p, qu'il refacturera au client de son composant. Si la contrainte de qualité de service était violée, le Détaillant de R demandera une pénalité égale à 10*p au Détaillant du composant serveur.

Soit F un composant muni d'une facette hébergeant une méthode m() compatible avec la méthode attendue par R. Le Détaillant de F garanti à 99,2% un temps de réponse en x ms, en contrepartie duquel il exige un montant f(x), avec une rétrocession de C*f(x) en cas de non respect de la garantie.

Pour aboutir à une relation entre nos deux composants R et F, nos fournisseurs de service vont devoir passer par les étapes suivantes :

- Le fournisseur de F construit sa facette métier en se basant sur une facette technique du composant. Cette facette métier défini une contrainte de temps **générale**, une contrepartie, et une relation entre les deux pour l'utilisation de m(), en prévoyant un dédommagement en cas de violation de la contrainte.
- Le fournisseur de R construit son réceptacle métier, à partir du réceptacle technique du composant. Ce réceptacle métier défini une contrainte **générale**, une contrepartie et une relation entre les deux pour l'utilisation de la méthode m(), en prévoyant la clause de non respect de la garantie.

Les facettes et réceptacles métier définis, les fournisseurs de F et R tentent de trouver un accord entre contraintes et contreparties pour associer leurs *composants métier* respectifs. Comme fruit de leur accord, les facettes et réceptacles métier **généraux**, seront projetés en une *facette et un réceptacle de collaboration*, **spécifique** à la relation entre les deux composants.

- Le fournisseur de F va construire une facette pour son composant F, garantissant une exécution de m() en 100 ms, pour une facturation de 3m€, et une couverture de 30 m€ en cas de dépassement.
- Le fournisseur de R va construire son réceptacle pour son composant R, demandant un temps de réponse inférieur à 100 ms, pour un coût de 3ms, et un dédommagement de 30m€.

Pour la résolution de leur contrainte et contrepartie, nos fournisseurs peuvent faire appel à un Fournisseur de Services Tiers indépendant, qui centralisera le Timer afin de prévenir toute contestation entre les intervenants, et un Gestionnaire de Facturation apportant le service de facturation nécessaire.

Nous allons dans la suite de cette partie passer en revue les mécanismes utilisés au niveau du serveur d'application permettant la mise en place de ce comportement.

4.4 Les facettes et réceptacles métier

4.4.1 Les facettes métier

Les facettes métier sont matérialisées sous la forme d'un document XML (*nomComposant.bfac*), généré durant la phase de conditionnement du composant. Elles sont dépendantes de la facette technique issue du développement (*nomComposant.tec*), et sont destinées à être spécialisées en facettes et réceptacles de collaboration au moment du déploiement du composant.

Ce document XML énumère les facettes métier créées à partir des facettes techniques du composant sous-jacent, en utilisant pour ce faire la notation XPath. La notation XPath ci-dessous précise que la facette métier `"vip-facet"` étend la facette `"Facet1SIPExecute"` déclarée dans la facette technique du composant.

```
  <facet name="vip-facet1"
extends="myComp:/definitions/portType[name=Facet1SIPExecute]/name">
```

Après avoir listé les paramètres libres de la facette métier (tags `<parameter>`), qui seront fixés lors de la définition des facettes de collaboration, le document recense au moyen des tags `<constraint>` et `<counterparty>`, les contraintes garanties et contreparties offertes à l'intérieur de la facette. Pour qu'une contrainte et/ou contrepartie exprimée du coté d'une facette soit interprétable du coté du réceptacle ou vice-versa, notre architecture doit s'appuyer sur un système de représentation des données standard (dont la définition dépasse la portée de ce manuscrit), et ainsi assurer une totale interopérabilité entre fournisseurs de services. De plus, ces interfaces métiers ayant vocation à définir les termes de la collaboration de nos composants, les contraintes et contreparties exprimées à ce niveau doivent correspondre à des fonctionnalités apportées par l'environnement d'exécution de nos composants.

Comme déclaration explicite des contraintes nous avons adopté la notation *ressource.registre.attribut*. Les ressources éligibles peuvent être la méthode appelée, le contexte dans lequel la méthode s'exécute, la requête ou la réponse retournée par la méthode. Le registre qui dépend de la ressource accède à un certain aspect de la ressource. Par exemple le registre `"execution"` de la ressource `"method"` concentre une série de paramètres relatif à l'exécution en cours de la méthode, ce qui permet de poser une contrainte sur la durée ou l'horaire d'exécution de la méthode. Un autre registre `"invocation"` pourrait regrouper les attributs liés au mécanisme d'invocation de la méthode, pour en limiter le nombre d'appels concurrents ou consécutifs.

```
method.execution.duration
method.execution.startingTime
method.invocation.concurrentCall
method.invocation.limit
```

Appliquée à notre service "tableau blanc" du paragraphe 3.5.5.4, la contrainte `method.invocation.limit` permettrait à partir d'une unique facette technique contenant la méthodes `addElement(Element)`, de proposer deux facettes métiers. Une première facette métier, dont l'accès serait conditionné par un service d'authentification et d'autorisation, proposerait un service de tableau blanc pouvant contenir un nombre illimité d'éléments. A l'opposé, une seconde facette de notre service de tableau blanc, en accès libre cette fois, utiliserait notre contrainte en vue de limiter le nombre d'éléments insérables en une session de service.

Outre la nécessité d'être interprétables et donc non ambiguës, les contraintes que nous considérons doivent également être mesurables par les deux parties de la relation facette/réceptacle. En effet, une fois adoptée par les composants clients et serveurs, l'évaluation de la contrainte se fait à la fois du coté de la facette et du coté du réceptacles, chacun étant à même de constater le nom respect d'une contrainte et donc d'appliquer le traitement convenu.

Pour être pleinement interprétable d'un fournisseur de service à l'autre, la déclaration des contreparties doit également faire appel à une nomenclature standardisée. Un service de facturation pourra ainsi être référencé sous le nom générique `services.common.billing`, chaque fournisseur de service ayant ensuite la liberté d'implémenter le service selon son environnement technique.

Une facette métier énumère enfin les méthodes métier construites à partir des méthodes présentes sur la facette technique du composant. Un exemple de facette métier, construit à partir de la facette technique exposée en Figure 43 pourrait être :

```
<business-facets name="MyComponent"

targetNamespace="http://www.enst.fr/myComponent.bfac"
                 xmlns:myComp="http://www.enst.fr/myComponent.tec">
  <facet name="business-facet"
         extends="myComp:/definitions/portType[name=Facet1SIPExecute]/name">
    <parameters>
      <parameter type="long
                 name="duree"/>
      <parameter type="long"
                 name="limite"/>
      <parameter type="double"
                 name="prix"/>
      <parameter type="double"
                 name="penalite"/>
      <parameter type="long"
                 name="myAccount"/>
      <parameter type="long"
                 name="remoteAccount"/>
    <parameters>
    <constraints>
      <constraint name="chronometre"
                  value="method.execution.duration"/>
      <constraint name="limiteur"
                  value="method.invocation.limit"/>
    </constraints>
    <counterparties>
      <counterparty name="paiement"
                    value="services.common.billing"/>
    </counterparties>

    <methods>
      <method ope-name="facet:/operation[name=m1]/name">
        <cust-parameters>
          <cust-parameter name="method:param[name=custParam]/name"
                          value="m1() from facet business-facet"/>
        </cust-parameters>
        <constraints>
          <constraint constraint-id="constraint1"
```

```
                    constraint-ref=”chronometre">
            <const-param name="operator1" value="strict-inf"/>
            <const-param name="operand1" value="duree"/>
          </constraint>
          <constraint constraint-id="constraint2”
                    constraint-ref=”limiteur">
            <const-param name="limit" value="limite"/>
          </constraint>
        </constraints>
        <counterparties>
          <counterparty counterparty-ref="counterparty1">
                      counterparty-id="paiement">
            <count-param name="amount" value="prix"/>
            <count-param name="from" value="remoteAccount"/>
            <count-param name="to" value="myAccount"/>
          </counterparty>
        </counterparties>
        <indemnities>
          <indemnity constraint="chronometre">
            <counterparty counterparty-id="counterparty2">
                        counterparty-ref="paiement">
              <count-param name="amount" value="penalite"/>
              <count-param name="from" value="myAccount"/>
              <count-param name="to" value="remoteAccount"/>
            </counterparty>
          </indemnity>
        </indemnities>
      </method>
    </methods>
  <facet>
</business-facets>
```

Figure 47 La description d’une facette métier

Une méthode métier commence par fixer les valeurs prisent par les variables de personnalisation du comportement des méthodes du composant, définies dans la facette technique (tag `<cust-parameter>`). Il s’ensuit une succession de tags `<constraint>`, délimitant les contraintes garanties par le composant. La définition de ces contraintes ne demande pas d’informations quant à l’implantation de leurs évaluations dans le processus d’exécution de la méthode. De fait, les contraintes définies à ce niveau ne sont pas de simples tests logiques, mais de véritables briques logicielles dédiées à une contrainte particulière (cf. paragraphe 4.5.3), qui viendrons se greffer sur le code d’implémentation de la méthode à laquelle elles se rapportent. Par exemple la contrainte `method.execution.duration` va utiliser une brique interceptant l’appel et le retour de la méthode, pour évaluer le temps de traitement de la méthode, et ainsi déterminer qui de la contrepartie ou de l’indemnité doivent être appliqués.

La déclaration de la méthode métier se poursuit avec une énumération de tags `<counterparty>` faisant appel cette fois à une brique logicielle générique, qui s’exécutera à l’achèvement du retour de la méthode, si aucune contrainte n’a empêché le fil d’exécution d’aller jusqu’à elle. La tag susnommé permet de communiquer à cette brique logicielle générique le service à appeler et les paramètres à joindre à cet appel.

Enfin la méthode métier inventorie les indemnité à considérer en fonction du non respect des contraintes garanties par le composant. Les indemnités fonctionnent sur le même modèle que les contreparties classiques.

4.4.2 Les réceptacles métier

Les réceptacles métier sont construits sur le même modèle que les facettes métier. L'utilisation d'un stub nous permet d'implanter les briques logicielles dédiées aux contraintes autour de l'appel de méthode local, effectué sur le stub. Un réceptacle métier produira donc un fichier XML analogue à la Figure 47.

4.5 Les facettes et réceptacles de collaboration

4.5.1 Présentation

Après une phase de conditionnement, nos composants maintenant équipés de facettes et réceptacles métier vont être déployés au sein d'un serveur d'application. Ce déploiement sera l'occasion de satisfaire d'une part les réceptacles exhibés par le composant, et d'autre part de proposer de nouveaux services aux composant existants.

Les facettes et réceptacles précédemment définis précisaient les vœux exprimés par les fournisseurs de service quant au contexte d'exécution de leurs composants. Cependant les contraintes et contreparties évoquées dans ces vœux restent générales, et nécessitent une spécialisation associée à une collaboration particulière entre deux composants clairement identifiés. L'ultime étape de notre structure de facettes et de composants sera ainsi la production de facettes et réceptacles de collaboration.

La facette de collaboration est l'interface finale du composant, directement proposée à l'utilisateur. Nos services étant construits sur une collection de composants, les points d'entrée d'un service seront constitués par les facettes de collaboration de son composant frontal, à partir desquelles s'enchaîneront des interactions entre réceptacles et facettes de collaboration.

4.5.2 La définition des facettes et réceptacles de collaboration

Une facette de collaboration (resp. réceptacle de collaboration) est une implémentation spécifique d'une facette métier (resp. réceptacle métier) dans le cadre d'une relation entre deux classes de composants. Facettes et réceptacles de collaboration peuvent maintenir des informations de classe relatives au composant distant. Ces informations ne sont pas destinées à interagir directement avec l'utilisateur final, mais à faciliter les interactions entre business actors. Ainsi un numéro de compte bancaire maintenu dans une facette de collaboration permettra à un composant serveur de débiter le compte du fournisseur du composant client, qui refacturera le coût du service à l'utilisateur final.

La déclaration d'une facette de collaboration commence par préciser la facette métier qu'elle implémente, puis fixe les variables libres déclarées dans cette même facette métier, d'après le contrat établit entre les deux fournisseurs de services.

```
<facet name="subscriber-facet"
       implements="myComp:/business-facets/facet[name=business-facet]/name>
  <parameters>
```

```
        <parameter name="facet:parameters/parameter[name=myAccount]"
                   value="123456789"/>
        <parameter name="facet:parameters/parameter[name=remoteAccount]"
                   value="987654321"/>
     </parameters>
```

Si une facette métier précisait les contraintes et contreparties proposées par le fournisseur de service, la facette de collaboration est le fruit d'un accord entre deux fournisseurs de services. En conséquent, les méthodes métiers reprisent dans la facette de collaboration réutilisent un sous-ensemble des contraintes et contreparties définies au niveau de la facette métier. La déclaration de chaque méthode de collaboration ne fait que lister les contraintes et contreparties désirées, les variables définies au niveau de la facette de collaboration, les contraintes et contreparties déclarées dans la facette métier étant des informations suffisantes pour la construction du contexte d'exécution de la méthode de collaboration.

Un exemple de facettes de collaboration peut être :

```
<collaboration-facets name="MyComponent"
                      targetNamespace="http://www.enst.fr/myComponent.cfac"
                      xmlns:myComp="http://www.enst.fr/myComponent.bfac">
   <facet name="subscriber-facet"
          implements="myComp:/business-facets/facet[name=business-facet]/name>
      <parameters>
         <parameter name="facet:parameters/parameter[name=duree]"
                    value="100"/>
         <parameter name="facet:parameters/parameter[name=prix]"
                    value="0.03"/>
         <parameter name="facet:parameters/parameter[name=penalite]"
                    value="0.3"/>
         <parameter name="facet:parameters/parameter[name=myAccount]"
                    value="123456789"/>
         <parameter name="facet:parameters/parameter[name=remoteAccount]"
                    value="987654321"/>
      <parameters>
      <methods>
         <method ope-name="facet:methods/method[ope-name=m1]/ope-name">
            <constraints>
               <constraint constraint-id="method:constraints/constraint[constraint-id
                  =constraint1]/constraint-id"/>
            </constraints>
            <counterparties>
               <counterparty counterparty-id="method:counterparties/counterparty
                   [counterparty-id=counterparty1]/counterparty-id"/>
            </counterparties>

            <indemnities>
               <indemnity constraint="method:constraints/constraint[constraint-id
                  =constraint1]/constraint-id">
                  <counterparty counterparty-id="method:indemnities/counterparty
                     [counterparty-id=counterparty2]/counterparty-id"/>
               </indemnity>
            </indemnities>
         </method>
      </methods>
   <facet>
   <facet name="free-facet"
          implements="myComp:/business-facets/facet[name=business-facet]/name>
      <parameters>
         <parameter name="facet:parameters/parameter[name=limite]"
                    value="25"/>
      <parameters>
      <methods>
         <method ope-name="facet:methods/method[ope-name=m1]/ope-name">
            <constraints>
               <constraint constraint-id="method:constraints/constraint[constraint-id
                  =constraint2]/constraint-id"/>
            </constraints>
            <counterparties/>
            <indemnities/>
         </method>
```

```
      </methods>
   <facet>
</collaboration-facets>
```

Figure 48 La description d'une facette de collaboration

Ce document déclare deux facettes de collaboration, implémentant la même facette métier. Une première facette, présentée à des utilisateurs abonnés au composant (le mécanisme d'authentification et d'autorisation n'est pas mentionné dans la facette), donne accès à notre méthode m1(), en garantissant un temps de réponse de 100 ms, avec un coût de 3 m€. La seconde donne toujours accès à notre méthode m1(), sans contrepartie cette fois, mais avec un nombre d'invocations limité à 25 au sein d'une même session.

4.5.3 L'interaction entre réceptacles et facettes de collaboration

Facettes et réceptacles de collaborations sont abordés de façon diamétralement opposées quand il s'agit de considérer leurs interactions avec le code d'implémentation du composant. D'un coté les références maintenues dans les facettes de collaborations font qu'il est aisé de partir de la facette de collaboration, utilisée par le client du composant, pour remonter vers la facette métier puis la facette technique, directement prise en charge par le code d'implémentation du composant ; de l'autre le code d'implémentation ayant été figé au terme de la phase de développement, il n'est pas envisageable de lier directement ce code d'implémentation aux réceptacles de collaboration définies pendant la phase de déploiement.

Comme premier support de résolution de ce problème, nous pouvons réutiliser les informations de dépendances entre facettes et réceptacles techniques d'un même composant (cf. Figure 43), apportées avec la définition des facettes techniques. En effet, un réceptacle n'a de raison d'être que s'il contribue à au moins l'exécution d'une méthode du composant. Transposée au niveau des interfaces de collaboration du composant, cette assertion nous enseigne qu'un réceptacle de collaboration n'a lieu d'être que s'il contribue à au moins l'exécution d'une méthode d'une facette de collaboration du composant. Ainsi si le problèmes des interactions entre composants est une affaire de relations entre réceptacles et facettes de composants distants, il est aussi une question de relation entre facettes et réceptacles d'un même composant.

Conformément à ce que nous avions écrit dans la partie 3.5.4, c'est au broker que sont destinées les informations relatives aux relations entre facettes et réceptacles techniques (cf. Figure 44), définies au moment du conditionnement du composant. Le broker disposait à ce moment des informations du type :

- `La méthode "m1()" sur la facette technique "facet1" dépend du réceptacle technique "receptacle1".`

Dans notre architecture nous faisons du broker une entité capable de rapprocher les réceptacles techniques manipulés dans le code d'implémentation, et les réceptacles de collaboration utilisés pour la communication effective entre composants. Le broker doit structurer les données issues du déploiement du composant pour être capable d'identifier par exemple que :

- Les facettes de collaboration `"subscriber-facet"` et `"basic-facet"` du composant nouvellement déployé sont toutes deux dérivées d'une facette métier faisant référence à la facette technique `"facet1"`.

- Les réceptacles `"premium"` et `"best-effort"` sont tous deux dérivés d'un réceptacle métier faisant référence au réceptacle technique `"receptacle1"`.
- La facette de collaboration `"subscriber-facet"` dépend du réceptacle de collaboration `"premium"` et la facette de collaboration `"basic-facet"` dépend du réceptacle de collaboration `"best-effort"`.

Les deux premiers points sont des informations directement interprétables après une lecture des facettes et réceptacles métier. En revanche le dernier point demande une intervention au moment du déploiement du composant, pour préciser les liens entre facettes et réceptacles de collaboration sur un même composant.

Dans la pratique, quand le code d'implémentation fera appel au broker pour obtenir une référence à un stub, l'algorithme de la méthode `resolve(nom_receptacle_technique)` utilisée sera le suivant :

1. Identifier la méthode qui fait appel au broker.
2. Identifier la facette de collaboration par laquelle l'invocation de la méthode a été acheminé.
3. Retourner une référence à un stub associé au réceptacle de collaboration en adéquation avec les données des points 1/ et 2/.

Cet algorithme repose sur l'hypothèse que facettes et réceptacles de collaboration sont construits sur le même modèle. En effet, si le code d'implémentation fait appel au broker pour localiser le stub qui lui permettra d'effectuer son appel distribué, il en va de même pour le conteneur de composants, qui passera par le broker pour localiser le squeletton capable de traiter un appel de méthode entrant. Ainsi le broker, pièce maîtresse omnisciente du serveur d'application, est capable de rapprocher appels distribués entrants et sortants, pour assurer les mises en relations légitimes entre facettes et réceptacles de collaboration.

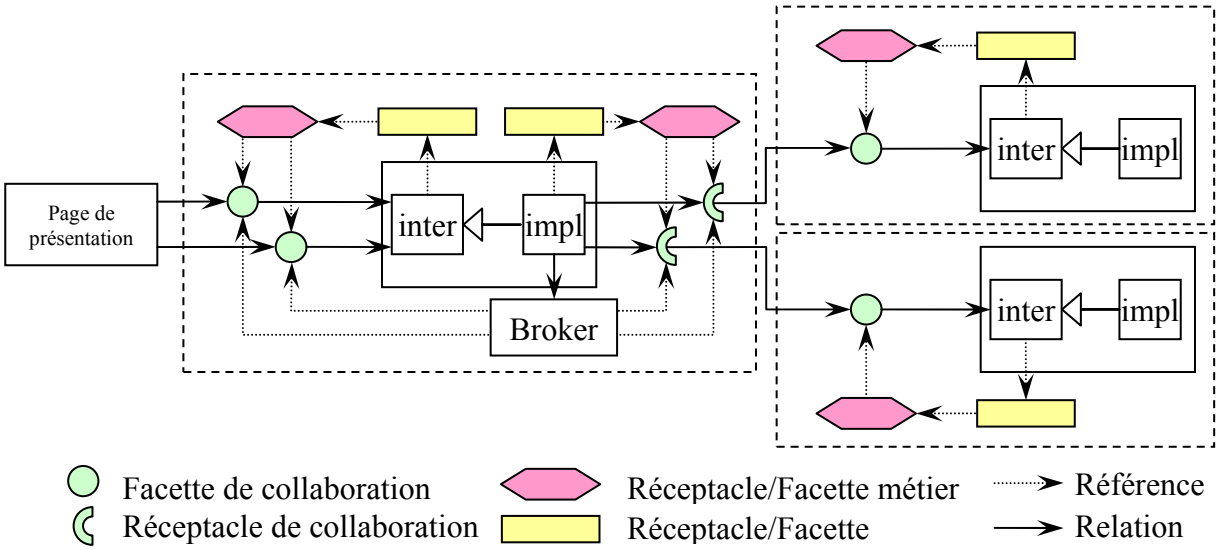


Figure 49 Liens entre facettes et réceptacles

Dans l'exemple de la Figure 49, nous présentons un service externalisant deux facettes de collaboration, issues de son composant frontal. Ces deux facettes de collaborations proviennent de la même facette technique, et sont dons dépendantes de deux réceptacles de collaboration provenant eux aussi du même réceptacle technique. Pour chaque invocation d'une méthode frontale du service, le broker a la charge de mettre en relation le code

d'implémentation des facettes du composant frontal, avec la facette adéquate d'un composant tiers.

4.6 L'architecture d'exécution

Cette partie descriptive de l'environnement d'exécution de nos composants s'inspire de travaux d'epérimentations effectués sur le serveur d'application JBoss. Pour préciser comment facettes et réceptacles de collaboration sont pris en charge dans notre architecture, nous allons détailler l'utilisation d'un réceptacle de collaboration par le code d'implémentation d'un composant client. La Figure 50 nous présente visuellement les différents éléments architecturaux concourant à la mise en relation de composants.

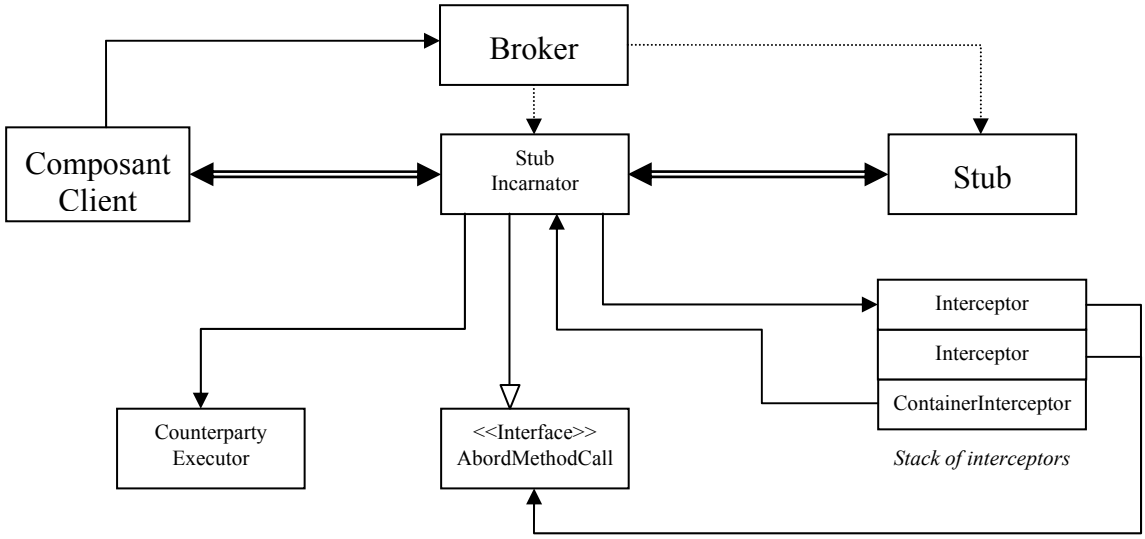


Figure 50 L'incarnation d'un réceptacle de collaboration

Un composant rencontre dans son code d'implémentation un besoin en interaction avec un composant distant. Afin d'effectuer son appel de méthode dans le contexte et sur le composant approprié, notre composant client fait appel au broker. En tenant compte de la facette et de la méthode ayant mené à la demande du composant client, le broker active une instance de *stub incarnator* chargée de coordonner la mise en place du réceptacle de coordination adéquat. Le broker retourne au composant client une référence vers le stub incarnator nouvellement instancié.

Sur le modèle du "transparent proxy" de l'architecture .NET, le stub incarnator présente localement au composant client l'interface distante qu'il attend. Il se chargera ensuite des traitements locaux visant à l'évaluation des contraintes et l'application des contreparties précisées dans le contrat de collaboration liant les deux classes de composants, avant de faire appel su stub qui procèdera à l'appel du composant distant.

Pour le traitement des contraintes nous avons réutilisé la notion de pile d'intercepteurs, au cœur du serveur d'application JBoss. Chaque intercepteur de cette pile est dédié au traitement d'une des contraintes définies dans le réceptacle de collaboration. A cet effet ils présentent au stub incarnator les méthodes suivantes :

- `preInvoke(Context, AbordMethodCall)`, appelée avant l'appel de la méthode distante. Cette méthode peut être l'occasion pour l'intercepteur d'initialiser un traitement qui prendra fin avec le retour de la méthode.
- `postInvoke(Context, AbordMethodCall)`, appelée avant la présentation du retour de la méthode au client. Cette méthode permet d'une part d'achever un traitement qui aurait été initié lors de l'appel à la méthode preInvoke, et d'autre part de trancher quant au respect ou non de la contrainte suivie par l'intercepteur.

Face aux contraintes exprimables dans nos facettes et réceptacles de collaboration, les intercepteurs peuvent avoir deux comportements distincts. D'un coté nous aurons des intercepteurs dont l'action sera simultanée à l'exécution de la méthode distante, et de l'autre les intercepteurs qui se contenteront de l'évaluation d'un paramètre au terme de l'exécution de cette méthode distante.

Classiquement le premier type d'intercepteur est utilisable pour des contraintes de contrôle de flux au niveau application, de temps ou de qualité d'exécution mesurée en cours de traitement. Outre la caractéristique de s'exécuter en parallèle avec la méthode distante, cette classe d'intercepteurs permet de suspendre à tout moment l'exécution de la méthode. Comme exemple de cette catégorie d'intercepteur nous pouvons reprendre notre intercepteur Timer, dont le code pourrait ressembler à :

```
Public class TimerInterceptor extends Interceptor
{
  private String id;
  private long time;
  private Timer timer;
  private AbordMethodCall abord;

  public TimerInterceptor(String id, long time);
  {
    this.id = id;
    this.time = time;
  }

  public void preInvoke(Context context, AbordMethodCall abord)
  {
    this.abord = abord;
    timer = new Timer();
    timer.schedule(new AbordTask(), this.time);
    context.getStack().next().invoke(context, abord);
  }

  public void postInvoke(Context context, AbordMethodCall abord)
  {
    timer.cancel();
    context.getStack().previous().invoke(context, abord);
  }

  class AbordTask extends TimerTask
    public void run()
    {
      abord.abordMethod(this.id);
    }
}
```

Cet intercepteur hérite de la classe `Interceptor`, ce qui lui confère des propriétés communes à tout intercepteur géré par le stub incarnator. Le constructeur d'un intercepteur prend comme argument un identifiant d'intercepteur et les données utiles à son fonctionnement. L'identifiant d'intercepteur est réutilisé par le stub incarnator pour faire le lien entre contraintes et contreparties dans le cas où la contrainte est violée. Les données de paramétrage de l'intercepteur (l'intervalle de temps pour notre exemple), sont directement issues du fichier de configuration du réceptacle de collaboration, pris en charge par le stuc incarnator.

Les méthodes `preInvoke` et `postInvoke` reçoivent en argument le contexte d'exécution du composant, maintenu par le conteur, à partir duquel il est possible d'importer les données résultant de l'exécution d'intercepteurs précédents, ou de connaître pour un intercepteur son successeur dans la pile d'intercepteurs. Comme second argument nos méthodes reçoivent également une référence vers une interface implémentée par le stub incarnator, permettant l'annulation de l'exécution de la méthode, suite à une constatation de la violation de la contrainte gérée par l'intercepteur.

Le second type d'intercepteur est utilisable pour les évaluations ponctuelles de contraintes ou le traitement des contreparties en cas de respect des contraintes. En effet, comme nous l'avons représenté en Figure 51, contraintes et contreparties peuvent tous deux être modélisés par un intercepteur. Nos intercepteurs étant à la fois parcourus à l'aller (via la méthode preInvoke) et au retour (via la méthode postInvoke), il suffit d'adopter une règle de construction de la pile d'intercepteurs qui place les contreparties au sommet et les contraintes en bas de la pile. Avec cette convention, sur le retour de la méthode distante, les contraintes commenceront par être évaluées, et ne permettront le passage dans les intercepteurs de contrepartie que si l'ensemble des intercepteurs de contrainte ont validé le respect de leur contrainte propre.

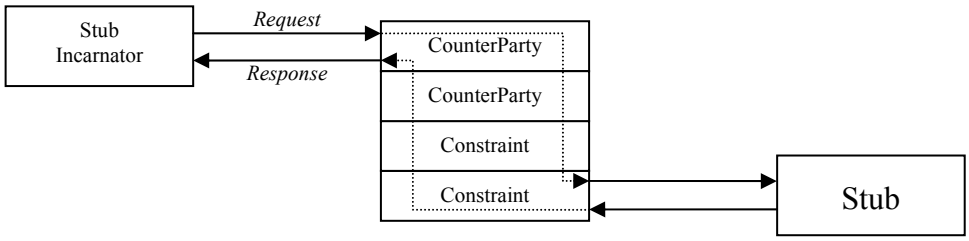


Figure 51 Schéma de construction d'une pile d'intercepteurs

Comme exemple d'intercepteur de contrepartie nous pouvons donner un aperçu de ce que pourrait être la prise en charge d'une contrepartie de facturation pour notre facette de collaboration `"subscriber-facet"` de la Figure 48 :

```
Public class BillingInterceptor extends Interceptor
{
  private long fromAccountNumber;
  private long toAccountNumber;
  private long sessionId;
  private double amount;

  public BillingInterceptor(long fa, long ta, long id, double
amount);
  {
    this.fromAccountNumber = fa;
    this.toAccountNumber = ta;
```

```
    this.sessionId = id;
    this.amount = amount;
  }

  public void preInvoke(Context context, AbordMethodCall abord)
  {
    this.abord = abord;

    Broker broker = new Broker(”broker.socgen.fr”);
    Object ref = broker.getHome(”ChargerHome”);
    ChargerHome home = ChargerHomeHelper.narrow(ref);
    Charger charger = home.findSessionById(this.sessionId);
    Charger.addUser(this.accountNumber);

    context.getStack().next().invoke(context, abord);
  }

  public void postInvoke(Context context, AbordMethodCall abord)
  {
    Broker broker = new Broker(”broker.socgen.fr”);
    Object ref = broker.getHome(”ChargerHome”);
    ChargerHome home = ChargerHomeHelper.narrow(ref);
    Charger charger = home.findSessionById(this.sessionId);
    Charger.requestCharge(this.fromAccount, this.toAccount);

    context.getStack().previous().invoke(context, abord);
  }
}
```

Le constructeur de notre intercepteur va prendre en paramètre les données utiles à l'exécution de la contrepartie, ici en l'occurrence un service de facturation. Comme nous l'avons vu dans la partie 4.5.3, les contreparties sont exigées par une extrémité d'une relation de collaboration, le plus souvent le composant serveur, et accordées de l'autre coté de la relation, le composant client. Ainsi toute contrepartie convenue dans un contrat de collaboration génère un intercepteur de contrepartie du coté de la facette et un autre intercepteur de contrepartie du coté du réceptacle de collaboration. Notre exemple ci-dessus nous présente l'intercepteur coté composant serveur (facette de collaboration). Les deux intercepteurs, perçus comme de simples clients par ce service de facturation distant hébergé dans le domaine `socgen.fr`, font appel au broker du domaine pour obtenir une référence vers la home du composant frontal du service, en utilisant un nom logique `broker.getHome(”ChargerHome”)`. Le réceptacle de collaboration étant parcouru avant la facette de collaboration, c'est le réceptacle qui sur la home demandera l'ouverture d'une session de service en appelant la méthode `createSession()`. La home retournera alors une référence vers une instance de composant `Charger` à l'intercepteur coté réceptacle, dont l'identifiant parviendra jusqu'à l'intercepteur de la facette de collaboration (paramètre id du constructeur), pour que ce dernier puisse partager la session de service. Développé selon la pattern introduite dans le paragraphe 3.5.5, notre service de facturation nécessite un appel à la méthode `addUser(userId)` sur l'instance de composant `Charger`, afin de connecter au composant session les composants utilisateur dédiés à nos fournisseurs de composant client et serveur. Avec le retour d'exécution de la méthode proposée par la facette, c'est la facette de collaboration qui sera cette fois parcourue avant le réceptacle. La méthode `postInvoke` de notre intercepteur est appelée si aucune contrainte n'a été violée. L'intercepteur se charge alors d'appeler la méthode `requestCharge()` sur le service de facturation, afin d'initier une transaction entre nos deux fournisseurs. Arrivé du

coté du réceptacle, l'intercepteur appellera la méthode `acknowledgeCharge()` sur le composant session, afin de valider la transaction. Le service de facturation procèdera ainsi aux opérations de débit et crédit sur les composants clients qui lui sont associés.

Comme indiqué dans la Figure 50, une pile d'intercepteurs se termine par un intercepteur nommé `ContainerInterceptor`, dont l'action se limite à retourner la main au Stub incarnator, qui se chargera d'appeler le stub lors du chemin aller, ou de retourner le résultat de la méthode pour le chemin retour. D'autre part, le Stub incarnator implémentant l'interface `AbordMethodCall`, c'est lui qui recevra les notifications de contraintes non respectées. Dans ce cas il devra appliquer les contreparties prévues par la facette ou le réceptacle de collaboration, et remonter une exception au composant client. Notons qu'avec notre construction, il est également possible de lever une exception dans le cas où l'application d'une contraprtie rencontrerait un problème. Par exemple, un appel de la méthode `requestCharge()`, qui serait refusé par le service de facturation peut occasionner un appel de la méthode `abordMethod()` sur le Stub incarnator, pour annuler la relation de collaboration.

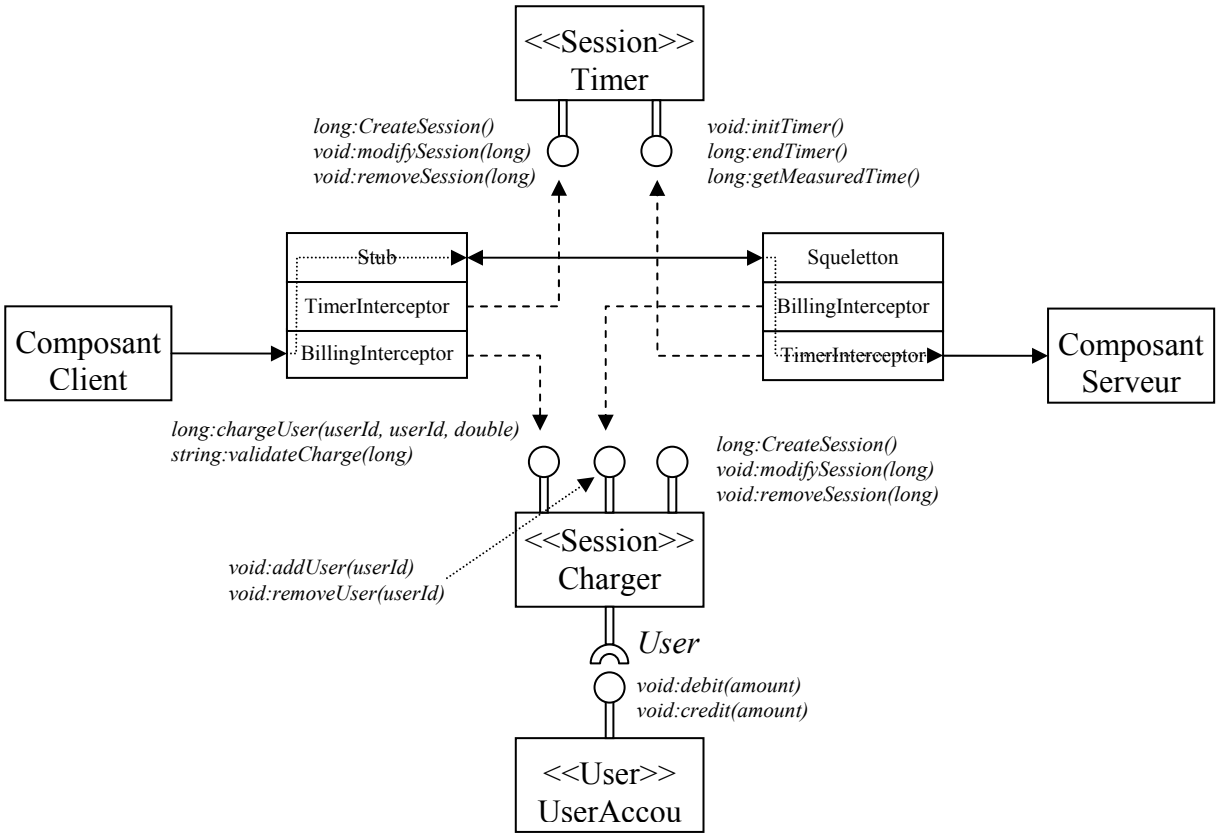


Figure 52 L'architecture d'exécution de la méthode m1()

Ainsi l'architecture d'exécution (représentée en Figure 52) de la méthode que nous avions introduit dans l'exemple de la partie 4.3 mobilise les intercepteurs suivants :

Coté Client :

`BillingInterceptor :`

preInvoke() Création d'une session de service pour le service de facturation
Ajout du fournisseur comme utilisateur de la session.

postInvoke() Acceptation du montant à payer, comme acquittement du service rendu.

`TimerInterceptor :`

preInvoke() Création et initialisation d'une session de service pour le service timer.

postInvoke() Interrogation de l'état du service Timer {Expired, NotExpired }.

Coté Serveur :

`BillingInterceptor :`

preInvoke() Ajout du fournisseur comme utilisateur de la session.

postInvoke() Demande de paiement, comme acquittement du service rendu.

`TimerInterceptor :`

preInvoke() Démarrage du service.

postInvoke() Interrogation de l'état du service Timer {Expired, NotExpired }.

Notre architecture basée sur les contraintes et contreparties, prises en charge par des intercepteurs nous permet une grande souplesse d'adaptation aux décisions commerciales de collaboration qui peuvent être prise dans le cadre d'interactions de service. En effet, le comportement des intercepteurs décrits ci-dessus suppose que les deux intercepteurs TimeInterceptor font appel à la même instance de service Timer offert par un fournisseur tiers. Mais un intercepteur ou les deux intercepteurs peuvent être modifié librement pour utiliser par exemple des services Timer locaux à chaque fournisseur de service.

4.7 Conclusion

Nous avons vu au cours de cette partie comment nous appréhendions les interactions entre composants, au moyen d'interfaces descriptives. Dans un second temps nous avons donné un aperçu d'une implémentation possible de ces interfaces, pour une prise en charge à l'exécution.

5 L'exploitation du service

5.1 Introduction

Les concepts avancés sur les services et leurs composants peuvent jusqu'à présent s'appliquer à des domaines fonctionnels allant au-delà du monde des télécommunications. Aussi pour montrer l'apport de nos propositions au monde des télécoms, nous allons proposer dans cette partie une mise en œuvre de nos services dans une architecture télécom.

Pour suivre l'activité de nos utilisateurs, nous avons réutilisé le service de présence comme point central de notre architecture. Pour répondre aux besoins de notre architecture, nous avons développé les capacités de ce service de présence en lui permettant de gérer outre l'état des utilisateurs, le statut de chacun des services souscrits, ce qui le place en une sorte de superviseur de l'état général des utilisateurs et services sur le réseau.

Pour véhiculer notre signalisation liée aux notifications de statut des services et demandes de collaboration, nous aons réutilisé le protocole SIP. Si notre contribution ne propose pas de nouveau message SIP (nous ne reprenons que le protocole standard muni de ses extensions [ROS02]), nous proposons en revanche les mécanismes de traitement des messages SIP au niveau des composants, en réutilisant les recommandations introduites dans JAIN SIP [JAS02], et ainsi permettre aux composants de créer ou modifier une session de services.

Dans un second temps nous présenterons comment offrir la possibilité à l'utilisateur final de personnaliser l'utilisation de ses services, en réutilisant la notion de contrats de coordination. Là encore, si notre contribution n'a pas introduit le concpet de contrat, elle a le mérite de proposer une formaliser les règles figurant dans ces contrats, dans le but de lier l'exécution d'une règle à des échanges entre composants (ou en d'autres termes à la consommation ou la production de messages SIP). D'autre part, notre contribution proposera un site d'implantattion de ces contrats dans notre architecture de services, ceux-ci étant à la destination directe de l'utilisateur final, pour lui offrir la possibilité de poser des pré ou post conditions sur l'exécution de ses services, et ainsi les adapter à ses besoins.

5.2 Présentation

La mise en exploitation du service se focalise sur les aspects dynamiques du service, à savoir son interaction avec d'autres éléments de services, et sa personnalisation à la fois par le fournisseur et l'utilisateur final. Pour ce faire, nous avons généralisé l'utilisation de contrats, pris en charge à l'exécution des services par le conteneur, à plusieurs niveaux de notre architecture. Pour appréhender les contraintes liées au contexte d'utilisation et d'interaction des services, nous avons placé le service de présence au cœur de notre architecture, comme observateur de l'état de nos utilisateurs.

5.3 L'utilisation du Service de Présence

Dans le réseau téléphonique classique, l'utilisateur n'a vis à vis du réseau guère plus de choix qu'entre les états libre, occupé, voire hors connexion s'il retirait son équipement du réseau. Avec l'arrivée de terminaux intelligents et de systèmes de communication ouverts, notamment la voix sur IP, cette notion d'état s'est considérablement enrichie, intégrant des concepts bien plus évolués tels que de retour dans 10 minutes, en conférence qui permettent une meilleure adéquation entre les demandes de mise en relation des uns et les disponibilités

des autres. Outre le fait que cette évolution permette aux opérateurs d'améliorer la rentabilité de leurs installations, ce raffinement dans la description de la disposition réelle de l'utilisateur a pour conséquence de permettre une meilleure intégration des services dans les situations de vie des abonnés. Alors qu'aujourd'hui il est tout au plus possible d'activer et désactiver ses services à l'unité, il sera possible demain de configurer un panel de services, qui variera au gré de nos changements d'états.

Cependant, si les systèmes émergeants basés IP sont capables de prendre en charge une quantité abondante d'états de leurs utilisateurs, ils trouvent leurs limites, mises en évidence par l'IETF [LRS00] dans la définition d'états personnalisés par les abonnés eux-mêmes. En effet, si les états libre ou occupé du système téléphonique actuel ont l'avantage de présenter des états interprétables par toute entité du réseau, l'état j'attends un appel urgent de Mr Dupont n'a pas de signification empirique, dans les réseaux actuels. De ce fait, il y a aujourd'hui un besoin non seulement de raffiner les états des utilisateurs, mais d'avantage une nécessité de définir un mécanisme qui permette de prendre en considération le comportement qui leurs sont associés.

Pour rester dans la philosophie de SIP [RLS99], le protocole utilisé pour assuré la signalisation à l'intérieur de notre architecture (cf partie 5.4), nous sommes revenus sur le mécanisme de recherche de la localisation des utilisateurs. Ainsi dans l'état un message INVITE arrivant au niveau d'un proxy SIP provoquera l'interrogation du location server du domaine, pour déterminer si l'utilisateur se trouve bien actuellement dans le domaine visité. Cette quête de la localisation de l'utilisateur constitue pour nous une démarche autour de laquelle nous avons construit notre recherche de l'état de l'utilisateur. En effet, notre architecture doit inclure un mécanisme qui tienne non seulement compte de la localisation de l'utilisateur, mais également de l'état de ses terminaux et de l'activation des services auxquels il a souscrit. Pour répondre à cette attente, nous avons fait appel au service de présence, et ce comme pivot conditionnant les relations entre services et utilisateurs.

Le service de présence, tel que présenté dans [LRS00] est basé sur une architecture à quatre niveaux (Figure 53), les deux niveaux supérieurs sont pris en charge par le Détaillant, tandis que les deux niveaux inférieurs sont directement implantés sur le terminal de l'utilisateur. Ces niveaux sont :

- Le serveur de présence
- Les éléments de présence
- Les agents utilisateur
- Les applications utilisatrices

L'immense majorité des travaux portant sur le service de présence se limitent à l'étude des interactions entre les trois premières entités de l'architecture de présence, le dernier point étant laissé à la libre interprétation des constructeurs d'applications. Notre architecture ayant vocation à proposer une solution au problème de la composition de services, nous attacherons une importance particulière à la définition de l'interaction entre les trois premières entités de l'architecture de présence et les applications utilisatrices.

Dans une architecture de présence, les utilisateurs sont dans la pratique les applications jouant le rôle d'interface entre l'utilisateur physique et le réseau. Ces applications (Microsoft Messenger, Yahoo Messenger, …) maintenant très répandues sur les terminaux dotés de capacités d'échanges de données entre utilisateurs servent à la fois au recueil des actions

utilisateur occasionnant des interactions avec le réseau, et de support pour l'exécution de services implantés sur le réseau.

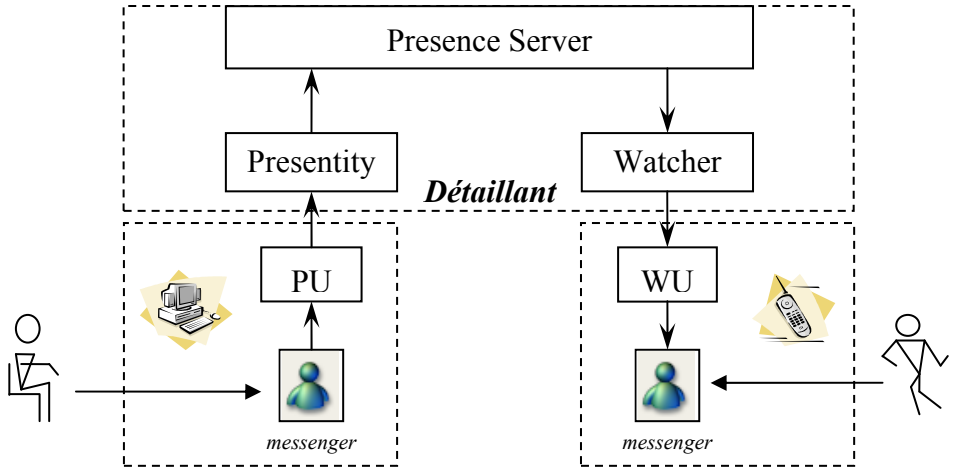


Figure 53 Le Service de présence

Tels qu'introduits dans [LRS00], les agents utilisateur, sont chargés d'assurer la translation entre actions coté utilisateur et changements d'état sur le réseau. Ces agents sont la plupart du temps directement couplés aux applications utilisatrices, dont ils captent un ensemble d'événements. Ces agents peuvent être classés suivant deux catégories : la première (Presence User Agent : PUA) écoute les actions utilisateur pour mettre à jour le cas échéant l'état de l'utilisateur maintenu par le réseau, alors que la seconde (Watcher User Agent : WUA) initie des actions sur l'application cliente, consécutivement à l'observation d'un changement d'état.

Tout comme les agents utilisateur, les éléments de présence peuvent être séparés en deux catégories : les Presentity et Watcher. Dans toute architecture de présence, un presentity modélise l'état global d'un utilisateur physique, dans lequel sont référencés les états particuliers de l'ensemble de ses PUA. Un presentity dispose d'un protocole de présence (SIP a été proposé à cet effet dans [ROS02]) pour communiquer la configuration de son état au serveur de présence. Un tel protocole doit intégrer d'un coté des messages permettant de mettre à jour le serveur de présence en tenant compte des états des différents agents connues du presentity, et de l'autre un ensemble de messages facilitant la souscription et la notification des changements d'états à des utilisateurs distants. A cet effet, l'architecture de présence propose les watchers, qui pour le compte d'un utilisateur, peuvent ponctuellement, ou périodiquement, s'enquérir auprès du serveur de présence, de l'état d'un utilisateur.

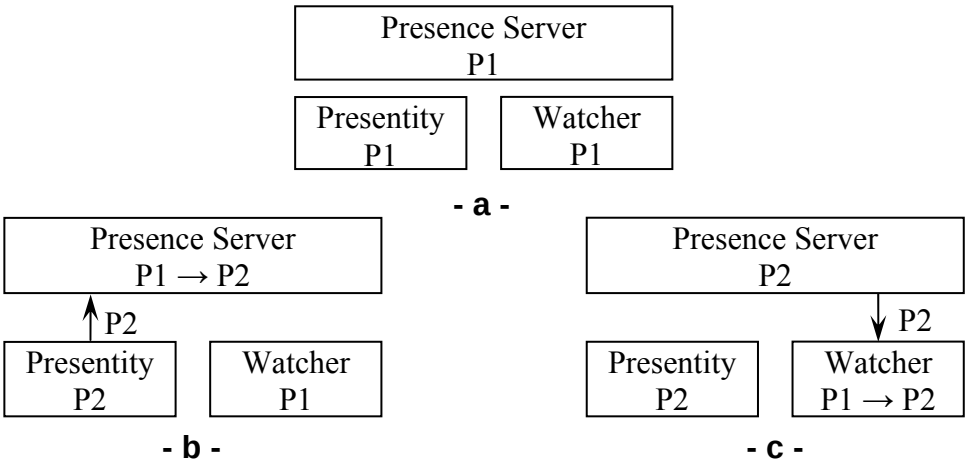


Figure 54 Le Processus de notification d'un changement d'état

Au sommet de cette architecture se trouve le serveur de présence, chargé de communiquer aux watchers les informations de présence émanant des presentities. Dans un souci d'optimiser les échanges entre éléments de présence, le serveur de présence gère en cache les états des presentities, qu'il communiquera aux watchers. Il reçoit également les demandes d'abonnement de la part des watchers désirant suivre les évolutions des états des presentities, en ayant la possibilité de les accepter ou les refuser, selon les recommandations des utilisateurs.

La Figure 54 nous présente le principe de la notification de changement d'état d'un utilisateur. Si nous admettons qu'un presentity est dans un état P1 en Figure 54-a, le passage dans un état P2 déclenchera l'envoi d'un message à destination du presence server, décrivant le nouvel état de présence du presentity (Figure 54-b). A réception de ce message, le presence server modifiera l'état de l'utilisateur qu'il maintient en interne, cette modification d'état occasionnant l'envoi d'un message de notification de changement d'état aux watchers liés au presentity (Figure 54-c)

5.4 SIP

5.4.1 Le choix de SIP

Lors de la recherche d'un protocole extensible, capable à la fois de prendre en charge la signalisation associée au fonctionnement des services et les informations de présence, notre choix s'est porté sur SIP. Le Session Initiation Protocol est un protocole de signalisation point à point permettant l'établissement et le contrôle de liaisons de données sur un réseau IP. Contrairement à l'architecture H323, qui est un regroupement de plusieurs signalisations (H245 et H225), SIP réalise le tour de force de proposer une signalisation, simple et éprouvée, fortement inspirée du protocole HTTP, offrant un modèle d'invocation de services autour duquel peuvent s'intégrer une multitude de services supplémentaires.

5.4.2 L'architecture autour de SIP

Le réseau IP cœur de notre architecture d'étude, géré par le fournisseur de services au détail (Détaillant), interconnecte les réseaux locaux hébergeant nos utilisateurs. Conformément à l'architecture SIP, sur le réseau local nous retrouvons un proxy redirecteur, chargé de

véhiculer les messages SIP du réseau local vers le sous réseau. Chaque business domain de l'architecture est ainsi accessible via un proxy, servant de point d'entrée vers les services offerts par le domaine. Sur la base de cette articulation, SIP prévoit la possibilité d'étendre le spectre de compétence du proxy en lui associant directement des agents. [ROS02] présente ainsi une extension de SIP pour prendre en charge un Presence Agent, capable de traiter des messages SIP spécifiques au service de présence, et assurant le lien entre le proxy et les informations de présence des utilisateurs.

Nous allons réutiliser ce modèle d'agents, ou d'intercepteurs de messages SIP, pour créer un agent spécialisé dans le traitement de contrats de collaboration, dans le but de composer ou personnaliser le comportement des services. De plus, être directement implanté sur le proxy permet la définition d'un modèle générique. En effet, la notion de proxy se retrouve à la fois sur le réseau cœur, et sur le terminal de l'utilisateur. De ce fait, il est possible d'implanter ses contrats en trois points du réseau. Le fournisseur de services pourra lui-même déployer sur son réseau, des contrats communs à tous ses abonnés, tandis qu'une entreprise ou un utilisateur itinérant aura, au niveau du réseau local, la possibilité de créer des contrats communs a un groupe plus restreint d'utilisateurs. Enfin, un utilisateur pourra créer sur son terminal un ensemble de contrats intimement lié aux applications à sa disposition.

5.4.3 Le principe de fonctionnement de SIP

SIP est un protocole de signalisation au niveau application utilisé pour la création, la modification et la destruction de sessions mobilisant un ou plusieurs utilisateurs. Ces sessions permettent essentiellement d'effectuer des appels téléphoniques ou multimédia au dessus d'Internet, grâce à l'utilisation de requêtes et réponses émises et réceptionnées par un SIP User Agent présent sur tout terminal SIP. Les concepteurs de SIP ayant souhaité retrouver la simplicité d'HTTP, SIP est un protocole basé sur des échanges de messages "plain text", réutilisant la même nomenclature que HTTP. SIP défini six requêtes qui peuvent être classées en deux catégories : Les requêtes REGISTER et OPTIONS, utilisées hors du contexte d'une session, et les requêtes INVITE, ACK, CANCEL et BYE directement liées à la gestion d'une session.

Pour ouvrir une session, un SIP User Agent Client (UAC) envoi un message INVITE à destination d'un SIP User Agent Server (UAS). Ce message contient, comme tout message SIP, en entête la liste des paramètres utiles aux proxy, serveurs de localisation … pour router la requête jusqu'à son destinataire. Cet entête commence par préciser le type du message, en indiquant l'URI auquel il est destiné. Le lien peut faire aussi bien faire référence à un utilisateur qu'à un service, un serveur SIP ayant la possibilité de passer par un serveur DNS si besoin pour résoudre l'adresse demandée. Les champs From et To permettent de préciser les appelant et appelé, le proxy ayant la possibilité de modifier leur valeur au gré par exemple de transferts de connexions. Enfin le champ call-ID permet à tout expéditeur d'une requête SIP, de faire le rapprochement avec une réponse reçue. Une exemple d'entête de message SIP INVITE peut être :

```
INVITE sip: UserA@wanadoo.fr SIP/2.0
Via: SIP/2.0/UDP aol.com:5060;branch=z9hG4bK74bf9
Max-Forwards: 70
From: Elie <sip:UserB@aol.com>;tag=9fxced76sl
To: Arnaud <sip:UserA@wanadoo.fr>
```

```
Call-ID: 12345601@aol.com
CSeq: 1 INVITE
Contact: <sip:UserB@blake.aol.com>
Content-Type: application/sdp
Content-Length: 147
```

SIP n'étant pas un protocole orienté connexion (il n'y a pas d'automate modélisant l'état d'un appel SIP comme dans le réseau intelligent), chacun des messages SIP tout au long de la session doivent contenir l'ensemble des informations susceptibles d'aider un serveur SIP dans son traitement.

A l'opposé, le corps du message INVITE encapsule des données exploitable par le destinataire (l'UAS) pour l'établissement de la connexion. Classiquement SIP utilise le langage SDP (cf. champ Content-type) pour décrire les différentes ressources média accessibles sur le terminal du client. Un corps de message INVITE peut ainsi être :

```
v=0
o=UserA 2890844526 2890844526 IN IP4 aol.com
s=Session SDP
c=IN IP4 blake.aol.com
t=0 0
m=audio 49172 RTP/AVP 0
a=rtpmap:0 PCMU/8000
```

En Figure 55 nous traitons un scénario issu de [DJS02], dont le contenu détaillé des messages est donné en Annexe 1, relatant l'établissement d'une connexion entre deux utilisateurs User A et User B, chacun étant accessible via un proxy. Ce scénario commence par une demande d'authentification de User A auprès du proxy, de par l'envoi d'une réponse 407 suite à la réception du message INVITE. Cette réponse 407 contient l'ensemble des données nécessaires à l'authentification attendue par le proxy. Le message ACK est alors utilisé par User A, pour préciser que la demande d'authentification à été prise en compte par l'utilisateur, qui retournera ensuite son message INVITE initial, complété par les données d'authentification présentées sous la forme :

```
Proxy-Authorization:Digest username="UserB", realm="security.com",
 nonce="wf84f1ceczx41ae6cbe5aea9c8e88d359", opaque="",
 uri="sip:UserA@wanadoo.fr", response="42ce3cef44b22f50c6a6071bc8"
```

Outre la réponse 407, SIP est capable de prendre en charge une grande variété de retours sur un message INVITE, classés selon six catégories :

1xx: Réponse provisoire – la requête a bien été reçue, son traitement est toujours en cours coté serveur.

2xx: Succès – l'action induite par la requête a été convenablement reçue, comprise et traitée.

3xx: Redirection – une action supplémentaire doit être effectuée pour achever l'exécution de la requête.

4xx: Erreur coté Client – la requête ne peut pas être traitée sur le serveur pour un problème de formatage.

5xx: Erreur coté Serveur – le serveur ne parvient pas à traiter une requête d'apparence valide.

6xx: Erreur Globale – la requête ne peut être traitée sur aucun serveur.

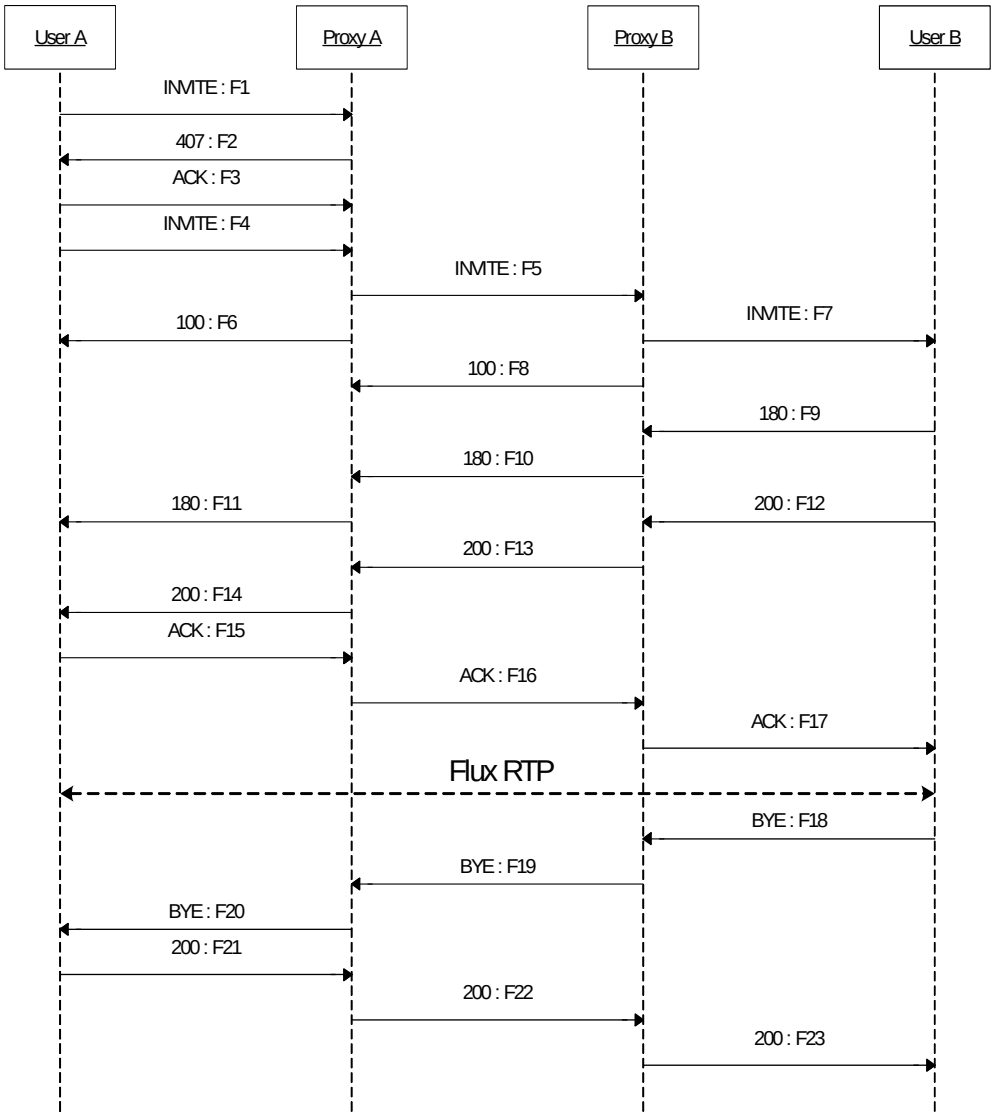


Figure 55 Etablissement d'une connexion RTP en utilisant SIP

A réception d'un message INVITE, l'UAS retournera un message 180 comme réponse temporaire, signifiant que le terminal alerte l'utilisateur en vue de l'établissement de la connexion RTP. Le message INVITE accepté par UserB, un flux RTP peut être établi entre nos deux utilisateurs, en utilisant un format et des paramètres de connexion conformes aux informations véhiculées par le message. Une fois ouverte, les paramètres de raccordement d'un utilisateur à la session peuvent être modifiés (changement du format des données multimédia…) par un de ses participants, en envoyant un nouveau message INVITE. Ce message reprendra l'ensemble des paramètres de la session active, auxquels seront ajoutés les données permettant un détachement et un rattachement de l'utilisateur à la session selon ses nouvelles contraintes.

Les messages CANCEL et BYE également utilisés autour d'une session SIP, permettent respectivement d'abroger le processus d'ouverture d'une session, typiquement suite à la

survenue d'une erreur, et de mettre fin au rattachement d'un utilisateur à une session. En effet, une fois la session ouverte, un utilisateur peut notifier son détachement de la session tout en permettant à la session de perdurer, tout comme un utilisateur extérieur à la session peut en demander son rattachement, en envoyant un message INVITE aux participants de ladite session.

5.4.4 L'accès aux services

Avec la généralisation de l'utilisation des clients légers, nous pouvons imaginer que la plupart des services télécom seront accessibles via un navigateur hébergé sur le terminal de l'utilisateur. Ainsi le fournisseur de services implémente un document XML décrivant les informations à recueillir auprès de l'utilisateur, et la manière de construire le message SIP à destination du service. Ce document XML, accompagné d'une feuille de style est importé sur le terminal de l'utilisateur au moment de l'accès à la page présentation du service.

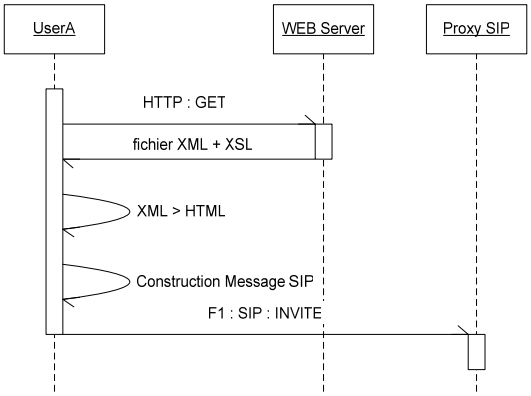


Figure 56 La demande d'accès au service

Le terminal compile ainsi le fichier XML pour générer une page HTML présentant l'accès au service. La page ainsi générée contient une forme (voir encadré ci-dessous) chargée de collecter les données utiles au service .

```
<form action="sip:cardCall@example.com">
...
     <input type="text" name="To">
     <input type="text" name="card_number">
...
     <input type="submit" value="lancer le service">
</form>
```

Figure 57 La page d'accès au service

Une fois la forme validée (sur le clic du bouton de validation "lancer le service"), le service est contacté via l'Uri : sip:cardCall@example.com (se reporter à [CAS01] pour le contrôle du contexte d'exécution de services par l'utilisation d'Uri SIP). SIP n'étant pas à priori un protocole pris en charge par tous les navigateurs WEB, la page de présentation peut être accompagnée d'un script s'exécutant sur le terminal du client, capable de traiter les données entrées par l'utilisateur, afin de construire le message SIP à envoyer au service. Ce script sera capable de réutiliser le document XML importé initialement par le client, pour connaître la sémantique des données recueillies, pour être capable par exemple d'associer la variable To

au champ To du message SIP, et ajouter le champ card_number au contexte d'exécution du service. L'entête du message SIP généré peut ressembler à l'exemple donné ci-dessous :

```
INVITE sip:cardCall@aol.com; card_number="12345678"
SIP/2.0
Via: SIP/2.0/UDP aol.com:5060
Max-Forwards: 70
From: UserA <sip:UserB@aol.com>
To: UserB <sip:UserA@wanadoo.fr>
Call-ID: 87654321@aol.com
CSeq: 1 INVITE
Contact: <sip:UserB@blake.aol.com>
Content-Type: application/sdp
Content-Length: 140

...
```

A ce niveau nous avons utilisé un message SIP INVITE pour véhiculer la signalisation à destination d'un service de mise en relation par carte, la charge utile du message contenant les informations nécessaires à l'établissement de la connexion RTP. Pour ce message INVITE, nous avons utilisé la possibilité offerte par SIP, introduite dans [RLS99], de différencier les champs To et l'Uri request du message. L'intérêt de cet artifice est de permettre un premier routage du message vers notre service **cardCall** au sein du domaine **example.com**, pour ensuite permettre un routage classique vers UserB, une fois le service exécuté.

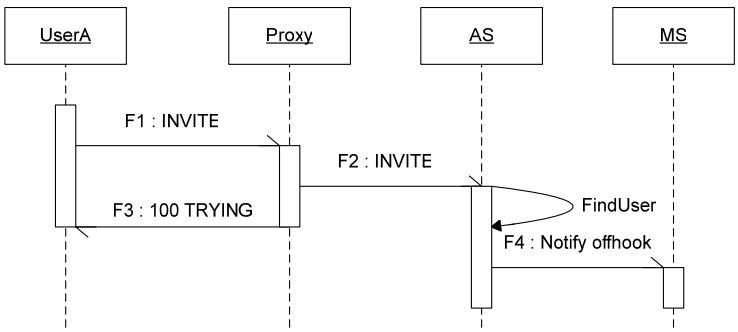


Figure 58 L'accès au Service

Arrivant dans le domaine example.com, le message INVITE sera pris en charge par un proxy, chargé de localiser le serveur d'application disposant d'une implémentation du service d'appel par carte. Si nous admettons dans cette optique que le serveur as_enst, héberge une instance de notre service, localement référencée sous le nom appel_par_carte, le proxy relaiera le message suivant au serveur d'application :

```
INVITE sip:service_carte@as.free.fr;
card_number="12345678" SIP/2.0
Via: SIP/2.0/UDP free.fr:5060;branch=1
Via: SIP/2.0/UDP aol.com:5060
Max-Forwards: 69
From: UserA <sip:UserB@aol.com>
To: UserB <sip:UserA@wanadoo.fr>
Call-ID: 87654321@aol.com
CSeq: 1 INVITE
```

```
Contact: <sip:UserB@blake.aol.com>
Content-Type: application/sdp
Content-Length: 140

...
```

Pendant ce temps, le Proxy retourne un message SIP TRYING, en réponse à F1, pour signifier à l'utilisateur A qu'un traitement est en cours, ce qui permet par exemple de rediriger UserA vers une page de mise en attente spécifique au service exécuté.

```
F3 Trying Proxy -> UserA

100 Trying SIP/2.0
Via: SIP/2.0/UDP free.fr:5060;branch=1
Via: SIP/2.0/UDP aol.com:5060
From: UserA <sip:UserB@aol.com>
To: UserB <sip:UserA@wanadoo.fr>
Call-ID: 87654321@aol.com
CSeq: 1 INVITE
Content-Length: 0
```

A réception de F2, le serveur d'application active une instance du service appel_par_carte en implantant dans son contexte d'exécution le paramètre card_number. L'instance activée peut à ce moment appliquer la logique propre au service.

5.4.5 Une organisation autour de serveurs

Les concepteurs de SIP ont délibérément choisi de ne pas associer le protocole à une architecture physique rigide et par conséquent peut évolutive. Aussi SIP est prévu pour fonctionner autour de nœuds appelés génériquement Serveurs SIP, à l'intérieur desquels sont déployés un ensemble de serveurs logiques. Les trois serveurs logiques frontaux que sont le Proxy Serveur, le Redirect Serveur et le Registrar permettent la prise en charge de serveurs secondaires comme le serveur de localisation des utilisateurs, un serveur DNS, … Un serveur physique tel qu'exposé dans la Figure 59 (le détail des messages SIP est donné en Annexe 2) peut héberger un Registrar, un Proxy et un Location Serveur. Ce serveur physique sera aussi bien mobilisé lors de l'enregistrement d'un utilisateur auprès du réseau (message REGISTER), que pour le traitement du message INVITE et la résolution de son destinataire (interrogation du Location Serveur). A cet effet, SIP laisse la totale liberté d'interprétation sur la relation entre les serveurs frontaux et les serveurs secondaires, dont la variété peut ne pas avoir de limite. Ainsi les messages STORE, QUERY et RESPONSE défini dans notre exemple correspondront à des services fournis par une API "location service" dont l'implémentation peut être assurée par un serveur LDAP ou un composant faisant appel à un simple fichier texte.

Cette architecture totalement ouverte nous permet de définir librement notre propre serveur de composition de services, utilisé comme serveur secondaire d'un proxy serveur, et utilisé lors du traitement de chaque requête INVITE.

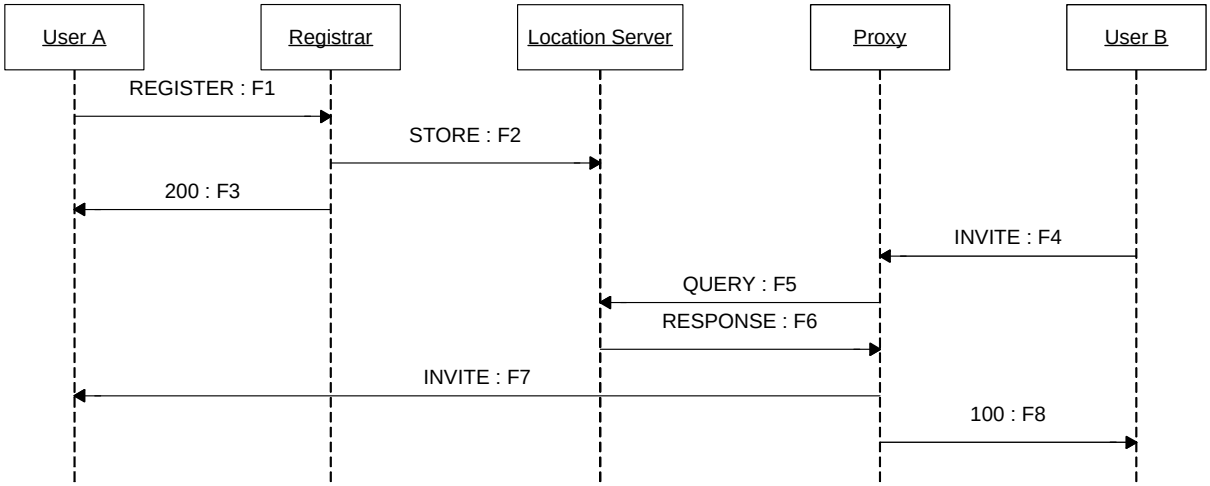


Figure 59 L'utilisation d'un serveur de localisation

5.4.6 SIP comme protocole de présence

L'ouverture de l'architecture autour de SIP fait qu'il est possible d'ajouter le serveur de présence derrière la notion générique de serveur SIP [ROS02] présente les extension à apporter à SIP, pour faire de ce protocole un protocole de présence, capable de prendre en charge le mécanisme d'abonnement d'un watcher auprès du serveur de présence, et la notification de changement d'état d'un presentity. Dans une architecture SIP, un serveur de présente est constitué d'un proxy comme serveur frontal, et de Presence Agent (PA) en services secondaires, capable de traiter les requêtes SIP SUBSCRIBE et NOTIFY, spécialement créées pour l'extension de SIP comme protocole de présence.

Dans cette utilisation de SIP, un utilisateur (`fontaine@wanadoo.fr`) désirant suivre l'état d'un autre utilisateur (`najm@enst.fr`), envoi un message SIP SUBSCRIBE à destination du PA adressable par l'URI `sip:najm@enst.fr`. Ce PA centralise les informations maintenues par les presentities de notre utilisateur, cet utilisateur ayant la possibilité de définir plusieurs PA, comme autant de facettes de son identité. A réception du message SUBSCRIBE, le PA a la possibilité de demander à `afontaine@wanadoo.fr` de s'authentifier, en réutilisant les mécanismes classiques SIP, puis d'accepter ou refuser la demande de souscription d'après les contraintes définies par l'utilisateur. Dans son message de souscription `afontaine@wanadoo.fr` a la possibilité de préciser via le champ `accept`, les formats d'informations de présence qu'il est capable de traiter ([AKL03] et [CAR02] sont des travaux portant sur les formats supportés par SIP), le PA se chargeant de la transformation entre le format d'informations de présence envoyé par `najm@enst.fr` et le format attendu par `fontaine@wanadoo.fr`.
Le message de souscription reçu par le proxy et transféré au PA se formule comme suit :

```
SUBSCRIBE sip:najm@enst.fr SIP/2.0
Via: SIP/2.0/TCP watcherhost.wanadoo.fr;branch=z9hG4bKnashds7
To: <sip:najm@enst.fr>
From: <sip:fontaine@wanadoo.fr>;tag=xfg9
Call-ID: 2010@watcherhost.wanadoo.fr
CSeq: 17766 SUBSCRIBE
Max-Forwards: 70
Event: presence
```

```
Accept: application/cpim-pidf+xml
Contact: <sip:fontaine@watcherhost.wanadoo.fr>
Expires: 600
Content-Length: 0
```

Après traitement du PA, le proxy retournera le message OK 200 suivant, en ayant pris soin d'indiquer ses coordonnées dans le champ Contact :

```
SIP/2.0 200 OK
Via: SIP/2.0/TCP watcherhost.wanadoo.fr;branch=z9hG4bKnashds7;received=10.11.12.13
To: <sip:najm@enst.fr>
From: <sip:fontaine@wanadoo.fr>;tag=xfg9
Call-ID: 2010@watcherhost.wanadoo.fr
CSeq: 17766 SUBSCRIBE
Event: presence
Expires: 600
Contact: sip:proxy.enst.fr
Content-Length: 0
```

Les travaux traitant de l'utilisation de SIP comme protocole de présence ne prévoient pas de faire appel à SIP pour prendre en charge le mécanisme de mise à jour du PA sur changement d'état des PUA. En effet, les PUA n'étant pas à priori normalisés, faute d'accord des protagonistes du secteur, il est difficile de prévoir un message SIP normalisé entre PUA/Presentity et PA. Cependant, le PA dispose du message SIP NOTIFY, véhiculant les informations de présence au format PIDF depuis le PA vers les watchers.

```
NOTIFY sip:afontaine@watcherhost.wanadoo.fr SIP/2.0
Via: SIP/2.0/TCP proxy.enst.fr;branch=z9hG4bKna998sl
To: <sip:fontaine@wanadoo.fr>
From: <sip: najm@enst.fr>;tag=xfg9
Call-ID: 2010@watcherhost.example.com
CSeq: 8776 NOTIFY
Event: presence
Subscription-State: active;expires=543
Max-Forwards: 70
Contact: sip:proxy.enst.fr
Content-Type: application/cpim-pidf+xml
Content-Length: 514

<?xml version="1.0" encoding="UTF-8"?>
<presence  xmlns="urn:ietf:params:xml:ns:pidf"
           entity="pres:najm@enst.fr">
  <tuple id="sg89ae">
    <status>
      <basic>open</basic>
    </status>
    <contact priority="0.8">tel:+33-1-45817709</contact>
  </tuple>
</presence>
```

5.4.7 SIP et nos composants

L'adaptation de SIP aux architectures logicielles orientées objets ou composants étant loin d'être marginale, nous ne nous risquerons pas à émettre une nième proposition dans ce domaine. En revanche nous allons montrer comment nous pouvons réutiliser des solutions

comme GOSIP [GOS03], une implémentation de la spécification Jain SIP Lite [JAS02], pour doter nos composant de capacités d'échange basées sur l'utilisation de SIP. La spécification Jain SIP Lite apporte une API de haut niveau, permettant à une application objet d'émettre ou recevoir des messages SIP, sans avoir à se soucier de détails de bas niveau, comme les numéros de séquence ou autres informations de contrôle de l'intégrité des messages contenues dans l'entête.

D'après le schéma de la Figure 50, relatif à l'incarnation des facettes et réceptacles de nos composants, l'utilisation de cette API peut se faire au niveau de nos stubs et squelettons. Ainsi ces derniers joueront le rôle de "SIP Wrapper", en proposant des mécanismes de traduction d'un appel de méthode en un message SIP et réciproquement. Ce rôle de wrapper assumé par nos stubs et squelettons est complètement en phase avec ce qui est pratiqué dans les architectures RMI/CORBA ou Webservice/SOAP, dans la mesure où le composant sous-jacent doit demeurer indépendant du mode de communication instauré entre les composants.

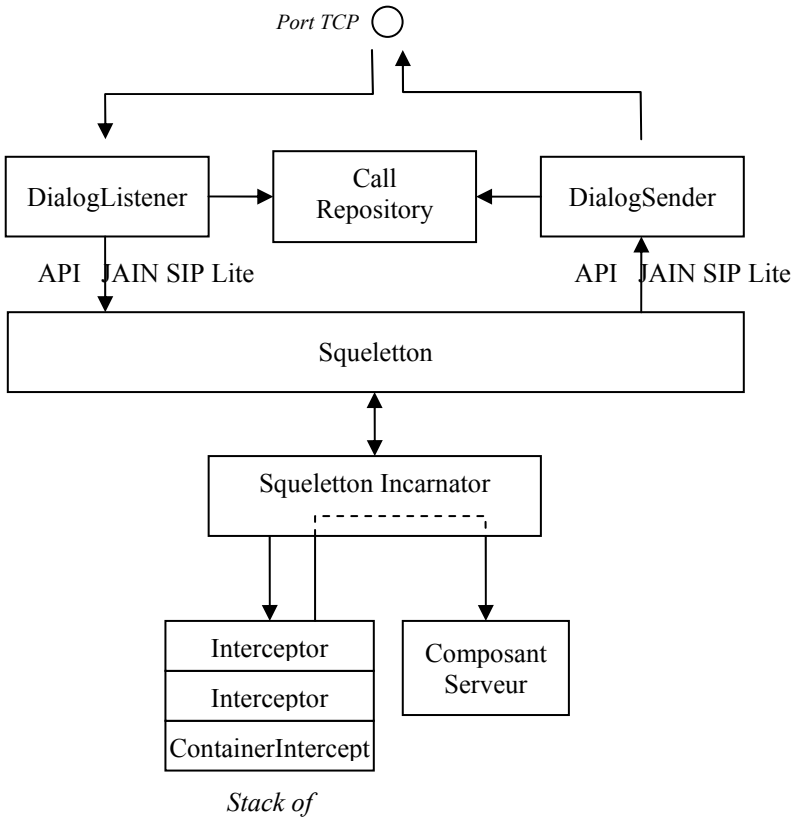


Figure 60 L'utilisation de l'API Jain SIP Lite

La spécification Jain SIP Lite défini deux niveaux d'abstraction des échanges entre client et serveur SIP. Un premier niveau "call" modélise la relation pérenne établie entre deux entités, sur laquelle repose un second niveau "dialog" assimilé cette fois à un échange ponctuel. Un "dialog" est constitué par une série de messages SIP, avec une requête initiale (par exemple INVITE) optionnellement un ensemble de messages intermédiaires (par exemple QUEUED 182) et un retour définitif (par exemple OK 200).

Comme nous l'expose la Figure 60, le point d'entrée d'une application communicant via SIP est marqué par un DialogListener qui écoute un port TCP, dans l'hypothèse de l'arrivée d'un message SIP. A réception d'un message SIP, le DialogListener effectue une opération

”d’unmarshalling”, typique des architectures réparties, pour ”objetiser” et ainsi rendre manipulable aux stubs et squelettons le flot binaire reçu. Le squeletton mentionné dans l’entête du message SIP reçoit à ce moment le message, sous la forme d’un objet `SIPMessage` (spécifié dans Jain SIP Lite), qu’il devra consulter pour déterminer les actions à effectuer sur le composant. Réciproquement une fois le traitement du composant achevé, et son retour communiqué au squeletton, celui-ci doit construire une structure XML, qui constituera le corps du message SIP construit par le DialogSender. Pour ce faire l’interface du DialogSender définie dans la spécification Jain SIP Lite propose au squeletton une méthode :

```
public void createResponse(SIPMessage incomingMessage,
                           String status,
                           String reason,
                           String contentType,
                           Byte[] payload);
```

Comme la signature de cette méthode l’indique, le `DialogSender` est vu comme un service sans état, capable de construire un message de réponse à partir d’un message aller (`incomingMessage`), de prérogatives sur le type de retour escompté (`status` et `reason`) et d’un contenu (`contentType` et `payload`). Si cette méthode permet de construire une réponse à un message SIP en restant à un haut niveau d’abstraction, elle nécessite cependant une prise en charge de certains aspects techniques au niveau du squeletton :

- Le squeletton étant lié à une instance particulière du composant, il lui est possible de conserver le SIPMessage à l’initiative du ”dialog”, pour le
- Maintenir des règles de transformation, permettent d’interpréter la structure XML du message initial, pour appeler un traitement du composant.
- Maintenir des règles de transformation, permettent la construction d’une la structure XML à partir des données retournées par le composant.
- Gérer les règles fonctionnelles permettant de déterminer le type de message SIP (1XX, 2XX, …) à retourner en fonction du comportement du composant.

La construction du message SIP de retour est ainsi effectuée par le DialogSender, sur la base du ”dialog”, et compte tenu des données spécifiques au ”call”. A cet effet, Les DialogListener et DialogSender doivent gérer une CallRepository, leur permettant de maintenir les informations de bas niveaux, qui resteront inaccessibles aux composants.

La prise en charge de SIP du coté d’un stub fait appel à la même architecture, avec néanmoins une permutation dans le séquencement entre DialogListener et DialogSender. En effet, le stub étant associé à un réceptacle de collaboration, il aura l’initiative de la création d’un ”dialog” avec un composant distant. Pour ce faire il commencera donc par utiliser le DialogSender, pour l’envoi de son message initial, puis attendra une notification de la part du DialogListener. L’utilisation du DialogSender se fait grâce à l’utilisation de la méthode :

```
public void createRequest(String method,
                          String contentType,
                          Byte[] payload);
```

Comme le squeletton, le stub doit construire de sa propre initiative le corps du message SIP à envoyer. De plus, la méthode générique proposée par le DialogSender permet de préciser le type du message SIP à envoyer (INVITE, NOTIFY, SUBSCRIBE , OPTIONS, …) ce dernier assumant de façon autonome le mécanisme de construction adapté au message demandé.

5.5 La PA-architecture

5.5.1 Vue d'ensemble

D'après nos objectifs, le modèle de présence introduit dans [LRS00] présente trois limitations. Tout d'abord la signalisation entre application utilisatrice et PUA est exclusivement destinée à la l'expression de l'état des ressources de l'utilisateur. Pour être pleinement utilisable, l'architecture de présence doit proposer une relation application - PUA enrichie, permettant au PUA d'outrepasser son simple rôle consultatif et ainsi d'interagir avec les applications du terminal. Ensuite disposer d'un unique PUA pour superviser l'état global du terminal, donc par transitivité l'état des ressources du terminal, n'est pas d'une granularité adaptée au contrôle des services du terminal depuis les entités hébergées dans le domaine du Détaillant. Enfin, si un presentity permet d'unifier les différentes ressources de l'utilisateur, et apporte ainsi la notion d'utilisateur universel (concept central des réseaux de troisième génération, l'architecture de présence n'offre pas la possibilité à l'utilisateur de définir des profiles de présence, fonctionnalité qu'apporte le PA dans [ROS02].

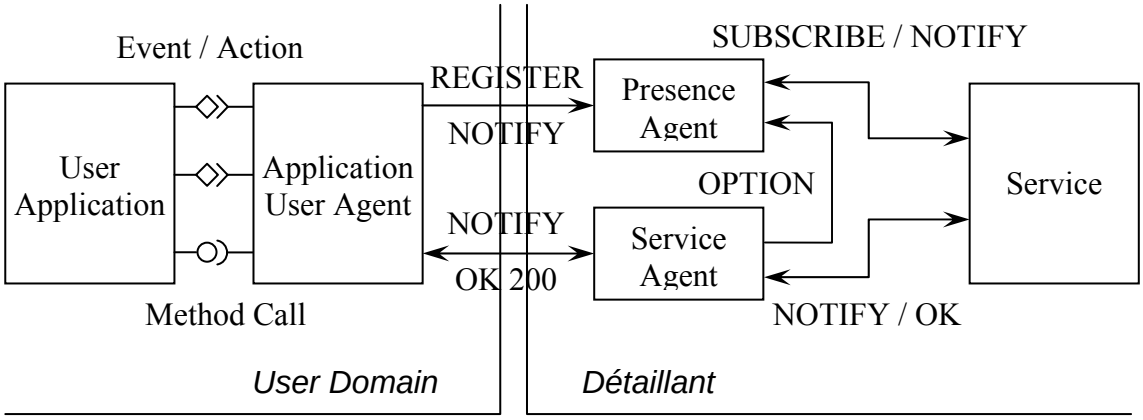


Figure 61 Notre architecture de présence

Fort de ces trois limitations nous avons conçu une architecture à quatre niveaux (cf. Figure 61), généralisant celle du service de présence, répartie sur les domaines fonctionnels de l'utilisateur et du Détaillant. Dans le domaine utilisateur, délimité par son terminal, nous retrouvons les applications clientes directement utilisées par l'utilisateur, accompagnées d'Application User Agent (AUA). Ces agents, séparés des applications, jouent le rôle d'interfaces entre les applications et les services implantés sur le réseau. Ils sont chargés d'une part de capter les événements émanant des applications pour le cas échéant les associer à des actions ou des changements de statut. D'autre part ils sont utilisés par les services du Détaillant comme point d'entrée vers les applications du terminal. Dans ce dernier cas les AUA présentent aux services une interface d'accès aux applications standardisée, les AUA ayant la responsabilité d'assumer la traduction entre requêtes standards et opérations spécifiques aux applications.

Dans le domaine du Détaillant, outre les services nous retrouvons un Presence Agent et un ensemble de Service Agent (SA) dédiés à un utilisateur en particulier. Dans cette relation triangulaire, le PA agrège et rend disponible les informations de présence, relatives à un utilisateur, aux autres éléments architecturaux. Pour ouvrir l'architecture de présence à la problématique de composition de services, nous avons introduit la notion de *statut* des applications utilisatrices, maintenu au niveau des AUA. Avec cette approche, l'état de présence considéré par le PA correspond à la réunion des statuts particuliers des applications,

interrogeables par les services et Service Agent. Le SA est une entité liée à l'utilisateur, ayant une connaissance de la structure externe des services. En jouant le rôle de client des services, un SA doit faire la relation entre une action demandée par une application utilisateur, par l'intermédiaire d'un AUA, et les méthodes disponibles sur les différentes facettes du service auquel il est lié.

Mis à part les applications clientes, le plus souvent réduites à l'état de simple IHM, les différents éléments de notre architecture peuvent être implémentés sous la forme composants, aux facettes et réceptacles de collaboration dotés de capacités de traitement de messages SIP. Les services sont assumés par un framework de composants session (cf partie 3.5.5), indépendants des données des utilisateurs. Les SA, regroupant les intérêts d'un utilisateur pour un service donné, sont implémentés sous la forme de composants utilisateurs. De même, le PA est un composant dont les facettes permettent aux AUA et aux services d'accéder aux informations de présence, et les réceptacles de notifier aux services le changement d'état d'un utilisateur. L'AUA enfin est un composant particulier, offrant facettes et réceptacles aux composants du Détaillant, mais accédant en direct aux applications déployées sur le terminal.

5.5.2 Présentation détaillée de l'architecture

5.5.2.1 La relation application-AUA

5.5.2.1.1 La gestion de la signalisation associée au fonctionnement des applications

Nous appelons application un bloc logiciel implanté sur le terminal de l'utilisateur, constitué exclusivement d'éléments de présentation. D'après le système de classes associé au langage d'implémentation de notre application, les éléments de présentation sont constitués d'un certain ensemble d'événements et de méthodes à la base du comportement de l'application. Ainsi doter une application de capacités de communication avec les services du réseau revient à définir des événements au niveau de l'application qui devront être captés par des AUA. Ces événements applicatifs ayant pour vocation d'interagir avec les services coté Détaillant, ils devront prendre comme argument la totalité des informations nécessaires à l'exécution du service. En outre la définition de ces événements demande de corréler la production de l'événement à une action effectuée en amont sur l'application. La construction d'une application cliente demande ainsi d'identifier les processus qui resteront locaux à l'application, et réciproquement les opérations qui génèreront la production, et donc la communication avec des services du réseau.

Si nous reprenons notre exemple du tableau blanc, le déplacement de la souris sur la surface du tableau, le drag d'un élément à ajouter au tableau et son déplacement sur le tableau sont des événements reliés à des opérations qui restent limitées à l'application cliente. En revanche, nous pouvons imaginer que le drop de ce même élément sur la surface du tableau, provoque une modification de l'apparence du tableau qui devra être répercutée sur les tableaux maintenus par les applications clientes distantes, en passant par le service de synchronisation du Détaillant. Synchroniser l'ajout de l'élément sur les tableaux distants demande la création d'un événement d'ajout d'élément, ayant pour attribut l'élément à reconstituer à distance.

```
<application>
  <name>WhiteBoard</name>
```

```
  <events>
    <event event-name="create"
           event-type="fr.enst.whiteBoard.CreateElementEvent">
      <event-attribute type="fr.enst.whiteBoard.Element"
                       name="element"/>
    </event>
  </events>

  <handlers>
    <handler handler-name="handler1"
             event-name="create"
             class="fr.enst.whiteBoard.ElementCreationHandler"/>
  </handlers>
  <sources>
    <source class-name="fr.enst.whiteBoard.BoardFrame"
            method-name="add"
            event-name="create">
      <method-arg type="fr.enst.whiteBoard.Element"
                  name="arg1"/>
      <mapping event-attribute="element"
               source-attribute="arg1"/>
    </source>
  </sources>
  <listeners>
    <aua class-name="fr.enst.whiteBoard.WhiteBoardUA"
         method-name="createElement">
      <binding handler-name="handler1"
               receptacle-name="WhiteBoardSA"
               output-method="addElement"/>
    </aua>
  </listeners>
</application>
```

Figure 62 La relation entre applications et événements à destination des services

Pour permettre à l'utilisateur de personnaliser le comportement de ses applications, nous avons fait le choix d'inclure les correspondances entre action et événement au sein d'un fichier de configuration XML, dont un exemple est donné dans la Figure 62. Grâce à ce fichier de configuration il est possible de définir :

- De nouveaux événements et les informations qu'ils véhiculent (tag `<event>`).
- Les gestionnaires d'évènements dédiés aux événements nouvellement créés (tag `<handler>`), utilisables par les AUA pour capter les actions effectuées sur les applications du terminal.
- Les sources d'événements (tag `<source>`), caractérisées par un type et une méthode signée.
- Les correspondances entre les attributs des événements et les arguments ou variables utilisables au niveau de la source d'événement (tag `<mapping>`).

La gestion de ce fichier de configuration peut être abordée de deux façons antagonistes. La première, que nous pourrons appeler statique, consiste à anticiper les événements définissables par l'utilisateur, manipulés à l'état latent par l'application. L'ajout de ces événements dans le fichier de configuration ne fait dans ce cas qu'activer une partie du code qui était auparavant non opérationnelle. La seconde approche plus dynamique, que nous

allons présenter plus en détail, consiste à recompiler le code de l'application déployée sur le terminal, tel qu'il est commun de le faire avec le noyau du système d'exploitation Linux.

Une recompilation du code d'une application déployée sur le terminal, suite à une modification de son fichier de configuration, va générer un ensemble de classes utiles à la prise en charge de l'événement nouvellement créé, que nous expliciterons en utilisant Java.

La première de ces classes sera la classe événement elle-même. Chaque tag `<event>` génère une classe héritant de la classe générique d'événement (en l'occurrence `EventObject` en Java), dont le nom sera équivalent à l'attribut XML `event-type`. A cette classe nous ajouterons autant d'accesseur que le tag `<event>` contient de fils `<event-attribute>`. La classe événement générée à partir de notre fichier de configuration aura ainsi la structure :

```
package fr.enst.whiteBoard;

public class CreateElementEvent extends EventObject
{
  private Element element;

  public CreateElementEvent(Object obj)
  {
    super(obj);
  }

  public Element getElement()
  {
    return this.element;
  }

  public void setElement(Element element)
  {
    this.element = element;
  }
}
```

Une fois les événements créés, la régénération du code doit s'appliquer aux sources des événements, pour prendre en charge la création des événements au sein des méthodes précisées dans le fichier de configuration. Ainsi chaque classe mentionnée dans un tag `<source>` sera étendue et ses références manipulées dans des classes tierces seront remplacées par cette classe spécialisée. Ainsi la classe `fr.enst.whiteBoard.BoardFrame` citée dans notre fichier de configuration sera étendue avec une classe `ExtendedBoardFrame`. Cette nouvelle classe défini un nouvel attribut de type `CreateElementEvent`, et de nom `"create"`, en rapprochement avec l'attribut XML `event-name`. Associé à cet événement, la classe devra créer également un écouteur d'événement, utilisé par le handler pour connecter sources et récepteurs d'événements. Notre classe contiendra ainsi un attribut `"listener"` de type `CreateElementListener`, accompagné de la méthode setListener(), permettant au récepteur d'événement (l'AUA) de s'abonner à l'événement `create`. Enfin la méthode `add(Element)` est étendue pour initier l'événement qui sera notifié au récepteur enregistré. Le code de cette classe générée sera ainsi :

```
package fr.enst.whiteBoard;

public class ExtendedBoardFrame extends BoardFrame
```

```
{
  public  CreateElementEvent    create;
  private CreateElementListener listener;
  public ExtendedBoardFrame()
  {
    super();
  }

  public void add(Element element)
  {
    create = new CreateElementEvent(this);
    create.setElement(element);
    listener.createElement(create);
    super.add(element);
  }

  public void setListener(CreateElementListener listener)
  {
    this.listener = listener;
  }
}
```

Finalement le processus de régénération de l'application va se concentrer sur les agents des applications (AUA). Les cycles d'exécution des AUA et des applications utilisateur peuvent être synchronisés ou non. Dans la négative, les AUA sont démarrés avec la session utilisateur, un service du terminal assure à ce moment l'indexation des AUA utilisables par les applications. Lors de son démarrage, une application doit ainsi localiser son AUA, pour que ce dernier puisse être à même de capter les événements provenant d'actions utilisateur. Si par contre les cycles d'exécution étaient synchronisés, applications utilisateur et AUA seraient démarrés conjointement, l'AUA étant instancié avec une référence directe à l'application utilisateur.

Un tag `<aua>` produit un agent d'application, dont le type est équivalent à l'attribut XML `"class-name"`. Pour chaque tag fils `<binding>` du tag `<aua>`, le constructeur de notre agent demande à l'event handler mentionné dans l'attribut XML `"handler-name"`, d'écouter pour lui l'événement auquel il est dédié, et de lui communiquer sur une méthode de son choix la notification de survenue de l'événement. Le tag `<binding>` permet enfin d'assurer la correspondance entre l'événement observé, et la méthode appelée sur un des réceptacles de l'AUA, connectés à des composants du Détaillant. De fait, quand l'utilisateur ajoutera un élément graphique à son tableau blanc, l'appel à la méthode `add()` sur la classe `ExtendedWhiteBoard` produira un appel de la méthode addElement() sur le composant connecté au réceptacle WhiteBoardSA (en l'occurrence le Service Agent du service de synchronisation du tableau blanc. La prise en charge de notre événement sera ainsi possible en utilisant le code suivant :

```
public WhiteBoardUA(ExtendedWhiteBoard whiteBoard)
{
     whiteBoard.create = new ElementCreationHandler(createElement);
     ...
}
private void createElement(Object sender, CreateEventArgs sent)
{
  Broker broker = new Broker();
  Object ref = broker.resolve("WhiteBoardSA");
```

```
  WhiteBoardAUASAFacet fac = WhiteBoardAUASAFacetHelper.narrow(ref);
  fac.addElement(sent);
}
```

Figure 63 La prise en charge des événements au niveau de l'AUA

Lors du démarrage de l'application cliente, un AUA (WhiteBoardUA) dédié à l'application tableau blanc sera instancié en faisant appel au constructeur `WhiteBoardUA(WhiteBoard)`. La référence passée en paramètre du constructeur permet de lier l'appel à la méthode `createElement(Object, CreateEventArgs)` de l'instance de `WhiteBoardUA` à la levée de l'événement `create` sur l'instance de `WhiteBoard`. La méthode `createElement` fait ensuite appel au service broker local au terminal, pour résoudre et ainsi obtenir une référence vers la facette du composant connecté au réceptacle `WhiteBoardSA`. En revenant à l'environnement d'exécution de nos composants exposé dans la Figure 60, l'appel à la méthode `addElement(sent)` sur le réceptacle de l'AUA sera concrètement associé à la production d'un message SIP, à l'initiative du stub utilisé en aval de la classe d'implémentation de notre composant AUA. Le stub chargé initialement par le composant de véhiculer l'appel de méthode jusqu'au composant serveur se rapprochera du code suivant :

```
public WhiteBoardAUASAFacetStub()
{
  ...
}
private void addElement(CreateEventArgs sent)
{
  XMLElement = ElementHelper.toXml(sent.getElement());
  XMLAddElement = WhiteBoardAUASAFacetHelper.addElement(XMLElement);
  payload = XMLAddElement.toBytes();
  dialogSender.createRequest("NOTIFY", "application/xml", payload);
}
```

Figure 64 L'envoi du message SIP par le Stub

Comme nous le précisions dans la partie 5.4.7, le stub a la responsabilité de la construction de la charge utile du message SIP, associé à l'appel distant exigé (cf. Figure 69). Cette charge utile est constituée par une enveloppe XML explicitant l'appel à la méthode distante `addElement(Element)`. Pour construire cette enveloppe XML nous nous appuyons sur deux classes utilitaires, d'une part une classe `ElementHelper` qui produit une représentation XML de l'élément ajouté au tableau blanc (récupéré en passant par l'accesseur `getElement()` sur l'événement `sent`), et d'autre part une classe qui produit structure XML d'un appel de méthode, tel que pratiqué par SOAP.

```
XMLElement = ElementHelper.toXml(sent.getElement());
XMLAddElement = WhiteBoardAUASAFacetHelper.addElement(XMLElement);
```

Cette hypothèse suppose que les règles de projection de nos éléments de tableau blanc soient communes aux applications du terminal et aux services du Détaillant, pour l'interprétation des données. Enfin la méthode `addElement()` se poursuit avec la création du message et son envoi vers le Service Agent par le biais du `dialogSender` en lien avec notre stub.

5.5.2.1.2 La gestion du statut des applications

La gestion du statut des applications du terminal reprend le modèle utilisé pour la signalisation entre applications et services. Pour ce faire le fichier de configuration introduit dans la Figure 62 sera complété pour déclarer :

- Les sources d'événement implantées sur l'application.
- Les gestionnaires d'événement utilisables par les AUA.
- Les événements de statut observés par les AUA.
- Les changements de statut notifiés au Presence Agent.

Après avoir défini les différents événements exploitables par l'application, notre fichier de configuration délimite sur les applications du terminal, les actions utilisateur qui occasionneront la production de ces événements. L'AUA constitue la pièce maîtresse de ce processus. En effet, c'est lui qui sélectionne les événements de statut qu'il souhaite écouter, parmi les événements écoutables. C'est lui également qui assure la translation entre les événements de statut émanant des applications, et les notifications de changement de statut adressées au Presence Agent (PA), implanté dans le domaine du Détaillant.

Dans notre architecture de présence le Presence Agent reprend à la fois les rôles de location et de presence serveur, en généralisant néanmoins son spectre d'activité à l'ensemble des applications présentes sur les terminaux de l'utilisateur. Il est ainsi utilisé par les services du Détaillant pour localiser et connaître l'état des applications de service dans un contexte particulier de vie de l'utilisateur. En effet, concrètement implémenté sous la forme d'un composant utilisateur, le PA est identifiable par une adresse logique SIP du type `SIP:afontaine@wanadoo.fr`, afin d'accéder aux applications que l'utilisateur physique `arnaud.fontaine@mortimer.enst.fr` veut bien rendre disponible au monde extérieur. Dans cette optique, l'AUA assure l'articulation entre la notion d'utilisateur physique, correspondant à la situation réelle de l'utilisateur, et la notion d'utilisateur logique, maintenue par le Presence Agent.
Si dans le monde réel la notion d'utilisateur physique est clairement définie; dans l'univers des architectures distribuées nous appellerons utilisateur physique un couple (identifiant de session, identifiant de terminal) correspondant à une identité authentifiée au niveau d'un terminal. Suivant cette propriété nous faisons l'hypothèse que l'utilisateur a la possibilité d'enregistrer ses identités logiques au niveau du terminal, pour que ce dernier instancie autant d'instances de composants AUA par application que d'identités logiques connue, afin de communiquer le statut des applications qu'il héberge aux différents PA de l'utilisateur physique. Pour ce faire un AUA externalise un réceptacle sur lequel se connectera la facette du PA hébergeant les méthodes de mise à jour du statut des applications de l'utilisateur. La Figure 65 illustre ce mécanisme au moyen d'un terminal hébergeant deux applications. Sur ce terminal l'utilisateur a pu s'authentifier en tant que *arnaud.fontaine@mortimer.enst.fr*, et ainsi ouvrir une session au niveau du système d'exploitation. Admettons que notre utilisateur soit en lien avec deux Détaillants, un privé et un professionnel, associés aux domaines respectifs *enst.fr* et *wanadoo.fr*, qui lui permettront d'assumer deux identités logiques : *afontaine@enst.fr* et *gabolaud@wanadoo.fr*.

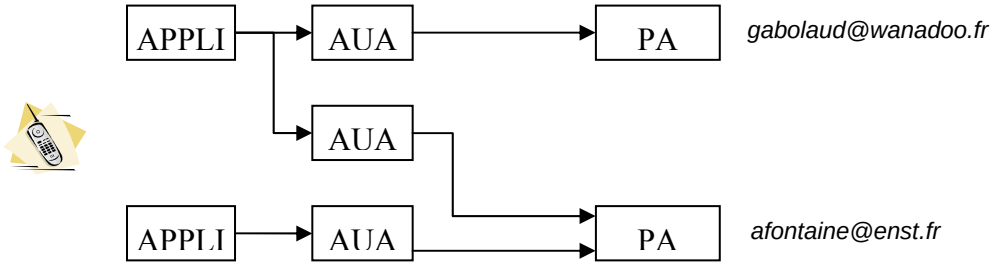


Figure 65 La relation entre AUA et PA

A l'ouverture de la session du terminal, les différents AUA vont devoir entrer en relation avec leur PA respectif, pour mettre à jour l'état de l'application à laquelle ils sont dédiés. En fonction des souscriptions aux événements de statut, les AUA observeront ou n'observeront pas les actions de l'utilisateur, qui pourront occasionner un changement d'état de l'application auprès du PA. L'état réel d'une application pouvant différer de son état relatif connu par le PA, chaque AUA peut décider de l'information qu'il produira réellement pour le PA.

Le fichier de configuration de la relation entre application et AUA commence par déclarer les événements de statut, communs aux différentes applications du terminal, pris en charge par des gestionnaires d'événements, spécifiques cette fois aux applications. Il précise ensuite pour chaque application, les sections du code à l'origine de la production d'événements de statut, avant de lister les méthodes de notification d'événement sur les AUA (attribut XML `method-name`), qu'il liera avec les gestionnaires d'événement de statut précédemment définis. En dédiant chaque gestionnaire d'événement à un événement de statut particulier (`"connecting", "closing", "drawing"`), notre construction introduit une certaine finesse de gestion des événements permettant à un AUA de sélectionner les événements qu'il souhaitera remonter jusqu'au PA de l'utilisateur. Ainsi l'ajout d'un tag `<mapping>` génèrera un code semblable à celui déjà évoqué dans la Figure 63, liant l'AUA à la source de l'événement de par l'utilisation d'un "event handler", et construisant un message SIP NOTIFY cette fois, à destination du PA de l'utilisateur. Si les applications $APPLI_1$ et $APPLI_2$ mentionnées dans la Figure 65 correspondent respectivement aux services whiteBoard et iPhone les configurations des différents AUA peuvent prendre la forme suivante :

```
<application>
  <events>
    <event event-name="connecting"
           event-type="fr.enst.common.StatusEvent">
      <event-attribute type="String"
                       name="status"/>
    </event>
    <event event-name="closing"
           event-type="fr.enst.common.StatusEvent">
      <event-attribute type="String"
                       name="status"/>
    </event>

    <event event-name="drawing"
           event-type="fr.enst.common.StatusEvent">
      <event-attribute type="String"
                       name="status"/>
    </event>
  </events>
```

```
<handlers>
  <handler handler-name=”handler1”
           event-name=”connecting”
           class=”fr.enst.whiteBoard.StatusHandler”/>
  <handler handler-name=”handler2”
           event-name=”closing”
           class=”fr.enst.whiteBoard.StatusHandler”/>


  <handler handler-name=”handler3”
           event-name=”drawing”
           class=”fr.enst.whiteBoard.StatusHandler”/>
  <handler handler-name=”handler4”
           event-name=”hangup”
           class=”fr.enst.iPhone.StatusHandler”/>
  <handler handler-name=”handler5”
           event-name=”hangdown”
           class=”fr.enst.iPhone.StatusHandler”/>
</handlers>
<sources>
  <source class-name=”fr.enst.whiteBoard.BoardFrame”>
          method-name=”online_click”
          event-name=”connecting”>
    <method-arg type=”System.Object”
                name=”sender”/>
    <method-arg type=”System.EventArgs”
                name=”sent”/>
    <mapping event-attribute=”status”
             source-attribute=”connecting”/>
  </source>
  <source class-name=”fr.enst.whiteBoard.Application”>
          method-name=”exit”
          event-name=”closing”>
    <mapping event-attribute=”status”
             source-attribute=”closing”/>
  </source>
  <source class-name=”fr.enst.whiteBoard.DragSourceDragEvent”
          method-name=”startDrag”
          event-name=”drawing”>
    <mapping event-attribute=”status”
             source-attribute=”drawing”/>
  </source>
  <source class-name=”fr.enst.iPhone.KeyBoardFrame”>
          method-name=”send_click”
          event-name=”hangup”>
    <method-arg type=”System.Object”
                name=”sender”/>
    <method-arg type=”System.EventArgs”
                name=”sent”/>
    <mapping event-attribute=”status”
             source-attribute=”hangup”/>
  </source>
  <source class-name=”fr.enst.iPhone.KeyBoardFrame”>
          method-name=”cancel_click”
          event-name=”hangdown”>
    <method-arg type=”System.Object”
                name=”sender”/>
    <method-arg type=”System.EventArgs”
                name=”sent”/>
```

```
      <mapping event-attribute="status"
               source-attribute="hangdown"/>
    </source>
  </sources>
  <listeners>
    <aua class-name="fr.enst.whiteBoard.WhiteBoardUA"
         method-name="statusChanged"
         agent-id="enst.fr">
      <binding handler-name="handler1"
               receptacle-name="WhiteBoardPA"
               notify-status="en ligne"/>
      <binding handler-name="handler2"
               receptacle-name="WhiteBoardPA"
               notify-status="hors ligne"/>
      <binding handler-name="handler3"
               receptacle-name="WhiteBoardPA"
               notify-status="en cours de dessinage"/>
    </aua>
    <aua class-name="fr.enst.iPhone.IPhoneUA"
         method-name="statusChanged"
         agent-id="enst.fr">
      <binding handler-name="handler4"
               receptacle-name="WhiteBoardPA"
               notify-status="décroché"/>
      <binding handler-name="handler5"
               receptacle-name="WhiteBoardPA"
               notify-status="raccroché"/>
    </aua>
    <aua class-name="fr.enst.iPhone.IPhoneUA"
         method-name="statusChanged"
         agent-id="wanadoo.fr">
      <binding handler-name="handler4"
               receptacle-name="WhiteBoardPA"
               notify-status="online"/>
      <binding handler-name="handler5"
               receptacle-name="WhiteBoardPA"
               notify-status="offline"/>
    </aua>
  </listeners>
</application>
```

5.5.2.1.3 Le retour des services vers les applications

Dans un contexte de retour des services vers les applications, l'AUA présente un ensemble de réceptacles standards, comme points d'entrée des services du réseau vers les applications du terminal. A ce titre l'AUA a la responsabilité de la réception, l'interprétation et l'exécution de la requête applicative induite par les messages SIP reçus. Ces messages SIP peuvent être de deux types, avec d'un coté les retours classiques SIP (1xx, à 6xx cf. partie 5.4.3) et de l'autre les demandes d'exécution (message SIP NOTIFY) émanant des services. Si ces deux types de messages se traduisent par un appel de méthode classique sur l'application du terminal, afin d'appliquer les requêtes émanant du service du Détaillant, ils peuvent également être associés à une mise à jour du statut de l'application au niveau du PA.

5.5.2.2 La relation AUA-PA

Agents Utilisateur des Applications (AUA) et Agents de Présence (PA) de l'utilisateur sont liés par une relation SIP de type REGISTER/NOTIFY. Ainsi chaque AUA mentionné dans le fichier de configuration des applications du terminal va entrer en relation avec le Presence Agent précisé dans l'attribut XML `agent-id`, pour au préalable enregistrer sa localisation et son état initial, puis mettre à jour le statut de l'application au gré des actions utilisateur. Pour ce faire, les PA manipulables par les AUA peuvent être déclarés au sein d'un fichier de configuration annexe :

```
<presence>
    <agents>
        <agent agent-id=″agent1″
               domain=″wanadoo.fr″
               host=″msjupiter″
               user=″gabolaud″/>
        <agent agent-id=″agent2″
               domain=″enst.fr″
               host=″infres″
               user=″afontaine″/>
    </agents>
    <applications>
        <application app-name=″WhiteBoard″>
            <reference agent-id=″agent2″
                       ref-id=″board″/>
        </application>
        <application app-name=″IPhone″>
            <reference agent-id=″agent1″
                       ref-id=″comApp″/>
            <reference agent-id=″agent2″
                       ref-id=″cellPhone″/>
        </application>
    </applications>
</presence>
```

Après avoir concentré les informations utiles à la localisation des Presence Agent (`host, domain, user`), ce fichier de configuration assure par application la correspondance entre nom local et nom dans le domaine du Détaillant. Ainsi notre application de tableau blanc, connue localement sous le nom `″WhiteBoard″` fait partie d'un service offert par le Détaillant, connu de lui sous le nom `″board″` dans le domaine `″enst.fr″`. Compte tenu de ce fichier de configuration, **au démarrage de la session utilisateur**, le terminal enregistre ses différentes applications au moyen de messages SIP REGISTER dont le contenu est donné ci-dessous :

```
REGISTER sip:infres.enst.fr SIP/2.0
From: Arnaud <sip:afontaine@mortimer.enst.fr>;tag=a73kszlfl
To: afontaine@enst.fr
Expires: 1800
Call-ID: 123456789@enst.fr
CSeq: 1 REGISTER
Contact: <sip:WhiteBoard@mortimer.enst.fr;application=board>
Content-Length: 0
```

Le message REGISTER est exclusivement destiné à référencer la localisation actuelle de l'application de l'utilisateur auprès du PA. Grâce au fichier de configuration de présence, le

message sera adressé directement au serveur hébergeant les composants PA du domaine du Détaillant.

```
REGISTER sip:infres.enst.fr SIP/2.0
```

Les champ `From` et `To` du message `REGISTER` sont utilisés pour assurer la correspondance entre utilisateur physique et utilisateur logique. Le champ `From` précise ainsi le contexte de session par lequel l'application de service est joignable sur le terminal, alors que le champ `To` indique au PA l'utilisateur logique à considérer. A réception du message REGISTER, le DialogListener présente le contenu du message au composant dont l'identité correspond au champ `To`. Enfin le champ `Contact` commence par mentionner les informations de localisation de l'application de service, accompagnées de son nom interprétable par le PA. Comme tout message SIP, il fait l'objet d'une réponse :

```
SIP/2.0 200 OK
From: Arnaud <sip:afontaine@mortimer.enst.fr>;tag=a73kszlfl
To: sip:afontaine@enst.fr
Call-ID: 123456789@enst.fr
CSeq: 1 REGISTER
Contact: <sip:WhiteBoard@mortimer.enst.fr;application=board>
Contact: <sip:TableauBlanc@blake.enst.fr;application=board>
Content-Length: 0
```

Outre cette nouvelle localisation, la réponse du PA comprendra, pour l'application `"board"` et l'utilisateur logique considérés, la liste synthétisant les localisations connues par le PA, pouvant faire l'objet de sessions de service. Au démarrage, ou plus généralement lors de modifications du statut de l'application `WhiteBoard`, l'AUA peut notifier son nouveau statut au PA en s'appuyant sur le message SIP NOTIFY :

```
NOTIFY sip:infres.enst.fr SIP/2.0
From: Arnaud <sip:afontaine@mortimer.enst.fr>;tag=a73kszlfl
To: afontaine@enst.fr
Call-ID: 123456789@enst.fr
CSeq: 8776 NOTIFY
Event: application-state
Subscription-State: active;expires=12548
Max-Forwards: 70
Content-Type: application/cpim-pidf+xml
Content-Length: 299

<?xml version="1.0" encoding="UTF-8"?>
<app:presence xmlns:app="http://www.ietf.org/xml/ns/pidf"
              entity="pres:afontaine@enst.fr">
  <app:tuple id="aua1_sg89ae">
    <app:status>
      <app:basic>en ligne</app:basic>
    </app:status>
    <pres:contact
priority="1">WhiteBoard@mortimer.enst.fr</pres:contact>
  </app:tuple>
</app:presence>
```

Par rapport aux implémentations du service de présence dont fait partie [ROS02], assumer la correspondance au niveau de l'AUA, entre les événements locaux à l'application et les statuts des applications de l'utilisateur maintenus par le PA, nous permet de concevoir des messages standards de notification d'état de l'AUA vers le PA. Comme pour le message REGISTER, le

message NOTIFY utilise les champ `From` et `To` pour assurer la correspondance entre utilisateur physique et logique. Le champ `Event` précise la nature de la notification, ici en l'occurrence `application-state` nous informe que le corps du message contient des informations se rapportant à l'état d'une application. Le champ `Content-Type` nous indique enfin la structure attendue pour le corps du message, `application/cpim-pidf+xml` établissant l'utilisation d'un corps XML dérivé du modèle cpim-pidf introduit dans [AKL03]. Ce corps de message repose sur une structure en `<tuple>`, chaque occurrence étant caractérisée par le nouvel état `<app:status>` de l'application correspondant au tag `<pres:contact>`.

A réception de ce message SIP NOTIFY, la problématique du PA est de rapprocher les informations reçues avec les applications utilisateur référencées. Pour ce faire, chaque composant PA peut maintenir une table de connaissance ayant la structure :

Application	User	Session	Local Name	Status
board	afontaine	afontaine@blake.enst.fr	TableauBlanc@blake.enst.fr	hors ligne
board	afontaine	afontaine@mortimer.enst.fr	WhiteBoard@mortimer.enst.fr	en ligne
cellPhone	afontaine	afontaine@mortimer.enst.fr	IPhone@mortimer.enst.fr	raccroché

La colonne `Application` reprend les noms génériques des applications connus en tant que tel par les services du Détaillant. La colonne `User` mentionne le nom logique de l'utilisateur alors que la colonne `Session` relève le nom physique, utilisé pour différentier les enregistrements des différents instances d'application apparaissant dans la colonne `Local Name`. Le PA doit maintenir un statut pour chacun de ces enregistrements, que notre message NOTIFY va mettre à jour dès que nécessaire. La détermination de l'enregistrement à mettre à jour passe par l'utilisation de l'attribut `entity` du tag `<app:presence>` à rapprocher avec la colonne `User` des données du PA, et du tag `<pres-contact>` équivalent à la colonne `Local Name`. La colonne `Status` est alors mise à jour simplement d'après le contenu du message NOTIFY. Cette information peut être réduite à sa plus simple expression, comme dans notre exemple de message NOTIFY :

```
<app:status>
  <app:basic>en ligne</app:basic>
</app:status>
```

Dans ce cas le PA ne fait que reprendre la valeur `"en ligne"` réceptionnée, dans l'attente de mises à jour ultérieures de la part de l'AUA. Cette définition des statuts des applications peut cependant être perfectionnée, pour tenir compte des spécificités propres à chaque application. En effet, une application de travail collaboratif peut par exemple gérer des statuts intermédiaires pour qu'une application en cours de téléchargement d'un document en ligne ne soit pas perturbée par quelque action de services du Détaillant. Dans cette démarche, c'est au constructeur du couple service/application de définir les statuts intermédiaires de son application, pour que ces statuts soit gérables par l'AUA et interprétables par le PA. Pour la construction de messages SIP NOTIFY intégrant de tels statuts, nous avons adopté la convention qu'un statut intermédiaire partait d'un état basic du service pour revenir vers ce même état basic. Le statut intermédiaire est composé d'un libellé de statut et d'un timbre horaire, précisant l'expiration du statut. Le corps d'un message NOTIFY aura ainsi la structure :

```
<app:presence xmlns:app="http://www.ietf.org/xml/ns/pidf"
              xmlns:ms="http://www.microsoft.com/pidf-status-
type/board"
              entity="pres:afontaine@enst.fr">
  <app:tuple id="aua1_sg89ae">
```

```
    <app:status>
      <app:basic>en ligne</app:basic>
      <ms:intermediate>
        <ms:basic>verrouillé</ms:basic>
        <ms:timestamp>2003-06-01T16:49:29Z</ms:timestamp>
      </ms:intermediate>
    </app:status>
    <app:contact priority="1">WhiteBoard@mortimer.enst.fr</app:contact>
  </app:tuple>
</app:presence>
```

De cette présentation du PA nous pouvons déduire que notre composant présente une première facette générique `AUAPAFacet`, à destination des AUA du terminal, pouvant être étendue en `ApplicationAUAPAFacet` pour tenir compte d'opérations de présence spécifiques aux applications. Cette facette doit au minimum proposer des méthodes d'ajout et de retrait d'une localisation d'une application, et une méthode de mise à jour de son statut.

5.5.2.3 La relation AUA-SA

Un Service Agent (SA) est un composant Utilisateur (cf. partie 3.5.5) dont les facettes et réceptacles de collaboration ont été enrobées de capacités de traitement des messages SIP. L'utilisation de SIP prend tout son sens à ce niveau, en tirant partie de la structure des messages SIP et de son modèle d'échange.

Les informations véhiculées dans l'entête d'un message SIP (nombre de retransmissions du message, temps d'acheminement maximum, …), s'adaptent particulièrement bien à notre problématique de composition de services multi-fournisseurs. En effet, des données à destination des serveurs relais du message peuvent être intégrées au message SIP, en réutilisant un nombre important de contraintes de qualité de service présentent au niveau des facettes et réceptacles de collaboration. De plus, le caractère extensif de SIP permet la prise en charge d'une variété illimitée de corps de messages, s'adaptant ainsi à toutes les relations possibles entre applications, composants utilisateur et composants session.

Utilisé comme protocole de communication entre nos composants de service, les retours SIP intermédiaires classiques (1XX) ou en erreur (2XX, 3XX, ...) permettent une gestion des états des applications adaptable aux opérations effectuées sur les services. En effet, après avoir invoqué un appel de méthode sur un Agent de Service, un AUA peut faire évoluer le statut de l'application de laquelle il dépend, au fur et à mesure des retours intermédiaires du SA, comme dans le scénario suivant :

Action	Statut
L'AUA demande au SA de se rattacher à une session de service.	L'AUA notifie le PA du passage de l'application à l'état "tentative de connexion"
Le SA retourne un message 100 (Trying) à l'AUA.	L'AUA notifie le PA du passage de l'application à l'état "connexion en cours"
Le SA reçoit un retour 182 (Queud) du composant session, qu'il relaie à l'AUA.	L'AUA notifie le PA du passage de l'application à l'état "demande de connexion mise en attente"
L'AUA reçoit un retour 200 (OK) de son SA	L'AUA notifie le PA du passage de l'application à l'état "connecté"

Si c'est la relation AUA-SA qui assure la signalisation entre applications et services, c'est l'enregistrement de l'AUA auprès du PA qui permettra au SA de se connecter à l'AUA avant de pouvoir traiter les messages SIP à destination ou en provenance des services du Détaillant.

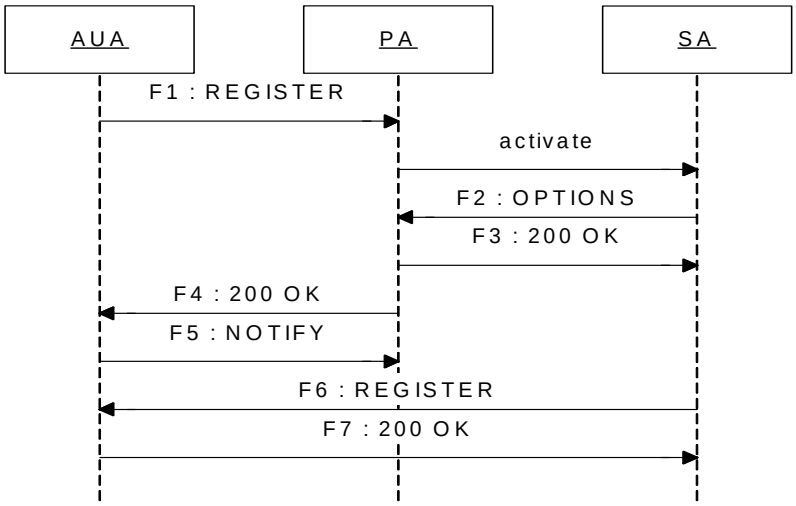


Figure 66 L'initialisation de la relation AUA-SA

Une fois la phase d'enregistrement de l'AUA accomplie auprès du PA (cf. partie 5.5.2.2 pour les détails des messages F1 et F4), le PA va se connecter à la home du composant utilisateur dédié au service en relation avec l'application utilisateur à l'origine de la procédure d'enregistrement, pour y invoquer la méthode `createUser(utilisateur_physique)`. La problématique sera à ce moment là de connecter l'instance de composant utilisateur à notre AUA. Pour se faire, dans l'implémentation de sa méthode `createUser()` notre composant doit interroger le PA de l'utilisateur physique, via l'utilisation d'un message SIP OPTIONS, pour connaître les informations utiles à la connexions des facettes et réceptacles de collaboration.

```
F2 : OPTIONS sip:afontaine@infres.enst.fr SIP/2.0
     From: sip:boardSynchronizer@enst.fr
     To: sip:afontaine@mortimer.enst.fr
     Call-ID: 123456789@enst.fr
     CSeq: 4765 OPTIONS
     Accept: application/xml
```

La première ligne directrice stipule que ce message est adressé à la machine `infres`, hébergeant le PA de notre utilisateur logique `afontaine` appartenant au domaine `enst.fr`. Pour permettre au PA de faire le rapprochement entre la demande d'information du SA et l'enregistrement de l'AUA, le message OPTIONS intègre dans son champ `From` l'identité du service auquel notre composant utilisateur est dédié, et dans son champ `To` l'identité de l'utilisateur physique reçue par le SA. Le champ `Call-ID` est identique à celui mentionné dans le message F1, pour que le PA puisse synchroniser le cas échéant l'arrivée du message OPTIONS avec l'envoi d'un retour intermédiaire. Le message retourné enfin par le PA au SA ressemblera à :

```
F3 : SIP/2.0 200 OK
     From: sip:boardSynchronizer@enst.fr
     To: sip:afontaine@mortimer.enst.fr
```

```
    Call-ID: 123456789@enst.fr
    CSeq: 4765 OPTIONS
    Accept: application/xml

  <agent-relation>
    <aua local-name="WhiteBoard@mortimer.enst.fr">
    <sa authorization-level="subscriber">
  </agent-relation>
```

L'objectif de notre travail n'a pas été de spécifier le format des données retournées dans le message F3. Cependant, compte tenu de notre architecture, nous pouvons préciser que ce message doit contenir les informations nécessaires au paramétrage du composant utilisateur, ce qui correspond principalement en la déterminisation des facettes et réceptacles de collaboration, à connecter d'une part à l'AUA et d'autre part au service du Détaillant (cette opération ne sera réalisée dans la pratique qu'au moment d'un appel à la méthode `addSession(sessionId)`). Dans cette optique, le message F3 donné en exemple apporte premièrement au SA (via le tag `<aua>`) les données nécessaires et suffisantes pour localiser l'application utilisateur. Deuxièmement, le tag `<sa>`, contenu dans le corps du message, requière auprès du SA l'utilisation de facettes et réceptacles de collaboration assimilés à un niveau d'autorisation `"subscriber"`.

Le passage de l'information entre PA et SA, du niveau d'autorisation d'accès, repose sur l'hypothèse que PA et SA peuvent ne pas être supportés par le même acteur. En effet, si le PA appartient nécessairement au domaine du Détaillant, le service demandé par notre utilisateur peut dans les faits être déployé dans le domaine d'un fournisseur de service tiers. Dans ce cas de figure, pour minimiser la gène de l'abonné, le processus d'authentification de l'utilisateur, compatible avec les attentes des fournisseurs de services, devra être centralisé par le Détaillant. Le Détaillant aura ainsi la charge de présenter aux fournisseurs de services soit l'identité authentifiée de l'utilisateur, soit un ensemble de rôles par eux communiqués au Détaillant. Ces fonctions assumées par le PA supposent ainsi qu'outre les informations de présence, l'agent de présence de l'utilisateur peut avoir accès aux souscriptions de l'utilisateur aux services, pour renvoyer à ces derniers les autorisations adéquates.

Les données reçues du PA, l'intercepteur SIP du composant utilisateur peut envoyer un message SIP REGISTER à l'AUA contenant les références aux facettes de collaboration compatibles avec le niveau d'autorisation communiqué par le PA. Après avoir associé ses réceptacles de collaboration aux références de facettes reçues, notre AUA envoi à l'AS en retour ses propres facettes, pour que le SA connecte à son tour ses réceptacles de collaboration à l'AUA. La connexion établie, AUA et SA sont disposés à véhiculer la signalisation que s'échangent les applications du terminal et les services sur le réseau.

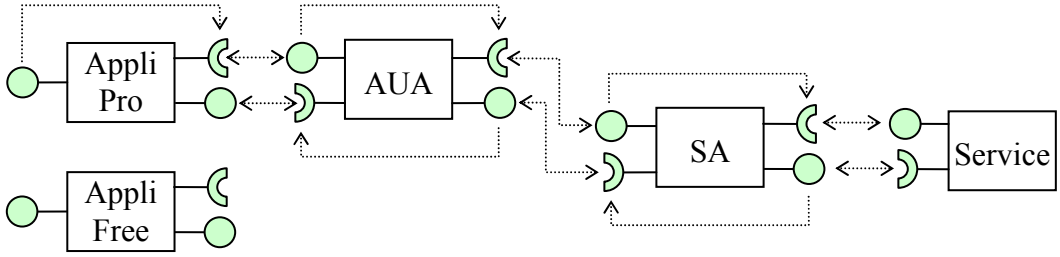


Figure 67 Les relations entre AUA, SA et Services

Conformément à ce que nous présentions dans la partie 4.5.3, c'est l'interface directement utilisée par le client qui conditionne l'intégralité de l'enchaînement entre facettes et réceptacles de collaboration. Comme nous le schématisons dans la Figure 67, si un service du réseau est exploitables via deux applications clientes (une gratuite et une payante), chacune accompagnée de son propre AUA, l'utilisation d'une application plutôt que l'autre ne provoquera pas les mêmes connexions entre Réceptacles et facettes de collaboration des AUA et SA. De plus, par le lien unissant facettes et réceptacles d'un même composant, les connexions seront également différentes entre le SA et le service en bout de chaîne. Le rapprochement entre l'application utilisée par le client et la facette de collaboration à présenter à l'AUA ne peut se faire sans l'intervention du PA. Ainsi c'est avec le retour du PA au message OPTIONS (message F3), que la topologie de la connexion est déterminée. Un corollaire à cette propriété est que le PA gère des identifiants d'application, qui soient assez fins pour que l'AUA précise les facettes et réceptacles de collaboration qu'il attend, par le seul nom d'application précisé dans le champs contact du message REGISTER :

```
Contact: <sip:WhiteBoard@mortimer.enst.fr;application=board>
```

Dans le cas de notre exemple, le PA peut déclarer à son niveau des applications `"board"` et `"board-lite"`, à partir desquelles il sera possible de préciser les composants AUA et SA concernés, ainsi que les connexions possibles entre facettes et réceptacles de collaboration.

Les AUA et SA connectés, applications et services disposent d'une infrastructure de communication, permettant aux services de faire remonter des traitements jusqu'à l'utilisateur et réciproquement. Dans cette construction, AUA et SA utilisent des messages SIP NOTIFY encapsulant une enveloppe SOAP, pour spécifier les appels ou retours à des méthodes distantes. L'URI mentionnée en première ligne de l'entête permet de router le message vers l'instance de composant adéquate [CAS01]. Comme nous l'indique la Figure 68, l'entête SIP classique, précède un corps de message composé d'une enveloppe SOAP. L'entête SIP permet une interaction entre DialogSender et DialogListener, avec entre autre la question de la qualité de service, l'entête SOAP peut être utilisé pour préciser les interactions au niveau des instances de composants. Enfin le corps de l'enveloppe SOAP explicite la structure et le contenu des données à traiter par le destinataire du message.

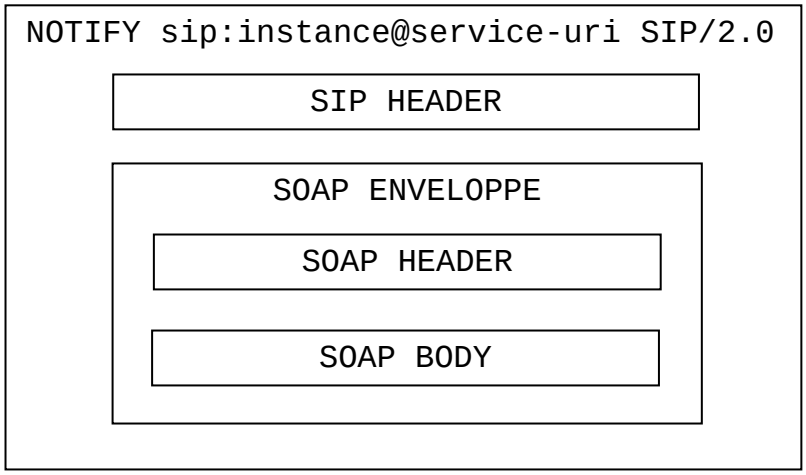


Figure 68 La structure d'un message SIP NOTIFY de corps SOAP

Pour notre exemple de tableau blanc, nous donnons en Figure 69 un message SIP NOTIFY que pourrait adresser à son SA, l'AUA d'une application utilisateur, suite à l'ajout d'un

élément sur le tableau. Le corps de la partie SOAP est issu de la projection de l'élément métier et de l'appel de la méthode en XML, assumé par le stub et interprété par le squeletton.

```
NOTIFY sip:afontaine123@boardSynchronizer.enst.fr SIP/2.0
From: WhiteBoard@mortimer.enst.fr;tag=a73kszlfl
To: afontaine123@boardSynchronizer.enst.fr
Call-ID: 123456789@enst.fr
CSeq: 8776 NOTIFY
Max-Forwards: 70
Content-Type: application/soap
Content-Length: 299

<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/soap-envelope">
  <env:Body>
    <m:addElement xmlns:p="http://infres.enst.fr/board">
      <a:element>
        <a:line>
          <h-location>325</h-location>
          <v-location>1456</v-location>
          <h-length>12</h-length>
          <h-length>964</h-length>
          <depth>3</depth>
          <color>#129</color>
        </a:line>
      </a:element>
    </m:addElement>
  </env:Body>
</env:Envelope>
```

Figure 69 Le message SIP NOTIFY associé à l'appel de la méthode addELement(Element)

5.5.2.4 La relation entre Services et PA/SA

Les services proposés par un Détaillant ou un Fournisseur de Services Tiers sont matérialisés par un ensemble de composants session (pouvant être réduit à un unique composant). L'ouverture d'une session de service demande l'envoi d'un message SIP INVITE (cf. partie 5.4.4) regroupant dans son entête l'identité du service demandé, l'application utilisateur à l'origine de la demande d'utilisation du service, et optionnellement une information relative à l'application utilisateur appelée. A réception de ce message SIP, le DialogListener localisé sur le serveur d'application va appeler la méthode createSession() sur la Home du composant session adéquate. L'instance de service ainsi créée va adresser dans un second temps un nouveau message INVITE à destination du ou des composants utilisateur, pour ajouter les SA de nos utilisateurs à la session de service. Pour reprendre notre service de tableau blanc évoqué dans la Figure 69, l'instanciation du service sera effectuée grâce au message SIP suivant :

```
INVITE sip:boardSynchronizer@enst.fr SIP/2.0
Via: SIP/2.0/UDP WhiteBoard@mortimer.enst.fr
From: afontaine@enst.fr
To: boardSynchronizer@enst.fr
...
```

Ce message INVITE va créer une instance du composant session incarnant le service `boardSynchronizer`, dans le domaine du fournisseur de services `enst.fr`. A réception de ce message, le service va entrer en contact avec le PA de l'utilisateur logique mentionné dans le champ `From` (en utilisant un message SIP OPTIONS), afin de localiser le SA dédié à l'application `WhiteBoard@mortimer.enst.fr`, ajoutée au message par l'AUA, dans une entrée `Via`. Cette référence obtenue, notre composant session va envoyer un nouveau message INVITE à destination de l'agent de service de l'utilisateur, pour attacher l'utilisateur logique à la session de service. Ce message INVITE prend cette fois la forme :

```
INVITE sip:afontaine123@boardSynchronizer.enst.fr SIP/2.0
Via: SIP/2.0/UDP board987@boardSynchronizer.enst.fr
Via: SIP/2.0/UDP WhiteBoard@mortimer.enst.fr
From: afontaine@enst.fr
To: boardSynchronizer@enst.fr
...
```

Après avoir pris soin d'ajouter une référence vers lui-même dans une entrée `Via`, l'instance du service de synchronisation de tableau blanc va modifier le destinataire immédiat du message en précisant la référence au SA : `afontaine123@boardSynchronizer.enst.fr` de l'utilisateur `afontaine@enst.fr`, sans toutefois modifier le champ To, qui d'après [HSS99] doit rester inchangé. Les messages SIP INVITE et OK 200 échangés entre composants session et composants utilisateur, notre service et le SA sont considérés comme liés, et donc capables de traiter la signalisation liée à l'utilisation du service. Pour ce faire, SA et services s'échangeront des messages SIP NOTIFY, sur le même modèle que la relation SA-AUA.

Concevoir des services dans une architecture télécom ne peut s'envisager sans considération de l'état de l'utilisateur. En conséquence, les traitements proposés par les services, et appliqués sur les Agents des Services, doivent être accompagnés de gardes portant sur le statut des applications utilisateur. Pour revenir à notre exemple de tableau blanc, si A ajoute un élément sur son tableau blanc local, le service de synchronisation du Détaillant ne pourra répercuter cet ajout sur le tableau blanc de B, tant que le statut de l'application "WhiteBoard" de B aura un statut différent de `online`. Cette fonctionnalité demande une adaptation des service aux statuts des Agents des Services (le statut d'un SA est le reflet du statut de l'application utilisateur dont il dépend).

Si ce mécanisme est indispensable pour l'adaptation des services aux conditions de vie des utilisateurs, il ne doit pas en revanche être directement assumé par les composants. En effet, avec des considérations d'état particulier des utilisateurs, les gardes sur l'état de l'utilisateur paraissent incompatibles avec un code du composant regroupant des traitements métier génériques. De plus, dans un environnement multi-fournisseurs, où les services et agents de présence peuvent être maintenus par des acteurs différents, il n'est pas envisageable pour des problèmes d'interopérabilité qu'un tel mécanisme soit directement pris en charge au niveau des composants. Pour prendre en compte cette fonctionnalité dans la relation entre composants, nous avons donc introduit le concept de réceptacle à géométrie variable, ayant un impact au niveau des stubs, des Agents de Présence et du Broker.

Outre la capacité à assurer la correspondance entre un appel de méthode objet et le traitement de messages SIP, le stub a également la responsabilité de la gestion du pool de composants serveurs connectés au réceptacle, dans le cas de réceptacles multiplex. Aussi à ce niveau la définition de réceptacles a géométrie variable revient à demander au stub coté service, de

maintenir des groupes de facettes, fonction des états des applications dont les SA dépendent. Le stub peut ainsi gérer des envois inconditionnels de messages SIP, et en parallèle attendre une notification pour l'envoi vers un second groupe de facettes.

Cette notification vient du broker lui-même, qui centralise les états des utilisateurs appartenant à une session, de manière à adapter la géométrie des réceptacles qu'il retourne aux demandes de résolution des composants. Pour ce faire, le broker adopte le principe de fonctionnement suivant :

- Un composant fait une demande de résolution pour un réceptacle, auquel est attaché une garde g portant sur le statut dans lequel doivent être l'application des utilisateurs pour que le traitement du service soit vers eux adressé.
- Le broker connaissant le statut des applications, retourne au composant une référence vers un stub auquel il a communiqué au préalable la structure de deux groupes de facettes : les facettes inconditionnelles et les facettes conditionnelles.
- Pendant ce temps le broker va s'abonner (envoi d'un message SIP SUBSCRIBE) auprès des PA associés aux SA apparaissant dans le groupe des facettes conditionnelles, pour pouvoir être informé du changement de statut de l'application (au moyen d'un message NOTIFY).
- Sur une notification de statut d'un PA, indiquant le passage de l'application dans un statut compatible avec la garde g, le broker informe le stub de la nécessité d'envoyer le message SIP à cet utilisateur.
- A la destruction du composant service, le broker se désabonne des PA, et prend soin de vérifier la destruction des stubs en attente d'envoi de messages vers des facettes encore conditionnées.

Pour être généralisables à des utilisateurs appartenant à des Détaillants distincts, les gardes gérées par le broker demandent l'utilisation de libellés standards de statut spécifiques au service (ie. l'utilisation de statuts créés par l'utilisateur n'est pas envisageable). Nous verrons enfin dans la partie 5.6 que les messages SUBSCRIBE adressés par le broker aux PA, occasionne la création de contrats de collaboration au niveau du composant PA, pour lier la réception d'une notification de changement de statut émanant de l'AUA, avec l'envoi de la notification au service.

5.5.3 Exemple de session de service

Nous allons présenter brièvement l'ouverture et l'utilisation du service de tableau blanc entre deux utilisateurs A et B. La Figure 70 nous présente la séquence de messages échangés suite à l'ouverture de la session utilisateur, puis le démarrage de l'application de tableau blanc sur le terminal de A. Nous avons adopté le principe que notre terminal disposait d'un service d'indexage des AUA, permettant l'initialisation de l'ensemble des agents des applications du terminal, dès l'ouverture de la session utilisateur (voir commentaires de la Figure 66 pour plus de détails). Le démarrage de l'application provoque un événement immédiatement capté par l'AUA, qui provoquera une modification du statut de l'application auprès du PA de notre utilisateur A.

Son application démarrée, notre utilisateur A peut manipuler son tableau blanc comme un simple logiciel de dessin, dont l'exécution restera cantonnée à son terminal. Admettons qu'un

élément de l'interface graphique de l'application permette la création d'une session de service (cf. Figure 71). L'utilisation de cet élément graphique provoquera à nouveau un événement observé par l'AUA, générant d'une part un changement de statut de l'application auprès du PA, en "connexion demandée", et d'autre part l'envoi d'un message INVITE à destination du service de synchronisation de tableau blanc.

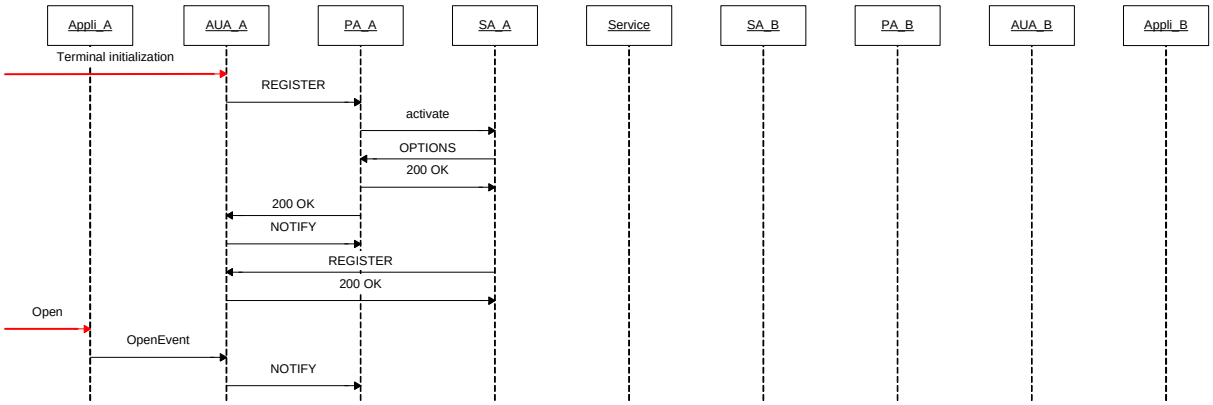


Figure 70 Ouverture de l'application utilisateur

A réception du message INVITE, le service peut retourner un message SIP de retour intermédiaire (TRYING 100), permettant à l'AUA de suivre l'évolution de sa demande d'ouverture de session de service, et en offrant par la même occasion la possibilité de mettre à jour le statut de l'application avec un statut du type "connexion en cours". Outre le service demandé, le message INVITE envoyé par l'AUA précise l'identité de l'application utilisateur émettrice, permettant au service d'interagir avec le gestionnaire de PA (via un message SIP OPTIONS), pour déterminer l'identité de l'utilisateur logique et son SA à ajouter à la session de service. Le SA identifié, le service lui adresse un message INVITE, relayé à l'AUA comme tout message INVITE réceptionné par un Agent de Service. L'AUA détectant que ce message NOTIFY fait référence à la même session que le message INVITE envoyé précédemment au service, l'acceptation de l'invitation par l'utilisateur sera considérée comme implicite, l'AUA répondra donc au SA un message SIP OK 200 de sa propre initiative. L'invitation de l'agent de service de l'utilisateur achevée, le service retourne un message OK 200 à l'AUA, modifiant à nouveau le statut de l'application à "en ligne". Le retour du service est également utilisé par l'AUA pour modifier la configuration de l'interface graphique de l'application.

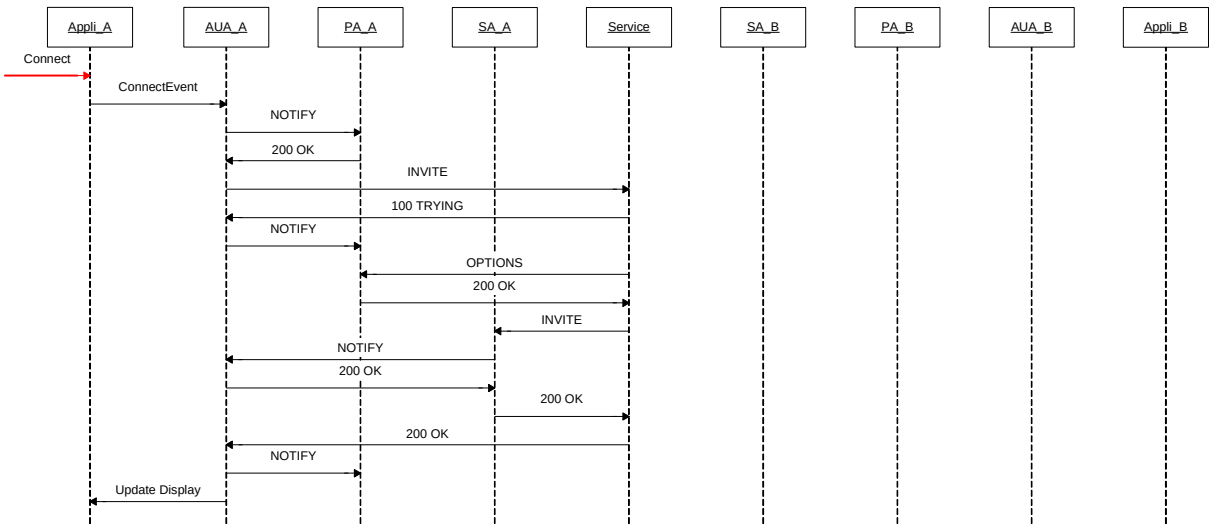


Figure 71 Ouverture d'une session de service

Après avoir ouvert une session de service dans laquelle il est pour le moment l'unique membre, notre utilisateur A manipule son application pour demander au service de rattacher un utilisateur distant B à sa session de service (cf. Figure 72). Comme toute opération effectuée au sein de la session de service, suite à la captation d'un événement utilisateur l'AUA émet un message NOTIFY à destination du SA de notre utilisateur A, après avoir au passage mis à jour le statut de l'application. La méthode du composant utilisateur de A mentionnée dans le message NOTIFY (`attachUser`), génère un message INVITE permettant d'ajouter un utilisateur B à la session de service. L'entête du message SIP envoyé par le SA_A prendra la forme :

```
INVITE sip:board987@boardSynchronizer.enst.fr SIP/2.0
Via: SIP/2.0/UDP afontaine123@boardSynchronizer.enst.fr
From: afontaine@enst.fr
To: najm@enst.fr
...
```

Adressé à l'instance `board987@boardSynchronizer.enst.fr` de composant session dédiée à notre utilisateur A : `afontaine@enst.fr`, ce message SIP comprend la référence au SA de l'utilisateur A `afontaine123@boardSynchronizer.enst.fr`, et l'identité de l'utilisateur logique que l'on cherche à contacter (cf. champ `To`). Après avoir retourné un message SIP intermédiaire au SA, utilisé pour informer notre utilisateur A de l'avancée du processus, le service reproduit le schéma vu précédemment, pour la recherche cette fois de l'agent de service de l'utilisateur B. Il commence à cet effet par interroger le PA_B pour localiser le SA_B, puis lui adresse un message INVITE, remontant cette fois jusqu'à l'utilisateur pour que son ajout à la session de service puisse être par lui avalisé. Après acceptation de l'invitation, la session de service est composée de deux utilisateurs, capables de s'échanger de l'information.

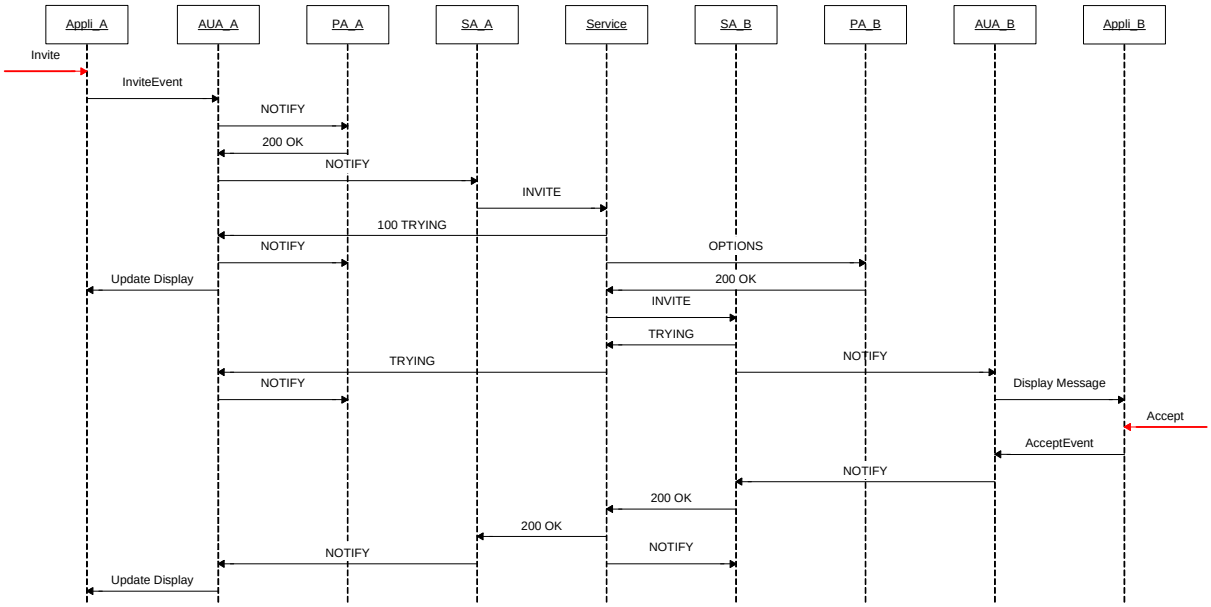


Figure 72 L'invitation d'un utilisateur

L'échange d'informations à l'intérieur de la session de service (cf. Figure 73) occasionne la production de messages NOTIFY, permettant d'une part que chaque utilisateur soit notifié

des changements d'état des autres utilisateurs (consécutivement à un événement DragEvent), et d'autre part d'entrer en relation avec le service au centre de la session.

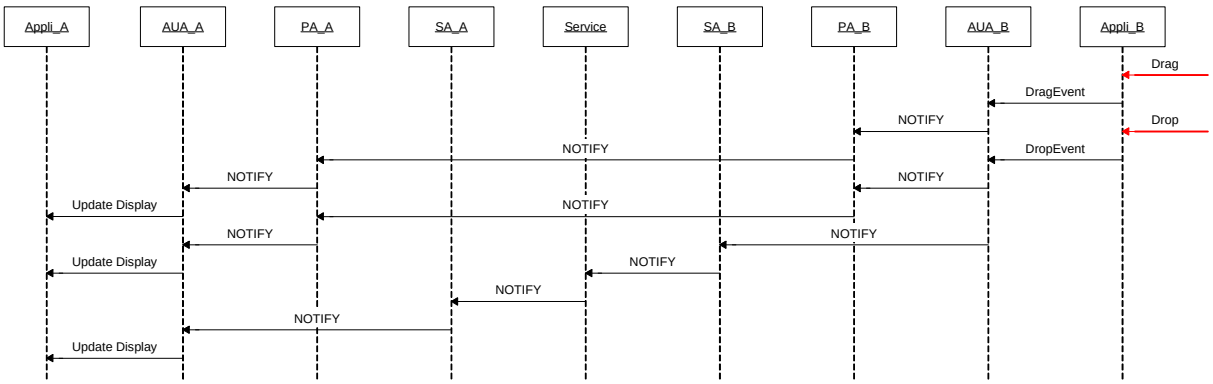


Figure 73 Le flux d'information entre applications utilisateur

Le détachement d'un utilisateur à la session de service (cf. Figure 74) passe par une première étape où l'application de l'utilisateur change de statut, et où ce nouveau statut "leaving" est communiqué à tous les utilisateurs membres de la session de service. L'AUA adresse ensuite un message NOTIFY au SA, lui demandant de se retirer de la session de service, par l'envoi au service d'un message SIP BYE du type :

```
BYE sip:board987@boardSynchronizer.enst.fr SIP/2.0
Via: SIP/2.0/UDP najm456@boardSynchronizer.enst.fr
Max-Forwards: 68
From: najm@enst.fr
To: board987@boardSynchronizer.enst.fr
...
```

Réceptionné par le service, ce message sera relayé à tous les Agents de service des utilisateurs, en ayant préalablement modifié le destinataire en `afontaine123@boardSynchronizer.enst.fr` (SA_A) et ajouté une entrée Via pour conserver l'information du passage par le service.

```
BYE sip:afontaine123@boardSynchronizer.enst.fr SIP/2.0
Via: SIP/2.0/UDP board987@boardSynchronizer.enst.fr
Via: SIP/2.0/UDP najm456@boardSynchronizer.enst.fr
Max-Forwards: 68
From: najm@enst.fr
To: board987@boardSynchronizer.enst.fr
...
```

Reçu par le SA de notre utilisateur `afontaine@enst.fr`, ce message provoquera l'envoi d'un message de notification vers l'AUA, pour que le PA de l'utilisateur A stoppe sa surveillance du statut de l'application de l'utilisateur B, et pour mettre à jour l'affichage ç l'intérieur de l'application de A.

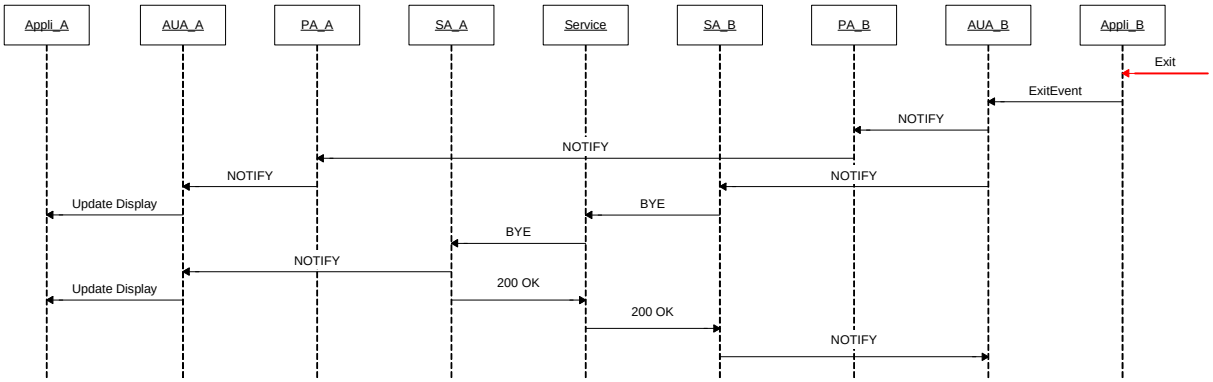


Figure 74 Le retrait d'un utilisateur

5.6 La gestion du comportement

5.6.1 Les bases du comportement

5.6.1.1 Etat de présence – Statut de l'application

Comme état ou statut nous n'avons évoqué jusqu'à présent, que le statut particulier des applications de l'utilisateur, sans aborder son état global, qui est l'état habituellement considéré dans les architectures de présence.

L'état global d'un utilisateur est caractérisé par un libellé du type "je suis parti manger", standard ou défini par l'utilisateur, dont l'expression est à priori indépendante des statuts des services. Le changement d'état d'un utilisateur peut être explicitement demandé par ce dernier, via un logiciel dédié sur son terminal, ou corrélé au changement de statut d'une application. Typiquement l'utilisateur pourra passer automatiquement dans l'état libellé "en conférence" consécutivement à l'ouverture d'une session de service mobilisant son application de tableau blanc.

L'état d'un utilisateur ne se limite pas à une simple étiquette. Premièrement, il déclare les applications et leurs services associés utilisables à l'intérieur de l'état de présence. Cette déclaration peut être accompagnée par un ensemble de contraintes portant sur les statuts des applications. Ainsi un état de l'utilisateur peut arrêter le statut d'une application, et ainsi ne pas tenir compte des notifications issues de l'AUA. Un état "en conférence" peut par exemple placer un ensemble d'applications secondaires dans un statut "busy", pour interdire aux services l'accès à ces applications et ainsi privilégier le fonctionnement de notre application de tableau blanc. Deuxièmement, un état de présence peut être associé à un comportement, matérialisé sous la forme de contrats de coordination, précisant les échanges entre services et application.

Tout comme les statuts des applications, par état de l'utilisateur nous entendons la situation de l'utilisateur logique, maintenue au niveau d'un Presence Agent, un utilisateur physique pouvant être à un même instant dans des états différents, suivant ses différentes identités logiques.

5.6.1.2 CPL

La définition du comportement associé à l'état de l'utilisateur ne s'est pas inscrite dans la continuité de CPL, car ce langage de scripts n'a pas la modularité que nous recherchions. En effet, nos contrats étant dynamiquement modifié au cours d'une session de service, les règles qu'ils contiennent doivent être vues comme des éléments atomiques. De plus CPL n'a pas de support pour la gestion de l'héritage de scripts ni de scripts répartis, élément indispensable de notre architecture où les contrats répartis à plusieurs points du réseau doivent pouvoir concorder à une exécution satisfaisante pour tous les utilisateurs.

5.6.1.3 La programmation par contrat

La programmation objet, et par extension par composant logiciel, est maintenant une technologie reconnue pour faciliter la réutilisation de code et l'évolution des applications. Mais dans le cas où de nombreux composants devraient interagir ensemble, elle montre assez vite ses limites. En effet, les interactions fortes entre composants à l'intérieur du code d'implémentation rendent difficile l'ajout de nouveaux composants sans avoir à changer l'existant. Pour casser cette interaction forte, et ainsi permettre une meilleure adaptabilité des composants logiciels, il a été développé la notion de contrat de coordination. Cette approche, permet au moyen de règles conditionnées de paramétrer les interactions entre ses participants.

En Figure 75 nous présentons un exemple de contrat qui nous permet de modéliser la facturation d'un appel coté appelant, pour un montant correspondant au temps d'exécution d'un service. Dans ce contrat nous pouvons identifier en entête deux classes de composants qui constituent ses participants (2), un invariant qui permet de délimiter les instances de participants concernés (3) et des déclaration d'identificateurs locaux (5). Le corps du contrat, (7-11) est constitué de règles de coordination ayant un nom, un paramètre de synchronisation (8) par rapport à l'événement cité dans la clause when (9). Ainsi pour le contrat ci-dessous, un appel de la méthode FinishCall() sur un composant Call aboutira à un appel de la méthode charge() sur le composant Account, si l'attribut time était supérieur à 3.

```
1   Contract    Charging
2   Participants  x: Account, y : Call;
3   Invariant   x.tel_number:= y.caller_number;
    
5   local Integer time := y.CalculateCallTime();
    
7   Coordination
8   Rule_1 : after
9         When      * ->> y.FinishCall();
10        With      time > 3;
11        Do        x.charge(time);
12  End
```

Figure 75 Un exemple de contrat

Une telle approche permet de configurer l'interaction entre composants, sans avoir à modifier les composants eux-mêmes. Lorsqu'un appel de méthode est effectué entre deux composants participant tous deux au même contrat, le contrat intercepte l'appel et lui superpose le comportement défini dans le contrat. De façon à préserver le caractère dynamique de cette approche, ni le client, ni le fournisseur, n'ont connaissance de l'existence du contrat.

5.6.2 La définition des contrats

5.6.2.1 Les contrats

Un contrat est composé d'un entête, essentiellement utilisé à des fins d'administration, et d'un ensemble de règles de coordination, optionnellement accompagnées de déclarations de variables locales. La définition d'un contrat de coordination fait naturellement suite à celle des états de présence des utilisateurs auxquels il se rapporte. En effet, l'entête délimite en terme d'utilisateur et d'état de présence, la portée du contrat applicable aux composants déclarés comme participants.

```
Contract            Contract-Name
Presence-agent      User | Group
Presence-state      State-ref
Participants        Component-ref
Invariant           Constraint_on_participants
```

La déclaration d'un contrat commence par mentionner un nom de contrat, précédé du mot clé `Contract`. Cette première ligne du contrat peut exprimer une dépendance ou un *héritage explicite* envers un autre contrat. Par exemple un utilisateur qui souhaite créer un contrat spécifique héritant d'un contrat plus général, peut déclarer son contrat comme ceci :

```
Contract mySpecificContract : theGenericContract@Détaillant-domain
```

La relation d'héritage est automatique (ie. elle ne tient pas compte de l'état de l'utilisateur mentionné dans le contrat hérité). De plus, si un contrat héritait de plusieurs contrats, l'ordre d'apparition des contrats fait office de relation de priorité entre eux. Dans un contexte commercial, cette notion d'héritage explicite peut être l'occasion pour un fournisseur de services de proposer des règles de coordination de base entre services, que l'utilisateur peut être libre d'accepter ou de refuser, par un ajout d'une simple relation d'héritage dans son contrat de coordination. Cependant ce système permissif doit être contrebalancé par un *héritage* dit *implicite*, ne permettant pas cette fois à l'utilisateur d'interférer à la dépendance entre contrats. La notion d'héritage implicite repose sur le constat qu'un utilisateur appartient en réalité à un certain nombre de groupes d'utilisateurs, auxquels peuvent être rattachés des contrats de coordination apportant un comportement commun à tous les membres du groupe d'utilisateurs. Dans un contexte d'entreprise, ce type d'héritage peut par exemple être utilisé pour limiter l'accès aux services, pour une certaine classe d'utilisateurs. Prenons un utilisateur au profile :

```
<presence-agent user="afontaine">
    <groups>
        <group group-ref="infres">
        <group group-ref="thesard">
    </groups>
</presence-agent>
```

Cet utilisateur logique (géré par un Presence Agent) appartient aux groupes ”infres” et ”thesard”. Cette propriété implique qu'un contrat défini pour l'un de ces groupes s'appliquera automatiquement à notre utilisateur, sans la moindre référence explicite au niveau de ses contrats de coordination. Notons que pour lever une ambiguïté sur la construction de l'arbre des règles de coordination (cf. partie 5.6.2.2), nous conviendrons que l'héritage implicite ne sera pas récursif. En conséquence, si notre utilisateur appartient à un groupe G1, lui-même contenu dans un groupe G2, ces deux groupes devront être déclarés auprès de l'agent de présence ci-dessus.

Du fait de la relation d'héritage entre contrats, un composant utilisateur de notre architecture de présence peut être amené à gérer à un même instant des règles émanant de multiples contrats. Il convient donc de préciser les propriétés selon lesquelles nos contrats sont construits et pris en compte :

(P1) L'opération de construction de nos contrats est injective. Ainsi deux règles de coordination applicables à un même spectre d'utilisateurs dans les mêmes états de présence seront nécessairement définies dans le même contrat de collaboration. La priorité entre ces deux règles est ensuite résolue d'après leur ordre d'apparition dans le contrat de coordination.

(P2) S'il existe un contrat C applicable aux utilisateurs A et B, dans un état de présence E, contenant les règles de coopération $(R_i)_{1\leq i\leq n}$ et si A souhaite ajouter dans ce contrat une règle R_{n+1} pour lui seul applicable. Alors C devra être séparé en deux contrats C_1 et C_2 avec C_1 applicable à l'utilisateur B, dans l'état E et contenant les règles $(R_i)_{1\leq i\leq n}$ et C_2 applicable à l'utilisateur A, dans l'état E et contenant les règles $(R_i)_{1\leq i\leq n+1}$.

(P3) Les contrats hérités implicitement seront toujours prioritaires sur les contrats hérités explicitement, eux-mêmes prioritaires sur le contrat courant.

(P4) La relation de dépendance est récursive. Si un contrat C1 hérite d'un contrat C2, héritant lui-même d'un contrat C3, C1 héritera dans l'ordre de priorité de C3 puis C2.

Outre le nom du contrat, l'entête liste l'utilisateur ou le groupe d'utilisateurs auxquels le contrats s'applique, grâce au mot clé `Presence-Agent`, également pris en compte pour les relations éventuelles d'héritage implicite. L'entête précise également l'état ou les états de présence dans lesquels il peut être applicable à l'utilisateur, via une entrée `Presence-state`. Les états mentionnés peuvent aussi bien être des états standards, comme des états définis par l'utilisateur, notre contrat se chargeant de définir le comportement à associer à ce nouvel état. L'entrée `Participants` liste enfin les types d'agents de service mentionnés dans le contrat, un `Invariant` pouvant définir des contraintes en vue d'en délimiter les instances (en se référant à leur relation avec un agent de présence), utilisable pour les problématiques d'indexation des contrats au niveau de l'agent de présence.
.

5.6.2.2 Les règles de coordination

Les règles de coordination permettent de synchroniser l'exécution d'un ou plusieurs services avec l'arrivée d'un message SIP sur un composant utilisateur, en présentant en sortie une série d'actions éventuellement matérialisées par l'envoi de messages SIP. Une telle règle est spécifiée comme il suit :

```
<name> : <priority> : <synchronization>
when <trigger>
with <condition on trigger>
do   <set of actions>
```

Figure 76 La spécification d'une règle de coordination

La déclaration d'une règle commence par la définition d'un nom unique à l'intérieur du contrat, réutilisé à l'intérieur des règles de manipulation de règles. La classe de la règles est ensuite déterminée par deux attributs `priority` et `synchronisation`, accompagnant l'identifiant de la règle. L'attribut `priority`, qui peut prendre les valeurs MUST ou MAY, est utilisé pour donner une priorité à certaines règles. Ainsi une règle de type MAY ne pourra être exécutée que si et seulement si l'ensemble des règles de type MUST applicables dans le contrat ont déjà été appliquées avant elle. Cependant, même si son trigger était actif, une règle de type MUST peut ne pas être exécutée, si une règle de type MUST plus prioritaire qu'elle a modifié entre temps le cours de l'exécution du service. Le caractère prioritaire des règles est utilisé principalement dans les problématiques d'héritage de contrats, une règle de type MUST d'un contrat hérité étant prioritaire sur une règle de type MUST du contrat courant. Le second attribut `synchronisation`, permet de positionner la règle dans le processus d'exécution du composant. Ses valeurs `before`, `replace` ou `after` précisent au gestionnaire du contrat la synchronisation entre l'exécution de la règle et le traitement du composant associé au message SIP réceptionné. Compte tenu du fait que les règles peuvent s'appliquer à la fois sur le chemin aller ou sur le retour du traitement du composant, nous pouvons distinguer six catégories de synchronisation de règles de coordination, en faisnat varier les critères :

- Orientation de la règle parmi "aller" ou "retour".
- Synchronisation de la règle parmi "before", "replace" et "after".

Avec les règles d'aller nous aurons un trigger de la forme `someone ->> SIP : MESSAGE-TYPE` signifiant d'après l'opérateur "`->>`" que la règle s'applique pour tout message SIP de type MESSAGE-TYPE en provenance de `someone`, la valeur `*` signifiant une origine quelconque. Le paramètre de synchronisation, en entête de la règle, nous permet à ce moment de distinguer trois classes de règles d'aller :

- Les règles `before`, dont l'exécution précède la présentation du message à son destinataire légitime, ce qui permet de définir des actions, ayant des répercussions sur les données du message, voir remettre en question la présentation du message, par exemple en définissant une règle de filtrage.
- Les règles `replace` consomment le message intercepté. Ce type de règle nous est utile par exemple sur occupation, pour générer un message d'erreur au niveau du gestionnaire de contrat, sans remonter jusqu'à l'utilisateur final.
- Les règles `after` sont utiles pour notifier les notifications asynchrones d'événements sans allonger le temps d'attente du message intercepté, ce qui concerne les services d'archivage par exemple.

L'ensemble des règles de retour, sont pareillement divisées en trois classes de règles relativement à leurs synchronisations. Le trigger sera cette fois de la forme `SIP : MESSAGE2 <<- SIP : MESSAGE1`. L'opérateur "`<<-`" signifiant que la règle est activable sur réception d'un message SIP de type `MESSAGE2` (élément gauche de l'opérateur) faisant réponse à un message SIP de type `MESSAGE1` mentionné dans l'élément droit de

l'opérateur, la jointure entre les deux messages se faisant d'après l'identifiant de jambe présent dans l'entêtes des messages SIP.

La partie active du contrat (entrées `when`, `with`, et `do`) commence par la définition d'une contrainte d'activation, appelée aussi trigger. Tout changement d'état d'un composant, modification de la valeur d'une variable, appel d'une méthode pouvant à priori être l'occasion d'activer une règle, nous avons décidé, dans un soucis de limiter la complexité du phénomène d'interaction entre règles de coordination, de limiter la définition de triggers aux seules arrivées de message SIP. La définition d'un trigger sur le chemin aller (vers le composant) respecte la syntaxe suivante:

```
SIP: <sender> ->> SIP: <msg-type>((<msg-field> <ident>)*)
```

Hérité des contrats de collaboration de Fiadeiro le champ `sender` mentionne le composant à l'origine de la demande du traitement. Sa valeur '`*`' par défaut ne défini aucune restriction sur l'émetteur, qui par ailleurs peut être difficile à identifier dans le cas de composants distants, mais pour des composant hébergés sur le même serveur d'application, une valeur du type `boardsynchronizer@enst.fr` peut filtrer les cas d'activation de la règle. Derrière l'opérateur d'aller "`->>`", la règle déclare un message SIP, en définissant au passage des identifiants relatifs aux entrées de l'entête du message SIP , utilisables dans les champs `with` et `do` de la règle. Un exemple de trigger peut ainsi être :

```
When * ->> SIP: INVITE(From f, Via v)
```

Réciproquement, la définition d'un trigger retour (depuis le composant) respecte la syntaxe suivante :

```
SIP: <msg-type>((<msg-field> <ident>)*) <<- SIP: <msg-type>((<msg-field> <ident>)*)
```

Pour être capable d'évaluer une telle contrainte, le gestionnaire de contrat doit disposer du message aller et du message retour. Nous verrons ainsi dans la partie 5.6.3 que le gestionnaire de contrat devra s'interfacer entre les `DialogListener/DialogSender` et le Squeletton, pour disposer de ces informations. Comme exemple de trigger retour nous pouvons avoir :

```
When SIP: ERROR 408 <<- SIP: INVITE(From f)
```

L'entrée `With` de la règle de collaboration permet de délimiter les cas d'activation de la règle. Cette délimitation peut être opérée selon trois principe : le statut d'une application de l'utilisateur, un égalité ou inégalité sur les identifiants définis au niveau du trigger ou un filtre sur le corps du message SIP reçu. Le test du statut d'une application fait appel à la notation :

```
With pa[login=afontaine]/applications[application=board]/statut=online
```

Cette condition précise que la règle sera activable pour une application `board` définie auprès de l'agent de présence de l'utilisateur `afontaine`, ayant pour statut la valeur `online`.Une égalité ou inégalité portant sur l'entête du message SIP pourra prendre la forme :

```
With a=najm@enst.fr AND v=*@boardSynchronizer.enst.fr
```

Cette entrée `With` peut également faire appel à la notation XPath, pour introspecter le corps XML du message SIP reçu, et ainsi activer la règle d'après le contenu du message. Par exemple si en marge de notre service de tableau blanc, notre utilisateur veut définir une règle réduisant la taille des éléments dessinés par les utilisateurs distants, du fait de la fiable

résolution de son terminal, il pourra créer une règle à la condition suivante, portant sur le message de la Figure 69.

```
With env:/Enveloppe/addElement/element/line[h-locationname>100]
```

Enfin, une règle de coordination déclare une itération d'actions, pouvant être classées suivant deux catégories: l'envoi d'un message SIP, et la manipulation de règles tierces. L'envoi d'un message SIP peut être soit l'occasion de redéfinir le message capté dans le trigger, pour en modifier l'entête voir même le corps. Le gestionnaire de contrat utilisant l'API Jain SIP Lite, nous n'avons pas la nécessité de préciser l'ensemble des informations de bas niveau liées au message SIP. Ainsi la modification de l'entête d'un message SIP pourra être assumée par une règle du type :

```
Rule R1 : MAY : replace
  When * ->> SIP: INVITE(Target t, From f)
  With t=*@board.enst.fr
  Do   SIP: INVITE(Target afontaine987@board.wanadoo.fr, Via t)
```

Cette règle est activée sur réception d'un message SIP INVITE à destination de l'agent du propriétaire du contrat pour le service `board` du domaine `enst.fr` (nous appelons Target l'adresse référencée dans la ligne principale le l'entête d'un message SIP). L'action déclarée va transférer le message INVITE reçu vers un agent de service tiers, en ajoutant au passage une entrée `Via` à l'entête du message. Cependant compte tenu du niveau d'abstraction offert par notre architecture, cette action n'a pas à préciser le devenir du corps du message, qui implicitement restera inchangé. D'après notre construction exprimée en Figure 60, le `DialogSender` gère la logique d'implémentation du protocole SIP, pour déterminer le message intermédiaire à retourner à l'émetteur du message INVITE, pour l'informer du transfert de sa demande. Néanmoins, nous pouvons surcharger ce mécanisme au moyen d'une nouvelle règle :

```
Rule R2 : MUST : after
  When SIP INVITE(Via v) <<- SIP: INVITE(Target t , From f)
  With t=*@board.enst.fr AND v=t
  Do   Moved Temporarily <SIP: 302()>
```

Cette règle R2 intercepte le message INVITE retourné par la règle R1, pour lui associer l'envoi d'un message SIP 302 au client, afin de lui signifier le transfert dû à un déplacement de l'utilisateur. Notons que pour ne pas alourdir la déclaration d'un message SIP dans une action, nous ne précisons pas les champs du message qui restent inchangés par rapport au message figurant dans le trigger, ce qui ne génère par conséquent aucune ambiguïté Les deux premières règles exposées ne modifient pour le moment que l'entête du message SIP, sans altérer pour le moment son corps. Les messages SIP manipulés par nos services faisant appel à des corps SOAP cf. Figure 68), nous allons utiliser XSLT et XPATH pour transformer un corps de message, et générer un nouveau corps à partir du corps reçu. Typiquement cette fonctionnalité est utilisée pour rediriger un appel de méthode sur un composant utilisateur donné. Par exemple un utilisateur peut demander à ce que chaque notification de modification d'un élément de notre exemple de tableau blanc, réceptionnée par son agent de service, soit transformée en une création d'un nouvel élément, par l'utilisation de la règle suivant :

```
Rule R3 : MUST : replace
  When* ->> SIP: NOTIFY(Target ta)
  Witht a=*@board.enst.fr AND
       name(env:/Enveloppe/last())=modifyElement
  Do   SIP: NOTIFY()
       {
```

```
    <env:Enveloppe>
     <addElement>
      <xsl:value-of select=”env:Enveloppe/modifyElement/”
     </addElement>
    </env:Enveloppe>
}
```

De part la structure de l'enveloppe SOAP, nous récupérons la méthode appelée par le client via l'instruction XPath `name(env:/Enveloppe/last())`. Notre règle construit à ce moment une nouvelle enveloppe SOAP relative à la méthode `addElement`, dont le contenu sera identique aux données aux données initialement envoyées à la méthode `modifyElement`. Si les trois classes de règles exposées dans exemples R1, R2 et R3 rejoignent sur certains point les fonctionnalités apportées par CPL, notre second type de règle permet de dynamiser nos contrats en créant, remplaçant ou supprimant une règle existante, via l'utilisation de mots clés `create_rule`, `modify_rule` ou `delete_rule` suivit de l'identifiant de la règle et son contenu dans le cas de la création. Une telle règle peut prendre la forme :

```
Rule R4 : MUST : before
  When * ->> SIP: INVITE(Target t , From f)
  With t=*@board.enst.fr AND f=najm@enst.fr
  Do delete_rule R1
     create_rule R5 : MUST : replace
       When * ->> SIP: INVITE(Target t , From f, Via v)
       With t=*@board.enst.fr v=*@board.wanadoo.fr
       Do   Busy Here <SIP: 486(Target t , From t)>
```

Cette règle R4 précise qu'un message INVITE à destination d'une instance du service `board.enst.fr`, et en provenance de l'utilisateur `najm@enst.fr`, occasionne le retrait d'une règle R1 du contrat (qui pourra éventuellement être ajoutée ultérieurement par une autre règle), puis la création d'une règle R5, et enfin la poursuite de la présentation du message SIP INVITE au squeletton du composant.

5.6.2.3 Les dépendances entre contrats

Le gestionnaire de contrats en charge de l'application de nos règles de coordination devra dans la pratique gérer une multitude de contrats. La prise en compte des règles de coordination de composants se fera donc en deux phases : une première étape va identifier les contrats applicables au contexte d'utilisation du composant et une seconde étape appliquera proprement dite des règles, dans leur ordre de priorité. La phase d'identification des contrats doit se réaliser selon les étapes suivantes :

- Sélectionner le contrat applicable selon le seul critère de l'utilisateur, parmi l'univers des contrats connus du composant. D'après les propriétés P1 et P2 (cf. page 141) l'utilisateur ne peut avoir plus d'un contrat applicable pour un couple (identité logique, état de présence) donné.
- Construire un ensemble de contrats en ajoutant récursivement les contrats hérités explicitement et implicitement des contrats déjà présents dans l'ensemble.

Dans un soucis d'optimisation des performances, la construction de cet ensemble de contrats ne sera opérée qu'à l'instanciation du composant. Tout comme les services, Le gestionnaire de contrats liée au composant utilisateur nouvellement créé s'abonnera auprès de l'Agent de Présence de l'utilisateur dont il dépend, pour reconstruire l'ensemble des contrats de l'utilisateur. Ce processus sera reproduit pour chaque changement de l'état global de

l'utilisateur. La Figure 77 nous donne un exemple d'ensemble de contrats, issu de la première étape précisée ci-dessus. C1 est le contrat associé au couple (utilisateur, état de présence) et G1, G2, G3 sont les groupes auxquels appartient notre utilisateur. D'après les hypothèses définies au moment de la définition de l'héritage implicite, nous pouvons construire un premier arbre, que nous appèlerons arbre d'héritage implicite, des contrats compatibles avec l'état de présence de l'utilisateur, issus des groupes d'appartenance de notre utilisateur. Nous appèlerons réciproquement arbre d'héritage explicite l'arbre construit sur les dépendances explicites entre contrats. Ces deux arbres possèdent le contrat unique de l'utilisateur comme racine.

La seconde étape de traitement, réévaluée cette fois à chaque appel de méthode sur le composant, consiste à construire un ensemble de règles ordonné pour chacune des six classes de règles introduites dans la partie 5.6.2.2. D'après la propriété P3, l'arbre d'héritage implicite est prioritaire sur l'arbre d'héritage explicite. Aussi la construction de chaque ensemble ordonné de règles commence par un traitement de l'arbre d'héritage implicite, avant de s'intéresser à l'arbre d'héritage explicite. Le traitement d'un arbre d'héritage passe par un premier parcours en profondeur, à la recherche des règles exécutables (dont le trigger et la condition sont compatibles avec le message SIP reçu) de type MUST, puis un second parcours pour les règles exécutables de type MAY. Chaque contrat étant ordonné, le parcours en profondeur des arbres d'héritage produira une première suite ordonnées de règles MUST, puis une seconde de type MAY, provenant d'une série ordonnée de contrats contenant eux-mêmes des séries de règles ordonnées.

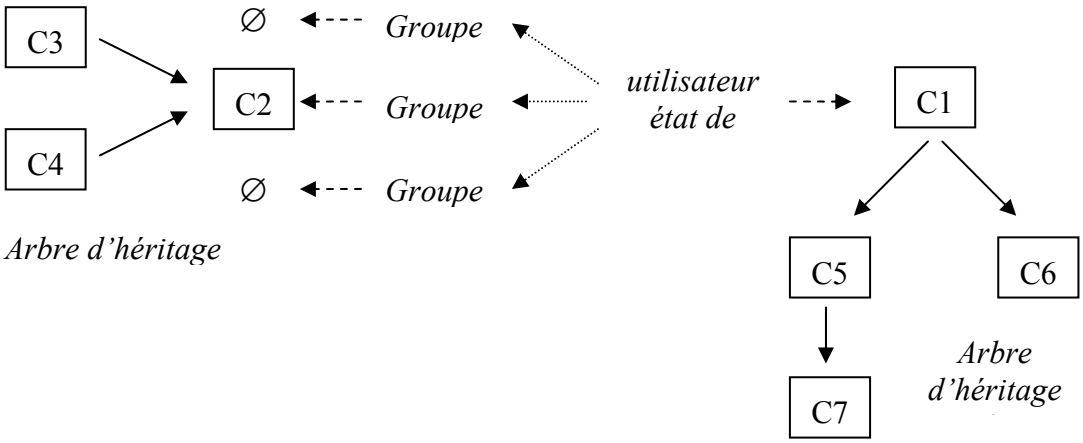


Figure 77 L'héritage de contrats

Pour les quatre situations de synchronisation (aller `before` et `after`, retour `before` et `after`), notre composant devra utiliser l'ensemble de règles ordonné adéquat, pour appliquer successivement la règle de type MUST de plus haute priorité non encore appliquée, puis la règle de type MAY de plus haute priorité non encore appliquée, tant qu'aucune règle ne demande à suspendre ou détourner l'appel de méthode originel. L'enchaînement suivra la même logique pour les deux situations de synchronisation de type `replace`, si ce n'est que l'application successive des règles ne s'achèvera pas cette fois par un appel initial à la méthode du composant.

5.6.3 La prise en charge des contrats

Destinés à coordonner l'exécution de services suite à une demande initiale, les contrats ont à priori vocation à être utilisés au niveau des facettes de collaboration, même si aucune

contrainte technique n'empêche leur implantation au niveau de réceptacles. En faisant référence à notre architecture d'exécution (cf. Figure 50 et Figure 60), la prise en charge de nos contrats de coordination peut être effectuée grâce à une entité logicielle capable de s'interfacer entre les DialogListener, DialogSender et Squeletton de la facette de collaboration (cf. Figure 78). D'après l'API Jain SIP Lite, cette entité captera sur le chemin aller la représentation objet du message SIP reçu par le `DialogListener`. Réciproquement, elle disposera du message initial et du corps du message adressé au `DialogSender` lors du retour.

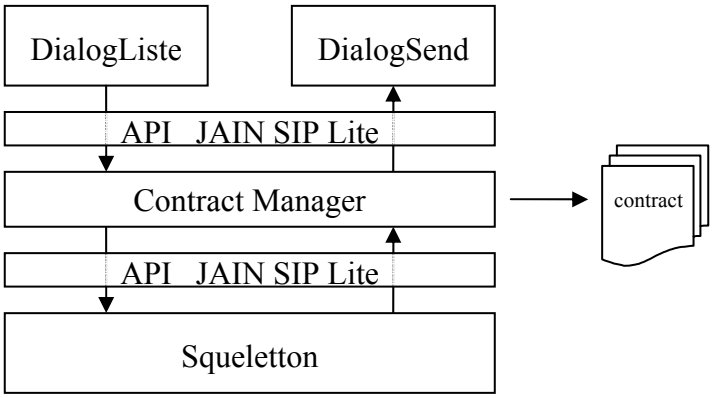


Figure 78 Le gestionnaire de contrats dans notre architecture d'exécution

Les contrats de collaboration étant intimement liés à la structure des utilisateurs, des groupes d'utilisateurs et leurs états de présence, leur stockage doit être naturellement assumé par les Agents de Présence des utilisateurs. Chaque SA souhaitant utiliser ces contrats se verra retourner ces contrats en réponse à sa demande de configuration, matérialisée par l'envoi d'un message SIP OPTIONS (cf. partie 5.5.2.3). De même le SA aura la possibilité de s'abonner, pour suivre l'évolution de l'état de l'utilisateur, et le cas échéant reconstruire l'ensemble des contrats de l'utilisateur.

5.6.4 L'utilisation des contrats de coordination

5.6.4.1 La composition de services

5.6.4.1.1 La composition mono-contrat

La composition de services souscrits par l'utilisateur est destinée à être l'utilisation la plus courante de nos contrats de coordination. Pour ce faire, l'utilisateur doit commencer par définir, par état de présence, les agents de service qu'il souhaite rendre disponible, en précisant au passage leur statut par défaut.

Admettons que notre utilisateur logique `afontaine@wanadoo.fr` souhaite définir un état de présence `"en conférence"`, dans lequel il déclare deux de ses services, en utilisant les identifiants de services définis au niveau de son PA. Il sélectionne un premier service : `PicturePrinter`, proposé par son Détaillant `wanadoo.fr`, pour imprimer et envoyer automatiquement par voie postale des images numériques. Ce service dispose d'un agent centralisant l'ensemble des données utiles à son fonctionnement pour le compte d'un utilisateur (format d'impression préféré, adresse postale, système de paiement préféré,…), dont l'instance dédiée à notre utilisateur est accessible via l'adresse SIP

afontaine123@picprinter.wanadoo.fr. Notons au passage que notre Détaillant peut n'assumer que la partie utilisateur du service, l'exécution proprement dite pouvant faire l'objet d'un accord avec un Fournisseur de Services Tiers, pour que le composant utilisateur associé à l'agent de service, entre par exemple en relation avec un composant session du domaine kodak.com. Notre utilisateur choisi d'ajouter à son état de présence un second service, capable de réceptionner des messages multimédia rejoués ensuite sur une application du terminal de l'utilisateur, qu'il fixe cette fois dans un statut "offline". Fixer ce statut de l'application ne signifie pas que physiquement l'application du terminal sera rendue indisponible, mais que le PA ne tiendra pas compte des notifications de changement de statut de l'AUA, pour présenter invariablement un statut "offline" à tout service tentant d'accéder au SA de l'utilisateur.

```
<presence user="afontaine"
          domain="wanadoo.fr"
          presence-state="en conference">
  <tuple>
    <status>
      <default>available</default>
    </status>
    <service>
      <name>PicturePrinter</name>
      <agent>afontaine123@picprinter.wanadoo.fr</agent>
    </service>
  </tuple>
  <tuple>
    <status>
      <static>offline</static>
    </status>
    <service>
      <name>MultimediaMessaging</name>
      <address>afontaine@mms.wanadoo.fr</address>
    </service>
  </tuple>
</presence>
```

Sur la base de cet état de présence, notre utilisateur peut bâtir ses contrats de coordination, en faisant référence aux identifiants d'agents de services définis. Admettons que notre utilisateur souhaite rediriger l'ensemble des messages présentés à son agent de service MultimediaMessaging hors ligne, vers son agent du service d'impression PicturePrinter.

```
Contract        Arnaud en conférence <conference>
Presence-agent  afontaine@wanadoo.fr
Presence-state  en conference
Participants    PicturePrinter p, MultimediaMessaging m
Invariant       p.pa = m.pa = afontaine@wanadoo.fr

Coordination
Rule_1 : MUST : replace
   When  * ->> SIP: NOTIFY (Target t, From f)
   With  t=m AND name(env:/Enveloppe/last())=sendMessage
```

```
          SIP: NOTIFY(Target p, Via m)
          {
            <env:Enveloppe>
              <printPicture>
   Do           <xsl:value-of
          select=”env:Enveloppe/sendMessage/”
              </printPicture>
            </env:Enveloppe>
          }
End
```

Pour ce faire, il devra créer une règle de coordination, interceptant un appel à la méthode `sendMessage` du composant utilisateur associé à l'agent `MultimediaMessaging` destinataire d'un message SIP NOTIFY. Pour tester cet appel de méthode nous utilisons la notation XPath, afin de parcourir le corps du message SIP à la recherche de l'élément XML contenu dans l'enveloppe SOAP : `env:/Enveloppe/last()`. Notre règle précise ensuite l'entête du message NOTIFY à envoyer à l'agent du service d'impression `PicturePrinter`, en indiquant la méthode `printPicture` invoquée sur le composant sous-jacent, à laquelle nous fournissons en entrée le message multimédia initialement reçu sur l'agent du service `MultimediaMessaging`.

5.6.4.1.2 La composition multi-contrats

Dans un environnement multi-fournisseurs un utilisateur peut déployer des contrats en plusieurs points du réseau, dans des domaines de Détaillants distincts, ayant néanmoins la nécessité de coopérer pour concourir à un fonctionnement des agents des services de l'utilisateur en phase avec ses aspirations. Admettons qu'un utilisateur dispose d'un Détaillant `wanadoo.fr` pour son environnement privé, et d'un Détaillant `enst.fr` pour ce qui concerne son activité professionnelle. Notre utilisateur, grand consommateur du service de tableau blanc, a souscrit au service `boardSynchronizer` auprès de son Détaillant `wanadoo.fr` et au service `tableauBlanc` du Détaillant `enst.fr`. En complément de ces deux souscriptions, notre utilisateur souhaite définir un état de présence `”au travail”` pour son identité logique `afontaine@wanadoo.fr`, durant lequel il redirigera, pour une liste d'utilisateurs, les invitations à destination de son agent du service `boardSynchronizer` vers son second agent du service `tableauBlanc`, alors que pour les autres utilisateurs il retournera un message signifiant son indisponibilité temporaire. Parallèlement à cela, notre utilisateur souhaite créer un état de présence `”en séance de travail”` pour cette fois son identité logique `afontaine@enst.fr` dans laquelle tout utilisateur à l'origine d'une demande d'invitation vers son agent du service `tableauBlanc`, se verra retourner une proposition de rappel dès le changement d'état de l'utilisateur, excepté pour les invitations en provenance de son Détaillant `wanadoo.fr`, systématiquement honorées. Le premier contrat de collaboration déployé dans le domaine `wanadoo.fr` s'ordonnera ainsi :

```
Contract          Au travail à l'ENST <travail>
Presence-agent    afontaine@wanadoo.fr
Presence-state    Au travail
                  BoardSynchronizer wb, TableauBlanc tb,
Participants      Contacts c
Invariant         wb.pa = c.pa = afontaine@wanadoo.fr
                  tb.pa = afontaine@enst.fr
```

```
Coordination
Rule_1 : MUST : replace
   When  * ->> SIP: INVITE (Target t, From f)
   With  t=wb AND f in c
   Do    Call Is Being Forwarded <SIP: 181 ()>
         SIP: INVITE(To tb, Via wb)

Rule_2 : MUST : replace
   When  * ->> SIP: INVITE (Target t, From f)
   With  t=wb
   Do    Moved Temporarily <SIP: 302()>
End
```

Ce premier contrat contiendra deux règles comme autant de cas de figure du comportement des services demandé par l'utilisateur. Une première règle précise qu'un utilisateur appartenant à une liste de contacts à l'origine d'un message INVITE sur un agent de service `BoardSynchronizer` se verra retourner un message SIP 181, tandis que sa requête sera transformée en INVITE sur un agent `TableauBlanc`, appartenant cette fois au domaine `enst.fr`. La définition d'une telle règle suppose que :

- Les agents de services sont adressables entre Détaillants, ainsi l'agent de service `TableauBlanc` du Détaillant `enst.fr` peut être déclaré au sein d'un état de présence de notre utilisateur logique `afontaine@wanadoo.fr`, pour être utilisable dans son contrat de collaboration. Cette propriété critique suppose que les différents opérateurs sont disposés à rendre publique les interfaces de leurs services, pour permettre la construction de contrats inter-Détaillants.
- Une règle peut occasionner un ensemble d'actions. De fait la règle Rule_1 ci-dessus commence par répondre un message SIP 181 à l'initiateur du message INVITE avant d'émettre un second message INVITE vers l'agent de service `TableauBlanc`.

Une seconde règle, moins prioritaire car placée derrière la première règle, précise qu'un message INVITE provenant d'un utilisateur hors de la liste des contacts recevra en retour un message SIP 302, sans autre alternative.
Notre utilisateur devra ensuite déployer un contrat dans le domaine `enst.fr`, précisé comme suit :

```
Contract          Séance de Travail en équipe <seance>
Presence-agent    afontaine@enst.fr
Presence-state    en seance de travail
Participants      WhiteBoard wb, TableauBlanc tb
Invariant         wb.pa = c.pa = afontaine@wanadoo.fr
                  tb.pa = afontaine@enst.fr

Coordination
Rule_1 : MUST : before
   When  * ->> SIP: INVITE (Target t, From f, Via v)
   With  t=tb AND v=afontaine789@WhiteBoard.wanadoo.fr
   Do    Proceed
```

```
Rule_2 : MUST : replace
   When  * ->> SIP: INVITE (Target t, From f)
   With  t=tb
         Queued <SIP: 182 ()>
          create_rule Rule3 : MAY : after
           When tb.aua ->> SIP: NOTIFY(From f, To t)
   Do      With f=WhiteBoard@mortimer.enst.fr AND
                   t=afontaine@enst.fr
         Do   SIP: INVITE (Target t, From f)
End
```

Une première règle intercepte les message INVITE en provenance du domaine wanadoo.fr. A ce effet, nous utilisons la possibilité offerte par SIP, qu'un proxy ou serveur SIP intermédiaire marque son intervention dans le processus de traitement du message, en ajoutant une entrée `Via` dans l'entête. Ainsi pour tout message SIP ayant emprunté l'agent de service `afontaine789@WhiteBoard.wanadoo.fr`, le message INVITE poursuivra son cours sans altération. En revanche, un message SIP à destination de l'agent `TableauBlanc`, n'ayant pas subi de transfert, sera mis en attente en retournant d'une part un message 182 à son émetteur, et en créant une règle (qui devra dans la pratique être implantée au niveau du PA), capable d'intercepter un message de changement de statut de l'application utilisateur, pour le cas échéant réveiller la demande d'invitation mise en veille.

Les deux contrats présentés ci-dessus sont des exemples de composition indirecte de contrats, ceux-ci n'ayant pas de lien direct entre eux, l'utilisation de l'entrée `Via` étant leur seul point de contact. L'héritage de contrat permet à l'opposé de définir une composition de contrats plus directe. Admettons par exemple que notre utilisateur `afontaine@enst.fr` souhaite définir un contrat `"Contacter d'urgence Elie"` précisant que tout invitation émanant de l'utilisateur `najm@enst.fr`, pour quelque service que ce soit, sera transformée en une mise en relation directe via un service `SoftPhone`. Ce contrat devra créer une règle consommant le message INVITE initial, pour le substituer à une invitation à une session se service mobilisant cette fois le service SoftPhone, auquel seront rattachés nos deux utilisateurs.

```
Contract        Contacter d'urgence Elie <urgent>
Presence-agent  afontaine@enst.fr
Presence-state  ANY
Participants    SoftPhone sp
Invariant       sp.pa = afontaine@enst.fr

Coordination
Rule_3 : MUST : replace
   When  * ->> SIP: INVITE (From f)
   With  f=najm@enst.fr
         SIP: INVITE(Target sp.srv, From afontaine@enst.fr, To
         sp.srv)
   Do
         SIP: NOTIFY(Target sp, From afontaine@enst.fr, To
         najm@enst.fr)
End
```

Un premier message INVITE créera ainsi la session de service, dans laquelle figure l'utilisateur `afontaine@enst.fr`, que le second message SIP NOTIFY étendra à l'utilisateur `najm@enst.fr`. Implicitement, cette règle ne donnera pas suite à la demande d'invitation initiale, pour provoquer une relation entre nos deux utilisateurs, via le service `SoftPhone`. Pour être pleinement utilisable, ce contrat doit être vu comme un contrat prioritaire, le contrat `Séance de Travail en équipe` introduit en page précédente sera ainsi déclaré en :

```
Seance : urgent@afontaine.enst.fr
```

D'après les hypothèses formulées sur le traitement de l'héritage de contrat, `Rule_3` sera toujours une règle prioritaire sur les règles `Rule_1` et `Rule_2`. De fait un message INVITE à destination du service `TableauBlanc` en provenance de l'utilisateur `najm@enst.fr` sera automatiquement consommé pour faire place à une mise en relation.

5.6.4.2 La personnalisation du comportement des services

Nos contrats de collaboration peuvent être également utilisés dans un second contexte de personnalisation par l'utilisateur du comportement des services. Cette spécialisation repose sur le principe qu'un message SIP INVITE peut véhiculer un certain nombre d'informations utile à l'édification de la session de service, que l'utilisateur pourra préciser à l'intérieur d'une règle de collaboration.

Admettons qu'un utilisateur définisse un premier contrat, qui reste pour le moment conforme aux règles précédemment exposées, chargé de transformer un message INVITE adressé à son Agent de service SoftPhone en un message NOTIFY envoyé cette fois à son agent du service de messagerie.

```
Rule : MUST : replace
   When  * ->> SIP: INVITE (Target t, From f, To t)
   With  f=softPhoneSA
   Do    SIP: NOTIFY(Target messagingSA, From f, To t)
```

Notre utilisateur peut à ce moment définir un second contrat, au niveau de son service de messagerie, pour préciser le comportement que devra adopter l'agent face à une demande d'exécution du service. L'implantation de ces contrats est ainsi données en Figure 79.

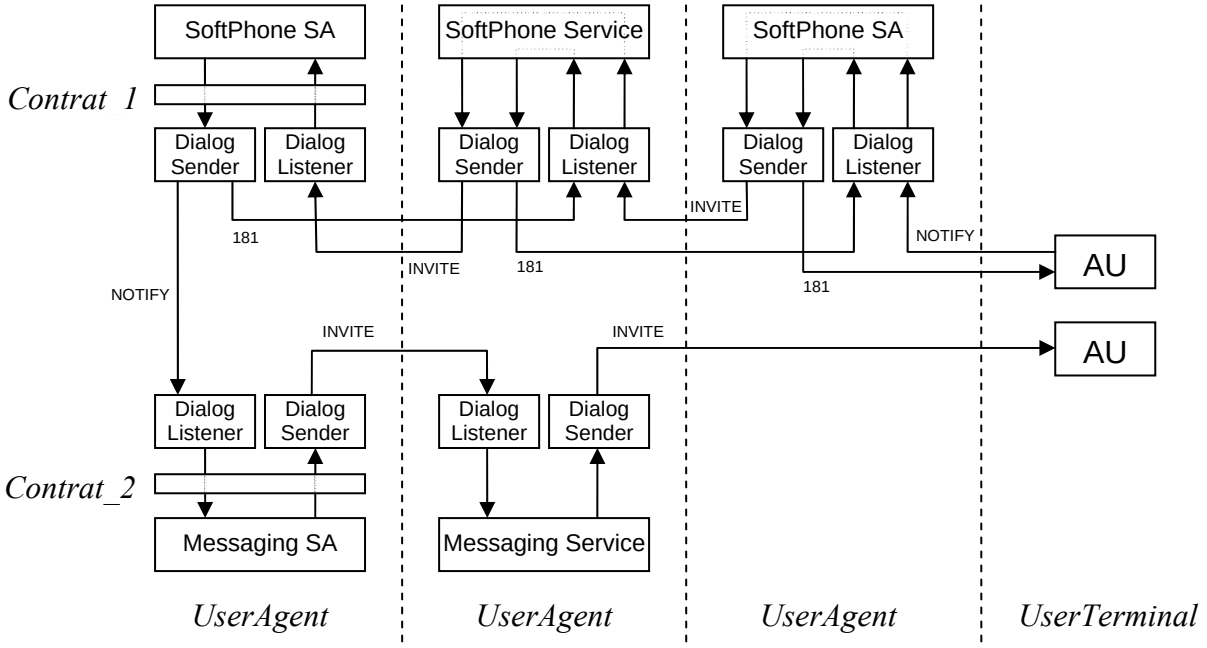


Figure 79 La répartition des contrats sur les agents de service

Arrivé auprès de l'agent du service de messagerie, le message NOTIFY et transformé en message INVITE, conformément au propriété exposées dans la partie 5.5.2.4, pour adresser au service une invitation à la session, à destination de l'utilisateur distant. Le service de messagerie a cependant de particulier qu'il ne nécessite pas l'utilisation d'un agent de service du coté de l'utilisateur invité, ce qui permet au service d'accéder directement à l'AUA de l'utilisateur. Cette opération passe donc par une interrogation du PA de l'utilisateur, par le service lui-même, pour localiser l'AUA compatible avec les contraintes d'implémentation du service de messagerie. Ainsi un service basé sur l'utilisation de flux audio pour la gestion des annonces, cherchera à localiser un AUA d'une application de gestion de tels flux sur le terminal de l'utilisateur, afin de lui envoyer le message INVITE, dans le but d'ouvrir une session RTP. A l'opposé un service de messagerie basé sur des pages web exposant par exemple une photographie et un texte rédigé par l'abonné demandera au service de localiser un AUA correspondant à un navigateur Web, pour lui signifier l'emplacement de cette page.

Une règle permettant à un agent de service de messagerie de diriger un utilisateur vers une annonce déposée par notre abonné sur un serveur vocal pourra ainsi être :

```
Rule : MUST : replace
   When   SIP: INVITE () <<- SIP: NOTIFY (Target t)
   With   t=messaging
          SIP: INVITE(Content-Type application/sdp)
          {
            v=0
            o=fontaine 2890844526 2890844526 IN IP4 wanadoo.fr
            s=Session SDP
   Do       c=IN IP4 100.101.102.103
            t=0 0
            m=audio 49172 RTP/AVP 0
            a=rtpmap:0 PCMU/8000
          }
```

Ainsi une fois le message INVITE construit par l'agent de service, nous lui ajoutons un corps SDP permettant de localiser le message d'annonce enregistré par l'abonné. Si d'aventure notre utilisateur souhaitait passer par une e page web d'accueil, pour se présenter il pourrait créer la règle suivante :

```
Rule : MUST : replace
   When  SIP: INVITE (To to) <<- SIP: NOTIFY (Target ta)
   With  ta=messaging AND to in Contacts
         SIP: INVITE(Content-Type application/sdp)
         {
           v=0
           i=Page d'accueil de la messagerie d'Arnaud Fontaine
   Do      u=http://www.enst.fr/afontaine/messagerie.html
           e=afontaine@enst.fr (Arnaud Fontaine)
           m=application 32416 udp wb
         }
```

Ces deux règles produisent un message SIP INVITE encapsulant un corps SDP, recueilli par l'agent de l'application du terminal pour se connecter sur une ressource de l'utilisateur. Parallèlement à cela, le composant session jouant le rôle d'intermédiaire entre l'agent de service et l'agent de l'application met en œuvre un mécanisme de collecte du message, qu'il pourra par exemple héberger sur un serveur vocal, pour retourner à l'agent de service (au sein du message OK 200) un corps de message SDP synthétisant les données nécessaire à sa lecture ultérieure par l'abonné.

Les règles de coordination peuvent à ce moment aider notre abonné à paramétrer le comportement de son agent de service, pour proposer à l'utilisateur distant de réécouter, voir de détruire son message enregistré. Si nous partons du principe que l'agent repose sur un composant utilisateur proposant les méthodes :

- `[messageId]  getAllMessage(String userId)`
- `SDPBody  readMessage(messageId)`
- `void  removeMessage(messageId)`

Notre utilisateur peut par exemple décider d'autoriser son correspondent à récupérer la liste des messages dont il est l'auteur, pour le cas échéant les réécouter ou les supprimer, grâce aux règles définies ci-dessous. Une première règle présente une méthode à l'agent de l'application de l'utilisateur distant, capable d'importer la liste des messages que l'abonné lui autorise à invoquer. Une seconde règle, exécutable sur le retour de l'appel à la méthode `getAllMessages()` véhiculant les identifiants de messages de l'agent de service vers l'agent d'application, autorise l'utilisateur distant à faire appel à aux méthodes `readMessage` et `removeMessage`, normalement au seul usage de l'abonné.

```
Rule : MUST : after
   When  SIP: * ->> 200 OK (Target ta, From f)
   With  ta=messagingSA
```

```
          SIP: NOTIFY(To f, From afontaine@enst.fr)
          {
           <env:Envelope xmlns:env="http://www.w3.org/soap-
          envelope">
              <env:Body>
                 <m:getAllMessages>
   Do              <a:userId>
                      <a:string>f</a:string>
                   </a:userId>
                 </m:getAllMessages>
              </env:Body>
           </env:Envelope>
          }

Rule : MUST : replace
   When   SIP: NOTIFY <<- SIP: NOTIFY (Target ta, From f)
          t=messagingSA AND
   With   name(env:/Enveloppe/last())=getAllMessages
          SIP: NOTIFY()
          {
           <env:Envelope xmlns:env="http://www.w3.org/soap-
          envelope">
              <env:Body>
                 <m:readMessage>
                   <a:messageId>
                      <a:string/>
                   </a:messageId>
   Do            </m:readMessage>
                 <m:removeMessage>
                   <a:messageId>
                      <a:string/>
                   </a:messageId>
                 </m:removeMessage>
              </env:Body>
            </env:Envelope>
           }
```

6 Conclusion et perspectives

Après nous être intéressé aux différentes phases de l'analyse d'un service, en étendant le formalisme UML classique, et en définissant un nouveau type de diagramme, nous avons proposé un modèle de composant alliant une partie programmable à des interfaces de description. Sur cette base de composant, nous avons proposé des extensions originales, permettant la prise en charge de contraintes et de contreparties liées au contexte d'exécution de nos services. Notre présentation s'est enfin attardée sur la mise en œuvre de notre architecture, de par l'utilisation du protocole SIP, du service de présence et des contrats de coordination.

Par l'ouverture de son éventail technique, ce manuscrit reflète la grande variété des technologies que j'ai pu rencontrer au cours de ma thèse. De fait ce manuscrit a été pour moi l'occasion de concilier des travaux parfois antagonistes (Réseau Intelligent, TINA, SIP, EJB, CCM, dotNET, …) pour porter dans un environnement télécom les dernières avancées informatiques que j'ai pu expérimenter. Aussi ce manuscrit de thèse constitue-t-il pour moi une première étape d'édification de principes s'inscrivant dans une démarche personnelle de construction d'une architecture de services.

L'architecture déjà ici définie se tourne naturellement vers l'utilisation de contrats, qui pour le moment permettent à l'utilisateur final de personnaliser directement le contexte et les enchaînements d'exécution de ses services. Cette utilisation des contrats reste cependant un aspect limité du potentiel dont ils disposent dans une architecture de composants. En effet, de tels contrats peuvent permettre d'appréhender entre autre des problèmes d'interaction maligne de service, les problèmes de prise de décision en cas conflit d'intérêt entre utilisateurs.

7 Bibliographie

[ADE99] M. Arango, A. Dugan, I. Elliott, C. Huitema, and S. Pickett, Media gateway control protocol (MGCP) version 1.0, Request for Comments 2705, IETF, October 1999.

[AKL03] D. Atkins, G. Klyne, Common Presence and Instant Messaging: Message Format, draft-ietf-simple-impp-cipm-msgfmt-08, Internet draft of IETF Internet Engineering Task Force, January 2003

[CAR02] B. Campbell, J. Rosenberg, CPIM Mapping of SIMPLE Presence and Instant Messaging, draft-ietf-simple-cipm-mapping-00, Internet draft of IETF Internet Engineering Task Force, August 2002

[CAS01] B. Campbell, R. Sparks, Control of Service Context using SIP Request-URI, Request for Comments 3087, IETF, April 2001

[CCM01] E. Christensen, F. Curbera, G Meredith, S. Weerawarana, Web Services Description Language (WSDL), W3C Note, http://www.w3.org/TR/2001/NOTE-wsdl-20010315, March 2001

[CGH00] F. Cuervo, N. Greene, C. Huitema, A. Rayhan, B. Rosen, and J. Segers, Megaco protocol 1.0, Request for Comments 3015, IETF, November 2000.

[DJS02] A. Johnston, S. Donovan, R.Sparks, C. Cunningham, D. Willis, J. Rosenberg, K. Summers, H. Schulzrinne, SIP Call Flow Examples, draft-ietf-sipping-call-flows-00, IETF, February 2002.

[GAL00] Lim, W., Galis, A., Active Networks Enterprise Model, proceedings of the 5th London Communications Symposium, University College London, September 2000

[GOS03] Project Open Source GoSIP, an implementation of JSR125, http://sourceforge.net/projects/gosip/

[GRR00] N. Greene, M. Ramalho, and B. Rosen, Media gateway control protocol architecture and requirements, Request for Comments 2805, IETF, April 2000.

[HAJ98] M. Handley, V. Jacobson, SDP: Session Description Protocol, Request for Comments 2327, IETF, April 1998.

[HSS99] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg, SIP: Session Initiation Protocol, Request for Comments 2543, IETF, Mars 1999.

[JAS02] Jain SIP Lite Specification, http://www.jcp.org/en/jsr/detail?id=125, public review july 2002.

[LRS00] M. Day Lotus, J. Rosenberg, H. Sugano, A Model for presence and instant messaging, Request for Comments 2778, IETF, February 2000.

[MCJ02] Mahy, Campbell, Johnston, Petrie, Sparks and Rosenberg, A Multi-party Application Framework for SIP, draft-ietf-sipping-cc-framework-00.txt, Internet draft of IETF SIPPING working group, February 2002

[MME01] R. Marvie, P. Merle, CORBA Component Model : Discussion and Use with OpenCCM, Special issue of Informatica, dedicated to component based development, June 2001

[PAR00] Group PARLAY http://www.parlay.org

[RLS99] J. Rosenberg, J. Lennox and H. Schulzrinne, Programming Internet Telephony Services, IEEE Network Magazine, Vol. 13, No. 3, May/June 1999, pg. 42-49

[RMS01] J. Rosenberg, Mataga and H. Schulzrinne, An Application Server Component Architecture for SIP, draft-rosenberg-sip-app-components-01.txt, Internet draft of IETF Internet Engineering Task Force, March 2001

[ROS02] J. Rosenberg, A Presence Event Package for the Session Initiation Protocol (SIP), draft-ietf-simple-presence-09, Internet draft of IETF Internet Engineering Task Force, December 2002.

[RSC02] J. Rosenberg and H. Schulzrinne, Guidelines for Authors of Extensions to the Session Initiation Protocol (SIP), draft-ietf-sip-guidelines-06, Internet draft of IETF Internet Engineering Task Force, November 2002.

[TIM98] P. Timmers, Business Models for Electronic Market, European Commission, April 1998

[TIN97] TINA Consortium, TINA Business Model and Reference Points, 22nd May 1997.

[UML04] UML – Unified Modeling Language http://www.uml.org.

8 Annexes

8.1 Annexe 1 : Fichier WSDL du composant service1

```
<?xml version="1.0" encoding="utf-8" ?>
<definitions targetNamespace="http://tempuri.org/"
             xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:s="http://www.w3.org/2001/XMLSchema"
             xmlns:s0="http://tempuri.org/"
             xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
             xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
             xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
             xmlns="http://schemas.xmlsoap.org/wsdl/">
   <types>
      <s:schema elementFormDefault="qualified" targetNamespace="http://tempuri.org/">
         <s:element name="m">
            <s:complexType>
               <s:sequence>
                  <s:element minOccurs="0" maxOccurs="1" name="p" type="s:string" />
               </s:sequence>
            </s:complexType>
         </s:element>
         <s:element name="mResponse">
            <s:complexType>
               <s:sequence>
                  <s:element minOccurs="0" maxOccurs="1" name="mResult" type="s0:R" />
               </s:sequence>
            </s:complexType>
         </s:element>
         <s:complexType name="R" />
         <s:element name="R" nillable="true" type="s0:R" />
      </s:schema>
   </types>
   <message name="mSoapIn">
      <part name="parameters" element="s0:m" />
   </message>
   <message name="mSoapOut">
      <part name="parameters" element="s0:mResponse" />
   </message>
   <message name="mHttpGetIn">
      <part name="p" type="s:string" />
   </message>
   <message name="mHttpGetOut">
      <part name="Body" element="s0:R" />
   </message>
   <message name="mHttpPostIn">
      <part name="p" type="s:string" />
   </message>
   <message name="mHttpPostOut">
      <part name="Body" element="s0:R" />
   </message>
   <portType name="Service1Soap">
      <operation name="m">
         <input message="s0:mSoapIn" />
         <output message="s0:mSoapOut" />
      </operation>
   </portType>
   <portType name="Service1HttpGet">
      <operation name="m">
         <input message="s0:mHttpGetIn" />
         <output message="s0:mHttpGetOut" />
      </operation>
   </portType>
   <portType name="Service1HttpPost">
      <operation name="m">
         <input message="s0:mHttpPostIn" />
         <output message="s0:mHttpPostOut" />
      </operation>
   </portType>
   <binding name="Service1Soap" type="s0:Service1Soap">
      <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document" />
      <operation name="m">
         <soap:operation soapAction="http://tempuri.org/m" style="document" />
         <input>
            <soap:body use="literal" />
         </input>
         <output>
            <soap:body use="literal" />
         </output>
```

```
      </operation>
   </binding>
   <binding name="Service1HttpGet" type="s0:Service1HttpGet">
      <http:binding verb="GET" />
      <operation name="m">
         <http:operation location="/m" />
         <input>
            <http:urlEncoded />
         </input>
         <output>
            <mime:mimeXml part="Body" />
         </output>
      </operation>
   </binding>
   <binding name="Service1HttpPost" type="s0:Service1HttpPost">
      <http:binding verb="POST" />
      <operation name="m">
         <http:operation location="/m" />
         <input>
            <mime:content type="application/x-www-form-urlencoded" />
         </input>
         <output>
            <mime:mimeXml part="Body" />
         </output>
      </operation>
   </binding>
   <service name="Service1">
      <port name="Service1Soap" binding="s0:Service1Soap">
         <soap:address location="http://localhost/WebService1/Service1.asmx" />
      </port>
      <port name="Service1HttpGet" binding="s0:Service1HttpGet">
         <http:address location="http://localhost/WebService1/Service1.asmx" />
      </port>
      <port name="Service1HttpPost" binding="s0:Service1HttpPost">
         <http:address location="http://localhost/WebService1/Service1.asmx" />
      </port>
   </service>
</definitions>
```

8.2 Annexe 2 : Flot de messages SIP, pour une connexion utilisant deux proxy.

Scénario d'une connexion entre User A (`fontaine@wanadoo.fr`) et User B (`najm@enst.fr`)

```
/***  User  A  réserve  le  port  TCP  49172  sur  son  terminal  (adresse  IPv4
      100.101.102.103), pour l'établissement d'une connexion RTP avec User B ***/
```

F1 INVITE User A -> Proxy A

```
INVITE sip:najm@enst.fr SIP/2.0
Via: SIP/2.0/UDP wanadoo.fr:5060;branch=z9hG4bK74bf9
Max-Forwards: 70
From: Arnaud <sip:fontaine@wanadoo.fr>;tag=9fxced76sl
To: Elie <sip:najm@enst.fr>
Call-ID: 12345601@wanadoo.fr
CSeq: 1 INVITE
Contact: <sip:fontaine@100.101.102.103>
Content-Type: application/sdp
Content-Length: 147

v=0
o=fontaine 2890844526 2890844526 IN IP4 wanadoo.fr
s=Session SDP
c=IN IP4 100.101.102.103
t=0 0
m=audio 49172 RTP/AVP 0
a=rtpmap:0 PCMU/8000
```

```
/*** Le Proxy A demande à User A de s'authentifier ***/
```

F2 407 Proxy Authorization Required Proxy A -> User A

```
SIP/2.0 407 Proxy Authorization Required
Via: SIP/2.0/UDP enst.fr:5060;branch=z9hG4bK74bf9;received=100.101.102.103
From: Arnaud <sip:fontaine@wanadoo.fr>;tag=9fxced76sl
To: Elie <sip:najm@enst.fr>;tag=3flal12sf
Call-ID: 12345601@wanadoo.fr
CSeq: 1 INVITE
Proxy-Authenticate: Digest realm="security.com",
   nonce="f84f1cec41e6cbe5aea9c8e88d359",
   opaque="", stale=FALSE, algorithm=MD5
Content-Length: 0
```

F3 ACK User A -> Proxy A

```
ACK sip:Najm@enst.fr SIP/2.0
Via: SIP/2.0/UDP wanadoo.fr:5060;branch=z9hG4bK74bf9
Max-Forwards: 70
From: Arnaud <sip:fontaine@wanadoo.fr>;tag=9fxced76sl
To: Elie <sip:najm@enst.fr>;tag=3flal12sf
Call-ID: 12345601@wanadoo.fr
CSeq: 1 ACK
Content-Length: 0
```

```
/*** User A réémet son message INVITE contenant les données cryptées
     d'authentification ***/

F4 INVITE User A -> Proxy A

INVITE sip:najm@enst.fr SIP/2.0
Via: SIP/2.0/UDP wanadoo.fr:5060;branch=z9hG4bK74bf9
Max-Forwards: 70
From: Arnaud <sip:fontaine@wanadoo.fr>;tag=9fxced76sl
To: Elie <sip:najm@enst.fr>
Call-ID: 12345601@wanadoo.fr
CSeq: 2 INVITE
Contact: <sip:fontaine@100.101.102.103>
Proxy-Authorization:Digest username="gabolaud", realm="security.com",
 nonce="wf84f1ceczx41ae6cbe5aea9c8e88d359", opaque="",
 uri="sip:fontaine@wanadoo.fr", response="42ce3cef44b22f50c6a6071bc8"
Content-Type: application/sdp
Content-Length: 147

v=0
o=fontaine 2890844526 2890844526 IN IP4 wanadoo.fr
s=Session SDP
c=IN IP4 100.101.102.103
t=0 0
m=audio 49172 RTP/AVP 0
a=rtpmap:0 PCMU/8000


/*** Proxy A valide les données d'authentification de User A et transmit le message
     INVITE au Proxy B. On fait l'hypothèse que Proxy A et Proxy B se sont
     préalablement authentifiés ***/

F5 INVITE Proxy A -> Proxy B

INVITE sip:najm@enst.fr SIP/2.0
Via: SIP/2.0/UDP proxyA.wanadoo.fr:5060;branch=z9hG4bK2d4790.1
Via: SIP/2.0/UDP wanadoo.fr:5060;branch=z9hG4bK74bf9;received=100.101.102.103
Max-Forwards: 69
Record-Route: <sip:proxyA.wanadoo.fr;lr>
From: Arnaud <sip:fontaine@wanadoo.fr>;tag=9fxced76sl
To: Elie <sip:najm@enst.fr>
Call-ID: 12345601@wanadoo.fr
CSeq: 2 INVITE
Contact: <sip:fontaine@100.101.102.103>
Content-Type: application/sdp
Content-Length: 147

v=0
o=fontaine 2890844526 2890844526 IN IP4 wanadoo.fr
s=Session SDP
c=IN IP4 100.101.102.103
t=0 0
m=audio 49172 RTP/AVP 0
a=rtpmap:0 PCMU/8000


/*** Proxy A informe User A qu'il tente de joindre User B ***/

F6 (100 Trying) Proxy A -> User A

SIP/2.0 100 Trying
Via: SIP/2.0/UDP wanadoo.fr:5060;branch=z9hG4bK74bf9;received=100.101.102.103
From: Arnaud <sip:fontaine@wanadoo.fr>;tag=9fxced76sl
To: Elie <sip:najm@enst.fr>
Call-ID: 12345601@wanadoo.fr
CSeq: 2 INVITE
Content-Length: 0
```

```
/*** Proxy B tente de joindre User B ***/

F7 INVITE Proxy B -> User B

INVITE sip:najm@enst.fr SIP/2.0
Via: SIP/2.0/UDP proxyB.enst.fr:5060;branch=z9hG4bK721e418c4.1
Via: SIP/2.0/UDP proxyA.wanadoo.fr:5060;branch=z9hG4bK2d4790.1;received=1.2.3.4
Via: SIP/2.0/UDP wanadoo.fr:5060;branch=z9hG4bK74bf9;received=100.101.102.103
Max-Forwards: 68
Record-Route: <sip:proxyB.enst.fr;lr>, <sip:proxyA.wanadoo.fr;lr>
From: Arnaud <sip:fontaine@wanadoo.fr>;tag=9fxced76sl
To: Elie <sip:najm@enst.fr>
Call-ID: 12345601@wanadoo.fr
CSeq: 2 INVITE
Contact: <sip:fontaine@100.101.102.103>
Content-Type: application/sdp
Content-Length: 147

v=0
o=fontaine 2890844526 2890844526 IN IP4 wanadoo.fr
s=Session SDP
c=IN IP4 100.101.102.103
t=0 0
m=audio 49172 RTP/AVP 0
a=rtpmap:0 PCMU/8000


F8 (100 Trying) Proxy B -> Proxy A

SIP/2.0 100 Trying
Via: SIP/2.0/UDP proxyA.wanadoo.fr:5060;branch=z9hG4bK2d4790.1;received=1.2.3.4
Via: SIP/2.0/UDP wanadoo.fr:5060;branch=z9hG4bK74bf9;received=100.101.102.103
From: Arnaud <sip:fontaine@wanadoo.fr>;tag=9fxced76sl
To: Elie <sip:najm@enst.fr>
Call-ID: 12345601@wanadoo.fr
CSeq: 2 INVITE
Content-Length: 0


/*** le terminal notifie son changement d'état à Proxy B ***/

F9 180 Ringing User B -> Proxy B

SIP/2.0 180 Ringing
Via: SIP/2.0/UDP proxyB.enst.fr:5060;branch=z9hG4bK721e418c4.1;received=2.3.4.5
Via: SIP/2.0/UDP proxyA.wanadoo.fr:5060;branch=z9hG4bK2d4790.1;received=1.2.3.4
Via: SIP/2.0/UDP wanadoo.fr:5060;branch=z9hG4bK74bf9;received=100.101.102.103
Record-Route: <sip:proxyB.enst.fr;lr>, <sip:proxyA.wanadoo.fr;lr>
From: Arnaud <sip:fontaine@wanadoo.fr>;tag=9fxced76sl
To: Elie <sip:najm@enst.fr>
Call-ID: 12345601@wanadoo.fr
Contact: <sip:najm@110.111.112.113>
CSeq: 2 INVITE
Content-Length: 0
```

F10 180 Ringing Proxy B -> Proxy A

```
SIP/2.0 180 Ringing
Via: SIP/2.0/UDP proxyA.wanadoo.fr:5060;branch=z9hG4bK2d4790.1;received=1.2.3.4
Via: SIP/2.0/UDP wanadoo.fr:5060;branch=z9hG4bK74bf9;received=100.101.102.103
Record-Route: <sip:proxyB.enst.fr;lr>, <sip:proxyA.wanadoo.fr;lr>
From: Arnaud <sip:fontaine@wanadoo.fr>;tag=9fxced76sl
To: Elie <sip:najm@enst.fr>
Call-ID: 12345601@wanadoo.fr
Contact: <sip:najm@110.111.112.113>
CSeq: 2 INVITE
Content-Length: 0


/*** le changement d'état de User B remonte jusqu'à User A ***/
```

F11 180 Ringing Proxy A -> User A

```
SIP/2.0 180 Ringing
Via: SIP/2.0/UDP wanadoo.fr:5060;branch=z9hG4bK74bf9;received=100.101.102.103
Record-Route: <sip:proxyB.enst.fr;lr>, <sip:proxyA.wanadoo.fr;lr>
From: Arnaud <sip:fontaine@wanadoo.fr>;tag=9fxced76sl
To: Elie <sip:najm@enst.fr>
Call-ID: 12345601@wanadoo.fr
Contact: <sip:najm@110.111.112.113>
CSeq: 2 INVITE
Content-Length: 0


/*** User B décroche. Le message retourne le port 3456 et l'adresse IP permettant à
     son tour à User A de se connecter sur le terminal de User B ***/
```

F12 200 OK User B -> Proxy B

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP proxyB.enst.fr:5060;branch=z9hG4bK721e418c4.1;received=2.3.4.5
Via: SIP/2.0/UDP proxyA.wanadoo.fr:5060;branch=z9hG4bK2d4790.1;received=1.2.3.4
Via: SIP/2.0/UDP wanadoo.fr:5060;branch=z9hG4bK74bf9;received=100.101.102.103
Record-Route: <sip:proxyB.enst.fr;lr>, <sip:proxyA.wanadoo.fr;lr>
From: Arnaud <sip:fontaine@wanadoo.fr>;tag=9fxced76sl
To: Elie <sip:najm@enst.fr>
Call-ID: 12345601@wanadoo.fr
Contact: <sip:najm@110.111.112.113>
CSeq: 2 INVITE
Content-Type: application/sdp
Content-Length: 147

v=0
o=najm 2890844527 2890844527 IN IP4 enst.fr
s=Session SDP
c=IN IP4 110.111.112.113
t=0 0
m=audio 3456 RTP/AVP 0
a=rtpmap:0 PCMU/8000
```

F13 200 OK Proxy B -> Proxy A

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP proxyA.wanadoo.fr:5060;branch=z9hG4bK2d4790.1;received=1.2.3.4
Via: SIP/2.0/UDP wanadoo.fr:5060;branch=z9hG4bK74bf9;received=100.101.102.103
Record-Route: <sip:proxyB.enst.fr;lr>, <sip:proxyA.wanadoo.fr;lr>
From: Arnaud <sip:fontaine@wanadoo.fr>;tag=9fxced76sl
To: Elie <sip:najm@enst.fr>
Call-ID: 12345601@wanadoo.fr
Contact: <sip:najm@110.111.112.113>
CSeq: 2 INVITE
Content-Type: application/sdp
Content-Length: 147
```

```
v=0
o=najm 2890844527 2890844527 IN IP4 enst.fr
s=Session SDP
c=IN IP4 110.111.112.113
t=0 0
m=audio 3456 RTP/AVP 0
a=rtpmap:0 PCMU/8000


/*** la notification du décrochage de User B remonte jusqu'à User A ***/

F14 200 OK Proxy A -> User A

SIP/2.0 200 OK
Via: SIP/2.0/UDP wanadoo.fr:5060;branch=z9hG4bK74bf9;received=100.101.102.103
Record-Route: <sip:proxyB.enst.fr;lr>, <sip:proxyA.wanadoo.fr;lr>
From: Arnaud <sip:fontaine@wanadoo.fr>;tag=9fxced76sl
To: Elie <sip:najm@enst.fr>
Call-ID: 12345601@wanadoo.fr
Contact: <sip:najm@110.111.112.113>
CSeq: 2 INVITE
Content-Type: application/sdp
Content-Length: 147

v=0
o=najm 2890844527 2890844527 IN IP4 enst.fr
s=Session SDP
c=IN IP4 110.111.112.113
t=0 0
m=audio 3456 RTP/AVP 0
a=rtpmap:0 PCMU/8000


/*** la notification du décrochage de User B est acquittée ***/

F15 ACK User A -> Proxy A

ACK sip:najm@110.111.112.113 SIP/2.0
Via: SIP/2.0/UDP wanadoo.fr:5060;branch=z9hG4bK74bf9
Max-Forwards: 70
Route: <sip:proxyB.enst.fr;lr>, <sip:proxyA.wanadoo.fr;lr>
From: Arnaud <sip:fontaine@wanadoo.fr>;tag=9fxced76sl
To: Elie <sip:najm@enst.fr>
Call-ID: 12345601@wanadoo.fr
CSeq: 2 ACK
Content-Length: 0


F16 ACK Proxy A -> Proxy B

ACK sip:najm@110.111.112.113 SIP/2.0
Via: SIP/2.0/UDP proxyA.wanadoo.fr:5060;branch=z9hG4bK2d4790.1
Via: SIP/2.0/UDP wanadoo.fr:5060;branch=z9hG4bK74bf9;received=100.101.102.103
Max-Forwards: 69
Route: <sip:proxyB.enst.fr;lr>, <sip:proxyA.wanadoo.fr;lr>
From: Arnaud <sip:fontaine@wanadoo.fr>;tag=9fxced76sl
To: Elie <sip:najm@enst.fr>
Call-ID: 12345601@wanadoo.fr
CSeq: 2 ACK
Content-Length: 0
```

F17 ACK Proxy B -> User B

```
ACK sip:najm@110.111.112.113 SIP/2.0
Via: SIP/2.0/UDP proxyB.enst.fr:5060;branch=z9hG4bK721e418c4.1
Via: SIP/2.0/UDP proxyA.wanadoo.fr:5060;branch=z9hG4bK2d4790.1;received=1.2.3.4
Via: SIP/2.0/UDP wanadoo.fr:5060;branch=z9hG4bK74bf9;received=100.101.102.103
Max-Forwards: 68
Route: <sip:proxyB.enst.fr;lr>, <sip:proxyA.wanadoo.fr;lr>
From: Arnaud <sip:fontaine@wanadoo.fr>;tag=9fxced76sl
To: Elie <sip:najm@enst.fr>
Call-ID: 12345601@wanadoo.fr
CSeq: 2 ACK
Content-Length: 0


/*** Une fois l'acquittement du décrochage revenue jusqu'à User B, une session RTP
     peut être ouverte ***/
/*** User B raccroche ***/
```

F18 BYE User B -> Proxy B

```
BYE sip:fontaine@100.101.102.103 SIP/2.0
Via: SIP/2.0/UDP enst.fr:5060;branch=z9hG4bKnashds7
Max-Forwards: 70
Route: <sip:proxyA.wanadoo.fr;lr> , <sip:proxyB.enst.fr;lr>
From: Arnaud <sip:fontaine@wanadoo.fr>;tag=9fxced76sl
To: Elie <sip:najm@enst.fr>
Call-ID: 12345601@wanadoo.fr
CSeq: 1 BYE
Content-Length: 0
```

F19 BYE Proxy B -> Proxy A

```
BYE sip:fontaine@100.101.102.103 SIP/2.0
Via: SIP/2.0/UDP proxyB.enst.fr:5060;branch=z9hG4bK721e418c4.1
Via: SIP/2.0/UDP enst.fr:5060;branch=z9hG4bKnashds7;received=110.111.112.113
Max-Forwards: 69
Route: <sip:proxyA.wanadoo.fr;lr>
From: Arnaud <sip:fontaine@wanadoo.fr>;tag=9fxced76sl
To: Elie <sip:najm@enst.fr>
Call-ID: 12345601@wanadoo.fr
CSeq: 1 BYE
Content-Length: 0
```

F20 BYE Proxy A -> User A

```
BYE sip:fontaine@100.101.102.103 SIP/2.0
Via: SIP/2.0/UDP proxyA.wanadoo.fr:5060;branch=z9hG4bK2d4790.1
Via: SIP/2.0/UDP proxyB.enst.fr:5060;branch=z9hG4bK721e418c4.1;received=2.3.4.5
Via: SIP/2.0/UDP enst.fr:5060;branch=z9hG4bKnashds7;received=110.111.112.113
Max-Forwards: 68
From: Arnaud <sip:fontaine@wanadoo.fr>;tag=9fxced76sl
To: Elie <sip:najm@enst.fr>
Call-ID: 12345601@wanadoo.fr
CSeq: 1 BYE
Content-Length: 0
```

F21 200 OK User A -> Proxy A

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP proxyA.wanadoo.fr:5060;branch=z9hG4bK2d4790.1;received=1.2.3.4
Via: SIP/2.0/UDP proxyB.enst.fr:5060;branch=z9hG4bK721e418c4.1;received=2.3.4.5
Via: SIP/2.0/UDP enst.fr:5060;branch=z9hG4bKnashds7;received=110.111.112.113
From: Arnaud <sip:fontaine@wanadoo.fr>;tag=9fxced76sl
To: Elie <sip:najm@enst.fr>
Call-ID: 12345601@wanadoo.fr
CSeq: 1 BYE
Content-Length: 0
```

F22 200 OK Proxy A -> Proxy B

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP proxyB.enst.fr:5060;branch=z9hG4bK721e418c4.1;received=2.3.4.5
Via: SIP/2.0/UDP enst.fr:5060;branch=z9hG4bKnashds7;received=110.111.112.113
From: Arnaud <sip:fontaine@wanadoo.fr>;tag=9fxced76sl
To: Elie <sip:najm@enst.fr>
Call-ID: 12345601@wanadoo.fr
CSeq: 1 BYE
Content-Length: 0
```

F23 200 OK Proxy B -> User B

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP enst.fr:5060;branch=z9hG4bKnashds7;received=110.111.112.113
From: Arnaud <sip:fontaine@wanadoo.fr>;tag=9fxced76sl
To: Elie <sip:najm@enst.fr>
Call-ID: 12345601@wanadoo.fr
CSeq: 1 BYE
Content-Length: 0
```

8.3 Annexe 3 : Flot de messages SIP, pour l'utilisation d'un location serveur.

/*** UserA du domaine wanadoo.fr informe le registrar "regist" du domaine "enst.fr" de sa localisation temporaire. ***/

F1 REGISTER User A -> Registrar

```
REGISTER sip:regist.enst.fr SIP/2.0
Via: SIP/2.0/UDP enst.fr:5060;branch=z9hG4bKnashds7
Max-Forwards: 70
From: Arnaud <sip:UserA@wanadoo.fr>;tag=a73kszlfl
To: Arnaud <sip:UserA@wanadoo.fr>
Call-ID: 123456789@enst.fr
CSeq: 1 REGISTER
Contact: <sip:UserX@mortimer.enst.fr>
Contact: <sip:+33-1-4581-8308@gw.enst.fr;user=phone>
Content-Length: 0
```

F2 STORE Registrar -> Location Serveur

Le registrar du domaine domaine "enst.fr" entre en relation avec le location serveur local pour mettre à jour les données de l'utilisateur UserA@wanadoo.fr , qui restera connu sous cette identité dans le domaine enst.fr. Le location serveur a ensuite la responsabilité de contacter le location serveur du domaine "wanadoo.fr", pour tenir compte de la nouvelle localisation de UserA. D'après son message REGISTER, UserA sera ainsi accessible selon deux nouveaux modes :

- Une adresse SIP précisant que UserA est connecté en tant que UserX sur la machine mortimer.enst.fr
- Un numéro de téléphone : "33-1-4581-8308" accessible via la gateway "gw.enst.fr"

```
/*** le registrar retourne tous les contacts connus vers UserA ***/
```

F3 200 OK Registrar -> User A

```
SIP/2.0 200 OK
Via: SIP/2.0/UDP enst.fr:5060;branch=z9hG4bKnashds7 ;received=mortimer.enst.fr
From: Arnaud <sip:UserA@wanadoo.fr>;tag=a73kszlfl
To: Arnaud <sip:UserA@wanadoo.fr>
Call-ID: 123456789@enst.fr
CSeq: 1 REGISTER
Contact: <sip:UserX@mortimer.enst.fr >;expires=3600
Contact: <sip:+33-1-4581-8308@gw.enst.fr;user=phone>;expires=3600
Contact: <mailto:UserA@wanadoo.fr>;expires=4294967295
Content-Length: 0
```

F4 INVITE User B -> Proxy

```
INVITE sip:UserA@wanadoo.fr SIP/2.0
Via: SIP/2.0/UDP aol.com:5060;branch=z9hG4bK74bf9
Max-Forwards: 70
From: Elie <sip:UserB@aol.com>;tag=9fxced76sl
To: Arnaud <sip:UserA@wanadoo.fr>
Call-ID: 12345601@aol.com
CSeq: 1 INVITE
Contact: <sip:UserB@blake.aol.com>
Content-Type: application/sdp
Content-Length: 147

v=0
o=UserB 2890844526 2890844526 IN IP4 aol.com
s=Session SDP
c=IN IP4 blake.aol.com
t=0 0
m=audio 49172 RTP/AVP 0
a=rtpmap:0 PCMU/8000
```

F5/F6 QUERY Proxy <-> Location Serveur

Le proxy du domaine ”aol.com” contacte le location serveur du domaine ”wanadoo.fr” (éventuellement en passant par un proxy propre au domaine wanadoo.fr), afin de connaître la localisation actuelle de UserA. Cette opération retournera l’adresse SIP : UserX@mortimer.enst.fr. Utiliser un proxy permet de gérer le changement de localisation de UserA, de façon transparente pour UserB. Le proxy modifie en conséquent le message INVITE F1.

F7 INVITE Proxy -> UserA

```
INVITE sip:UserX@mortimer.enst.fr SIP/2.0
Via: SIP/2.0/UDP aol.com:5060;branch=z9hG4bK74bf9
Max-Forwards: 70
From: Elie <sip:UserB@aol.com>;tag=9fxced76sl
To: Arnaud <sip:UserA@wanadoo.fr>
Call-ID: 12345601@aol.com
CSeq: 1 INVITE
Contact: <sip:UserB@blake.aol.com>
Content-Type: application/sdp
Content-Length: 147

v=0
o=UserB 2890844526 2890844526 IN IP4 aol.com
s=Session SDP
c=IN IP4 blake.aol.com
t=0 0
m=audio 49172 RTP/AVP 0
a=rtpmap:0 PCMU/8000
```

F8 (100 Trying) Proxy -> UserB

```
SIP/2.0 100 Trying
Via: SIP/2.0/UDP aol.com:5060;branch=z9hG4bK74bf9 ;received=blake.aol.com
From: Elie <sip:UserB@blake.aol.com>;tag=9fxced76sl
To: Arnaud <sip:UserA@wanadoo.fr>
Call-ID: 12345601@aol.com
CSeq: 1 INVITE
Content-Length: 0
```