UNIVERSITÉ DE SHERBROOKE

Faculté de génie
Département de génie électrique et de génie informatique

SERVICE DE PRÉSENCE ADAPTÉ AU CONTEXTE DES COMMUNICATIONS D'URGENCE MÉDICALE PRÉ-HOSPITALIÈRE

Mémoire de maîtrise es sciences appliquées
Spécialité : génie électrique

Élaboré par : Fakher BEN SALAH

Dirigé par : Alain C. HOULE

Composition du jury : Soumaya CHERKAOUI
Alejandro QUINTERO

Sherbrooke (Québec), CANADA JUILLET 2009

Your file Votre référence
ISBN: 978-0-494-53146-4
Our file Notre référence
ISBN: 978-0-494-53146-4

RÉSUMÉ

Notre groupe de recherche développe actuellement une plateforme de communication multimédia sur protocole IP. Cette plateforme vise une application dans le domaine des communications d'urgence médicale pré-hospitalière. Dans un tel contexte, il devient utile de joindre et d'inviter à une conférence multimédia, un spécialiste médical. La problématique que nous abordons est celle de pouvoir joindre, à tout moment, une telle ressource possédant certaines caractéristiques de spécialité et de disponibilité. Pour y répondre, nous suggérons la mise au point d'un service de présence adapté au contexte des communications d'urgence médicale pré-hospitalière. Ce service a la particularité de permettre une recherche de personnel qui répond à des critères spécifiques, tels que la spécialité, la localisation et/ou l'état de présence. ***L'objectif général de notre projet de recherche est donc de contribuer à l'avancement de la technologie des systèmes de présence en contexte médical.***

(Système de présence, information de présence, PIDF, télémédecine, médical, télécommunication, SIP, urgence médicale).

REMERCIEMENTS

Au nom de dieu, le tout miséricordieux, le très miséricordieux

Dieu dit dans ses tous premiers versés révélés à son prophète Mohammad (paix soit sur lui) en 640

1. Lis, au nom de ton Seigneur qui a créé,

2. qui a créé l'homme d'une adhérence [Alaq]

Chapitre [Sourat] 96 du Saint Coran : l'adhérence [Al Alaq]

À mon directeur de recherche M. Alain C. Houle :

Je suis très ravie de l'honneur que vous me faites en acceptant de diriger ce travail. J'ai toujours admiré votre bienveillance, l'amabilité de votre abord et la richesse de votre enseignement. Veuillez trouver dans ce travail l'expression de ma profonde gratitude! Vous étiez pour moi plus qu'un professeur par vos qualités humaines inégalables. Pourvu que la réussite fleurisse toute votre vie!

À la chaire de recherche industrielle en infrastructures et outils de communication:

Un grand merci d'avoir accepté mon projet et m'assuré un soutien financier.

À mes collègues de laboratoire, en ordre alphabétique : Alexis, Cyril, Éric, Jean, Mehdi, Manon, Mohamed, Mohamed Firass, Ramzi, Viviane et Wajdi;

Merci infiniment pour avoir discuté d'une tonne de sujets intéressants qui ont enrichi d'une manière inespérée mon expérience à la maîtrise. Je n'oublierai jamais les moments qu'on a partagés ensemble.

TABLE DES MATIÈRES

LISTE DES FIGURES

1. DÉFINITION DE PROJET DE RECHERCHE

1.1. Introduction

Un individu peut, à la suite d'un accident ou d'une détérioration soudaine de son état de santé, nécessiter des soins médicaux urgents. C'est ainsi qu'une chaîne d'actions se déclenche afin d'assurer rapidement et efficacement la prestation des soins requis. Au cœur de cette chaîne d'actions, on retrouve l'intervention des services ambulanciers (Laberge-Nadeau et al., 1998).

Lorsqu'un patient est pris en charge par les services ambulanciers, les moyens de communication entre l'ambulance et l'hôpital de première ligne pouvant recevoir le patient se limitent actuellement à des communications vocales via téléphone cellulaire ou radio mobile. Récemment, notre groupe de recherche a développé un prototype de plateforme de communication multimédia sur protocole IP dédiée aux communications médicales d'urgence pré-hospitalières (Dorais-Joncas, 2007), En plus de la communication vocale entre une ambulance et un hôpital, cette plateforme permet la communication des signaux physiologiques du patient. Dans un premier temps, les signaux physiologiques considérés sont des électrocardiogrammes. On mentionne qu'il s'agit d'une communication multimédia puisque plusieurs types de flots d'information (information vocale et physiologique) sont transmis.

Au-delà de la communication multimédia, le prototype permet également la mise en conférence de plusieurs entités en mode pleinement maillé. Dans un cas d'utilisation typique, un ambulancier établit d'abord une communication point-à-point entre l'ambulance et l'hôpital de première ligne pour permettre au médecin urgentologue en service de prendre connaissance de l'état du patient et de ses signaux physiologiques. En fonction de son appréciation de l'état du patient, le médecin urgentologue pourrait vouloir consulter un autre médecin ou un spécialiste. C'est ainsi que la communication point-à-point se transforme en une communication pleinement maillée où toutes les entités impliquées (ambulancier, médecin urgentologue, spécialiste) peuvent dialoguer et avoir accès aux signaux physiologiques du patient en temps réel. Il devient alors possible de prendre une décision éclairée quant au meilleur traitement à offrir au patient ainsi qu'à propos du meilleur endroit,

l'hôpital de première ligne ou une autre institution plus spécialisée, vers lequel il faut diriger l'ambulance pour assurer des soins optimaux. Cette communication médicale d'urgence pré-hospitalière permet aussi la préparation des équipes hospitalières avant même l'arrivée du patient à l'hôpital. Ce faisant, les chances de survie du patient sont maximisées tandis que les risques de séquelles sont minimisés.

L'intervention des services ambulanciers peut être requise à tout moment. Il n'est pas du tout évident que les entités (ambulancier, médecin urgentologue, spécialiste) qui doivent interagir lors d'une communication d'urgence médicale pré-hospitalière soient spontanément disponibles ou facilement joignables. Cette situation indésirable constitue le cœur de la problématique que nous proposons de solutionner. Pour ce faire, nous proposons la mise au point d'un service de présence spécifique au milieu médical. Ce service pourra éventuellement s'intégrer à la plateforme de communication multimédia sur protocole IP qui a été développée au sein de notre groupe de recherche.

Un service de présence permet aux utilisateurs de publier des informations personnelles telles leur disponibilité momentanée, les moyens de communication par lesquels ils peuvent être joints. Il permet aussi de contrôler l'accès des autres utilisateurs à ces informations. Contrairement aux services de présence qui sont dédiés aux systèmes de communication personnels (e.g. MSN Messenger, Skype, etc.), le service de présence que nous proposons inclut un outil de recherche qui se base non pas sur l'identité d'une personne mais plutôt sur sa spécialité et sa localisation. D'autres paramètres peuvent s'inclure dans cette recherche. En effet, un médecin urgentologue nécessitant l'avis d'un cardiologue n'a pas besoin de joindre un individu en particulier mais a plutôt besoin de joindre un individu spécialiste dans le domaine visé.

1.2. Objectifs du projet de recherche

La problématique que nous abordons est celle de pouvoir joindre, à tout moment, une ressource du milieu médical possédant certaines caractéristiques de spécialité et de disponibilité. Pour y répondre, nous suggérons la mise au point d'un service de présence adapté au contexte des communications d'urgence médicale pré-hospitalière. ***L'objectif***

général de notre projet de recherche est donc de contribuer à l'avancement de la technologie des systèmes de présence en contexte médical.

De manière plus précise, les objectifs spécifiques de notre projet de recherche sont les suivants :

- Définir un modèle d'informations de présence spécifique au contexte médical;
- Définir un modèle abstrait de l'architecture de notre service de présence, incluant la définition des entités présentes dans cette architecture;
- Définir les opérations que le service de présence proposé doit inclure pour être adéquatement fonctionnel;
- Proposer une implémentation du service de présence à l'aide de technologies actuelles telles la technologie XML (Annexe A.1) et la technologie SIP (Rosenberg et al., 2002);
- Développer un prototype du service de présence pour en faire la preuve de concept;
- Discuter de l'intégration du service de présence à une plateforme de communication multimédia sur IP vouée au contexte médical;
- Commenter certaines implications pratiques comme, par exemple, les modes de mise à jour des informations de présence et la sécurité informatique.

1.3. Méthodologie

Notre méthodologie, dont les résultats sont rapportés dans les prochains chapitres, débute par une étude de l'état de l'art des services de présence et une description du cadre de référence technologique dans lequel nous travaillons (chapitre 2). Par la suite, nous présentons les modèles abstraits du service de présence ainsi que les opérations à mettre au point pour réaliser le service de présence (chapitre 3). Au chapitre 4, nous proposons une implémentation du service de présence employant des technologies actuelles comme la technologie XML et la technologie SIP. Nous rapportons également le développement d'un prototype nous

permettant de faire la preuve de concept du service de présence. Au chapitre 5, nous testons et évaluons le prototype réalisé pour concrétiser ce projet. Finalement, le chapitre 6 traite de l'intégration du service de présence à une plateforme de communication multimédia sur IP pour le milieu.

2. ÉTUDE DE L'ART

2.1. Recherche Bibliographique

L'environnement de santé est un milieu très dynamique. L'intercommunication, la mobilité et l'accès à l'information des agents médicaux sont des parties intégrantes du paysage médical. Les agents médicaux se composent du personnel médical, des patients et des dispositifs. Par exemple, un ambulancier au lieu de l'accident a besoin de communiquer avec un spécialiste, pour lui transmettre un électrocardiogramme (ECG) pour fins d'analyse.

Cette communication se réalise par l'intermédiaire de différents dispositifs fixes et mobiles. Leur évolution rapide ainsi que l'apparition de nouvelles générations d'appareils utilisant différentes technologies de connexion ont diversifié l'accès à l'information. Cette diversité étend la zone d'accessibilité des appareils et par la suite élargit la zone d'action.

Les moyens de communication modernes permettent de concrétiser le dynamisme voulu de l'environnement de santé. Le partage rapide et efficace de l'information entre les agents médicaux aura un impact significatif sur le milieu médical. Ceci nous mène à introduire la notion de la télémédecine qui est l'intégration de la télécommunication dans le domaine médical.

Plusieurs technologies de communication touchent le domaine médical. On note un système générique appliqué à la notification d'urgence et à la surveillance d'évènement. Ce système a été appliqué à l'urgence et à la surveillance d'évènements médicaux (Arabshian et al., 2003a).

Le domaine de la télémédecine touche aussi le réseau de téléphonie cellulaire. On note un mécanisme de transmission de données médicale d'urgence sur les réseaux cellulaires (Arunachalan et al, 2007). Le langage HL7 est utilisé dans ce mécanisme au niveau de l'échange de l'information médicale. HL7 est un langage internationalement agréé pour l'échange de l'information médicale entre les systèmes informatiques.

La technologie de présence est en train de gagner un intérêt significatif dans plusieurs domaines. Elle devient un composant indispensable dans l'industrie récente des télécommunications. Les services basés sur la présence conduisent à de nouvelles opportunités

d'affaire et changent chaque aspect de notre vie quotidienne. On entend par service de présence, tout service permettant de stocker et de gérer les informations de présence. Ces informations sont mises à jours en temps réel, elles comprennent l'état de présence et/ou disponibilité et la localisation des utilisateurs.

Il existe un modèle abstrait de présence défini par le RFC2778 (Day et al., 2000a). Son but est de fournir un vocabulaire commun pour les futurs travaux sur les exigences des protocoles pour la présence et la messagerie instantanée.

Le protocole de communication utilisé dans plusieurs systèmes de présence est SIP (*session initiation protocol*) (Rosenberg et al., 2002). C'est un protocole d'initiation de sessions multimédias, il est utilisé aussi dans la notification et l'inscription des usagers. Le protocole de communication SIP est détaillé plus à l'annexe A.2.

La majorité des articles recommandent XML (*eXtensible Markup Language*) comme étant le langage le plus adapté pour le codage de l'information de présence. Cela est dû au fait que cette information a une structure hiérarchique et doit être entièrement extensible. PIDF (Presence Information Data Format) représente le format de données de l'information de présence. Elle est définie dans (Sugano et al., 2004). Des extensions d'état de présence se trouvent dans plusieurs ébauches (*drafts*) du groupe de travail SIMPLE (*SIP for Instant Messaging and Presence Leveraging Extensions*) de l'organisation IETF (Internet Engineering Task Force) .

L'article (Jin et al., 2004) détaille un service de messagerie instantanée et de présence en utilisant SIMPLE. Une démonstration a été réalisée avec l'*API JAINTM SIP/SIMPLE.* C'est une interface de programmation qui permet d'implémenter le protocole SIP/SIMPLE. Il y a aussi l'article (Peternel et al., 2008) qui met l'accent sur la collaboration effective en utilisant l'information de présence. Les localisations et les états de présence des agents sont détectés et regroupés dans le serveur de présence afin de les distribuer aux utilisateurs concernés.

Aussi l'article (Huh et al., 2007) utilise la notion de *RessourceList* (RL) Hiérarchique pour améliorer la collaboration entre les agents ; l'information notifiée par le serveur de présence englobe généralement plusieurs agents, le RL est une liste de ressources, gérée par le RS

Server (RLS), contenant un groupe d'agents. Au lieu d'envoyer pour chaque agent une notification, le serveur de présence aura juste à envoyer une seule notification contenant l'information relative à tous ces agents. Le RLS est donc une méthode pour regrouper les agents. On peut créer plusieurs listes différentes.

Afin de gérer toute une liste d'agents, le serveur de présence utilise le protocole XCAP (*XML Configuration Access Protocol*). Il permet à un client de lire, écrire et modifier l'information de configuration d'applications stockées en format XML dans un serveur. Ce protocole est utilisé dans (Jin et al., 2004), (Peternel et al., 2008) et (Huh et al., 2007).

L'information de présence doit être protégée. Elle peut contenir des informations strictement confidentielles. Les méthodes de sécurité et de confidentialité sont parmi les grands soucis des utilisateurs du service de présence (Lucenius, 2008). Dans cet article, on a mis l'accent sur la sécurité et la confidentialité dans les services de présence.

Actuellement les systèmes sont assez intelligents pour connaître l'état de présence ou la localisation des agents sans intervention humaine. L'information est retirée, par exemple, des capteurs ou constatée de quelques variables (Wu, 2007), grâce à l'utilisation des nouvelles technologies en relation avec la présence et la localisation.

La localisation est une information de présence. Le système RFID (*Radio Frequency ID*) est généralement utilisé pour le traçage d'agents, l'identification de patients ou de professionnels ou la surveillance des entrées de zones à accès limitées. Il y a aussi les *wifi tags* qui servent à localiser les agents. La zone d'accès de ces systèmes est limitée (Chiang et al., 2008). Il y a le système GPS (*Global Positioning System*). Il permet la localisation satellitaire sur une zone géographique étendue, mais il ne fonctionne pas dans les zones intérieures. De surcroît, sa précision n'est souvent pas suffisante pour une application dans le domaine qui nous intéresse.

L'utilisation du protocole de communication sans fils *Zigbee* dans plusieurs systèmes de surveillance de santé est bien acceptée. Ce protocole fonctionne bien avec les systèmes à faible consommation d'énergie mais à bas débit. Il peut transmettre en temps réel les informations médicales recueillies des capteurs ECG par exemple. Une fois ces informations

intégrées au sein du service de présence, celles-ci sont transmises aux utilisateurs par le protocole SIP (Kim et al., 2007).

Les recherches actuelles en télémédecine intègrent les réseaux mobiles de troisième génération. Ce système permet de mettre tout le personnel médical en relation, de l'ambulancier jusqu'au médecin spécialiste. Il peut connecter aussi les patients et les met sous surveillance médicale par l'intermédiaire de capteurs placés sur le corps (Viruete Navarro et al., 2005), (Zou et al., 2004). D'ailleurs, il y a un cas où la télémédecine intègre le réseau cellulaire (Arunachalan et al., 2007).

Les recherches récentes du domaine médical on mit l'accent aussi sur les systèmes de conférence multimédia ; ces systèmes consistent en un ensemble d'agents mobiles et fixes qui s'échangent l'information. On trouve une application de conférence vidéo sécurisée basée sur SIP (Tulu et al., 2003). On peut s'inspirer aussi du système de collaboration dans le domaine de la santé adapté à la ville de Taiwan (Hsieh et al., 2006). Ce système permet le suivi en temps réel de l'état d'une situation d'urgence médicale et offre des instructions adaptées aux intervenants.

La communication entre les ambulanciers et le centre d'urgence est cruciale par rapport à la vie des patients dans plusieurs cas. Comme l'ambulance est un véhicule mobile, on a donc besoin d'un système de télémédecine sans fils. L'article (Navarro et al., 2006) traite d'un système de communication entre les ambulanciers et les médecins à travers plusieurs plateformes de réseaux.

La télémédecine utilise même la communication par satellite VSAT (Very Small Aperture Terminal) (Pandian et al., 2007). Ce système transmet les données médicales au moyen d'antennes paraboliques. Il est équipé d'un système de visioconférence. Il est pratique dans les zones rurales ou sinistrées, là où il n'y a pas d'infrastructures déployées.

Avec le progrès de la technologie de télécommunication et l'utilisation des appareils mobiles, il est devenu naturel d'utiliser de telles technologies pour améliorer l'efficacité des services de santé. Ces technologies permettent au personnel hospitalier de rester connecté à

ses systèmes vitaux sans se soucier de leur localisation dans les établissements médicaux ou ailleurs.

Cependant, il reste un problème : la disponibilité d'un utilisateur à exécuter une tâche spécifique à un moment donné. L'intégration d'un service de présence aux applications médicales s'avère essentielle afin de résoudre ce problème. L'article (Arabshian et al., 2003b) décrit un système de surveillance d'évènements médicaux basé sur SIP. Ce système intègre le service de présence, l'information de présence est transmise par le protocole SIP. Dans ce cas, la technologie *Bluetooth* est utilisée pour déterminer la localisation des médecins. Bluetooth est une spécification de l'industrie des télécommunications. Elle utilise une technologie radio courte distance destinée à simplifier les connexions entre les appareils électroniques.

Aussi, il existe un autre système détaillé par l'article (Lakas et al., 2005). Ce système est assez complet. Il interconnecte tout le personnel médical. Un service de présence est intégré à ce système, l'information de présence comprend la localisation et l'état de présence de chaque agent médical. La localisation est déterminée par les capteurs communicants par *Zigbee* et d'autres appareils mobiles. La sécurité est intégrée à ce système.

D'après ce qu'on a aperçu de la littérature, on remarque qu'il existe plusieurs applications médicales vraiment performantes. Elles intègrent les technologies récentes pour promouvoir l'efficacité de leurs systèmes et ensuite appuyer le domaine de la santé. Néanmoins, le cas où un professionnel médical a besoin d'une consultation urgente de la part d'un spécialiste n'est pas encore traité. Rappelons que nous avons énoncé, au chapitre 1, qu'un des objectifs spécifiques de notre projet concerne la définition d'un modèle d'informations de présence spécifique au contexte médical.

On va essayer, au cours de cette recherche, de résoudre ce problème. On va développer un service de présence appliqué au domaine médical et qui a la particularité de permettre une recherche de personnel qui répond à des critères spécifiques, tels que la spécialité, la localisation et/ou l'état de présence. Cette particularité fait de cette recherche une innovation dans le domaine des urgences médicales.

Dans cette partie, on a positionné ce projet par rapport à ceux existants. Au début, on a mentionné des articles se rapportant au domaine médical. Puis, on a introduit la notion de présence en citant quelques autres articles. Ensuite, on a clarifié les applications médicales en les appuyant par des articles relatifs à ce domaine. Puis, on a cité des articles se rapportant aux applications médicales intégrant le service de présence. Finalement, on a mis en évidence un des objectifs spécifiques du projet. Tout au long de ce développement, plusieurs technologies existantes ont été mentionnées par tous ces articles. On utilisera plusieurs de ces technologies dans ce projet. Cependant, seules les principales seront définies dans la partie suivante.

2.2. Cadre de référence

Dans cette section, on définira le système de présence et les différentes technologies principales impliquées qu'on utilisera dans ce projet. Dans la partie système de présence, on définira le modèle abstrait, le profil commun de présence, le flot de données dans un système de présence et le format de données de l'information de présence PIDF. Dans la partie technologies impliquées, on développera les mécanismes de stockage de l'information de présence incluant une brève introduction au langage XML dans une sous-partie intitulée base de données. On mettra aussi l'accent sur la communication client/serveur, on définira le protocole SIP et, pour finir, on clarifiera les considérations de sécurité.

2.2.1. Système de présence : modèle abstrait

On va présenter dans ce paragraphe le modèle abstrait du système de présence. Ce modèle est inspiré essentiellement du RFC2778 (Day et al., 2000a) (*A model of presence and instant Messaging*). Ce paragraphe se compose d'une première partie ' Architecture générale' et d'une deuxième partie ' Profil commun de présence CPP '.

Un système de présence assure la bonne gestion de présence d'un groupe d'utilisateurs qui veulent partager leurs informations de présence entre eux. Il accepte l'information de présence, la stocke dans une base de données et la distribue aux utilisateurs concernés. Il fournit aussi les moyens nécessaires pour garantir le bon déroulement de ces processus. Il traite l'accès aux informations de présence selon des règles conformes au protocole mis en place par l'administrateur.

Les moyens de communication qui peuvent être utilisés pour exploiter les informations de présence des utilisateurs sont nombreux : clavardage (chat), mms (Multimedia Messaging service), sms (Short Message Service), courriel, téléphone, vidéo, ... Chaque moyen de communication procure l'état de présence de l'utilisateur (PRINCIPAL en général) à un moment donné. Ceci s'appelle de l'information générale de présence : utilisateur connecté, non connecté, enregistré, absent, de retour,... On peut aussi trouver de l'information détaillée de présence telle que libre, parti manger, sur la route, sommeil, ... L'information de présence peut aussi contenir la localisation des utilisateurs (information géographique pertinente selon le contexte) par l'intermédiaire de dispositifs de localisation.

Architecture générale:

Les agents externes :

On définit le PRINCIPAL comme étant un humain, groupe, et/ou programme qui choisi d'apparaître au SERVICE DE PRÉSENCE comme un acteur singulier, distinct des autres PRINCIPALS, et qui utilise le système comme un moyen de coordination et de communication. Il est complètement en dehors du modèle.

Le PRINCIPAL interagit avec le système via un ou plusieurs USER AGENTS (PRESENCE USER AGENT, WATCHER USER AGENT). Les USER AGENTS sont séparés dans ce modèle bien que plusieurs implémentations combinent certains d'entre eux. Un USER AGENT est une entité qui intervient entre un PRINCIPAL et une PRESENTITY ou un WATCHER. C'est lui qui établit la communication avec le PRESENCE AGENT.

Les agents internes :

Le système de présence comprend deux ensembles d'entité : le service de présence et deux catégories de client. Certaines composantes de ces entités pourraient être combinées dans une implémentation, mais elles seront traitées séparément dans le modèle. Un ensemble de clients, appelé PRESENTITIES, fournit l'INFORMATION DE PRÉSENCE pour être stockée et distribuée. L'autre ensemble de clients, appelé WATCHERS, reçoit l'INFORMATION DE PRÉSENCE du service. Le service de présence accepte, stocke, gère et distribue l'information de présence.

Les clients : WATCHER et PRESENTITY

Il y a deux sortes de WATCHERS, appelés FETCHERS et SUBSCRIBERS. Un FETCHER demande simplement la valeur courante d'INFORMATION DE PRÉSENCE d'un certain PRESENTITY du SERVICE DE PRÉSENCE. Un SUBSCRIBER demande des notifications du SERVICE DE PRÉSENCE en cas de futurs changements dans l'INFORMATION DE PRÉSENCE d'un certain PRESENTITY. Il y a un type spécial de FETCHER, celui qui apporte l'information de façon régulière : il est appelé POLLER.

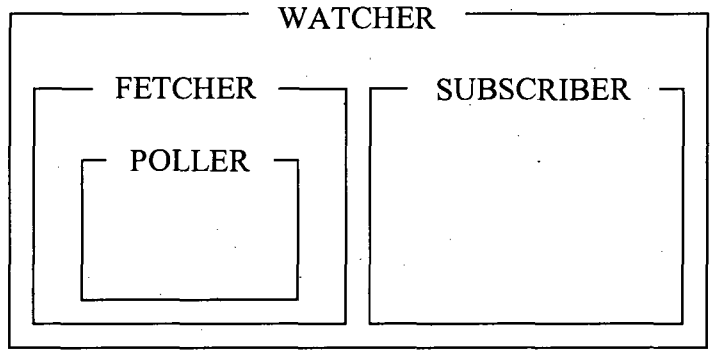


Fig. 1 : Variétés de WATCHER.

Un *presentity* se définit comme une entité logique qui projette ses informations de présence à ses parties intéressées (*watchers*). Il stocke toutes les informations de présence, les règles d'accès et les préférences du PRINCIPAL et par la suite notifie le service de présence chaque fois qu'il y a un changement dans ces informations. Il peut confier le traitement des droits d'accès au service de présence : ce dernier est alors averti de toutes les règles nécessaires.

Relation entre agents :

Le rôle du service de présence est de collecter et stocker toutes les informations de présence des PRINCIPALS dans un seul document (base de données) en parcourant toutes les PRESENTITIES. Il distribue aussi des informations exactes cumulées, concernant des PRESENTITIES spécifiés à des WATCHER spécifiés qui ont été déjà inscrits pour recevoir

de telles informations de présence. Des règles d'accès pourraient être imposées par le PRINCIPAL afin d'approuver ou non les demandes d'inscription et de filtrer et/ou changer les informations de présence expédiées. Tout cela est à la discrétion du PRINCIPAL et/ou l'administrateur.

Le SERVICE DE PRÉSENCE détient non seulement les informations des WATCHERS mais aussi leurs activités envers l'INFORMATION DE PRÉSENCE en termes de recherche ou inscription. De même, il peut distribuer certaines informations de présence de certains WATCHERS et leurs activités, à un certain WATCHER, en utilisant les mêmes mécanismes qui sont disponibles pour notifier. Cette technique consiste à vérifier la conformité de l'INFORMATION DE PRÉSENCE présente dans ce WATCHER à un moment donné et à intercepter des activités anormales de certains tiers. De cette manière, il peut détecter certaines anomalies pouvant être causées par des attaques malveillantes comme l'usurpation (spoofing).

Le partage de l'information se fait par l'intermédiaire de notifications des changements d'état de présence de chaque utilisateur concerné en permettant aux utilisateurs de se souscrire entre eux.

L'inscription se fait par l'intermédiaire d'une demande faite par un SUBSCRIBER au SERVICE DE PRÉSENCE afin d'avoir accès à des INFORMATIONS DE PRÉSENCE concernant un certain PRESENTITY ou un groupe de PRESENTITYs. Le SERVICE DE PRÉSENCE traite la demande puis décide s'il accepte de le notifier ou juste négliger sa demande. Le cas échéant, les changements dans l'INFORMATION DE PRÉSENCE se distribuent aux SUBSCRIBERS via notifications. La notification peut être limitée (les conditions d'accès à des INFORMATIONS DE PRÉSENCE d'un certain PRESENTITIE sont traitées dans ce prochainement).

Cas d'utilisation typique :

Les changements dans l'INFORMATION DE PRÉSENCE sont distribués aux SUBSCRIBERS via notifications. La figure suivante montre un cas d'utilisation permettant de comprendre les rôles et les relations des diverses entités impliquées dans un système de présence. Le schéma montre trois PRINCIPALS qui essaient d'interagir avec un serveur de

présence. Il détermine comment le flux d'information circule d'une entité à une autre et détermine aussi le changement de l'INFORMATION DE PRÉSENCE.

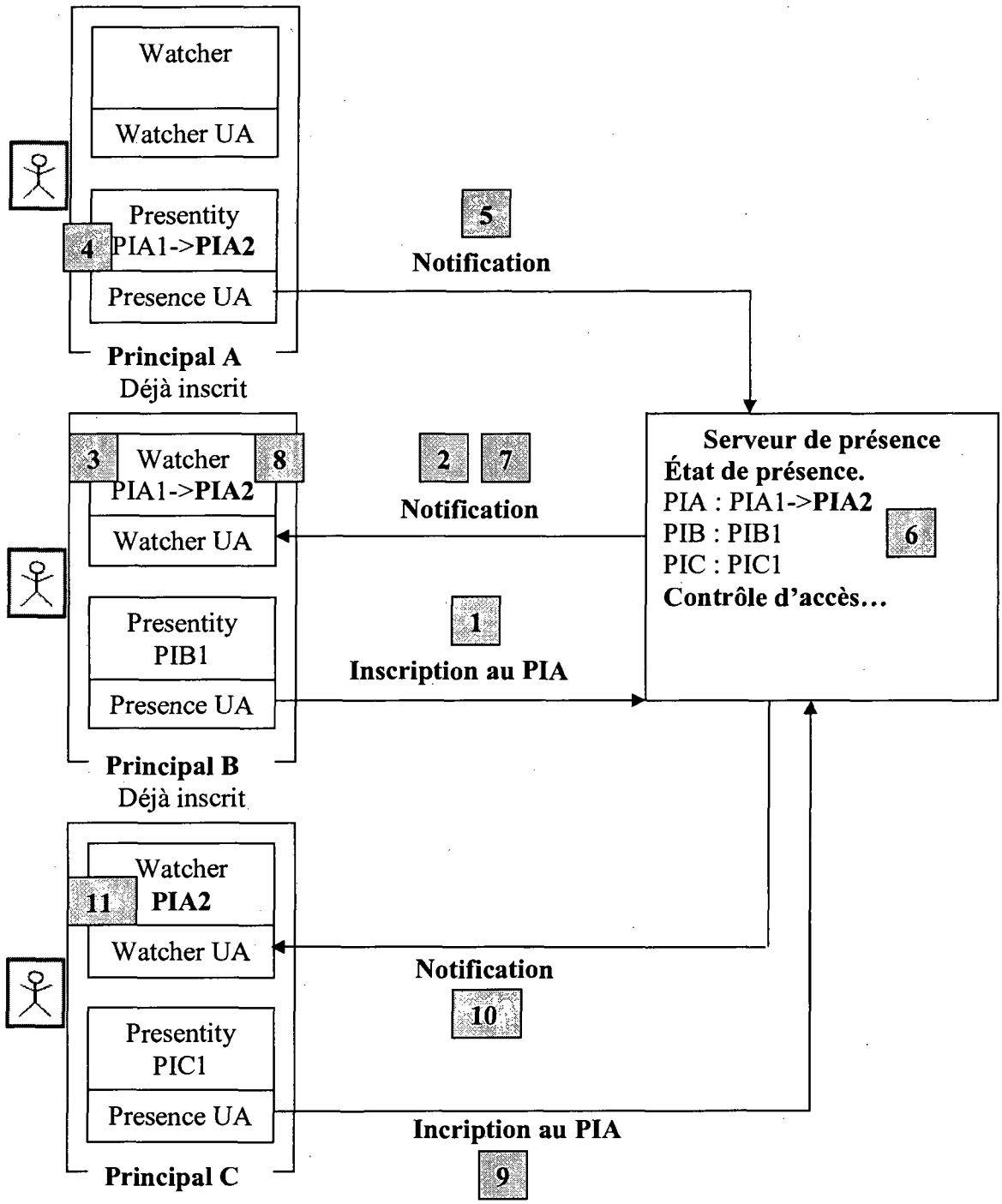


Fig. 2 : Architecture générale - interaction entre éléments.

PIA : Information de présence de A.

Interprétation de la figure 2 :

Le WATCHER B s'inscrit aux informations de présence de A, il envoie sa requête au service de présence (1). Ce dernier procède à quelques vérifications de droit d'accès, et peut interroger le PRINCIPAL concerné en envoyant sa demande à la PRESENTITY, et le cas échéant va notifier le WATCHER B en lui envoyant également un paquet de données contenant les informations de présence de A. Ces dernières peuvent être complètes, partielles ou modifiées suivant le résultat obtenu par l'enquête effectuée par le service de présence (ou l'investigation) (2). Le WATCHER B reconnaît l'information, l'accepte puis la mémorise (3).

Ensuite, A change son état de présence de PIA1 à PIA2, donc notifie le service de présence automatiquement. Ce service, de son tour, reconnaît le paquet d'information, l'identifie, et change l'information de présence appropriée de PIA1 à PIA2 (4), (5), (6).

Le service de présence enregistre un changement dans le PIA, donc va notifier tout WATCHER inscrit à cette information de présence, et en conséquent il va notifier B, qui à son tour reçoit la notification et change de PIA1 à PIA2. (7) et (8)

Subséquemment, le C envoie une requête d'inscription au service de présence concernant les informations de présence de A. Ce service de présence accepte et notifie C, et en conséquence le WATCHER de C reconnaît le paquet et mémorise l'information de présence (9), (10), (11).

Profil commun de présence : *CPP*

On a présenté dans le paragraphe précédent l'architecture générale d'un système de présence. Dans ce qui suit, on présentera le profil commun de présence « CPP : *Commun Presence Profile* » définit par le RFC3859.

Il existe plusieurs protocoles de présence. Cependant, ceux-ci sont généralement peu interopérables. Pour corriger cette situation, le CPP a été élaboré. Celui-ci facilite la création des passerelles entre les services de présence.

Les comportements du service sont décrits abstraitement sous forme d'opération entre le fournisseur de service et le client. Chaque service de présence doit donc décrire comment ces

comportements influencent ses propres interactions de protocole. Par exemple, une stratégie permet de transmettre de l'information de présence comme paire (valeur, clé) ou une représentation binaire compacte ou *nested container.*

Chaque service de présence doit indiquer comment des exemples bien formés de la représentation abstraite sont encodés comme une série concrète de bits.

Notez que cette spécification admet que le protocole de présence adapté à CPP fourni la livraison fiable des messages; il n'a aucune couche d'application qui assure la livraison des messages dans cette spécification. Dans ce qui suit, on va définir les différentes opérations associées à ce profil.

Opération inscription

Pour qu'une application s'inscrive à l'information de présence, elle fait appel à l'opération 'inscription' qui comporte les paramètres *watcher, target, duration, subscriptID et tansID.* Le *watcher* et *target* désignent respectivement l'adresse SIP du *watcher* et de la *presentity.* La *duration* spécifie le nombre de secondes maximal pendant laquelle l'inscription devrait être active (qui peut être 0). Le *subscriptID* crée une référence à l'inscription utilisé par la désinscription. Le *transID* est l'identifiant unique utilisé pour lier l'opération d'inscription avec la réponse et vice versa. Les passerelles devraient être capables de manipuler des *transID* et *subscriptID* de longueur allant jusqu'à 40 octets.

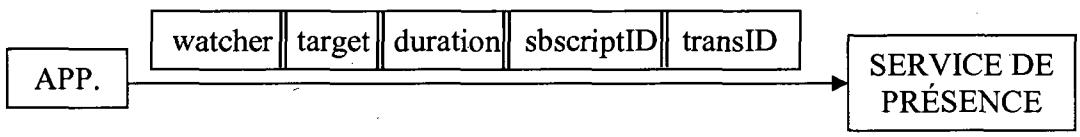


Fig. 3 : L'opération « inscription »

'APP.', qui représente l'application, peut annuler une inscription à n'importe quel moment en réappelant l'opération « inscription » avec une durée zéro avec le même *SubscriptID* que l'opération d'inscription originale. L'opération de notification appelée comme réponse à cette inscription peut être ignorée.

Opération réponse

Dès la réception de l'opération « inscription », le service répond immédiatement en appelant l'opération « réponse » contenant le même *transID*. L'opération « réponse » a les attributs suivants : *status*, *transID* et *duration*. Le *status* indique si l'opération d'inscription a réussi ou échoué. Le *transID* de la réponse est le même que celui de l'inscription. La *duration* peut être différente de la valeur demandée.

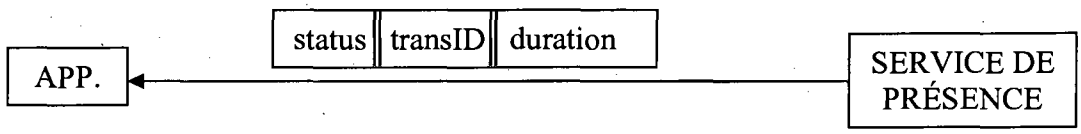


Fig. 4 : L'opération « réponse »

Opération notification

Le service appelle l'opération « notification » aussitôt que l'opération « réponse » réussit. L'opération « notification » contient les attributs suivants : *watcher*, *target* et *transID*. Les deux premiers attributs sont identiques à ceux donnés dans l'opération « inscription » qui a déclenché cette opération. Le *transID* est un identifiant unique pour cette notification. L'opération de « notification » contient aussi de l'information de présence (contenu détaillé dans la section ...).

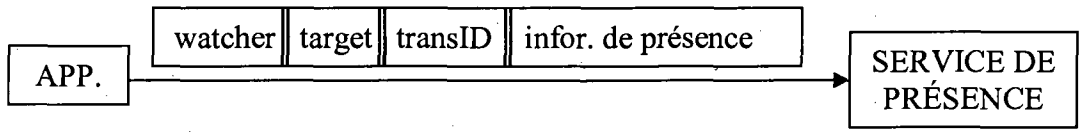


Fig. 5 : L'opération « notification »

Le service appelle l'opération de notification jusqu'à la durée spécifiée par le paramètre *duration* qui est supposé non nul. La notification se produit dès qu'il y a n'importe quel changement dans l'information de présence des *presentitys*. Si la durée est zéro, alors une seule notification est appelée, réalisant une seule fois l'interrogation de l'information de

présence. Il n'y a pas d'accusé de réception pour l'opération de notification défini dans le CPP.

Flot de données dans un système de présence

Afin de mieux comprendre les flots de données circulant lors d'une communication, une architecture à trois couches a été adoptée et est accompagnée par un schéma. Cette architecture comprend un cas réel, une requête d'initiation de session suivie d'une session multimédia. La façon dont les données circulent entre les entités est à retenir.

Dans l'architecture 'trois couches' de la fig. 6, un système de traitement d'appel (dont l'écriture est en italique) accompagné d'un système de présence ont démontré la situation illustrée par la façon qu'un système de communication fonctionne. Il a deux types de composantes, *user agents* et réseau de serveurs. Au cours de l'envoi d'un message d'initiation de session, un *user agent* permet à un appelant (ou un *watcher* dans un système de présence) d'initier une requête et permet également à un appelé (ou un *presentity* dans un système de présence) de répondre à la requête.

Un *presence user agent*, est un composant logique dans les systèmes de présence, prend en charge les terminaux d'un *presentity* (téléphone, cellulaire, *Personal Digital Assistant (PDA)* etc.), manipule l'information de présence et livre les données de présence fournies par les terminaux aux *Presence Agent*. Le contrôle d'accès se fait dans ce cas par l'intermédiaire de services personnalisés, qui sont programmés par les usagers, gérés par le serveur de contrôle d'accès, et exécutés par les *user agents* ou les *presence agents*.

Un message d'initiation de session peut contenir l'information expliquant une session média (ex. audio, vidéo, etc.). La session se produit après l'initiation. Le serveur dirige les messages d'initiation de session vers leur destination. Un *registrar* accepte les requêtes et garde l'information des usagers pour fournir le service de localisation, considéré comme la clé de la mobilité. Se trouvant en haut de la couche transport, les messages d'initiation de sessions peuvent transmettre des données : description de session, messages instantanés, documents de présence, etc. Une fois la session s'établie, les entrées médias temps réel sont échantillonnées,

converties en un format numérique et encapsulées. Ces entrées médias sont en conséquence distribuées.

Une différence dans le traitement de service entre les systèmes ci-dessus est constatée : Le système de traitement d'appel fournit un service synchronisé à un appelant et à un appelé, initié par l'envoi d'un message d'initiation de session INVITE d'un appelant. Le système de présence fournit un service non-synchronisé à un *watcher* et un *presentity*, initié par l'envoi d'un message SUBSCRIBE d'un *watcher* et suivi par un message NOTIFY non-synchronisé de la *presentity*. Notant ainsi que la présente mémoire met l'accent sur le système de présence (Jiang et al., 2005b), (Jiang et al., 2005a).

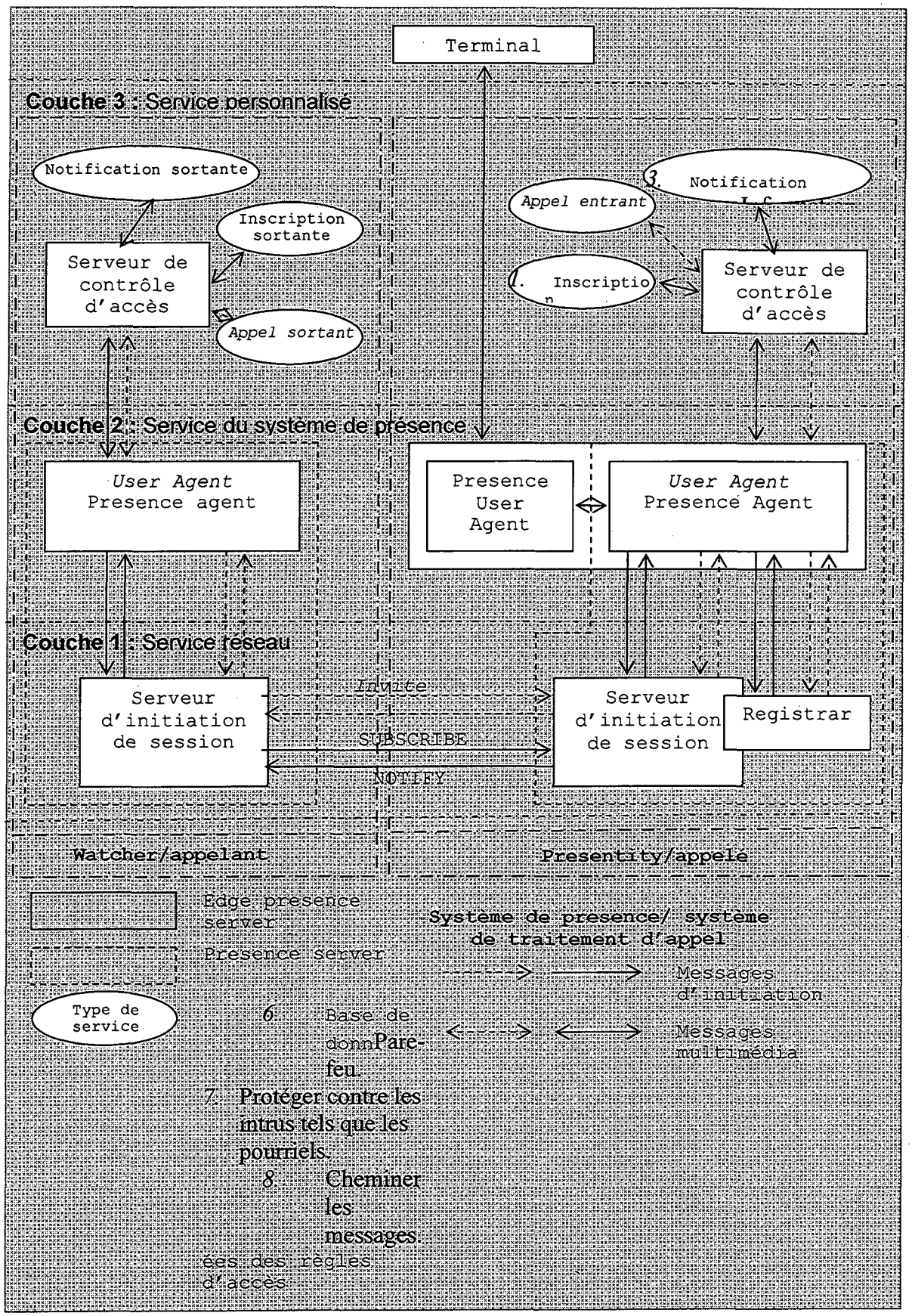

Terminal
Couche 3 : Service personnalisé
Notification sortante
Inscription sortante
Serveur de contrôle d'accès
Appel sortant
Appel entrant
Notification
Inscriptio n
Serveur de contrôle d'accès
Couche 2 : Service du système de présence
User Agent Presence agent
Presence User Agent
User Agent Presence Agent
Couche 1 : Service réseau
Serveur d'initiation de session
Invite
SUBSCRIBE
NOTIFY
Serveur d'initiation de session
Registrar
Watcher/appelant
Presentity/appelé
Edge presence server
Presence server
Système de presence/ système de traitement d'appel
Messages d'initiation
Messages multimédia
Type de service
6 Base de donn Pare-feu
7. Protéger contre les intrus tels que les pourriels.
8 Cheminer les messages.
ées des règles d'accès

Fig. 6 : Architecture trois couches - flot de données

Format de données de l'information de présence «PIDF»

Introduction sur PIDF :

Le format de données de l'information de présence PIDF « *Presence Information Data Format* » défini dans (Sugano et al., 2004), est le format de référence pour toutes les implémentations de présence de l'IETF. Le PIDF est ainsi conçu à la suite du profil commun de présence CPP et il est le format de données commun pour tous les systèmes conformes à CPP. Le PIDF a été conçu dès le début pour être extensible, flexible et ainsi convenable pour les systèmes de présence courants et futurs.

Le format spécifié dans cette partie définit de format de présence de base. Il définit aussi un ensemble minimum d'état de présence requis pour former un système de présence. On va énumérer aussi quelques extensions fortement recommandées, voire même indispensables et très utiles pour la fiabilité et la compréhension de l'information exigée par la majorité des domaines. Une étude plus profonde est réalisée prochainement afin de concevoir un format de présence adapté au contexte médical.

Décision de conception:

On a adopté le modèle décrit dans (Sugano et al., 2004) comme le point de départ. Ce modèle contient plusieurs déclarations au sujet de l'information de présence qui peut être considérée comme un ensemble de base de contraintes pour la conception du format.

Le modèle minimal :

Ce modèle est basé sur le travail du groupe IMPP «*Instant Messaging and Presence Protocol*» appartenant à l'IETF «*Internet Engineering Task Force*» (Internet Society, s. d.). Cette dernière organisation à pour mission de produire des documents technique et d'ingénierie appropriés qui influent la manière dont les gens conceptualise, utilise et gère l'internet de manière à rendre l'internet plus efficace. Ces documents comprennent des

standards de protocole, les meilleures pratiques actuelles, et des documents informationnels de différents types.

L'information de présence se compose d'un ou plusieurs **tuples de présence** *«presence tuples»*; ce dernier se compose à sont tour d'un **statut** *«status»*, **adresse de communication** *«communication adress»* qui est optionnelle et d'autres **extensions de présence** *«presence extension»*. Notez que dans ce document, l'**adresse de contact** *«contact address»* réfère à un URI «*Universal Ressource Identifier*» (RFC2778 : sec.3).

L'ensemble minimum de la valeur *status* est *open* et *close*. Il y a d'autres valeurs de *status* qui peuvent être combinées ou échangé avec ceux-ci (RFC 2778 : Sec.3, RFC2779 : Sec.4.4.1 - 4.4.3 (Day et al., 2000b). Ces valeurs de *status* peuvent influencer le traitement des requêtes par les récepteurs concernés, ils peuvent aussi déclencher des processus spéciaux comme la génération automatique de messages ou redirection d'appels. Un *status* peut se composer d'une ou plusieurs valeurs (RFC2778 : Sec.2.4).

Il doit y avoir un moyen d'extension du format de présence commun pour représenter d'autres informations complémentaires qui ne sont pas incluses dans le format commun. Les mécanismes d'extension et d'enregistrement doivent être définis dans le schéma de l'information de présence, incluant de nouveaux état de *status* et de nouvelles formes à d'autres extensions de présence (RFC2779 : Sec.3.1.4 - 3.1.5).

Le format commun de présence doit disposer d'un moyen qui identifie d'une manière unique la *Presentity* dont l'information de présence est rapportée (RFC2779 : Sec.3.1.2).

Le format de présence commun doit permettre à la *Presentity* de sécuriser l'Information de Présence envoyé au *Watcher*. Il doit aussi permettre l'intégrité, la confidentialité et l'authenticité pour être appliqué à l'information de présence (RFC2779 : Sec5.2.1, 5.2.4, 5.3.1, 5.3.3).

Caractéristiques supplémentaires :

Comme supplément au modèle minimum décrit au dessus, le format spécifié dans cette spécification a les caractéristiques suivantes :

- On peut spécifier **les priorités relatives** *(relative priorities)* aux adresses de contact dans notre modèle. Ils vont permettre à la source de l'information de présence d'indiquer au récepteur «*watcher user agent*» ses préférences sur des moyens de contact multiples. Par exemple un spécialiste qui veut recevoir toutes les communications sur son cellulaire au lieu de son ordinateur met la priorité relative à son cellulaire la plus haute.

- Le format de présence est capable de contenir le *timestamp* (heure et date), il permet aux récepteurs de savoir l'heure et la date de la création de l'information de présence reçue. En conséquence, il peut être utilisé pour détecter une «replay-attack», indépendamment du mécanisme fondamental de signature.

2.2.2. Technologies impliquées

Dans la dernière partie, on a vu le modèle abstrait d'un système de présence. Dans cette partie, on présentera les technologies principales impliquées dans ce projet. On commencera par définir la base de données XML, puis la communication client serveur, ensuite le protocole de communication SIP pour finir avec les considérations de sécurité dont il faut tenir compte.

La base de données XML

Introduction

XML (*Extensible Markup Language*, « langage de balisage extensible ») est un langage informatique de balisage générique. Le World Wide Web Consortium (W3C), promoteur de standards favorisant l'échange d'informations sur Internet, recommande la syntaxe XML pour exprimer des langages de balisages *spécifiques*.

Son objectif initial est de faciliter l'échange automatisé de contenus entre systèmes d'informations hétérogènes (interopérabilité). XML retient les principes essentiels comme :

- la structure d'un document XML est définissable et se valide par un schéma;

- un document XML est entièrement transformable dans un autre document XML.

Pour plus amples informations concernant le langage XML, référez vous à l'annexe A.1.

Le format de données de l'information de présence encode l'information de présence en XML. En considérant les caractéristiques de l'information de présence qui a une structure hiérarchique et qui devrait être entièrement extensible, XML est considéré comme le cadre le plus adapté. L'information de présence est donc stockée dans un document XML, ce qui consiste la base de données du système de présence. Dans ce paragraphe, on va identifier les raisons du choix d'un document XML comme base de données et la façon de gérer l'information de présence. Pour plus amples informations sur la notion de base de données, référez-vous à l'annexe A.3.

Méthode de stockage de l'information de présence

L'information de présence est stockée dans une base de données, qui est dans ce cas un document XML. On a choisi cette méthode de stockage parce que

- XML est une méthode simple pour mémoriser des données structurées dans un fichier texte.
- Il est un moyen simple d'échanger des données à travers un réseau de données, ce qui concorde parfaitement à notre cas.
- Langage de balisage extensible et flexible d'utilisation
- Un standard international interprété par la majorité des systèmes informatiques, un simple éditeur de texte peut lire son contenu.
- Pas besoin d'un système de gestion de base de données puissant, XML satisfait à nos besoins.

Manipulation de l'information

Tout d'abord, le document de présence (la base de données) doit être analysé par un parseur afin de gérer l'information existante. Le parseur qui est choisi dans notre cas est *xerces-c DOM Parser*, parce que

- Il a été utilisé par un autre étudiant du groupe de recherche, sa compréhension et son intégration dans le système est plus facile.
- Il est implémenté en C++, le langage utilisé dans ce projet.
- Son API est ouverte pour tout le monde et il est bien documenté.
- Il est efficace, facile à intégrer et rapide d'accès de mémoire, il satisfait aux besoins.

Pour de plus amples informations sur la définition d'un parseur et sur la technologie *xerces-c DOM Parse,* référez-vous à l'annexe A.4.

Vue d'ensemble de la PIDF :

Le type de contenue 'application/pidf+xml' :

Le contenu de l'information de présence est de type pidf+xml, donc le paramètre *content type* de l'information de présence a la valeur 'application/pidf+xml'. Cette spécification suit les recommandations et conventions décrites dans le RFC3023 (Murata et al., 2001), incluant la convention d'appellation du type (*'+xml' suffix*).

Le contenu de l'information de présence :

Dans ce qui suit, les grandes lignes des informations qui peuvent se trouver dans un document de type 'application/pidf+xml' sont données. La définition complète du contenu de PIDF est dans la section suivante.

- Presentity URL : spécifie l'URL de la presentity.
- Liste des *tuples* de présence.
- *Identificateur* : marque pour identifier ce *tuple* dans l'information de présence.

Status : *open/close* et/ou des extensions.

1. *Communication address* : moyens de communication et adresse de contact de ce *tuple*. (Optionnelle)
2. *Relative priority* : valeur numérique spécifie la priorité de cette adresse de communication. (Optionnelle)

3. *Timestamp* : le temps et la date du changement de ce *tuple*. (Optionnelle)

4. *Readable comment* : des notes au sujet de ce *tuple*.

- Readable comment : des commentaires au sujet de la presentity.

Format de données de présence encodé par XML :

On va définir dans cette section le format de données de présence (PIDF) encodé par XML utilisé dans les systèmes conformes à CPP. La trame d'information de présence dans ce format qui se produit par la *presentity* (la source de l'information de présence), et qui se transporte au *watcher* par le serveur de présence, ou les passerelles, ne subit aucune interprétation ni modification. Étant donné que la PIDF est encodée par le langage XML, une brève description de ce langage est réalisée dans la section suivante.

L'élément <*presence*> :

Les éléments PIDF sont associés avec le nom d'espace-nom XML 'urn:ietf:params:xml:ns:pidf', ce nom est déclaré en utilisant l'attribut *xmlns*. L'espace-nom peut être déclaré par défaut ou associé avec quelques préfixes d'espaces-nom. Un **espace-nom** est un ensemble de ce qui est désignable dans un contexte donné par une méthode d'accès donnée faisant usage de noms symboliques (par exemple des chaînes de caractères avec ou sans restriction d'écriture).

La racine d'un objet 'application/pidf+xml' est l'élément <*presence*> associé à l'espace-nom de l'information de présence approprié. Ceci contient zéro à n éléments de type <*tuple*>, suivi de zéro à n éléments de type <*note*>, suivi par n'importe quel nombre d'extensions d'éléments optionnels à partir d'autres espaces-noms.

L'élément <*presence*> doit avoir un attribut '*entity*', la valeur de cet attribut est le 'pres' URL de la *presentity* publiant ce document de présence. Cette 'pres' URL pourrait être une adresse SIP. En conséquence, la valeur de l'attribut 'entity' pourrait avoir le format suivant : 'pres : personne@exemple.com'.

L'élément <*presence*> doit contenir une déclaration d'espace-nom ('xmlns') pour indiquer sur quel espace-nom le document de présence est basé. Le document de présence conforme à

cette spécification doit avoir l'espace-nom 'urn:ietf:params:xml:ns:pidf'. Il peut contenir d'autres déclarations d'espaces-nom pour les extensions utilisées dans le document XML de présence.

L'élément <*tuple*> :

L'élément <*tuple*> porte un tuple de présence, se composant d'un élément obligatoire <*status*>, suivi par n'importe quel nombre d'extension d'éléments optionnels (éventuellement d'autres espaces-nom), suivi par un élément <*contact*> optionnel, suivi par n'importe quel nombre d'éléments <*note*> optionnels, suivi par un élément <timestamp> optionnel.

Les *tuples* fournissent une manière de segmentation de l'information de présence. Les protocoles et les applications peuvent choisir de segmenter l'information de présence associée avec une *presentity* pour plusieurs raisons. Par exemple l'information de présence d'une *presentity* peut provenir de plusieurs dispositifs distincts ou applications différentes, ou être générée à des périodes différentes de temps.

L'élément <tuple> doit contenir un attribut '*id*' qui est utilisé pour qu'il se distingue dans la même presentity. La valeur d'un attribut '*id*' doit être unique par rapport aux valeurs des autres *tuples* de la même *presentity*. Une valeur '*id*' est employée par des applications traitant le document de présence pour identifier le *tuple* correspondant dans l'information de présence pré acquise de la même *presentity*. La valeur de l'attribut '*id*' est une chaîne de caractère arbitraire, qui n'a pas de signification au-delà de fournir des moyens pour distinguer les valeurs d'un <*tuple*>.

L'élément <statut> :

L'élément <*status*> contient un élément optionnel <*basic*>, suivi par n'importe quel nombre d'extensions d'éléments optionnels (possiblement d'autre espaces-nom), sous la restriction qu'au moins un élément enfant apparait dans l'élément <*status*>. Ces éléments enfants de <status> contiennent des valeurs de statut de ce tuple. En permettant des valeurs de statut multiples dans un seul élément <*tuple*>, différents types de valeurs de *status* tels que l'accessibilité et la localisation, peuvent être représentés par un <*tuple*>.

Il existe plusieurs extensions de valeurs de statut (accessibilité et localisation) définies par des définitions de schéma. On peut les employer dans l'élément *<status>* pour enrichir ou compléter l'information de présence. Le schéma désigné par l'URN (Universal Ressource Name) urn:ietf:params:xml:schema:pidf:status:rpid contient plusieurs d'extensions de ce genre.

Notez que, bien que l'élément *<status>* doit avoir au moins un élément de valeur de statut, ce statut peut ne pas être l'élément *<basic>*.

L'élément *<basic>* :

L'élément *<basic>* contient une des chaînes de caractères suivante : open et close.

Open : les éléments du tuple correspondant sont disponibles à des requêtes.

Close : les éléments du tuple correspondant ne sont pas disponibles à des requêtes

L'élément *<contact>* :

L'élément <contact> contient un URL de l'adresse de contact. Il a un attribut optionnel '*priority*', dont la valeur signifie une priorité relative à cette adresse de contact. La valeur de cet attribut doit être un nombre décimal entre 0 et 1 inclusivement avec au maximum trois chiffres après le point décimal. Plus la valeur est élevée, plus la priorité est élevée, voici des exemples de valeur de priorité : 0, 0.021, 0.5, 1.00. Si cet attribut est absent, les applications doivent assigner à l'adresse de contact la priorité la moins élevée. Si la valeur est hors limite alors les applications devraient ignorer la valeur et la traiter comme si l'attribut n'était pas présent.

Les applications devraient traiter les contacts qui ont les priorités plus élevées comme les plus préférées vis-à-vis à ceux qui ont les priorités les moins élevées. La façon dont les contacts de mêmes priorités sont manipulés dépend de l'implémentation.

L'élément *<note>* :

L'élément *<note>* contient une valeur de chaîne de caractère généralement employée pour des commentaires lisibles par l'humain. Cet élément peut apparaître entant qu'élément enfant

de *<presence>* ou de *<tuple>*. Dans le premier cas, le commentaire est au sujet de la presentity. Dans le deuxième cas, le commentaire concerne un tuple particulier.

Notez qu'un élément *<note>* ne devrait pas être employé et interprété comme un substitut non interopérable au statut de son élément parent.

Cet élément devrait avoir un attribut spécial 'xml :lang' pour spécifire le langage utilisé dans le contenu de cet élément. La valeur de cet attribut est l'identificateur de langage comme défini par le (Alvestrand, 2001). Il peut être absent quand le langage employé est indiqué implicitement par le contexte.

L'élément *<timestamp>*:

L'élément *<timestamp>* contient une chaîne de caractère qui indique la date et l'heure du changement de *status* de ce *tuple*. La valeur de cet élément doit suivre le '*IMPP datetime format*' (Newman et al., 2002). Les *timestamps* qui contiennent les lettres T ou Z doivent utiliser la forme majuscule.

Comme mesure de sécurité, cet élément devrait être inclus dans tout les *tuples*, à moins que l'heure exacte du changement du statut ne puisse pas être déterminée. Pour les directives de sécurité concernant la réception de l'information de présence par les *watchers* avec *timestamp*, voir la partie «Considérations de sécurité».

Une *presentity* ne doit pas générer des éléments successifs de *<presence>* qui ont le même *timestamp*.

Communication client/serveur

Dans ce projet, les terminaux doivent communiquer à distance, le protocole SIP s'avère adéquat pour ce genre de tâche. Il comporte toutes les signalisations nécessaires comme 'subscribe' et 'notify' mentionnées précédemment. En plus, il est une nouvelle norme internationale définie par le RFC3261.

L'implémentation de la transmission de données est faite avec la pile commerciale *M5T SIP safe*, et cela parce que

- L'équipe de recherche a accès au code source de la pile.
- La pile contient toutes les signalisations nécessaires dans ce projet.
- Il y a certains membres de l'équipe qui savent manipuler la pile et qui sont à notre disposition.
- La pile est implémentée avec C++ qui concorde avec le langage de programmation choisi dans ce projet.

Le protocole de communication SIP

Session Initiation Protocol (dont l'abréviation est **SIP**) est un protocole normalisé et standardisé par l'IETF (décrit par le RFC 3261 et complété par le RFC 3265) qui a été conçu pour établir, modifier et terminer des sessions multimédia. Il se charge de l'authentification et de la localisation des multiples participants. Il se charge également de la négociation sur les types de média utilisables par les différents participants. SIP ne transporte pas les données échangées durant la session comme la voix ou la vidéo. SIP étant indépendant de la transmission des données, tout type de données et de protocoles peut être utilisé pour cet échange.

SIP comprend plusieurs fonctionnalités. Notons *subscribe*, elle consiste en une méthode permettant d'inscrire les utilisateurs auprès du serveur de présence. Elle contient l'adresse SIP de l'utilisateur voulant s'inscrire. Cette adresse étant l'identificateur unique pour cet utilisateur, elle a la forme suivante : 'anonyme@domaine.com'. Notons aussi la fonction *notify*, c'est une méthode permettant de notifier les utilisateurs de nouvelles informations. Finalement la méthode *publish* qui permet aux utilisateurs de publier leurs nouvelles informations au serveur. Tous ces utilisateurs sont identifiés par leur adresse SIP.

L'annexe A.2 décrit plus profondément le protocole SIP, quoique le RFC3261 reste toujours le document officiel de SIP.

Considérations de sécurité:

Vu que l'information de présence est considérée confidentielle, le protocole utilisé pour son transport via le réseau doit être capable de la protéger contre les menaces possibles, telles que

l'écoute clandestine «*eavesdropping*», *corruption*, *tampering* et *replay attacks*. Des mécanismes de sécurité doivent être disponibles pour l'utilisation point à point entre les *presentities* et les *watchers*, même si ces deux emploient des protocoles de présence différents et communiquent à travers une passerelle. PIDF devrait suivre les recommandations normalisées pour l'usage de S/MIME (Ramsdell, 2004).

Notez que l'usage de *timestamps* dans PIDF peut fournir quelques protections rudimentaires contre les *replay attacks*. Si un *watcher* reçoit de l'information de présence périmée, il devrait l'ignorer. Les applications et les protocoles aussi sont avisés à adapter leurs propres règles pour déterminer quelle fréquence devrait-elle considérée pour rafraîchir l'information de présence. Par exemple, si l'information de présence semble être datée de plus d'une heure, elle pourrait être considérée comme de l'information périmée, une notification générée pour cette information de présence ne prendra pas tout ce temps là pour parvenir au *watcher*. Aussi si une *presentity* n'a pas rafraîchi son information de présence depuis x temps, on peut déduire qu'elle a été déconnectée ou débranchée sans préavis.

2.3. Conclusion

Dans la partie étude de l'état de l'art, on a parcouru les articles pour en tirer les informations pertinentes à ce projet. On a défini aussi les technologies principales qu'on utilisera prochainement. On a défini la base de données XML, puis la communication client serveur, ensuite le protocole de communication SIP pour finir avec les considérations de sécurité. Toutefois, il existe bien d'autres technologies impliquées dans ce projet qu'on définira dans l'annexe A.

3. CONCEPTION DE LA SOLUTION

Dans la partie cadre de référence, on a défini abstraitement le système de présence et ses composantes, puis on a défini les technologies principales impliquées dans ce projet. Dans cette partie conception de la solution, on définira le processus de fonctionnement, puis la base de données spécifique pour ce projet.

3.1. Processus de fonctionnement

On commencera dans cette partie par une vue d'ensemble du processus de fonctionnement. Puis, on expliquera la méthode d'enregistrement et d'inscription des personnels médicaux. Ensuite, on développera aussi la méthode de notification des personnels médicaux. Après, on définira le contrôle d'accès. Et, pour finir, on développera le scénario de demande d'un spécialiste spécifié. C'est un outil de recherche de spécialiste qui se base sur sa spécialité et sa localisation. D'autres paramètres peuvent s'inclure dans cette recherche. Ce système de recherche constitue l'originalité de ce projet vue qu'il n'y a aucun système de présence qui l'intègre.

3.1.1. Vue d'ensemble

On identifie dans cette partie les scénarios de communication possibles entre les entités du modèle de présence. Chaque scénario est présenté par un diagramme de séquence. L'intérêt de ces diagrammes de séquence est de montrer l'enchaînement des requêtes envoyées afin de diriger une communication.

L'utilisateur se connecte par l'intermédiaire d'un ou plusieurs moyens de communication. Le moyen de communication par défaut est constitué d'un programme de messagerie standard. Ce programme fonctionne moyennant un terminal informatique, par exemple un ordinateur, un PDA, ...

Chaque terminal est enregistré à un utilisateur. On aura un lien temporaire entre les deux. Le terminal échange l'information avec le serveur de présence. Chaque terminal a un *tuple* de présence qui lui est spécifique dans l'information de présence de l'utilisateur. Donc, chaque terminal a sa propre information de présence. C'est à travers ces terminaux que l'échange de l'information avec le serveur et les autres clients va être réalisé.

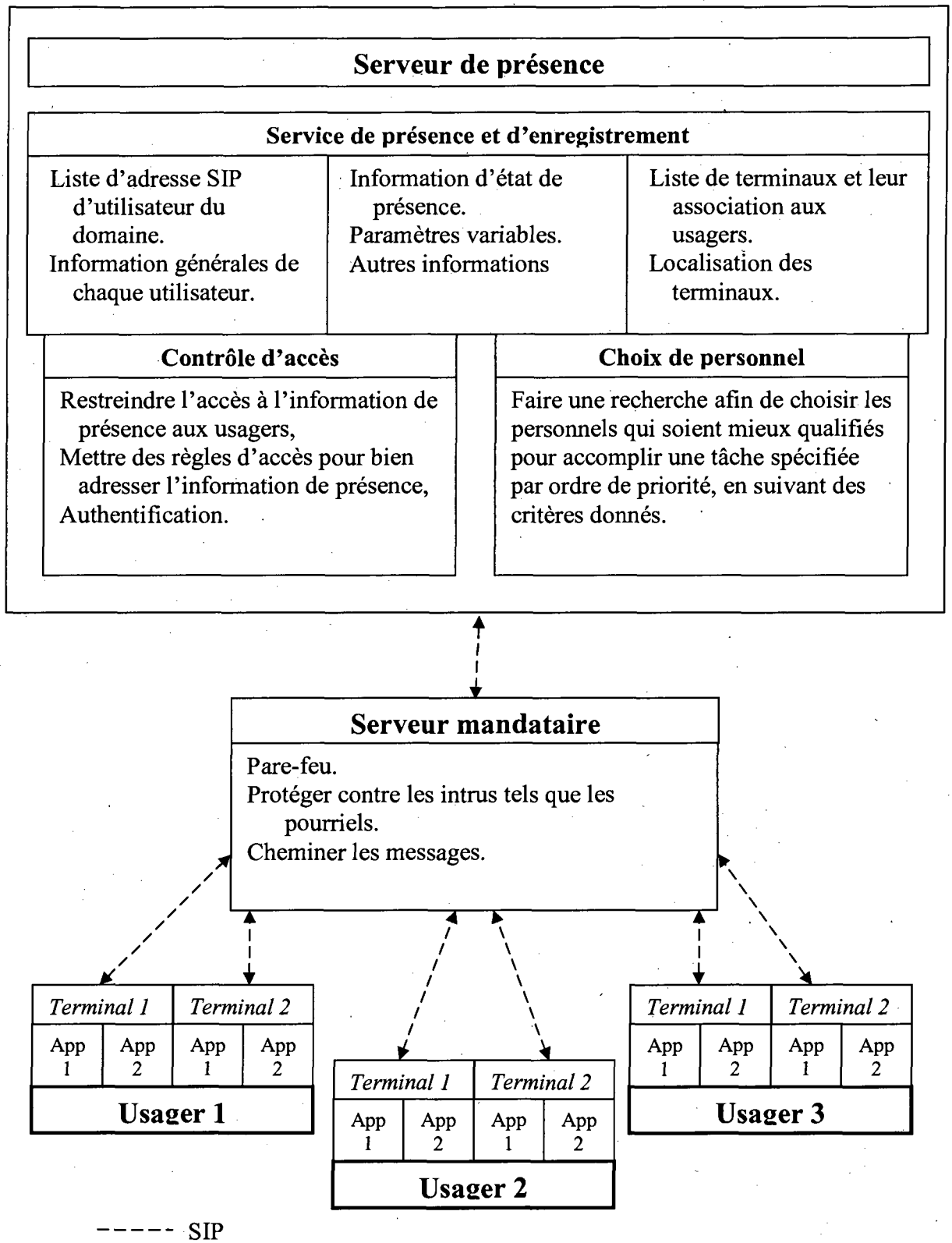


Fig. 7 : Schémas d'ensemble du système de présence

Comme le mentionne la figure 7, le serveur de présence s'occupe de la gestion de l'information de présence. Il maintient l'information de présence, les règles d'accès et tous les terminaux de communication enregistrés. Il s'occupe aussi d'établir un lien entre les adresses d'enregistrement et leurs moyens de contact. Il peut chercher les usagers les plus qualifiés pour faire une tâche donnée par ordre de priorité. De plus, il notifie par l'intermédiaire d'un serveur mandataire les entités concernées des nouvelles informations de présence. Le serveur de présence différentie les utilisateurs par leur adresse SIP (URI : *Uniform Resource Indicator*). Cette adresse a la forme suivante: sip:example@domaine.exp

3.1.2. Enregistrement et inscription du personnel médical

L'enregistrement au système de présence permet d'identifier un usager. Un responsable saisit les données concernant les usagers voulant s'enregistrer dans le système, telles que le nom, le prénom, la date de naissance, l'adresse, ... Ces informations étant fournies par l'usager, une adresse SIP lui sera affectée par le responsable. L'usager peut l'utiliser comme un moyen de communication. Cette adresse est utilisée par une application de communication. Cette application permet l'envoi des messages, l'envoi des fichiers et permet aussi la communication vocale.

Après s'être enregistré, l'utilisateur peut modifier, ajouter ou supprimer ses propres informations telles que ses préférences, ses numéros de téléphone, ses moyens de contact,... Il y a certaines de ses informations qu'il ne peut pas modifier, seul le responsable du domaine a l'autorisation de les modifier, sous présentation des pièces justificatives par le propriétaire e.g. nom, spécialité, date de naissance, ...

Pour permettre à un usager enregistré d'échanger de l'information concernant une ou plusieurs ressources avec le serveur, il faut qu'il s'inscrive au serveur de présence. L'opération d'inscription consiste à émettre au serveur de présence une requête *subscribe*. Elle doit contenir les ressources que le client veut cibler. Le serveur traite sa demande et lui envoie l'information de présence demandée et permise. Cette opération s'appelle notification.

Le diagramme de la figure 8 décrit deux scénarios; demande d'enregistrement et demande d'inscription.

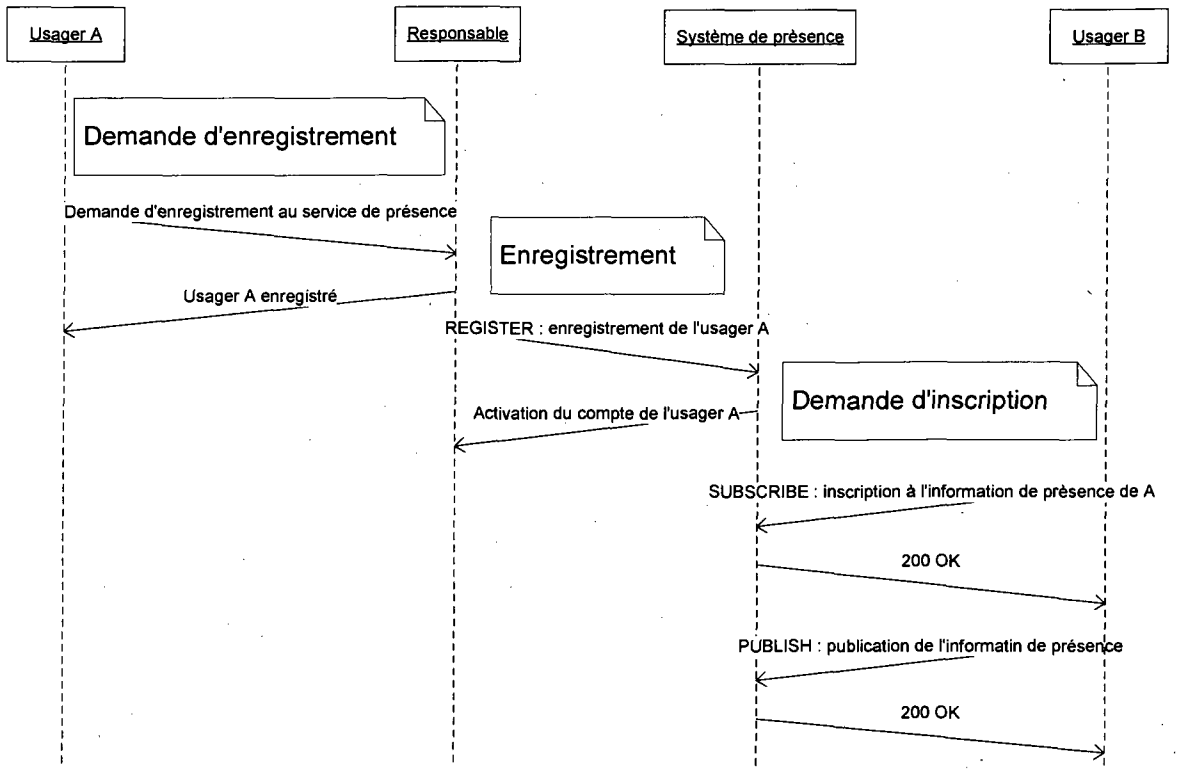


Fig. 8 : Diagramme de séquence de demande d'enregistrement et d'inscription

Le personnel publie ses propres informations de présence par l'intermédiaire de la fonction *publish*. Cette fonction ne demande pas l'établissement d'un dialogue « *dialog* » ou session entre les interlocuteurs, mais le serveur de présence vérifie l'identité de l'expéditeur avant d'agir.

3.1.3. Notification du personnel médical

Un utilisateur A peut demander l'information de présence d'un groupe d'utilisateurs B spécifique par l'intermédiaire de leur adresse SIP. L'utilisateur A envoie une demande d'inscription d'une durée spécifiée renouvelable au serveur de présence afin de savoir l'information de présence du groupe B à un moment donné. De cette façon, chaque fois qu'il y a un changement dans l'information de présence des personnes ciblées, le serveur de présence notifie le demandeur A des nouvelles informations de présence selon les permissions d'accès

soumises. Cette opération est appelée *notify*, Le diagramme de séquence de la figure 9 montre la façon dont la demande de notification est traitée.

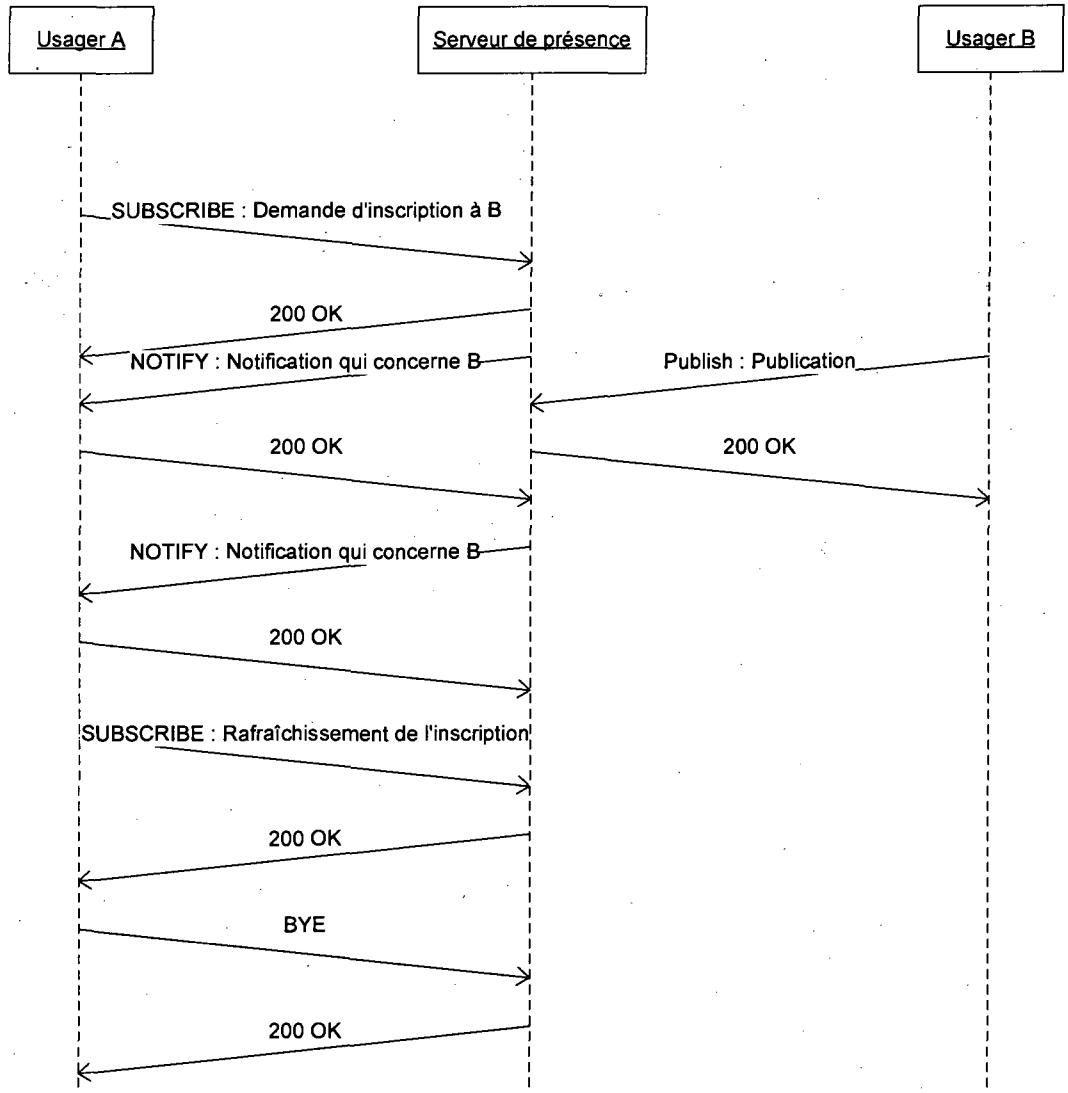


Fig. 9 : Diagramme de séquence de demande d'inscription à un usager

3.1.4. Contrôle d'accès

Afin de contrôler efficacement l'accès, seul le responsable du domaine peut enregistrer les utilisateurs. Il a aussi le pouvoir absolu de gestion du système de présence. Une fois

enregistré, l'utilisateur peut modifier, supprimer ou ajouter certains paramètres de son profil. Il ne peut pas toucher aux paramètres sensibles et utiles au système de présence tels que le nom, le prénom, la spécialité, la date de naissance, les diplômes obtenus, l'adresse SIP, ... Par contre, il peut modifier, supprimer ou ajouter une adresse civique, un numéro de téléphone, un moyen de contact, des données de présence, ...

L'utilisateur ou le personnel doit s'inscrire au serveur de présence afin d'échanger l'information de présence, seuls les personnels médicaux ont le droit de voir l'information de présence de n'importe quel autre utilisateur du même domaine. Un utilisateur peut créer une liste d'amis à partir d'adresses SIP qu'il connaît. Le serveur lui notifie alors les informations de présence de ses amis. Cette action n'est pas réciproque, c'est-à-dire que quelqu'un de sa liste d'ami n'est pas nécessairement notifié. Il n'y a aucune restriction au niveau de notification tant que les deux utilisateurs sont dans le même domaine. Le notifié doit connaître l'adresse SIP du notifiant en avance.

Seuls les usagers pouvant être impliqués dans une éventuelle situation d'urgence ont la permission d'accès au service de demande d'un spécialiste. C'est le responsable du domaine qui gère ces permissions.

3.1.5. Scénario de demande d'un spécialiste spécifié

Lors d'une situation d'urgence, un ambulancier aurait besoin d'un spécialiste pour le guider à identifier l'état du patient. Il peut aussi avoir besoin d'avis pour stabiliser son état. L'ambulancier veut aussi savoir vers quel endroit doit-on le transférer afin d'éviter les pertes de temps dues à un transfert inutile. Pour cela, un système de recherche de personnel qualifié a été mis en place.

Diagramme de séquence général

Lorsqu'un ambulancier, ou autre usager, a besoin de communiquer avec un spécialiste, il envoie alors une requête de recherche au serveur de présence. Cette requête comporte alors la spécialité voulue et d'autres critères pertinents à la recherche.

Le serveur de présence analyse ensuite la requête, puis cherche dans sa base de données une liste des spécialistes les plus adéquats. Une fois que le serveur établit cette liste, il l'envoie au

demandeur. Elle est rafraichie chaque fois qu'il y a un changement de la liste. Elle contient de l'information pertinente aux spécialistes trouvés tels que leur adresse SIP, leurs moyens de contact, leur nom, leur prénom, leurs numéros de téléphone, leur numéro de fax...

Par la suite, l'ambulancier peut se mettre en relation avec celui qu'il trouve le plus compatible avec la situation au moyen de terminal de communication à sa disposition. Dans le schéma de séquence de la figure 10, on a schématisé un cas de communication ou l'ambulancier établit une session multimédia avec un spécialiste. Cette session est initialisée avec le protocole SIP.

Du point de vue technique, les deux interlocuteurs ouvrent un dialogue entre eux par l'intermédiaire de la fonction *subscribe* qui établit un *dialog* identifié par le paramètre *dialogID.* Ce dialogue consiste en une demande d'information envoyée par l'usager qui contient plusieurs paramètres d'identification. Puis, le serveur lui répond par un *notify* dont la charge utile est un fichier XML qui contient les spécialistes trouvés par le système de recherche de personnel qualifié. À chaque fois que la réponse change, le serveur envoie un rafraîchissement, jusqu'à ce que l'utilisateur ferme la session.

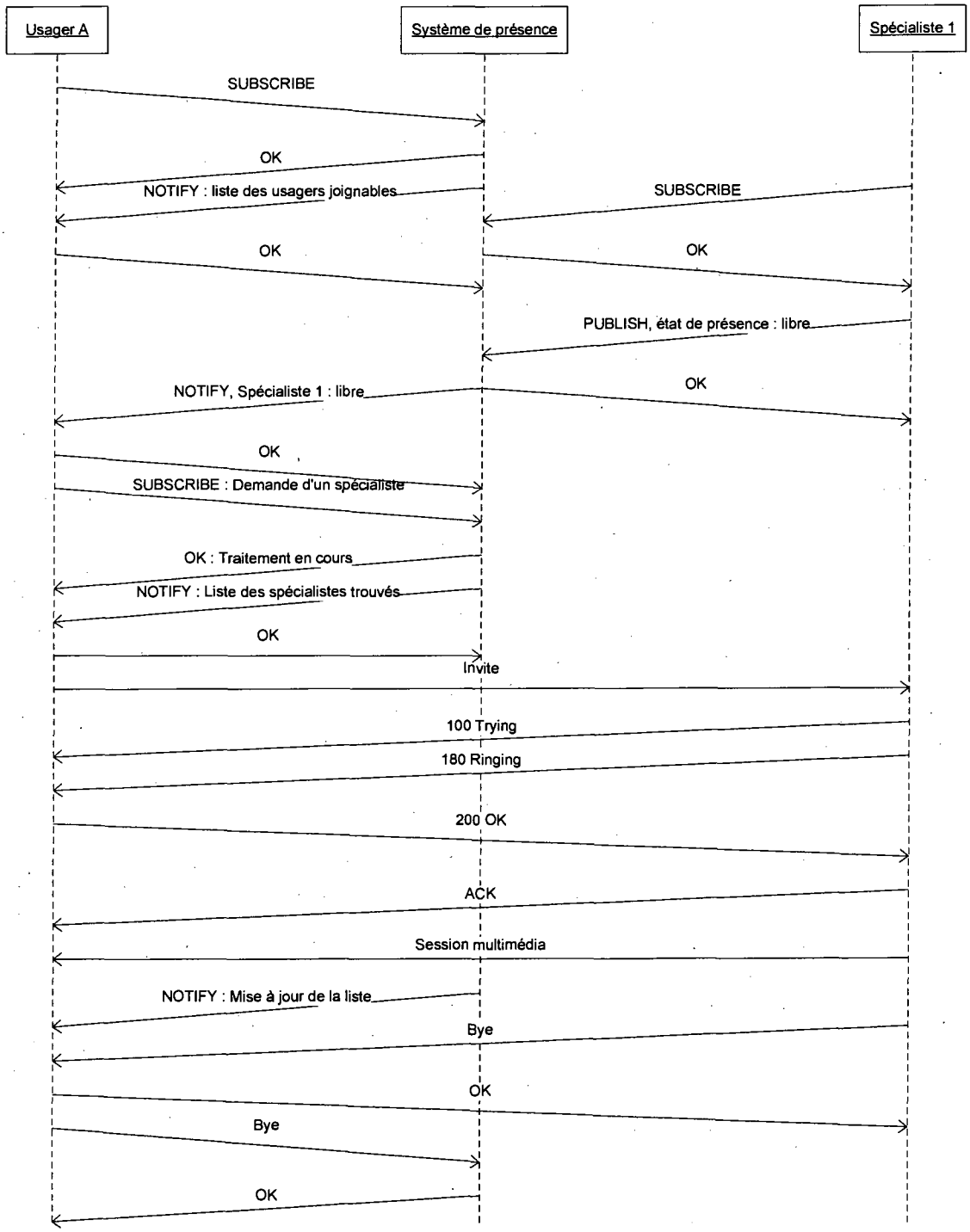


Fig. 10 : Diagramme de séquence général

Il arrive que le serveur de présence ne trouve aucun spécialiste qui répond aux critères exacts demandés. Le paragraphe suivant explique comment le serveur de présence gère ce genre de problème.

Algorithme de recherche de spécialiste

L'administrateur fixe les paramètres admissibles que le système utilise pour effectuer sa recherche. Il peut aussi les modifier à n'importe quel moment. Ces paramètres sont fixés par ordre de priorité. Il fixe aussi les listes d'éléments pertinents. Chaque liste contient des valeurs dont la première valeur est la principale. Si cette dernière ne correspond à aucun usager, il passe aux valeurs suivantes. Par exemple, si on prend le paramètre 'spécialité', on trouve les valeurs 'cardiologue' puis 'généraliste' dans la même liste. Au cas où le système ne trouve pas un cardiologue disponible, il peut chercher un généraliste.

Il y a certaines valeurs de ces paramètres qui sont relativement proches de point de vue signification, donc chacun de ces valeurs à une liste d'éléments du même type qui sont classés par ordre de proximité. Cette liste s'appelle liste d'éléments pertinents.

Pour l'instant, les paramètres qui sont pris en considération sont : 'état de présence', 'spécialité', 'localité' ou 'adresse'. L'utilisateur du système de recherche de spécialiste peut fixer un ou plusieurs de ces paramètres sauf 'état de présence', puisque c'est le système qui le gère. Si le système trouve dans une requête un paramètre non admissible, il n'est tout simplement pas pris en considération.

La liste d'éléments pertinents d'un élément change suivant la signification de ce dernier. Par exemple, si on prend la valeur 'libre' du paramètre 'état de présence', il y a la valeur 'sur appel' en deuxième lieu puis 'joignable' en troisième lieu. Pour le paramètre 'localité', c'est le lieu le plus proche du lieu recherché qui est pris en priorité.

Le système de recherche vise au début les usagers qui satisfont les critères de recherche, il effectue une liste dont le nombre maximum est fixé par l'administrateur. Au cas où il ne trouve aucun spécialiste qui répond aux critères exacts demandés, il étale la recherche sur les listes d'éléments pertinents. Le système parcourt les paramètres un par un, pour finalement

sortir la liste des spécialistes qui pourraient être admissibles aux critères de la recherche. Le parcours des éléments de ces lites se fait par l'ordre établi par l'administrateur.

Séquence d'une demande de spécialistes spécifiques

Lorsqu'un usager A veut communiquer avec un spécialiste, il lance une recherche avec les paramètres désirés. Son application alors envoie un *subscribe* mentionnant qu'il s'agit d'une demande d'un spécialiste «cardiologue». Le service de présence intercepte cette requête et lui répond par la liste qu'il a trouvée en suivant l'algorithme de recherche de spécialiste. Entretemps, un autre cardiologue change son état de présence vers «libre», donc le système de présence rafraîchit sa liste en ajoutant ce spécialiste à la liste trouvée et notifie l'usager A de cette nouvelle liste. La figure 11 illustre ce qu'on vient de l'expliquer.

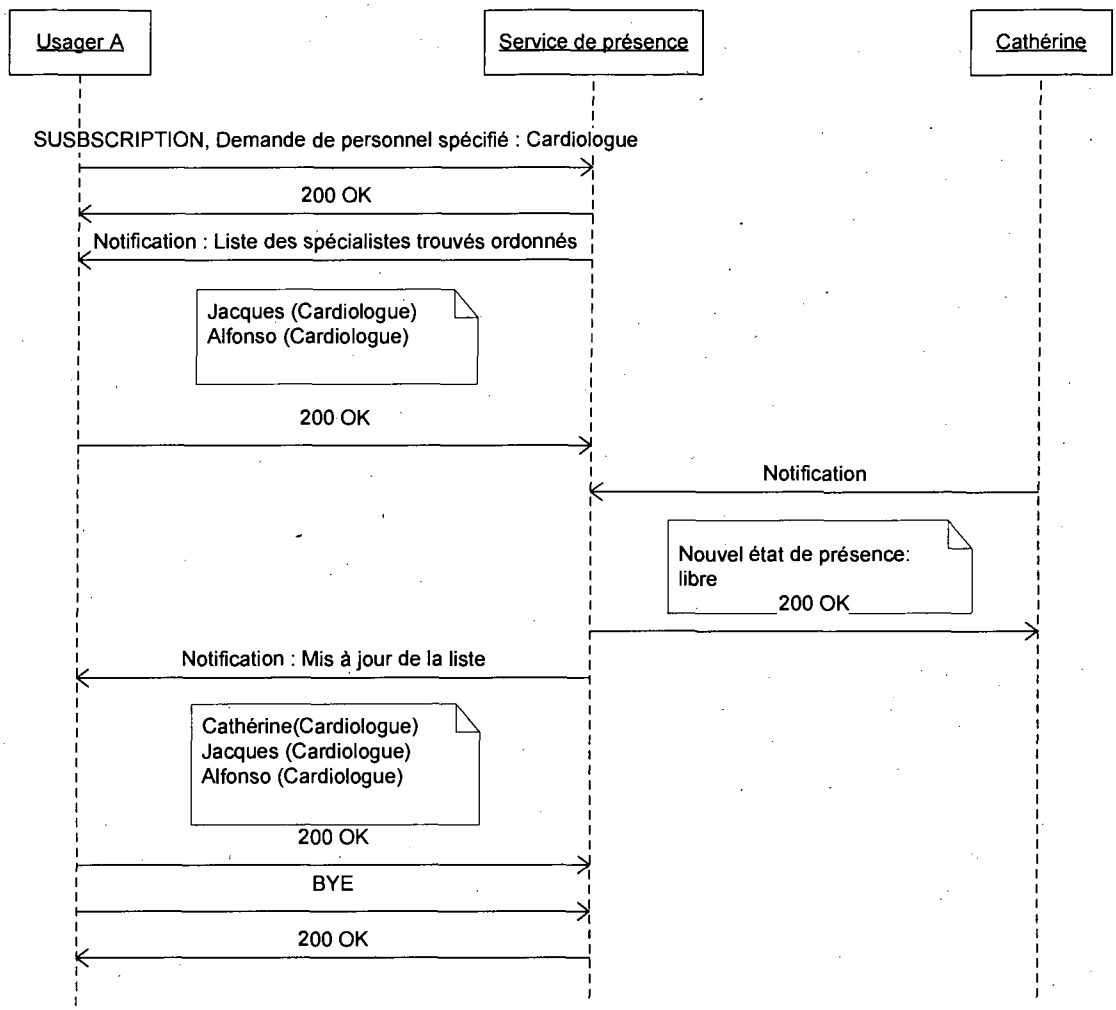


Fig. 11 : Diagramme de séquence d'une demande d'un spécialiste spécifique

Au début, ce système de recherche est un moyen d'aide pour la recherche temps réel d'usagers qui satisfont à des critères bien déterminés. Mais la décision finale revient à l'utilisateur : c'est lui qui choisit avec qui il va communiquer et de quelle façon, que ce soit en utilisant le système de recherche ou manuellement.

3.2. Réalisation de la base de données

3.2.1. Vue d'ensemble

La base de données de cette application contient les informations de tous les utilisateurs enregistrés ainsi que leurs données et leurs informations de présence. Elle consiste en un fichier XML bien formé et validé pour la structure. Des modifications et des extensions ont été faites dans le modèle PIDF décrit dans le RFC3863 afin de satisfaire les besoins de l'application.

Le document de présence est analysé par le parseur *xerces-c*, un programme en C++ a été mis en place pour gérer ce document. Ce programme analyse au début le document pour fixer les nœuds qui constituent les éléments XML. Pour arriver à faire les modifications nécessaires sur les nœuds, il peut aussi en ajouter et en retrancher. Toutes les modifications restent en mémoire jusqu'à ce qu'elles soient enregistrées dans le document de présence XML principal.

3.2.2. Modèle de l'information de présence

Dans cette partie, on va proposer un modèle de présence adapté au contexte médical. Tout d'abord, on va présenter le modèle standard et par la suite, les extensions proposées à ce modèle. Ceci constitue le modèle de présence qu'on va adopter dans ce projet.

Modèle standard

Dans ce qui suit, on trouve les grandes lignes des informations qui peuvent se trouver dans un document 'application/pidf+xml'. Ce modèle de données est expliqué en détail dans le RFC3863.

- Presentity URL : spécifie l'URL de la presentity.
- Liste des *tuples* de présence :
    - *Identificateur* : marque pour identifier ce *tuple* dans l'information de présence.
    - *Statut* : ouvert/fermé et/ou une extension de valeurs de statut.
    - *Adresse de communication* : moyens de communication et adresse de contact de ce *tuple*. (Optionnelle)

- *Priorité relative* : valeur numérique spécifie la priorité de cette adresse de communication. (Optionnelle)
- *Timestamp* : le temps et la date du changement de ce *tuple*. (Optionnelle)
- *Commentaire lisible* : des notes au sujet de ce *tuple*.

- Commentaire lisible sur la presentity : des commentaires au sujet de la presentity.

Extension proposée

Afin que le service de présence soit adéquat au contexte d'urgence médicale, on propose des nouveaux sous-éléments de l'élément « presence ». Ensuite, ceux du « tuple » qui est à son tour un sous-élément de « presence ». Il est important de savoir que l'élément « presence » concerne le personnel médical « entity » et l'élément « tuple » concerne un terminal appartenant à l'usager en question. Ces extensions qu'on va proposer constituent une contribution originale.

• *Entity* : c'est un attribut obligatoire de l'élément 'presence' qui associe à chaque membre du personnel médical une adresse SIP.

• nom, prénom, adresse, spécialité : ces éléments renferment les informations concernant l'usager en question. Ils sont importants puisqu'ils l'identifient. La spécialité aide le système à choisir un personnel adéquat à faire une tâche définie. Cet élément peut comporter d'autres sous-éléments comme statut (permanent, stagaire, partiel, ...), expérience de travail dans cette spécialité, titre (médecin, infirmier, ambulancier, ...).Des extensions d'élément comportant des informations pertinentes peuvent être ajoutées dans cet élément. Les éléments nom, prénom sont uniques pour chaque personnel et ils sont obligatoires. L'élément spécialité est obligatoire, mais un usager peut avoir plusieurs spécialités, tandis que l'élément adresse est optionnel, c'est-à-dire qu'un usager peut avoir zéro ou plusieurs adresses.

• *PresenceState* : il représente l'état de présence actuel de l'usager. Cet élément est obligatoire et unique, sa valeur est restreinte par une liste qui est définie dans la partie suivante.

• *Contact* : c'est des URI des contacts préférés d'un usager. Ce dernier peut avoir zéro ou plusieurs contacts à la fois.

• *Horaire* : Cet élément renferme l'horaire actuel de chaque usager. Il y a les heures de début et de fin de service. Ces informations peuvent être utilisées par le système de recherche de personnel qualifié.

• *Note* : une note qu'on peut ajouter concernant l'usager.

• *Tuple* : Tous les éléments qui existent dans le Tuple concernent seulement un moyen de communication appartenant à un usager. Le Tuple contient les éléments suivants :

- *Id* : c'est un attribut de l'élément 'Tuple' qui représente l'identificateur unique d'un terminal ou système de communication.
- *Statut* : indique l'état de présence soit du terminal s'il appartient à l'élément «Tuple», soit l'état de présence de l'usager.
- *Localisation* : Il contient les sous éléments suivants : «Lieu» comme maison, bureau, café, ..., « AdresseGéospatiale », c'est les coordonnées données par un GPS ou par triangualisation cellulaire et « AdresseCivique ». Une seule adresse peut suffire pour pouvoir localiser la cible.
- *TypeDuTerminal* : indique la description du terminal en question : «ordinateur, fax, téléphone, PDA, téléphone portable... Il peut contenir une description plus détaillée concernant le terminal : type et résolution de l'écran, s'il est doté d'une webcam, microphone, hauts parleurs... Ces informations peuvent être utiles. Par exemple au cas où un ambulancier veut envoyer un électrocardiogramme, il veut savoir par quel moyen. Si le terminal du destinataire n'est pas muni d'un écran, il peut chercher un autre moyen de communication que le destinataire utilise, comme un fax.

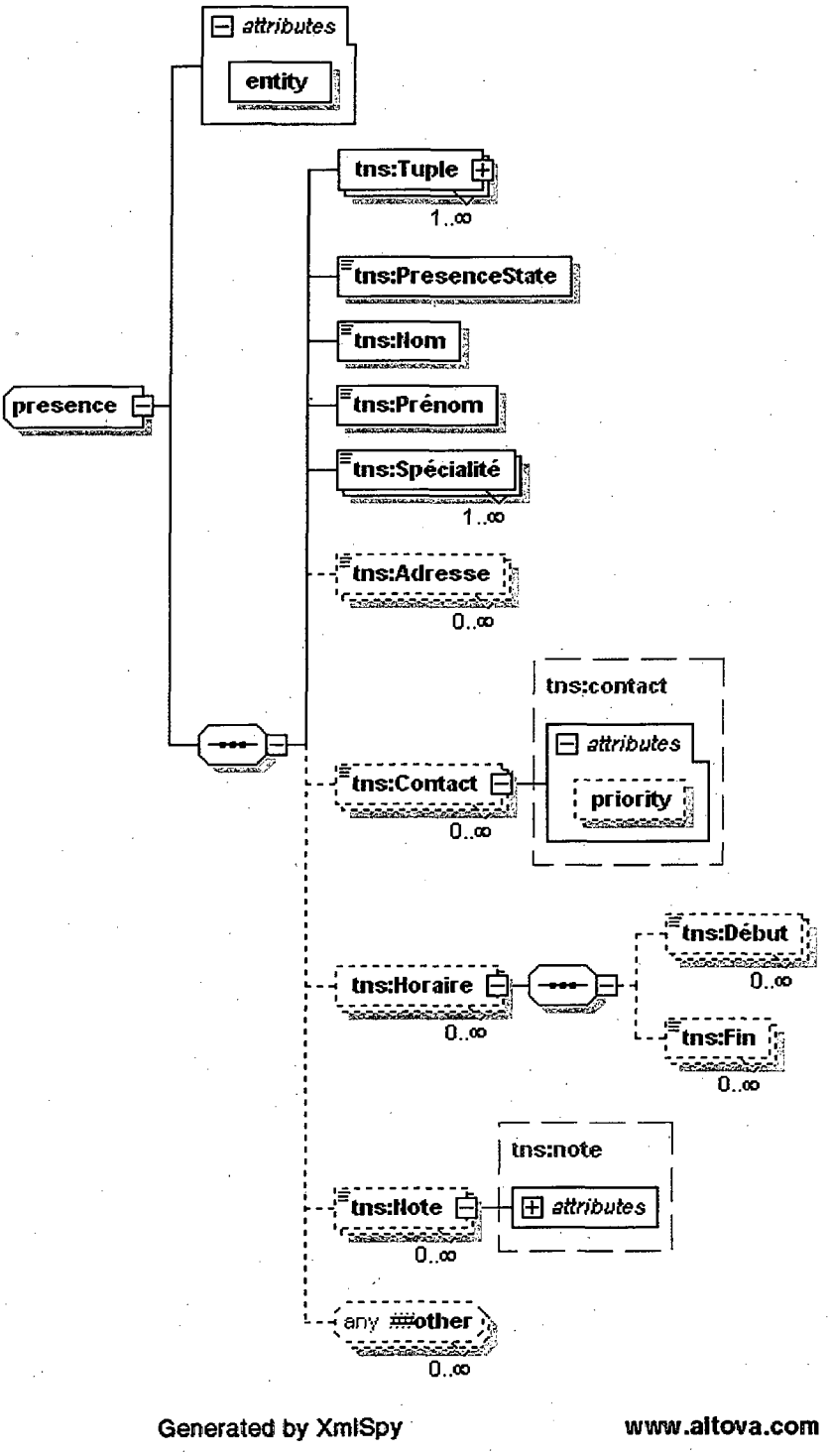


Fig. 12 : Modèle de l'information de présence

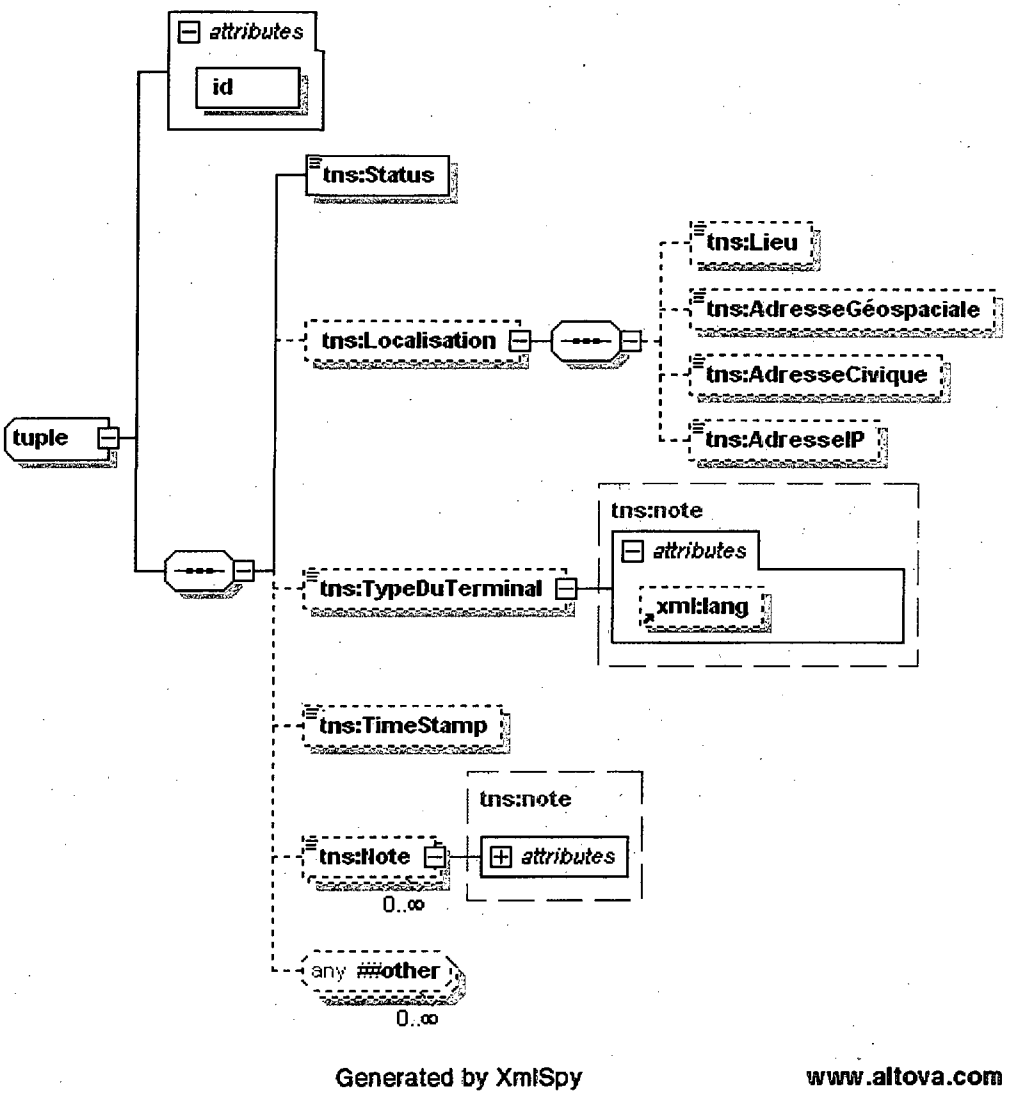


Fig. 13 : Modèle du Tuple de l'information de présence

Les deux figures 12 et 13 présentent le modèle de l'information de présence. Ils sont complémentaires, le deuxième schéma constitue le tuple de présence qui est un élément dans le premier schéma.

3.2.3. Schéma XML

Dans ce qui suit, le schéma XML du modèle de l'information de présence est schématisé dans les figures 12 et 13. Ce schéma XML est généré par le logiciel XmlSpy.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSpy v2006 rel. 3 sp2 (http://www.altova.com) by Jaque
(Norman) -->
<xs:schema xmlns:tns="urn:ietf:params:xml:ns:pidf"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="urn:ietf:params:xml:ns:pidf"
elementFormDefault="qualified" attributeFormDefault="unqualified">
  <!-- This import brings in the XML language attribute xml:lang-->
  <xs:import namespace="http://www.w3.org/XML/1998/namespace"
schemaLocation="http://www.w3.org/2001/xml.xsd" />
  <xs:element name="presence" type="tns:presence" />
  <xs:complexType name="presence">
    <xs:sequence>
      <xs:element name="Tuple" type="tns:tuple" maxOccurs="unbounded" />
      <xs:element name="PresenceState" />
      <xs:element name="Nom" />
      <xs:element name="Prénom" />
      <xs:element name="Spécialité" maxOccurs="unbounded" />
      <xs:element name="Adresse" minOccurs="0" maxOccurs="unbounded" />
      <xs:element name="Contact" type="tns:contact" minOccurs="0"
maxOccurs="unbounded" />
      <xs:element name="Horaire" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Début" type="xs:time" minOccurs="0"
maxOccurs="unbounded" />
            <xs:element name="Fin" type="xs:time" minOccurs="0"
maxOccurs="unbounded" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="Note" type="tns:note" minOccurs="0"
maxOccurs="unbounded" />
      <xs:any namespace="##other" processContents="lax" minOccurs="0"
maxOccurs="unbounded" />
    </xs:sequence>
    <xs:attribute name="entity" type="xs:anyURI" use="required" />
  </xs:complexType>
  <xs:complexType name="tuple">
    <xs:sequence>
      <xs:element name="Status" type="tns:status" />
      <xs:element name="Localisation" minOccurs="0">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="Lieu" type="xs:string" minOccurs="0" />
            <xs:element name="AdresseGéospaciale" minOccurs="0" />
            <xs:element name="AdresseCivique" minOccurs="0" />
            <xs:element name="AdresseIP" type="xs:hexBinary" minOccurs="0"
/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="TypeDuTerminal" type="tns:note" minOccurs="0" />
      <xs:element name="TimeStamp" type="xs:dateTime" minOccurs="0" />
      <xs:element name="Note" type="tns:note" minOccurs="0"
maxOccurs="unbounded" />
```

```
      <xs:any namespace="##other" processContents="lax" minOccurs="0"
maxOccurs="unbounded" />
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID" use="required" />
  </xs:complexType>
  <xs:simpleType name="status">
    <xs:restriction base="xs:string">
      <xs:enumeration value="connecté" />
      <xs:enumeration value="déconnecté" />
      <xs:enumeration value="hors service" />
      <xs:enumeration value="inanimé" />
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="PresenceState">
    <xs:restriction base="xs:string">
      <xs:enumeration value="libre" />
      <xs:enumeration value="occupé" />
      <xs:enumeration value="hors service" />
      <xs:enumeration value="injoignable" />
      <xs:enumeration value="joignable" />
      <xs:enumeration value="en conférence" />
      <xs:enumeration value="sur appel" />
      <xs:enumeration value="de retour" />
    </xs:restriction>
  </xs:simpleType>
  <xs:complexType name="contact">
    <xs:simpleContent>
      <xs:extension base="xs:anyURI">
        <xs:attribute name="priority" type="tns:qvalue" />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
  <xs:complexType name="note">
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute ref="xml:lang" />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
  <xs:simpleType name="qvalue">
    <xs:restriction base="xs:decimal">
      <xs:pattern value="0(.[0-9]{0,3})?" />
      <xs:pattern value="1(.0{0,3})?" />
    </xs:restriction>
  </xs:simpleType>
  <!-- Global Attributes -->
  <xs:attribute name="mustUnderstand" type="xs:boolean" default="0">
    <xs:annotation>
      <xs:documentation>
        This attribute may be used on any element within an optional
        PIDF extension to indicate that the corresponding element must
        be understood by the PIDF processor if the enclosing optional
        element is to be handled.
        </xs:documentation>
    </xs:annotation>
  </xs:attribute>
</xs:schema>
```

Ce schéma XML est décrit en détail dans l'annexe B. Dans le paragraphe suivant on définira les états de présence concernant les terminaux et les usagers.

Les états de présence

L'état de présence concerne deux types d'entités : le terminal et la personne. L'état de présence du terminal se compose des valeurs suivantes :

- connecté : lorsqu'un terminal se connecte au serveur.
- déconnecté : lorsqu'un terminal se déconnecte du serveur
- hors service : lorsqu'un terminal se déconnecte sans préavis
- inanimé : lorsqu'un terminal reste un bout de temps sans être utilisé.

L'état de présence de la personne comprend les valeurs suivantes :

- libre : lorsque l'usager est prêt pour intervenir,
- occupé : lorsque l'usager est occupé par son travail et ne peut intervenir dans une nouvelle situation d'urgence,
- hors service : lorsque l'usager est hors horaire de travail,
- injoignable : aucun terminal appartenant à un usager n'est connecté.
- joignable : au moins un terminal d'un usager est connecté.
- en conférence : l'usager est en réunion mais peut potentiellement intervenir en situation d'ugence.
- sur appel : lorsque l'usager n'est pas au travail, mais il peut intervenir en situation d'urgence.
- de retour : lorsque l'usager sera disponible dans un instant.

L'utilisateur et les terminaux ne gardent qu'un seul état de présence à un moment donné. L'état de présence d'un moyen de communication est fixé par le serveur de présence. Il reçoit l'état de présence du terminal et l'enregistre. En cas de problème technique, il déduit l'état de

présence en inspectant les trames échangées entre lui et le moyen de communication. L'état de présence peut être déduit et établi automatiquement, comme il peut être changé ou fixé par l'utilisateur.

Les états de présence des utilisateurs suivent les règles suivantes :

- Lorsque tous les moyens de communication d'un utilisateur ont soit l'état «déconnecté» soit l'état «hors service», l'état de présence du personnel s'établit automatiquement sur «injoignable». L'utilisateur ne peut pas choisir cet état de présence,
- L'état «joignable» se met automatiquement par le serveur lorsqu'au moins un moyen de communication se connecte. Il reste sur «joignable» jusqu'à ce que l'utilisateur le change ou il se déconnecte,
- L'utilisateur a le droit de choisir entre «libre», «occupé», «en conférence», «sur appel», «de retour»,
- L'état de présence de l'utilisateur peut être déduit à partir de son agenda.

4. IMPLÉMENTATION ET MISE EN PLACE

4.1. Vue d'ensemble

Nous avons réalisé une application prototype afin de concrétiser le système de présence conçu dans ce projet. C'est essentiellement un programme en C++ qui se compose de plusieurs modules. Il y a ceux qui sont essentiels comme le module de gestion de la base de données, le module de transmission de données et le module central de service de présence. Et il y a ceux qui sont complémentaires, comme le module de recherche de personnel qualifié et aussi le module de contrôle d'accès. Ces modules ont été développés dans la partie 3 « Conception de la solution ».

Lors du démarrage, le prototype donne le choix entre le démarrage du système de gestion d'informations de présence, celui d'un serveur de présence ou celui d'un client. Ces trois systèmes peuvent être séparés en trois programmes, et sont implémentés ensemble afin de faciliter l'exécution du programme.

Au début, l'application de présence lit et analyse les fichiers nécessaires pour le bon déroulement du processus. Ils sont décrits comme suit :

- config.txt : Le fichier de configuration de l'application.
- outputPresenceDocument.xml : Le document de présence principal. Il contient l'information de tous les usagers enregistrés. C'est la base de données du serveur de présence.
- user_informations.xml : Utilisé par le client de présence, contenant l'information de l'usager
- PersonnelChoice.xml : Utilisé par le client de présence, le résultat d'une demande de personnel qualifié.
- PresenceSubscription.dll : La DLL du module de communication client/serveur.
- PresenceTest.exe : L'exécutable de l'application du système de présence.

Suivant le système choisi, ces fichiers peuvent comporter le fichier de configuration des paramètres internes du système config.txt, et peuvent comporter aussi le document XML principal du serveur de présence outputPresenceDocument.xml ou le document propre à l'utilisateur user_informations.xml. Après tout ce processus, le système attend les instructions.

Le prototype se compose de plusieurs classes reflétant les modules du système. Les paragraphes suivants expliquent en détail les différents modules, les fichiers de configuration, suivis du diagramme de chaque classe du programme.

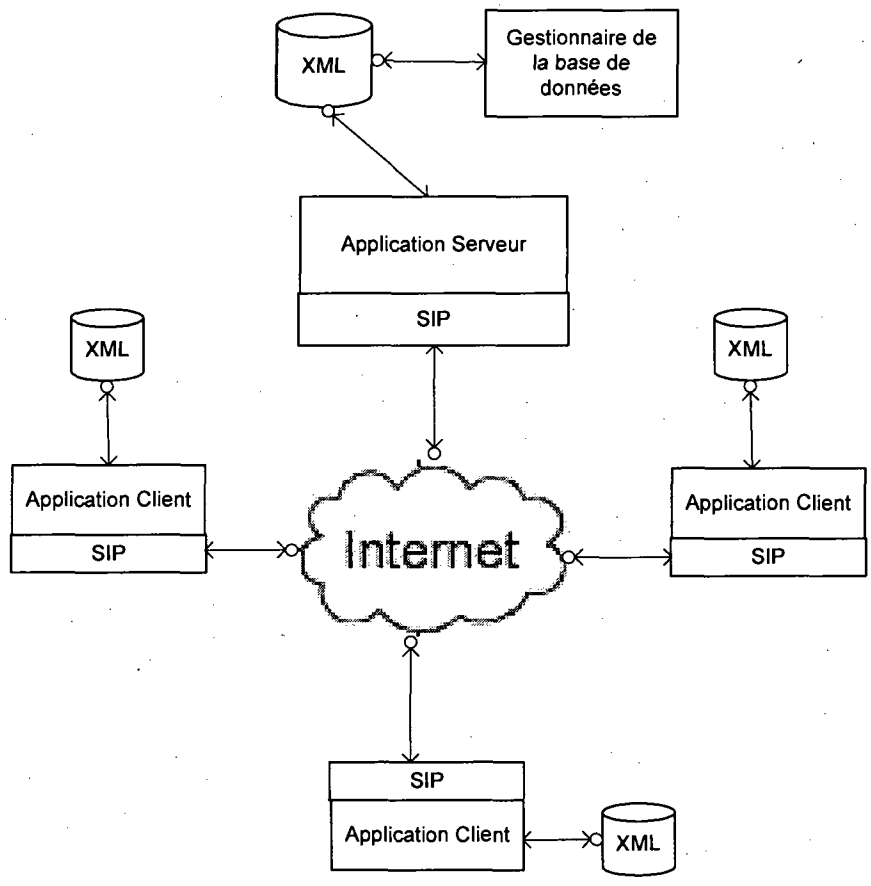


Fig. 14 : Schéma général du prototype

La figure 14 montre les différentes composantes du prototype. Elle montre aussi la manière dont la communication entre ces composantes est établie. Ces composantes sont décrites en détail dans les paragraphes 4.2.2, 4.2.3 et 4.2.4.

4.2. Description détaillée du prototype

Dans ce paragraphe, on va expliquer en détail les modules principaux du prototype, puis la composante de gestion de l'information de présence, ensuite le client de présence, après avec le serveur de présence pour finir avec le code source des classes et interfaces principales.

4.2.1. Les modules principaux

Les modules sont des entités logiques qui sont présentes au cœur du programme principal. Au début on va présenter la pile SIP de M5T, puis le système de détection d'événements pour finir avec le système de recherche de personnel qualifié.

Pile SIP de M5T

M5T SIP SAFE est une pile SIP commerciale basée sur le RFC3261 et implémentée en C++. Cette pile peut être utilisée pour la création des applications, des serveurs et des dispositifs SIP standards, avancés et sécurisés. Elle contient une couche de gestion de transaction et une couche de transport. Elle gère aussi les dialogues. Ses services sont modulaires, et le programmeur choisit dans ce cas le service qu'il veut activer.

La pile SIP se charge du transport de paquets, est responsable de la communication avec les réseaux IP et gère la communication avec les autres points d'accès. Elle gère également les transactions SIP. Plus spécifiquement, elle gère les retransmissions et les *timeouts* des requêtes et des réponses, détecte aussi et gère les retransmissions entrantes.

L'application reçoit des événements au cours de son exécution, Le cœur du système de la pile SIP définit un mécanisme de gestion d'événements, qui à son tour définit les événements reportés à l'application et à leur ordre. L'application interagit avec le cœur du système à travers un ensemble d'interfaces reçues des services utilisés par cette application. Ces interfaces sont implémentées par le programmeur afin de convenir aux besoins de chaque événement, lesquels seront détaillés plus tard.

Les services sont attachés au cœur du système, le programmeur choisit quel service à utiliser dans son programme. L'application peut utiliser ces services lors de l'exécution de différentes actions pouvant influencer le traitement des paquets SIP.

On a réparti la programmation du programme sur différents blocs qui sont expliqués en détail comme suit :

Une DLL permet au début la gestion de la communication client/serveur, puis elle initialise tous les composants de l'agent de l'utilisateur SIP et leurs dépendances. Elle configure les processus du cœur du système et les démarre. Ensuite, elle lit et analyse le fichier de configuration, puis elle ouvre les ports d'écoute afin de recevoir les trames. D'autres paramètres fixés seront expliqués lors de la présentation du code source des classes principales.

Les services utilisés afin de déterminer les fonctions principales du programme sont mentionnés et détaillés dans la partie code source des classes principales.

Dans le paragraphe suivant, on va définir le rôle du système de détection d'évènement. Ce système est situé au cœur du programme de l'application serveur.

Système de détection d'événement

Ce système permet de détecter le changement de l'information de présence de chaque utilisateur dans un serveur de présence, pour ensuite interagir avec ce changement. Il s'agit d'un objet qui, au début de l'exécution, extrait toutes les SIP URI des utilisateurs du document de présence principal et associe à chaque utilisateur un paramètre. À chaque fois qu'il y a un changement dans l'information de présence d'un utilisateur, ce paramètre devient *true*.

Cet objet s'ajoute au document de présence. Dans chaque méthode modifiant l'information de présence dans ledit document, l'objet est appelé pour mentionner ce changement auprès de l'utilisateur concerné par ce changement. Le programme principal intercepte ensuite ce changement et interagit avec lui. L'exemple suivant montre la façon avec laquelle l'objet détectant les événements, est intégré.

```
m_pEventCatcher = new EventCatcher();
m_pDocument->setEventCatcher( m_pEventCatcher );
```

Système de recherche de personnel qualifié

Ce service permet la recherche d'un ensemble d'utilisateurs satisfaisant un ensemble de critères. Les critères pris en considération dans la démo sont le lieu et la spécialité de l'utilisateur.

Au début, les critères considérés sont fixés. Au cas où le système reçoit un critère différent de ceux qui sont fixés, il le rejette. Dans la démo, les critères fixés sont la spécialité, l'état de présence et la localité. Ainsi la ligne de code qui définit les critères mises en considération se présente comme suit :

```
const char * const PersonnelChoice::consideredCriterium[
PersonnelChoice::criteriumNumber] = { "speciality" , "userPresenceState" ,
"locality" };
```

Ce système extrait les critères de recherche fixés par l'utilisateur et lance la procédure de recherche développée dans la partie conception de la solution pour finalement retourner un tableau contenant les utilisateurs recherchés. La taille maximale du tableau est fixée par l'administrateur au moment de la recherche, et est fixée à 5 dans le prototype.

```
choice->setChosenPersonnel(5);
```

4.2.2. Gestionnaire de l'information de présence

Ce gestionnaire est une application qui permet à l'administrateur du système de présence de gérer et de contrôler la base de données. Cette base de données est la même que celle gérée par l'application serveur comme il est illustré dans la fig.14. Ce gestionnaire permet d'enregistrer, retrancher et modifier les usagers, les terminaux ainsi que leurs caractéristiques.

Dans le menu de l'application, trois options se présentent :

Ajout d'un usager

Dans cette fonction, le programme demande à l'utilisateur d'entrer l'adresse SIP du nouvel usager, ensuite il vérifie l'existence de cette adresse. Si elle existe, il annule la procédure, sinon il demande l'ajout des sous-éléments. Le programme prend en considération le nombre permis d'un élément précis pour s'y ajouter. Exemple : l'usager peut avoir zéro ou plusieurs éléments 'contact'. Le programme continue à demander l'élément contact jusqu'à ce que

l'administrateur entre 'n/a' au clavier. Si un élément est obligatoire pour la création d'un usager, ce programme ne cesse d'appeler ce sous-élément jusqu'à ce qu'il soit validé.

Modification d'un usager

Cette fonction permet à l'utilisateur de modifier le contenu d'un élément de présence. Il demande au début l'adresse SIP en question. Si elle n'existe pas, il la demande de nouveau. Une fois trouvée, il parcourt ses sous-éléments en affichant leur valeur actuelle, puis il demande d'entrer la nouvelle valeur. Il donne aussi le choix de supprimer, modifier ou laisser inchangée la valeur existante. L'ajout d'une valeur supplémentaire d'un même sous élément est possible.

Suppression d'un usager

Le programme demande à l'utilisateur l'adresse SIP. Si elle n'existe pas, il essaie de nouveau. Sinon, il demande la confirmation de suppression. Dans ce cas, il enlève l'élément de présence entier correspondant du document de présence.

4.2.3. Client de présence

L'application client constitue le client de présence qui correspond à l'usager concerné. Elle gère ses propres inscriptions, les notifications reçues et les publications de son état de présence. Cette application permet aussi de gérer sa propre base de données qui est un fichier XML. Dans cette section, on va expliquer le démarrage du client, puis les commandes permises au clavier, ensuite l'envoi d'une inscription, après la réception d'un *notify* pour finir avec la demande de personne qualifié.

Démarrage du client

Lors du démarrage du client de présence, le programme lit et analyse le fichier de configuration, s'il trouve une anomalie il avertit l'utilisateur et abandonne l'exécution. Sinon, il initialise les paramètres de connexion puis il ouvre les ports d'écoute qui sont extraits de l'adresse locale dans le fichier de configuration. Le texte suivant montre le fichier de configuration du client de présence config.txt.

```
//Ce fichier de configuration ne tient pas compte des lignes vides, des espaces
//au début d'une ligne et des lignes qui commence par " " (ou espace blanc
```

```
//suivi par " ").
//-------------------------------------------------------------------------

//L'adresse IP locale suivie du port du user agent.
//-------------------------------------------------
local_addr=127.0.0.1:5062

//The peer IP address and port of the user Agent
//----------------------------------------------
peer_addr=127.0.0.1:5060

// La valeur de l'entête via qui va être mise dans toutes les requête envoyées
//par l'application.
//-------------------------------------------------------------------------
local_via=SIP/2.0/UDP 127.0.0.1:5062

// La valeur du champ de l'entête from lors de l'envoi d'une requête
//-----------------------------------------------------------------
from_name-addr=Docteur 2 <sip:docteur2@domain1.com>

// La valeur du champ de l'entête contact lors de l'envoi d'une requête, cette
//adresse sera utilisé pour contacter cet utilisateur.
//-------------------------------------------------------------------------
local_contact=sip:127.0.0.1:5062

// La valeur du champ Request-URI, c'est l'adresse où la requête sera
//acheminée.
//-------------------------------------------------------------------
call_request-uri=sip:127.0.0.1:5060

// La valeur du champ de l'entête to lors de l'envoi d'une requête
//---------------------------------------------------------------
to_name-addr=<sip:presence_server@domain1.com>
```

Après avoir analysé le fichier de configuration, le programme analyse le document de présence XML et construit un arbre. Cet arbre contient la liste des utilisateurs de ce domaine. Il initialise tous les états de présence '*presenceState*' du document de présence sur l'état injoignable.

Puis, il démarre un *thread* qui détecte et met à jour les changements dans l'information de présence. Finalement, le programme attend les nouvelles instructions, par clavier ou par notifications reçues du serveur de présence.

Les commandes permises au clavier

La commande 'e' termine l'inscription en cours, envoie un '*Terminate*' au serveur de présence, puis met à jour le document de présence.

Le commande 's' permet au client de s'inscrire auprès du serveur de présence, pour lui permettre de recevoir l'information de présence d'autres utilisateurs.

La commande 'c' affiche la configuration courante, la figure 15 montre un exemple.

```
Allowed command are:

  e:end subscription
  s:subscribe
  c:current configuration
  r:reset document
  d:save document
  p:send personal request
  q:quit
c
<015> ||||DF0|20|83517870|356|PresenceSubscriptionObject (003E7A68) - ::PrintCur
rentConfiguration ()

Current configuration:
Local Address: 127.0.0.1 : 5062
Peer Address: 127.0.0.1 : 5060
Local Via: SIP/2.0/UDP 127.0.0.1:5062
From name-address: Docuteur 2 <sip:docteur2@domain1.com>
Local contact: sip:127.0.0.1:5062
Call Request-URI: sip:127.0.0.1:5060
To name-address: <sip:presence_server@domain1.com>
Registrar URI: (null)
Address-Of-Record:
Registered contacts:
```

Fig. 15 : Le résultat de la commande 'c' d'un client de présence

La commande 'r' initialise le document de présence, et recharge le document de présence de nouveau, met tous les utilisateurs enregistrés sur l'état injoignable et initialise enfin tous les paramètres.

La commande 'd' enregistre le document de présence se trouvant dans la mémoire dans le fichier de présence s'intitulant : 'outputPresenceDocument.xml'.

La commande 'p' permet au client d'envoyer une demande de spécialistes qualifiés au serveur de présence.

La commande 'q' quitte l'application. Tous les utilisateurs inscrits seront désinscrits, les processus actifs et la pile SIP s'arrêtent.

Envoi d'une inscription

Pour qu'un usager puisse avoir les informations de présence des utilisateurs de son domaine, il doit s'inscrire au serveur de présence. C'est une SIP *subscribe* qui contient les paramètres du fichier de configuration. Ces paramètres permettent d'authentifier le *subscribe*

auprès du serveur de présence. Après l'inscription, le client recevra des *notifys* de la part du serveur de présence.

Envoi d'un publish

Pour qu'un usager se permette de publier sa propre information de présence, il envoie un *publish* au serveur de présence. C'est un message SIP qui contient les paramètres du fichier de configuration de cet usager ainsi qu'un fichier XML contenant son nouvel état de présence. Mais la fonction *publish* ne se trouve pas dans la pile SIP de M5T. Afin de le remplacer, un *subscribe* local est créé pour envoyer un *notify* vers le serveur de présence étant donné qu'il faut associer un *subscribe* pour ce *notify*. Lorsque le serveur de présence intercepte ce *publish*, il le traite comme étant un *notify* non valide puisqu'il n'y a pas de *subscribe* associé à lui.

Réception d'un *notify*

Dès l'interception du paquet *notify* du port d'écoute, le client extrait le fichier XML qui représente la charge utile et l'analyse pour en extraire l'information de présence. En analysant les *entitys* de l'élément *presence*, il s'aperçoit qu'il s'agit d'une notification concernant un ou plusieurs utilisateurs. Puis, il compare les valeurs reçues avec les valeurs se trouvant dans le document de présence du même élément de présence. Le système parcourt le fichier XML reçu et, pour chaque utilisateur, il parcourt le fichier principal de présence pour trouver l'élément de présence de cet utilisateur. Puis, il substitue ses sous-éléments qui ont changé de valeur.

Une *notify* contient un fichier qui a la même structure comme ce qui suit,

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
 <presenceDocument>
 <presence xmlns="http://tempuri.org/presenceDocument.xsd"
  entity="sip:docteur2@domain1.com">
   <tuple id="000001">
         <terminalPresenceState>Connecté</terminalPresenceState>
  </tuple>
  <locality>CHUS</locality>
  <speciality>cardiologue</speciality>
   <userPresenceState>joignable</userPresenceState>
   <login>docteur1</login>
  <password>sipdocteur1</password>
   <contact>sip:docteur3@domain1.com</contact>
```

```
    <contact>sip:docteur9@domain1.com</contact>
</presence>
<presence xmlns="http://tempuri.org/presenceDocument.xsd"
   entity="sip:docteur4@domain1.com">
   <tuple id="000003">
         <terminalPresenceState>Connecté</terminalPresenceState>
  </tuple>
   <locality>CHUS</locality>
   <speciality>cardiologue</speciality>
   <userPresenceState>joignable</userPresenceState>
    <userAvailabilityState>prêt pour intervention</userAvailabilityState>
    <login>docteur1</login>
    <password>sipdocteur1</password>
</presence>
</presenceDocument>
```

Demande de personnel spécifié

Cette fonction se déclenche manuellement par l'intermédiaire du client. Elle choisit l'option 'p' puis le système lui demande d'entrer la spécialité et le lieu du personnel recherché puis envoie la requête au serveur de présence. Ce dernier répond par une liste. Le client intercepte cette notification, l'analyse et s'aperçoit qu'il s'agit de la réponse à sa requête, extrait ensuite la liste et l'enregistre dans un fichier qui s'appelle '*PersonnelChoice.xml*'. Dans ce qui suit on montre un exemple du contenu de ce fichier.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<presenceDocument xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="presenceDocument.xsd">
 <presence entity="sip:personnel_choice@domain1.com">
   <contact>sip:docteur2@domain1.com</contact>
   <contact>sip:docteur4@domain1.com</contact>
   <contact>sip:docteur6@domain1.com</contact>
</presence>
</presenceDocument>
```

4.2.4. Serveur de présence

L'application serveur de présence s'occupe de gérer l'inscription, de la notification des clients ou usager, et de la gestion de la base de données de tous les usagers. Elle intègre le module de détection d'évènements qui lui permet de détecter les nouveaux changements d'état de présence dans le document XML de présence. Elle intègre aussi le module de recherche de personnel qualifié. Ces deux modules n'existent pas dans l'application client.

Démarrage du serveur

Lors du démarrage du serveur de présence, le programme lit et analyse le fichier de configuration. S'il trouve une anomalie, il avertit l'utilisateur et abandonne l'exécution. Sinon, il initialise tous les paramètres puis il ouvre les ports d'écoute qui sont extraits de l'adresse locale dans le fichier de configuration. Le texte suivant montre un exemple du fichier de configuration du serveur de présence config.txt.

```
//Ce fichier de configuration ne tient pas compte des lignes vides, des espaces
//au début d'une ligne et des lignes qui commence par " " (ou espace blanc
//suivi par " ").
//-------------------------------------------------------------------------

//L'adresse IP locale suivie du port du user agent
//-----------------------------------------------
local_addr=132.210.72.179:5060

// La valeur de l'entête via qui va être mise dans toutes les requêtes envoyées
//par l'application.
//-------------------------------------------------------------------------
local_via=SIP/2.0/UDP 132.210.72.179:5060

// La valeur du champ de l'entête to lors de l'envoi d'une requête
//---------------------------------------------------------------
from_name-addr=Server <sip:presence_server@domain1.com>

// La valeur du champ de l'entête contact lors de l'envoi d'une requête, cette
//adresse sera utilisée pour contacter cet utilisateur.
//-------------------------------------------------------------------------
local_contact=sip:132.210.72.179:5060

// La valeur du champ To lors de l'envoi d'une requête.
//-----------------------------------------------
to_name-addr=<sip:somebody@domain1.com>
```

Après avoir analysé le fichier de configuration, le programme analyse le document XML de présence et construit l'arbre d'éléments ou d'usagers correspondant. Il initialise tous les états de présence '*presenceState*' du document de présence sur l'état injoignable. Ensuite, il extrait les adresses SIP de tous les utilisateurs, les met dans une liste puis les initialise sur l'état injoignable. Cette liste lui sert comme un moyen de vérification de l'existence d'un tel utilisateur lors d'une requête entrante. On a associé un paramètre à chaque utilisateur. Ce paramètre indique s'il y a un changement dans l'information de présence propre à lui. Cela est dû au fait qu'il y a un système qui détecte les changements dans chaque élément de présence. Ce paramètre sert ainsi comme moyen de détection de changement de l'information de présence propre à un utilisateur puis il démarre un thread qui détecte et met à jour les changements d'information de présence. Finalement, le programme attend les nouvelles instructions, que ce soit par clavier ou par une commande reçue du port d'écoute, qui

correspond à une requête envoyée par un client. Lorsque le serveur reçoit une telle requête, il en extrait les paramètres nécessaires afin de faire suivre la réponse au client et les enregistre dans le profil de ce client. Ces paramètres sont l'adresse SIP du client *from_name-addr* et l'adresse de réponse *call_request-uri* qui se trouve dans le fichier de configuration du client.

Les commandes permises au clavier

La commande 'e' termine toutes les inscriptions en cours. Elle envoie un '*Terminate*' à tous les utilisateurs inscrits, puis met à jour le document.

La commande 'c' affiche la configuration courante, la figure 16 montre un exemple.

```
<015> ¦¦¦¦15D8¦20¦33777083¦354¦CUAStackController (003EAE0C) - ::InternalEvComma
ndResult (03D40308
<014> ¦¦¦¦15D8¦20¦33777084¦355¦Listening Succeeded!
Les commandes permises:

  e : Terminer les inscriptions
  c : Afficher la configuration courante
  r : Intialiser le document de presence
  d : Enregistrer le document de presence
  v : Afficher la liste des usagers enregistres
  q : Quitter l'application
c
<015> ¦¦¦¦714¦20¦33777698¦356¦PresenceSubscriptionObject (003EAB90) - ::PrintCur
rentConfiguration ()

Current configuration:
Local Address: 127.0.0.1 : 5060
Peer Address: 127.0.0.1 : 5062
Local Via: SIP/2.0/UDP 127.0.0.1:5060
From name-address: Server <sip:presence_server@domain1.com>
Local contact: sip:127.0.0.1:5060
Request-URI: (null)
To name-address:
Registrar URI: (null)
Address-Of-Record:
Registered contacts:
```

Fig. 16 : Le résultat de la commande 'c' d'un serveur de présence

La commande 'r' initialise le document de présence, elle recharge le document de présence de nouveau, met tous les utilisateurs enregistrés sur l'état *unsubscribe* et initialise tous les paramètres.

La commande 'd' enregistre le document de présence se trouvant dans la mémoire du fichier de présence qui s'intitule 'outputPresenceDocument.xml'.

La commande 'v' affiche l'état de présence de tous les usagers enregistrés dans le serveur de présence, montré par la figure 17 :

```
MgrAwaken(0, 6, 03D40A10)
<015> ||||B04|20|32740505|354|CUAStackController (003E7D8C) - ::InternalEvComman
dResult (03D40A10
<014> ||||B04|20|32740505|355|Listening Succeeded!
Les commandes permises:

  e : Terminer les inscriptions
  c : Afficher la configuration courante
  r : Intialiser le document de presence
  d : Enregistrer le document de presence
  v : Afficher la liste des usagers enregistres
  q : Quitter l'application
v
sip:personnel_choice@domain1.com -----> unsubscribed
sip:docteur1@domain1.com -----> unsubscribed
sip:docteur2@domain1.com -----> unsubscribed
sip:docteur3@domain1.com -----> unsubscribed
sip:docteur4@domain1.com -----> unsubscribed
sip:docteur5@domain1.com -----> unsubscribed
sip:docteur6@domain1.com -----> unsubscribed
sip:docteur7@domain1.com -----> unsubscribed
sip:docteur8@domain1.com -----> unsubscribed
sip:docteur9@domain1.com -----> unsubscribed
sip:docteur10@domain1.com -----> unsubscribed
sip:docteur11@domain1.com -----> unsubscribed
sip::docteur12@domain1.com -----> unsubscribed
sip::docteur13@domain1.com -----> unsubscribed
sip:docteur14@domain1.com -----> unsubscribed
Docteur14@domain1.com -----> unsubscribed
fakher -----> unsubscribed
alain@bozo.com -----> unsubscribed
```

Fig. 17 : Le résultat de la commande 'v' d'un serveur de présence

La commande 'q' quitte l'application, tous les utilisateurs inscrits se désinscrivent, le processus du système de détection d'évènement se termine et la pile SIP s'arrête aussi.

Réception d'une inscription

Dès l'interception d'une inscription de A, le serveur de présence consulte la liste des utilisateurs enregistrés, cherche cet utilisateur par son adresse SIP et change son état de présence de injoignable vers l'état joignable. Le système intercepte cet événement puis met à jour le document de présence. Ensuite, il détecte un changement de l'information de présence de l'utilisateur A. À cet effet, il notifie ce dernier de l'information de présence de tous les autres utilisateurs déjà inscrits : il envoie dans le message de notification un document XML contenant l'information de présence de tous les utilisateurs inscrits. Ensuite, le serveur de

présence notifie tous les autres utilisateurs de la nouvelle information de présence de l'utilisateur A. L'opération de notification sera expliquée ultérieurement.

Au cas où le serveur de présence ne trouve pas le client A enregistré dans la liste, le module de communication SIP se charge de l'envoi d'un message d'échec.

Réception d'un *publish*

Lorsqu'un serveur reçoit l'information de présence publiée par un client A, il extrait le document XML du paquet reçu, qui correspond à l'élément de présence de A. Il analyse le document pour en extraire l'adresse SIP du client, puis il cherche l'élément de présence correspondant à cette adresse dans le document principal. Il compare les sous-éléments de l'élément de présence reçu, avec ceux de l'élément de présence du document de présence. Pour chaque sous-élément, au cas ou sa valeur change, le système la remplace avec la valeur reçue.

En conséquence, le système détecte le changement de l'élément de présence, puis il notifie les utilisateurs inscrits du changement. L'opération de notification est expliquée ultérieurement.

Réception d'une demande de personnel spécifié

Lorsqu'un serveur de présence reçoit une demande de personnel qualifié, il analyse le fichier XML, puis il consulte la valeur de l'attribut *entity* de l'élément *presence*, s'aperçoit donc que cette requête est adressée à l'entité sip:personnel_choice@domain1.com qui est le système de recherche de personnels qualifiés. Il extrait ensuite les informations pertinentes relatives à sa recherche, qui sont la spécialité et la localité. Puis il envoie la requête au service de recherche de personnels qualifiés. Ce dernier lui fournit la liste du personnel trouvé dans un document XML. Ensuite, le serveur de présence envoie ce fichier XML au client concerné sous forme de *notify*. Voici un exemple de fichier XML reçu du client :

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<presenceDocument xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:noNamespaceSchemaLocation="presenceDocument.xsd">
  <presence entity="sip:personnel_choice@domain1.com">
      <speciality>cardiologue</speciality>
      <locality>CHUS</locality>
```

```
  </presence>
</presenceDocument>
```

Opération de notification

Un *notify* est un paquet SIP contenant un document XML ayant des éléments de présence concernant des utilisateurs spécifiques. L'opération de notification se déclenche au moment où le serveur envoie l'information de présence à un client. Ceci peut se produire dans trois cas :

- Lorsqu'il y a un changement dans l'information de présence d'un ou plusieurs utilisateurs, le serveur de présence doit notifier les autres utilisateurs inscrit.
- Lorsqu'un client s'inscrit, le serveur de présence doit notifier ce client de tous les utilisateurs inscris.
- Lorsqu'un client envoie une demande de personnels qualifiés, le serveur doit notifier ce client en lui envoyant la liste de spécialistes trouvée.

4.3. Diagramme de classe de l'application

La figure 18 montre le diagramme de classe de toute l'application du système de présence. Le cadre en bas désigne la partie de la pile SIP de M5T qu'on l'a adapté et implémenté selon nos besoins. Et le cadre en haut désigne la partie gestion de l'information de présence qu'on a implémentée en totalité. On a décrit chacune des classe qui figurent dans ce diagramme dans l'annexe C.

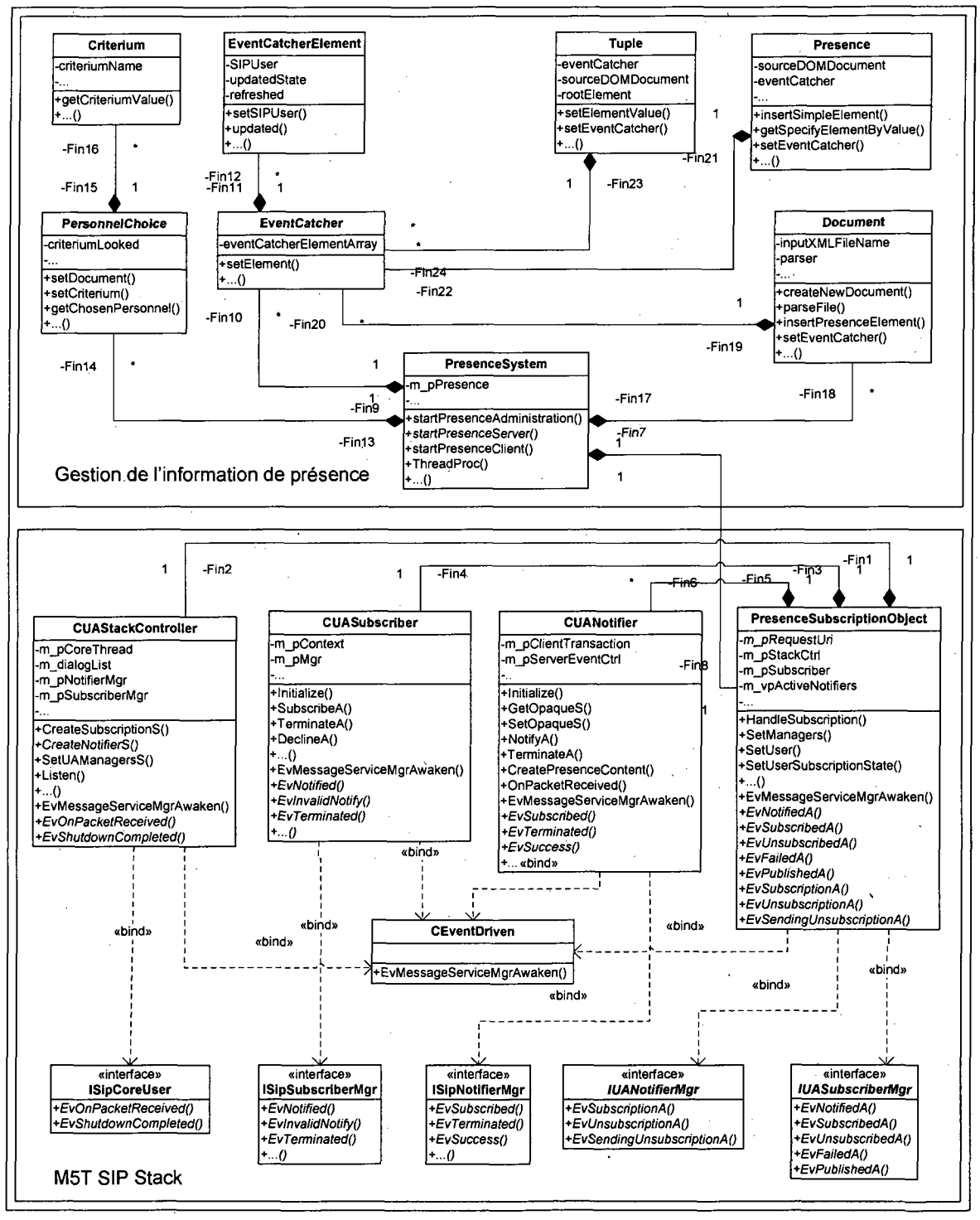


Fig. 18 : Diagramme de classe

Dans la partie 4 on a présenté l'implémentation et la mise en place de l'application du système de présence. On a aussi fait une description détaillée du programme de l'application tout en passant par la définition de ses modules principaux, la façon de gérer l'information dans la base de données ainsi la définition des différentes composantes de l'application tout en clarifiant la communication eux. Finalement on a défini le diagramme de classe de l'application.

Dans la partie suivante, on va étudier la possibilité d'intégration du service de présence à une plateforme multimédia médicale qu'on va définir.

5. TEST ET ÉVALUATION DU PROTOTYPE

Dans ce paragraphe, on va tester le fonctionnement de notre prototype. On va commencer par la configuration du test, puis la séquence subscribe/notify typique, ensuite la publication, on va tester après la recherche de spécialiste, puis la désinscription pour finir avec l'étude des paquets échangés.

5.1. Configuration du test :

Afin d'exécuter et d'appliquer le programme convenablement, on doit mettre tous les fichiers responsables du bon déroulement de l'exécution dans un même répertoire. Ces fichiers sont config.txt, outputPresenceDocument.xml, user_informations.xml, PresenceSubscription.dll et PresenceTest.exe. Ils sont expliqués dans la partie 4.

Le test du prototype est effectué avec 4 clients et un serveur de présence, l'adresse SIP du serveur est sip:presence_server@domain1.com, l'adresse SIP du système de recherche de personnels qualifiés est sip:personnel_choice@domain1.com et l'adresse SIP des clients sont comme suit : sip:docteur2@domain1.com, sip:docteur4@domain1.com, sip:docteur5@domain1.com et sip:docteur6@domain1.com.

5.2. Séquence subscribe/notify typique

Au début, on a démarré le serveur de présence et le premier client docteur2, voici une description du déroulement de cette partie du test

- Envoi d'un *subscribe* à l'aide de la commande 's' du docteur2,
- Interception du *subscribe* par le serveur
- Changement de l'état de présence du docteur2 dans le document de présence outputPresenceDocument.xml de l'état injoignable vers l'état joignable.

Et lorsqu'on a démarré le client docteur4,

- Envoi d'un *subscribe* au serveur de présence,

- Interception du *subscribe*, par le serveur de présence et report du nouvel état de présence du docteur4 au document de présence,

- Envoi d'un *notify* au docteur2 par le serveur de présence du nouvel état de présence du docteur4 et rafraichissement par un *notify* du docteur4 de l'état de présence du docteur2,

- Interception des *notifys* par le docteur2 et le docteur4 et mise à jour des documents de présence relatifs à ces docteurs.

Après, lors du démarrage du client docteur5,

- Envoi d'un *subscribe* au serveur de présence,

- Changement dans le document de présence du serveur de présence de l'état de présence du docteur5,

- Rafraichissement de l'état de présence du docteur 5 par l'envoi d'un *notify* du serveur au docteur2 et docteur4, rafraichissement aussi du docteur5 avec un seul *notify* de l'état de présence du docteur2 et du docteur4,

- Interception des *notifys* de tous les clients notifiés et report des modifications nécessaires aux documents de présence relatifs à ces clients.

Le même scénario s'est répété au docteur6, tout s'est déroulé sans fautes. On n'a enregistré aucune erreur.

5.3. Test de publication

Maintenant, tous les clients sont inscrits au serveur de présence, ils ont l'état de présence 'joignable'. Pour changer l'état de présence du docteur2 vers l'état 'libre', on a utilisé la commande 'n', voici une description du déroulement de cette partie du test

- Le programme demande le nouvel état de présence, on a entré 'libre',

- Envoi d'un *publish* du docteur2 vers le serveur de présence,

- Mise à jour du document de présence principal de l'état de présence du docteur2 vers 'libre',

- envoi d'un *notify* du nouvel état de présence du docteur2 vers tous les autres clients inscris. Mise à jour de leur document de présence.

Le même scénario est fait au docteur4 et au docteur5, tout s'est déroulé sans fautes.

5.4. Recherche de personnel qualifié ou de spécialiste

On va expliquer maintenant comment une demande de spécialiste est envoyée. Pour cela, on suppose que le docteur6 procède à cette demande.

Le système demande d'entrer la localité = CHUS et la spécialité = cardiologue, un *publish* vers le serveur de présence qui contient comme destinataire *To* = sip:personnel_choice@domain1.com est envoyé. Le système entame sa recherche dans le document de présence avec les paramètres suivants : localité = CHUS, spécialité = cardiologue et état de présence = libre. La réponse était docteur2 et docteur4, cette réponse est conforme aux règles de recherche.

On a essayé cette demande sur les autres clients, on a changé aussi l'état de présence des clients et les paramètres de recherche, la réponse du service de recherche de personnel qualifié était correcte.

5.5. Test de désinscription

Finalement on a désinscrit les clients et à chaque fois qu'un client se désinscrit, les autres clients inscrits se notifient et leur document de présence est mis à jour. Puis on a arrêté l'application des clients et du serveur, tous les modules sont éteints correctement et l'application s'est finalement arrêtée.

5.6. Étude des paquets échangés

On a aussi fait le suivi des paquets SIP échangés entre les terminaux avec le logiciel *Ethereal*. Ces paquets ont été conformes aux normes du RFC3261 qui définit le protocole SIP, à l'exception du *publish* vue qu'il n'est pas implémenté dans la pile SIP de M5T. Donc afin de

le remplacer, un *subscribe* local est créé pour envoyer un *notify* vers le serveur de présence. Lorsque le serveur de présence intercepte ce *publish*, il le traite comme étant un *notify* non valide par la méthode `CUASubscriber::EvInvalidNotify`.

Maintenant, on va étudier l'échange des paquets durant le fonctionnement du système de présence selon le nombre de clients inscrits. Pour cela, on va dresser un tableau qui indique le nombre de paquets échangés lorsqu'un usager envoi une demande d'inscription.

Clients inscrits	Paquets échangés
5	12
10	22
15	32
20	42
30	62
50	102

Supposant qu'un usager A envoie une demande d'inscription au serveur de présence, alors le serveur de présence consulte sa base de donnée et trouve par exemple 10 usagers inscrits. Il envoie donc 10 *notify* à ces usagers du nouvel état de présence de A et un *notify* à A de l'information de présence de tous les usagers inscrits.

5.7. Pistes de recherche identifiées

Suite a ce constat, plusieurs piste de recherche ont été identifiées, il s'agit de :

- L'interface graphique du prototype devrait être entièrement revue afin d'augmenter la facilité d'utilisation du logiciel, ce qui aura pour effet de simplifier grandement le logiciel.

- Système de contrôle d'accès : établir les règles d'accès aux différentes composantes, assurer la confidentialité et l'intégrité de l'information et l'authentification du personnel. Et assigner aussi des certificats personnels pour le personnel médical pour crédibiliser les transferts et assurer enfin la non répudiation de l'information.

- Étendre l'information de présence et le système de recherche aux locaux, aux terminaux, aux équipements médicaux.

- Système de contrôle interne de cohérence du document de présence
- Augmentation de l'efficacité du système de recherche de professionnels spécifié, ajout de paramètres, augmentation de la complexité de la méthode de recherche, étendre la recherche aux ressources médicales et les locaux.
- Renforcer la sécurité des transferts, chercher un protocole sécurisé de transmission comme le SIPS (Secure SIP) (Rosenberg et al., 2002).
- Élargir la zone de fonctionnement du système de présence sur plusieurs centres hospitaliers et établir un système de coopération entre eux.

6. INTÉGRATION DU SERVICE DE PRÉSENCE À UNE PLATEFORME MULTIMÉDIA

Une plateforme de communication multimédia sur IP pour le milieu médical est en cours de développement par notre groupe de recherche. Une première partie du projet est développée par Alexis DORAIS-JONCAS dans le cadre de son mémoire de maîtrise intitulé «Un mécanisme de conférence basé sur le protocole SIP pour la transmission simultanée de voix et de données médicales» (Dorais-Joncas, 2007). Ce projet consiste en un système de conférence multimédia pleinement maillé. Cela veut dire que chaque point de la conférence envoie de la voix à tous les autres points sans aucun intermédiaire.

Notre système de présence vient éclaircir le travail du système de conférence afin de le rendre plus efficace. Le système de présence vient assurer l'état de présence de chaque usager auprès des usagers eux-mêmes. De cette façon, chaque usager peut savoir la disponibilité des autres usagers avant de communiquer avec eux. Aussi, il peut faire une recherche de spécialistes disponibles en spécifiant la spécialité et la localité désirées.

Contrairement au système de communication, le système de présence est centralisé. Cette architecture est inévitable car il doit absolument avoir un serveur de présence centralisé qui gère tous les usagers simultanément. Cette architecture est essentielle pour assurer une disponibilité optimale de l'information de présence entre les différents membres du corps médical. Cependant le système de communication conserve son aspect pleinement maillé, et une fois une communication est débuté, elle se déroule sans aucune dépendance du serveur de présence.

L'intégration du système de présence constitue un complément primordial au système de communication dans une situation d'urgence médicale. Le mariage entre ces deux systèmes se caractérise par sa souplesse et son efficacité, ils peuvent fonctionner indépendamment l'un de l'autre. En cas de trouble du serveur de présence, le système de communication reste quand même fonctionnel. Chaque usager peut conserver une liste d'adresse de ses collègues. Il peut communiquer avec celui qu'il veut sans aucune intervention du serveur de présence. Mais il ne profite plus des services du serveur de présence (informations de présence, liste complète

du personnel, recherche de spécialiste). Ceci engendre la perte de temps qui n'est pas du tout appréciée dans une situation d'urgence médicale.

Pour mieux expliquer le fonctionnement des deux systèmes ensemble, on va procéder à une mise en situation. Dans une situation d'urgence médicale, un ambulancier qui se trouve dans son ambulance et qui est déjà connecté avec l'infirmier qui est tout près du patient, veulent communiquer avec un cardiologue. Pendant que l'infirmier essaye de stabiliser le cas du patient, l'ambulancier envoie une requête de recherche de cardiologue auprès du système de présence. Au moment où l'ambulancier reçoit la réponse, il demande au cardiologue trouvé 'C' de joindre la conversation entre lui et l'infirmier. Ceci permet à l'infirmier d'avoir de bonnes directives et à l'ambulancier d'anticiper les actions à entreprendre. La figure 19 illustre la mise en situation citée dans ce paragraphe.

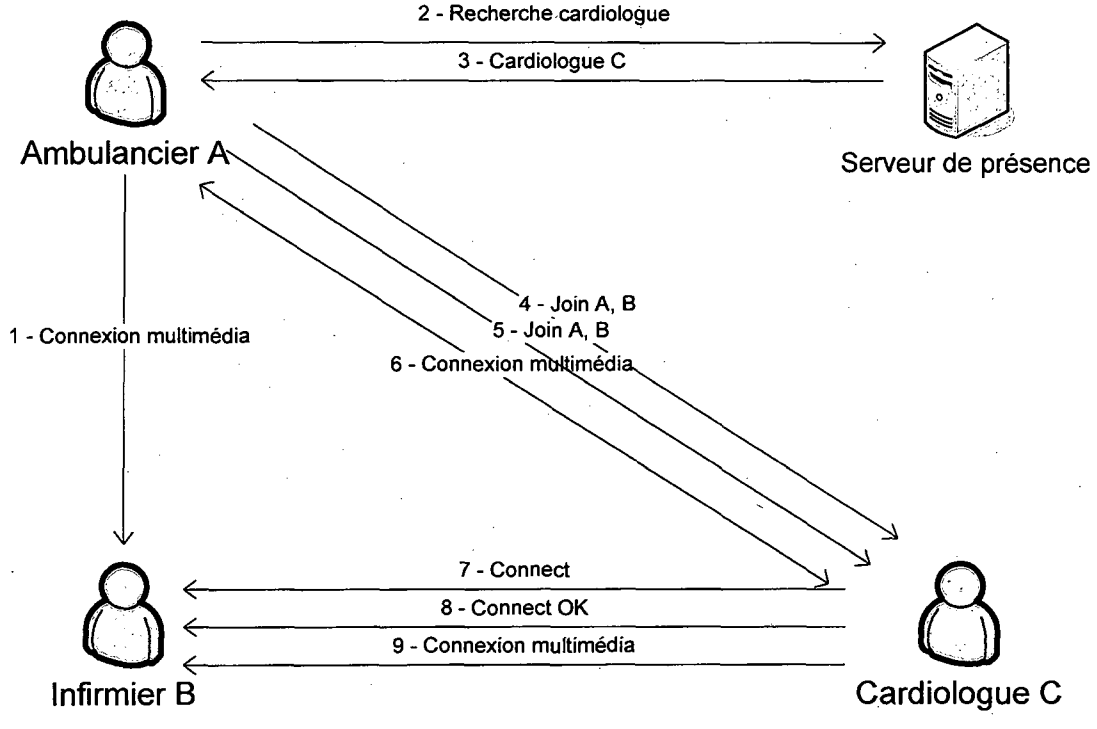


Fig. 19 : Intégration du système de présence à une plateforme multimédia

7. CONCLUSION

Ce projet de recherche vise globalement à contribuer à l'avancement de la technologie des systèmes de présence en contexte médicale. La problématique générale que nous avons abordée est celle de pouvoir joindre, à tout moment, une ressource du milieu médical possédant certaines caractéristiques de spécialité et de disponibilité. Effectivement, le système de présence que nous avons conçu répond parfaitement à l'objectif général fixé. Cet objectif a été atteint après une méticuleuse recherche des technologies de pointe existantes.

Dans cette conclusion, on va évaluer les objectifs spécifiques visés de notre projet de recherche. Puis on va proposer des sujets en relation directe avec notre projet. Finalement on va conclure cette mémoire.

Les objectifs spécifiques fixés de notre projet ont été atteints, en effet :

- On a défini un modèle d'informations de présence spécifique au contexte médical.
- On a défini un modèle abstrait de l'architecture de notre service de présence, incluant la définition des entités présentes dans cette architecture;
- On a défini les opérations que le service de présence proposé doit inclure pour être adéquatement fonctionnel. On note le système de recherche de personnel qualifié qui constitue une innovation très utile dans le contexte de communication d'urgence médicale.
- On a proposé une implémentation du service de présence à l'aide des technologies XML et SIP;
- On a développé un prototype du service de présence pour en faire la preuve de concept;
- On a discuté de l'intégration du service de présence à une plateforme de communication multimédia sur IP vouée au contexte médical;

- On a commenté certaines implications pratiques comme, par exemple, les modes de mise à jour des informations de présence et la sécurité informatique.

Tout au long de ce projet, plusieurs aspects intéressants ont émergé des travaux réalisés mais qui n'ont pu être approfondis à souhait. Les sujets offrent un potentiel de recherche très intéressant à l'avenir, nous citons principalement :

- Système de réservation de personnel et de ressource médicale : après avoir étendu l'information de présence aux locaux et aux ressources médicales, ce système peut être réalisé. Exemple, Imaginons un ambulancier en situation d'urgence veut transporter un patient vers un centre hospitalier pour intervention chirurgicale, il peut organiser cette opération par un simple clic. L'ambulancier sait toute l'information pertinente à cette opération comme le lieu, le local, la période, le matériel médical, le personnel,...

- Système de surveillance de patient à distance: l'information de présence peut contenir l'état de santé d'un patient en temps réel. Le patient peut être n'importe où cependant l'information concernant son état de santé est transmise au serveur de présence. S'il y a une anomalie quelconque, son médecin sera averti. Ce système peut rappeler le médecin de faire une certaine intervention pour ce patient à distance. Il peut également suivre son état de santé en permanence sans se déplacer. Tout en étant connecté au réseau sans fil, le système peut aussi capturer sa position géographique.

- Coupler le système de présence à un système de localisation RFID ou GPS pour faciliter la gestion du matériel (E.G. pompes à perfusion) ou assurer une certaine sécurité des patients dans un établissement hospitalier.

L'adoption d'un mécanisme sophistiqué de communication tel que celui présenté dans ce cas permettrait d'améliorer la qualité des soins pré-hospitaliers en situation d'urgence et ce, sans investissements majeurs. En traitant plus rapidement et de manière plus appropriée les patients, le taux de décès et de complications est appelé vers la diminution. De plus, une

économie au niveau des coûts de traitements pourrait être réalisée puisque la dégradation de l'état de nombreux patients serait ainsi évitée.

Dans le contexte démographique actuel du Québec, de telles technologies permettant d'améliorer l'efficacité du corps médical devront être introduites et utilisées dans le système de santé québécois afin de mieux répondre aux besoins d'une population vieillissante.

8. BIBLIOGRAPHIE

Alvestrand, H.T. (2001). RFC3066, Tags for the Identification of Languages. *Internet Engineering task Force (IETF).*

Arabshian, K. et Schulzrinne, H. (2003a). A Generic Event Notification System Using XML and SIP. *Columbia University,New York, USA.*

Arabshian, K. et Schulzrinne, H. (2003b). A SIP-based medical event monitoring system. *In* Anonyme, *5th International Workshop on Enterprise Networking and Computing in Healthcare Industry* (p. 66-70), Proceedings, 6-7 June 2003. Piscataway, NJ, USA, IEEE.

Arunachalan, B., Light, J. et Watson, I. (2007). Mobile Agent Based Messaging Mechanism for Emergency Medical Data Transmission Over Cellular Networks. *In* Anonyme, *Communication Systems Software and Middleware, 2007. COMSWARE 2007. 2nd International Conference on* (p. 1-6), Bangalore, India.

Chiang, C., Lin, P., Huang, C., Chou, Y., Chu, W., Chan, C. et Wang, P. (2008). Pervasive Communications for Care Professionals in Hospitals. *In* Anonyme, *Bioinformatics and Biomedical Engineering, 2008. ICBBE 2008. The 2nd International Conference on* (p. 1351-1354), 16-18 May 2008. Shanghai, China.

Day, M., Rosenberg, J. et Sugano, H. (2000a). RFC2778, A Model for Presence and Instant Messaging. *Internet Engineering task Force (IETF).*

Day, S. Aggarwal, G. Mohr, J. Vincent, M. (2000b). RFC 2779, Instant Messaging / Presence Protocol Requirements. *Internet Engineering task Force (IETF).*

Définition de parseur XML. *In* Anonyme . [En ligne]. http://www.alaide.com/dico.php?q=parseur+XML&ix=5238 (Juin 2006).

Dorais-Joncas, A. (2007). Un mécanisme de conférence basé sur le protocole SIP pour la transmission simultanée de voix et de données médicales. *Mémoire de Maîtrise.* Sherbrooke, QC, Canada (113 pages).

Extensible Markup Language (XML). *In* Anonyme . [En ligne]. http://www.w3.org/XML/ (2008).

Hsieh, S.L., Weng, Y.C., Hsieh, S.H., Lai, F.P., Hsu, K.P. et Tsai, W.N. (2006). Wireless/Wired Collaborative Remote Consultation Emergency Healthcare Information Systems Framework. *In* Anonyme, *TENCON 2006. 2006 IEEE Region 10 Conference* (p. 1-4), Hong Kong, China.

Huh, M.Y., Hyun, W. et Kang, S.G. (2007). Design Considerations on Subscription and Notification Function in the Presence Services for Hierarchical ResourceList. *In*

Anonyme, *The 9th International Conference on Advanced Communication Technology.*(p. 949-954), Phoenix Park, Korea.

Instant Messaging and Presence Protocol (impp). *In* Anonyme . [En ligne]. http://www.ietf.org/html.charters/OLD/impp-charter.html (2006).

Internet SocietyInternet Engineering Task Force (IETF). *In* Anonyme . [En ligne]. http://www.ietf.org/ (2006)

Jiang, D., Yeap, T., Liscano, R. et Logrippo, L. (2005b). Two Approaches for advanced presence services in SIP communications. *In* Anonyme, *Jointly held with the 2005 IEEE 7th Malaysia International Conference on Communication and 2005 13th IEEE International Conference on Networks* (p. 6 pp.), Kuala Lumpur, MALAYSIA, 16-18 nov 2005.

Jiang, D., Yeap, T., Logrippo, L. et Liscano, R. (2005a). Personalization for SIP multimedia communications with presence. *In* Anonyme, *Services Systems and Services Management, 2005. Proceedings of ICSSSM '05.*(p. 1365-1368).

Jin, Z., Jin, H. et Han, L. (2004). Instant messaging and presence services using SIMPLE. *In* Anonyme, *TENCON 2004. 2004 IEEE Region 10 Conference* (p. 157-159 Vol. 3), Chiang Mai, Thailand.

Kim, B., Kim, Y., Lee, I. et You, I. (2007). Design and Implementation of a Ubiquitous ECG Monitoring System Using SIP and the Zigbee Network. *In* Anonyme, *Future generation communication and networking (fgcn 2007)* (p. 599-604), Jeju, 6-8 dec 2008.

Laberge-Nadeau, C, Messier, M et Huot , I (1998). GUIDE DES SERVICES OFFERTS AUX BLESSÉS DE LA ROUTE, AU QUÉBEC. *In* Anonyme . [En ligne]. http ://www.crt.umontreal.ca/guide-blesses-route/reg2011.pdf (2007).

Lakas, A. et Shuaib, K. (2005). A framework for SIP-based wireless medical applications. *In* Anonyme, *Vehicular Technology Conference, 2005. VTC 2005-Spring. 2005 IEEE 61st* (p. 2755-2759), Stockholm, Suède.

Lucenius, J. (2008). Use of Presence and Location Information for Situational Awareness. *In* Anonyme, *ICDT '08. The Third International Conference on Digital Telecommunications, 2008.*(p. 117-125), Internet Engineering Task Force (IEEE).

Murata M., St.Laurent, S. et Kohn D (2001). RFC3023, XML Media Types. *Internet Engineering task Force (IETF).*

Navarro, E.A.V., Mas, J.R. et Navajas, J.F. (2006). Analysis and Measurement of a Wireless Telemedicine System. *In* Anonyme, *Pervasive Health Conference and*

Workshops, 2006 (p. 1-6), Internet Engineering Task Force IETF, Innsbruck, Austria.

Newman, C. et Klyne, G. (2002). RFC3339, Date and Time on the Internet: Timestamps. *Internet Engineering task Force (IETF).*

Pandian, P.S., Safeer, K.P., Shakunthala, D.T., Gopal, P. et Padaki, V.C. (2007). Internet Protocol Based Store and Forward Wireless Telemedicine System for VSAT and Wireless Local Area Network. *In* Anonyme, *ICSCN '07. International Conference on Signal Processing, Communications and Networking, 2007.*(p. 54-58), Chennai, India.

Peternel, K., Zebec, L. et Kos, A. (2008). Using presence information for an effective collaboration. *In* Anonyme, *Communication Systems, Networks and Digital Signal Processing, 2008. CNSDSP 2008. 6th International Symposium on* (p. 119-123),Graz, Austria.

Ramsdell, B. (2004). RFC3851, Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.1 Message Specification. *Internet Engineering Task Force (IETF).*

Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M. et Schooler, E. (2002). RFC3261, SIP : Session Initiation Protocol. *Internet Engineering task Force (IETF).*

SIP for Instant Messaging and Presence Leveraging Extensions (simple). *In* Anonyme . [En ligne]. http://www.ietf.org/html.charters/simple-charter.html (2006).

Sugano, H, Fujimoto, S, Klyne, G, Bateman, A et Carr, W (2004). RFC3863, Presence Information Data Format (PIDF). *Internet Engineering task Force (IETF).*

Tulu, B., Chatterjee, S., Abhichandani, T. et Li, H. (2003). Secured video conferencing desktop client for telemedicine. *In* Anonyme, *5th International Workshop on Enterprise Networking and Computing in Healthcare Industry* (p. 61-5), Proceedings, 6-7 June 2003. IEEE, Piscataway, NJ, USA,.

(Universal Ressource Name) urn:ietf:params:xml:schema:pidf:status:rpid. In Anonyme . [En ligne]. http://www.iana.org/assignments/xml-registry/schema/pidf/status/rpid.xsd (2006).

Viruete Navarro, E.A., Ruiz Mas, J., Femandez Navajas, J. et Pena Alcega, C. (2005). Enhanced 3G-Based m-Health System. *In* Anonyme, *Computer as a Tool, 2005. EUROCON 2005.*(p. 1332-1335), Glasgow, Scotland.

Wu, F. (2007). Presence Technology with Its Security and Privacy Implications. *In* Anonyme, *ISCE 2007. IEEE International Symposium on Consumer Electronics, 2007.*(p. 1-6), Texas, USA.

Xerces-c. *In* Anonyme . [En ligne]. http://xerces.apache.org/xerces-c/ (2006).

Zou, Y., Istepanian, R.S.H. et Bain, S.C. (2004). Policy driven mobile agents for ubiquitous medical diagnosis assistant system. *In* Anonyme, *IDEAS '04-DH. Proceedings. IDEAS Workshop on Medical Information Systems: The Digital Hospital, 2004.*(p. 3-7), Beijing, China.

Annexe A

Dans cette annexe, on va définir plus profondément les principales technologies impliquées dans ce projet. On va commencer par introduire XML puis définir le protocole SIP, ensuite on va donner une notion générale sur la base de données, finalement on va définir le parseur XML.

1. Introduction sur XML

Dans cette section, on va définir XML ainsi que ses principaux aspects qu'on a utilisé dans ce projet. On va définir ainsi les caractéristiques de XML, ensuite les syntaxes et terminologies, après la structure du document XML, puis les espaces nom «*name spaces*» pour finir avec la validation de données.

1.1. Caractéristiques de XML

XML signifie langage de balise extensible «eXtensible Markup Langage», et il est utilisé pour décrire les documents et les données dans un format texte normalisé qui peut être facilement transporté via des Protocoles internet standards. XML est basé sur la source de tous les langages balistiques «markup», qui est le SGML «Standard Generalized Markup Language».

1.1.1. Extensibilité

XML joue un rôle important dans la description des données structurées comme texte; le format est extensible. C'est-à-dire que n'importe quelle donnée qui peut être décrite comme texte et imbriqué dans des étiquettes XML «XML tags» va être généralement acceptée comme XML. Les seules limites imposées sur les données sont celles par les données eux-mêmes, à travers des règles de syntaxe et des directives de format imposé par l'utilisateur lors de la validation de données.

1.1.2. Structure

La structure de XML est habituellement complexe et difficile de suivre par l'œil nu, mais il est important de rappeler que XML bien que sommairement lisible, n'est pas conçu pour

qu'on le lise. Les parseurs XML et d'autres types d'outils qui sont conçu pour travailler avec XML assimilent facilement XML, même dans ses formes les plus complexes. Aussi, XML est désigné pour être un format d'échange de donnée ouvert.

1.1.3. Validité

Hormis les exigences obligatoires de syntaxe qui composent un document XML, les données représentées par XML peuvent optionnellement être validées pour la structure et le contenu, basées sur deux standards de validation de donnée séparés. Le standard original de validation de donnée est appelé définition de type de donnée «DTD : Data Type Definition», et l'évolution la plus récente de standard de la validation de donnée est le schéma XML «XML Schema».

1.2. Syntaxes et terminologies

Les documents XML bien formés sont des documents XML qui satisfont aux recommandations de configuration de document XML de W3C. Ces recommandations, qui peuvent être trouvées à www.w3.org/xml, sont très strictes en ce qui concerne les exigences de format qui font la différence entre un document texte contenant un ensemble de tag et un document XML réel. Toute cette exactitude et cette conformité sont établies puisque XML est désigné pour décrire des données et des documents. Un document XML bien formé peut contenir des éléments, des attributs et des textes.

1.2.1. Elément

Les éléments ressemblent à ceci et ont toujours des tags d'ouverture et de fermeture : **<élément></élément>**.

Il n'y a pas beaucoup de règles pour les éléments du document XML. Les noms des éléments peuvent contenir des lettres, des nombres, des traits d'union, des tirés bas et des deux points quand les espaces noms «namespaces» sont utilisés (les espaces noms : voir paragraphe 1.4 de cette annexe). Les noms des éléments ne peuvent pas contenir des espaces, ils sont remplacés par des tirés bas. Ils peuvent commencer par une lettre, tiré bas, ou deux points, mais ne peuvent pas commencer par un autre caractère non alphabétique, un nombre ou le mot XML.

Hormis les règles basiques, il est important de penser au sujet de l'utilisation des traits d'union et des points dans les noms des éléments. Ils peuvent être considérés comme une partie des documents XML bien formés, mais d'autres systèmes qu'utiliseront les données dans les noms des éléments tels que les systèmes de base de données relationnelles, auront souvent des troubles en travaillant avec les traits d'union et les points dans les identificateurs de donnée. Ils les prennent pour quelque chose autre qu'une portion du nom de l'élément.

1.2.2. Attributs

Les attributs contiennent des valeurs qui sont associées à un élément et font toujours partie de l'élément du tag d'ouverture : <element attribute="valeur"></element>.

Les règles et directives basiques pour les éléments s'appliquent aussi aux attributs, avec quelques additions. Le nom de l'attribut doit suivre un nom d'élément, ensuite la marque égale « = », après la valeur de l'attribut dans des guillemets simples ou doubles. La valeur de l'attribut peut contenir des guillemets, le cas échéant, un seul type de guillemet doit être utilisé dans la valeur et une autre autour de la valeur.

1.2.3. Texte

Le texte est localisé entre les tags d'ouverture et de fermeture d'un élément, il représente généralement les données réelles associées avec les éléments et attributs qui entoure le texte : <element attribute="valeur">text</element>.

Le texte n'est pas soumis aux mêmes règles syntaxiques des éléments et attributs, donc pratiquement n'importe quel texte peut être stocké entre les éléments du document XML. Il faut noter que lorsque la valeur est limitée au texte, le format du texte peut être spécifié comme un autre type de données par les éléments et attributs dans le document XML.

1.2.4. Elément vide

Les éléments qui n'ont pas d'attributs ou textes peuvent aussi être représentés dans un document XML : </element>.

Ce format est habituellement ajouté aux documents XML pour adapter une structure de données prédéfinie.

1.3. Structure du document XML

La plupart des document XML commence par un élément <?xml?> en haut de la page. Ceci s'appelle une déclaration du document XML. Cette déclaration est un élément optionnel qui est utile pour déterminer la version du document XML pour être bien formé dans la spécification 1.0 de XML de W3C. Voici la déclaration du document XML la plus habituelle : <?xml version="1.0" encoding="UTF-8"?>.

Il y a deux attributs inclus dans cette déclaration qui est vue habituellement mais pas souvent expliquée. La version XML est utilisée pour déterminer à quelle version de la recommandation XML de W3C le document XML adhère. Les parseurs XML utilisent cette information pour appliquer les règles syntaxiques spécifiques à la version du document XML. L'attribut «encoding» qui signifie codage, désigne quel jeu de caractère, «character-set» ou «charset», est utilisé dans la suite du document XML. Jusqu'à date, le type de codage le plus utilisé dans les documents XML est l'UTF-8.

Un document XML bien formé doit être disposé sous certaines règles structurelles et syntaxiques. Examinons la structure d'un document XML 1.0 bien formé très simple dans Listing 1.

```
<?xml version="1.0" encoding="UTF-8"?>
<ÉlémentRacine>
<PremierÉlément position="1">
        <Niveau1 Enfant="0">C'est le niveau 1 des éléments imbriqués
        </level1>
</PremierÉlément>
<DeuxièmeÉlément position="2">
        <Niveau1 Enfant="1">
                <Niveau2>C'est le niveau 2 des element imbriqués
                </Niveau2>
        </Niveau1>
</DeuxièmeÉlément>
</ÉlémentRacine>
```

Listing 1 : Document XML très simple bien formé.

Il faut faire attention à la structure du document incluant les éléments et les attributs, aussi à la syntaxe du document; les parseurs XML et les autres outils qui lisent et traitent les

documents XML ne sont guère tolérants à ces genres de fautes. XML a des règles très strictes à propos de la syntaxe utilisée pour représenter un document, contrairement à HTML.

1.4. Espace nom «*name space*»

Les espaces-nom sont une méthode pour séparer et identifier les noms d'élément XML dupliqués. Ils peuvent aussi être utilisés comme identificateurs pour décrire les types de données et autre information. La déclaration des espaces-nom peut être comparée comme définir un nom de variable court pour une variable longue (tel que pi=3.14149...) dans la programmation. Dans XML, l'assignation de variable est définie par une déclaration d'attribut. Le nom de la variable et sa valeur sont ceux de l'attribut. Afin de distinguer les déclarations d'espace-nom des autres types de déclarations d'attribut, un préfixe réservé xmlns: est employé en déclarant un nom et une valeur d'espace-nom. Le nom d'attribut après le préfixe xmlns: identifie le nom pour l'espace-nom défini. La valeur de l'attribut fournit l'identificateur unique pour l'espace-nom. Une fois l'espace-nom déclaré, le nom de l'espace-nom peut être utilisé comme un préfixe dans les noms d'élément.

`Listing 2` montre un document XML très simple qu'on a examiné dans `Listing 1`, mais cette fois on a ajouté quelques espaces-nom afin de différencier les éléments imbriqués.

```
<?xml version="1.0" encoding="UTF-8"?>
<ÉlémentRacine>
        <PremierÉlément xmlns:fe="http://www.benztech.com/schemas/verybasic"
        position="1">
                <fe:Niveau1 Enfant="0">C'est le niveau 1 des élément imbriqués
                </fe:level1>
        </PremierÉlément>
        <DeuxièmeÉlément
        xmlns:se="http://www.benztech.com/schemas/advanced"
        position="2">
                <se:Niveau1 Enfant="1">
                        <se:Niveau2>C'est le niveau 2 des elements imbriqués
                        </se:level2>
                </se:Niveau1>
        </DeuxièmeÉlément>
</ÉlémentRacine>
```

`Listing 2 : Un document XML très simple avec des espaces-nom.`

Dans cet exemple, on a employé deux espaces-nom comme identificateur pour différencier deux éléments Niveau1 dans le même document. L'attribut xmlns: déclare l'espace-nom pour un document ou une portion de document XML. L'attribut peut être placé dans l'élément ÉlémentRacine du document, ou dans n'importe quels éléments imbriqués.

Le nom de l'espace-nom de l'élément PremierÉlément dans la `Listing 2` est fe, et celui de l'élément DeuxièmeÉlément est se. Les deux utilisent deux URLs différentes comme valeur pour l'espace-nom. L'URL dans l'espace-nom mène souvent à une page WEB qui fournie la documentation concernant l'espace-nom. Il aboutit aussi à un document actuel, mais utilisé comme un détenteur de place pour la déclaration du nom de l'espace-nom.

La valeur de déclaration d'espace-nom n'a pas besoin d'être un URL ou de conduire à un URL actuel. De toute façon, il est préférable d'utiliser un URL qui peut conduire à une destination actuelle. De cette façon, les développeurs peuvent changer la structure et/ou la syntaxe du document XML concerné.

Quand utilise-t-on les espaces-nom?

Les espaces-nom sont des composantes optionnelles des documents XML de base. Cependant, les déclarations d'espace-nom sont recommandées si vos documents XML ont un potentiel actuel ou futur de partage avec d'autres documents qui peuvent partager les mêmes noms d'élément. Aussi, les nouvelles technologies basées sur XML, tels que les schémas XML, SOAP et WSDL, accroissent l'utilisation des espaces-nom XML afin d'identifier les types de codage de données et les éléments importants de leur structure.

1.5. Validation de données

D'après ce qu'on a vu, il y a des règles très strictes pour la structure et la syntaxe des documents XML bien formés. Il y a aussi plusieurs formats qui suivent la syntaxe XML bien formée qui fournissent des manières de représentation de types de données standardisées.

La validité d'un document XML est déterminée par un DTD «Document Type Definition» (définition du type de document). Il y a plusieurs formats de validation de données. On peut trouver une liste de formats de validation XML à http://www.oasis-

open.org/cover/schemas.html. Toutefois, les formats officiels ratifiés par W3C les plus communes sont le DTD «Document Type Definition» et le Schéma W3C.

Les documents XML sont comparés à des règles qui sont spécifiées dans un DTD ou un schéma. Un document XML bien formé qui rassemble toutes les exigences d'une ou plusieurs spécifications est appelé un document XML *valide*.

Notez que les documents XML ne se valident pas entre eux. La validation XML a lieu quand un document est analysé. La majorité des parseurs d'aujourd'hui ont leur propre fonctionnalité de validation, supportent généralement la validation des Schémas W3C et des DTDs et peuvent supporter d'autres types de validation, dépendamment du parseur. En plus, définir une variable ou appeler une classe différente dans le parseur peut souvent neutraliser la validation en ignorant les directives de DTD ou/et de Schéma dans le document XML. Les parseurs ou les classes de parseur qui ne supportent pas la validation sont appelés *des parseurs non validant*, et ceux qui la supportent sont appelés *des parseurs validant*.

Listing 3 montre le même document XML que celui du Listing 1, mais dans ce cas, il y a une référence d'un DTD et une autre d'un Schéma.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ÉlémentRacine SYSTEM "verysimplexml.dtd">
<ÉlémentRacine xmlns:xsi=http://www.w3.org/2001/XMLSchemainstance
xsi:noNamespaceSchemaLocation="verysimplexml.xsd">
<PremierÉlément position="1">
        <Niveau1 Enfant="0">C'est le niveau 1 des éléments imbriqués
        </Niveau1>
</PremierÉlément>
<DeuxièmeÉlément position="2">
        <Niveau1 Enfant="1">
                <Niveau2>C'est le niveau 2 des element imbriqués
                </Niveau2>
        </Niveau1>
</DeuxièmeÉlément>
</ÉlémentRacine>
```

Listing 3 : Document XML bien formé avec référence de DTD et de Schéma.

1.5.1. Validation des documents avec DTD

La définition de Type de document DTD est la méthode originale de validation de la structure du document XML et d'imposition de formatage spécifique du texte choisi. Cette méthode reste probablement la plus répandue. Bien que la déclaration au début du DTD laisse penser que c'est un document XML, les DTDs sont en fait des documents XML non-bien formés. Ceci est parce qu'ils suivent les règles syntaxiques de DTD au lieu de la syntaxe du document XML.

`Listing 4` montre le contenu du fichier verysimplexml.dtd référencié dans le document XML dans `listing 3` :

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT ÉlémentRacine (PremierÉlément, DeuxièmeÉlément)>
<!ELEMENT PremierÉlément (Niveau1)>
<!ATTLIST PremierÉlément
position CDATA #REQUIRED
>
<!ELEMENT Niveau1 (#PCDATA | Niveau2)*>
<!ATTLIST Niveau1
Enfant (0 | 1) #REQUIRED
>
<!ATTLIST DeuxièmeÉlément
position CDATA #REQUIRED
>
<!ELEMENT Niveau2 (#PCDATA)>
<!ELEMENT DeuxièmeÉlément (Niveau2)>
```

`Listing 4 : Contenu du fichier verysimplexml.dtd.`

1.5.2. Validation des documents avec schéma XML

Le Schéma W3C est la définition du Schéma officiellement ratifié. Contrairement au DTD, le format des Schémas W3C suit les règles des documents XML bien formés. De même, le Schéma permet un contrôle beaucoup plus précis des données décrites. Vu la forme d'organisation des données de XML et les contrôles de format détaillés, les Schémas tendent à être très complexes et souvent beaucoup plus longs que les documents XML décrits. Paradoxalement, ils sont généralement bien plus faciles à lire et à suivre par les développeurs, en raison de la nature moins cachée des références dans les Schémas par apport aux DTDs.

Voici la déclaration du Schéma du document XML dans `listing 3` :

```
<ÉlémentRacine xmlns:xsi=http://www.w3.org/2001/XMLSchemainstance
xsi:noNamespaceSchemaLocation="verysimplexml.xsd">
```

Dans ce cas, la déclaration d'espace-nom réfère à http://www.w3.org/2001/XMLSchemainstance. Cette dernière mène à un document actuel, qui est une description courte de la manière que le Schéma devrait être référencié. La valeur noNamespaceSchemaLocation indique qu'il n'y a pas de nom-espace prédéfini pour le Schéma. Cela dit que tous les éléments dans le document XML devraient être validés par le Schéma spécifié. La localisation de ce Schéma utilisé est verysimplexml.xsd parce qu'il n'y a pas de chemin défini. Le fichier du Schéma devrait être localisé dans le même répertoire que le fichier qui va être validé par ce Schéma.

On peut aussi définir la localisation du Schéma et la mappée à l'espace-nom spécifique par l'intermédiaire de la déclaration de l'attribut schemaLocation au lieu de noNamespaceSchemaLocation. Le cas échéant, on doit déclarer un espace-nom qui désigne la valeur de l'attribut schemaLocation. La déclaration doit être faite avant de référer le Schéma dans une assignation de l'attribut schemaLocation. Voici un exemple d'assignement d'un schemaLocation dans un élément racine d'un document XML :

```
<rootelement
xmlns:fe="http://www.benztech.com/schemas/verybasic"
xsi:schemaLocation="http://www.benztech.com/schemas/verybasic">
```

`Listing 5` montre le fichier verysimplexml.xsd référé dans le document XML du `listing 3`.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
    <xs:element name="PremierÉlément">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="Niveau1"/>
            </xs:sequence>
            <xs:attribute name="position" type="xs:boolean"
            use="required"/>
```

```
            </xs:complexType>
        </xs:element>
        <xs:element name="Niveau1">
            <xs:complexType mixed="true">
                <xs:choice minOccurs="0" maxOccurs="unbounded">
                    <xs:element ref="Niveau2"/>
                </xs:choice>
                <xs:attribute name="Enfant" use="required">
                    <xs:simpleType>
                        <xs:restriction base="xs:NMTOKEN">
                            <xs:enumeration value="0"/>
                            <xs:enumeration value="1"/>
                        </xs:restriction>
                    </xs:simpleType>
                </xs:attribute>
            </xs:complexType>
        </xs:element>
        <xs:element name="Niveau2" type="xs:string"/>
        <xs:element name="ÉlémentRacine">
            <xs:complexType>
                <xs:sequence>
                    <xs:element ref="PremierÉlément"/>
                    <xs:element ref="DeuxièmeÉlément"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
        <xs:element name="DeuxièmeÉlément">
            <xs:complexType>
                <xs:sequence>
                    <xs:element ref="Niveau1"/>
                </xs:sequence>
                <xs:attribute name="position" type="xs:byte"
                use="required"/>
            </xs:complexType>
        </xs:element>
</xs:schema>
```

2. SIP «Session Initiation Protocol»

Dans cette partie, on va définir le protocole de communication SIP. On va commencer par donner une introduction, puis donner un vue d'ensemble. Finalement, on va définir le format des différents messages SIP.

2.1. Introduction

Il y a beaucoup d'applications internet qui nécessitent la création et la gestion d'une session, quand une session est considérée comme un échange de données entre un groupe de participant. Les exigences et les pratiques des utilisateurs compliquent l'implémentation de ces applications : les usagers peuvent se déplacer d'un point à un autre, ils peuvent être adressables par plusieurs noms, et ils peuvent communiqués par plusieurs différents médias parfois simultanément. Plusieurs protocoles ont été conçus pour transporter différentes formes de sessions de multimédia et donnée en temps réel et parfois simultanément. Le Session Initiation Protocol (SIP) travaille en accord avec ces protocoles en permettant aux différents terminaux (user agents) de se découvrir et de se fixer les paramètres de la session qu'ils veulent se partager. Pour localiser les participants éventuels de session, et pour d'autres fonctions, SIP est capable de créer l'infrastructure serveurs proxy auxquels les user agents peuvent envoyer des inscriptions, invitation à une session, et d'autre demande. SIP est agile, un outil général pour créer, modifier, et terminer des sessions qui fonctionnent indépendamment des protocoles fondamentaux de transport et sans dépendance sur le type de session qui s'est établi.

2.2. Vue d'ensemble

SIP est un protocole de contrôle de la couche application qui peut établir, modifier, et terminer des sessions multimédia (conférence) tel que la téléphonie internet. SIP peut aussi inviter des participants à des sessions déjà existantes, tel que des conférences. Des médias peuvent être ajoutés (et retirés) à des sessions existantes. SIP supporte d'une manière transparente le mapping des noms et les services de redirections, qui supportent la mobilité personnelle : les usagers peuvent maintenir un seul identifiant visible de l'extérieur sans se soucier de la localisation de leur réseau. SIP supporte cinq facettes d'établissement et de terminaison de session multimédia :

- Localisation des usagers : détermination du terminal à employer pour la communication.
- Disponibilité des usagers : détermination de la bonne volonté de l'appelé à s'engager dans la communication.

- ❖ Capacité des utilisateurs : détermination du média et ses paramètres à employer.
- ❖ Installation de session : « sonner », établissement des paramètres de session pour les deux parties, l'appelé et l'appelant.
- ❖ Gestion de session : y compris le transfert et terminaison de sessions, modification des paramètres de session et appel à des services.

SIP n'est pas un système de communication intégré verticalement. Il est plutôt un composant qui peut être utilisé avec d'autres protocoles IETF pour développer une architecture multimédia complète. Typiquement, ces architectures vont contenir des protocoles tels que Real-time Transport Protocol (RTP, RFC1889) pour transporter les données en temps réel et fournir la qualité de service par la rétroaction. Le Real-Time Stream Protocol (RTSP, RFC2326) pour contrôler la livraison du flot de donnée, le Media Gateway Control Protocol (MEGACO, RFC 3015) pour contrôler le Public Switched Telephone Network (PSTN), et le *Session Description Protocole* (SDP) pour décrire les sessions multimédias. En conséquent, SIP devrait être utilisé en conjonction avec d'autre protocoles dans le but de fournir les services complet aux usagers. Toutefois, la fonctionnalité et l'opération fondamentale de SIP ne dépend d'aucun des ces protocoles.

SIP ne fournit pas de service. Il fournit plutôt les primitifs qui peuvent être utilisés pour implémenter différents services. Par exemple, SIP peut localiser un usager et délivrer un objet opaque vers sa localisation courante. Si ces primitifs sont utilisés pour délivrer une description de session écrite en SDP, par exemple, les terminaux peuvent être d'accord sur les paramètres d'une session. Si le même primitif est utilisé pour délivrer une photo d'un appelant aussi bien que la description de session, un service *caller ID* peut être implémenté facilement. Comme cet exemple le montre, un simple primitif est typiquement utilisé pour fournir plusieurs différents services.

SIP n'offre pas des services de contrôle de services tel que *floor control* ou *voting*, et ne prescrit pas la façon de gérer une conférence. SIP peut être utilisé pour initier une session qui utilise d'autre protocole de contrôle de conférence. Puisque les messages SIP et les sessions

qu'ils établissent peuvent passés à travers des réseaux entièrement différents, SIP n'est en aucun cas un fournisseur de n'importe quel réservation de ressources réseaux.

La nature des services fournis rend la sécurité particulièrement importante. À cet effet, SIP fournit une suite de service de sécurité, qui inclut la prévention contre le déni de service (DOS), l'authentification (deux usagers à usager et proxy à usager), la protection d'intégrité, le chiffrement et le service de protection de la vie privée. SIP fonctionne avec IPv4 et IPv6.

Dans ce travail, SIP s'avère indispensable pour la coordination des différents terminaux. On va l'intégrer dans l'initialisation des sessions, les demandes d'inscription, les notifications d'état de présence et la localisation « *subscribe-notify* » et la publication d'état de présence « *publish* ». On pourrait profiter des suites de service de sécurité qu'il offre.

2.3. Format des différents messages SIP

Un message SIP peut être à la fois une requête d'un client (terminal appelant) vers un serveur (terminal appelé), ou une réponse d'un serveur vers un client :

2.3.1. Requête SIP

<table>
<tr><td colspan="3">Request-line</td><td rowspan="2">Message-headers</td><td rowspan="2">CLRF</td><td rowspan="2">Message body (optionel)</td></tr>
<tr><td>Method</td><td>Request-URI</td><td>SIP-version</td></tr>
</table>

Fig. 20 : SIP-1 : Requête SIP d'un client

Method: Le document officiel de SIP, RFC3261, définit six méthodes qui sont, ACK, INVITE, BYE, CANCEL, OPTIONS, REGISTER. Il y a d'autres RFC qui définissent des extensions de méthode SIP, parmis eux on note, SUBSCRIBE, NOTIFY, MESSAGE, INFO, SERVICE, NEGOTIATE, REFER.

Request-URI : c'est un SIP ou SIPS URI. Il indique la ressource ou le service vers lequel cette demande est adressée. Par exemple «sip:docteur1@damain1.qc»

SIP-version : cet entête indique la version du protocole SIP à utiliser.

Message-header : un message SIP comprend un certain nombre de *Message-Header*. Ces *headers* ou entêtes contiennent des informations qui permettent au récepteur de mieux comprendre et manipuler le message correctement. On mentionnera les *Message-Headers* relatifs à ce projet dans la partie analyse du projet.

CRLF : permet de spécifier la fin du champ de *Message-Headers* et le début du champ *Message-Body*.

Message-Body : le paquet SIP peut contenir un champ *Message-Body*, l'interprétation du corps de ce champ dépend des *Message-Headers* suivants :

- *Content-Type* : indique le type de message que contient le *Message-Body*.
- *Content-Length* : la longueur en octet du champ *Message-Body*.

2.3.2. Réponse SIP

<table>
<tr><td colspan="3">Status-line</td><td rowspan="2">Message-headers</td><td rowspan="2">CLRF</td><td rowspan="2">Message-body (optionel)</td></tr>
<tr><td>SIP version</td><td>Status-code</td><td>Reason Phrase</td></tr>
</table>

Fig. 21 : SIP-2 : Réponse du serveur

Status-Code : est un code entier de trois chiffres qui indiquent le résultat d'une tentative de compréhension et de réponse à une demande. Le *Status-Code* est destiné à l'usage des automates.

Reason-Phrase : elle est destinée à donner une brève description textuelle. La *Reason-Phrase* est destinée à l'usage humain.

SIP-version : la version du protocole SIP utilisé.

2.4. Les Entêtes SIP «Headers»:

Un message SIP comprend un nombre d'entêtes message. Ces entêtes contiennent de l'information qui permet au récepteur de mieux comprendre le message ou le manipuler

proprement. Certains entêtes ont du sens seulement dans certaines requêtes ou réponses. Dans certains cas, la présence d'un entête dépend du contexte. La présence d'un entête spécial dans une réponse pourrait être raisonnable que si la réponse est délivrée à une demande spécifique.

2.4.1. Entêtes générales

Certains entêtes peuvent être utilisés dans les requêtes et les réponses ensemble. Ils sont connus sous le nom entêtes générales. Ces entêtes contiennent des informations de base. Par exemple, le champ d'entête 'To' désigne le destinataire de la requête, 'From' désigne l'expéditeur de la requête, 'Call-ID' est l'identificateur unique d'une invitation spécifique à une session.

Entêtes de requête

Les Entêtes de requête ne s'appliquent qu'aux requêtes SIP. Ils sont utilisés pour fournir des renseignements supplémentaires au serveur concernant la requête ou le client. Par exemple, 'Subject' peut être utilisé pour fournir une description textuelle du thème de la session. 'Priority' est utilisé pour indiquer si la requête est urgente ou non urgente.

Entêtes de réponse

Les champs d'entête de réponse s'appliquent uniquement aux messages réponse. Ces champs d'entête sont utilisés pour fournir de plus amples informations concernant la réponse et qui ne peuvent pas être inclus dans la ligne d'état. Par exemple, 'Unsupported' est utilisé pour identifier les éléments qui ne sont pas supportés par le serveur. 'Retry-After' indique quand un utilisateur appelé sera disponible, si ce dernier est actuellement occupé ou indisponible.

Exemple de message SIP

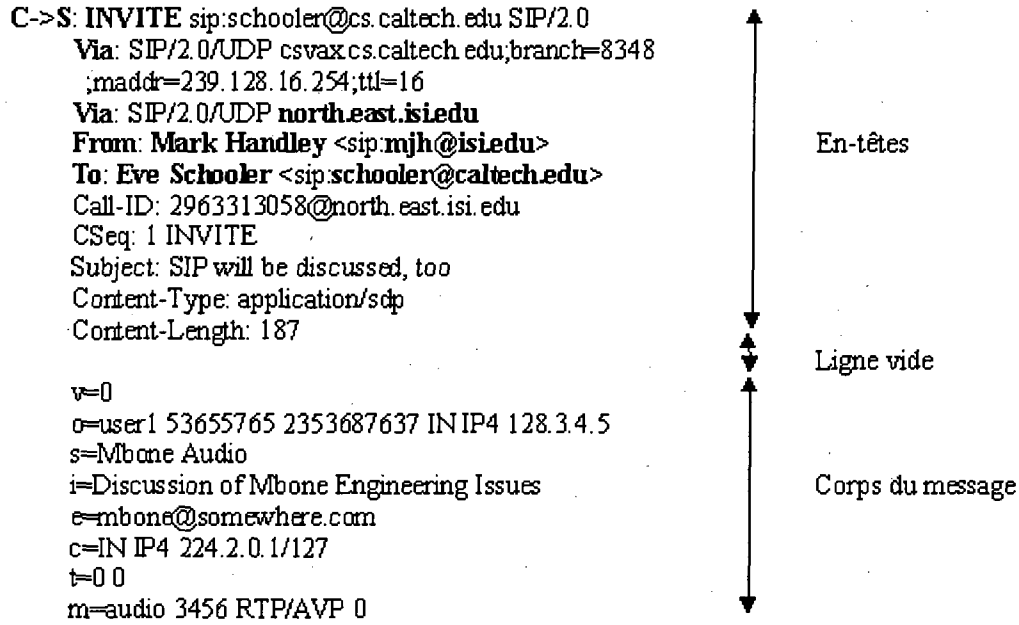


Fig. 22 : Exemple de message SIP

3. Base de données

On va définir dans cette section la notion de base de données. On va commencer par données une introduction, ensuite on va définir la fonction d'une base de données, et finalement on va définir la gestion d'une base de données.

3.1. Introduction

Une base de données (son abréviation est BD, en anglais DB, *database*) est une entité dans laquelle il est possible de stocker des données de façon structurée et avec le moins de redondance possible. Elle doit être conçue pour permettre une consultation et une modification aisée de son contenu, si possible par plusieurs utilisateurs ou programmes différents en même temps. Les données sont stockées dans des champs d'un type déterminé, et ces champs sont groupés dans des tables, reliées entre elles.

La notion de base de données est généralement couplée à celle de réseau, afin de pouvoir mettre en commun ces informations, d'où le nom de **base**. On parle généralement de système

d'information pour désigner toute la structure regroupant les moyens mis en place pour pouvoir partager des données.

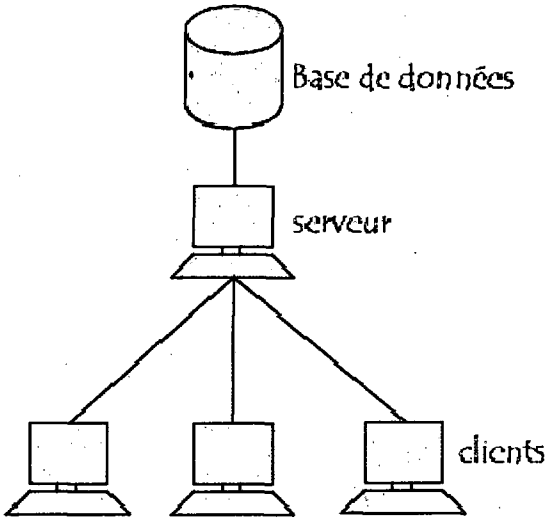


Fig. 23 : Système de base de données.

3.2. Fonction d'une base de donnée

Une base de données permet de mettre à la disposition de l'utilisateur des données pour une consultation, une saisie ou une mise à jour, tout en respectant les règles d'accès soumises. On remarque une augmentation de l'utilisation des bases de données dû à l'expansion du nombre des données informatiques. Plusieurs utilisateurs peuvent accédés à une même base de données en parallèle, ceci est un privilège majeur de son utilisation.

Une base de donnée peut être locale, utilisé localement par l'intermédiaire d'un utilisateur qui la gère sur une machine.

3.3. Gestion d'une base de donnée

Le contrôle d'une base de données est crucial, ainsi un système de gestion devrait être mis en place. La gestion de base de données se fait grâce à un système appelé **SGBD** (système de gestion de base de données) ou en anglais **DBMS** (*Database mangement system*). Un système de gestion de bases de données (SGBD) est un ensemble de service (applications logicielles) permettant de créer, de gérer et d'interroger efficacement une base de données indépendamment du domaine d'application, c'est-à-dire :

- permettre l'accès aux données de façon simple
- autoriser un accès aux informations à de multiples utilisateurs
- manipuler les données présentes dans la base de données (insertion, suppression, modification).

4. Parseur XML

Dans cette section, on va décrire la notion de parseur XML. On va commencer par définir le parseur XML pour finir avec la définition du *DOM Xerces-c parser.*

4.1. Définition d'un parseur XML

Dans le contexte de l'internet, c'est un analyseur syntaxique destiné à récupérer les informations contenues dans les balises d'un document XML. Cet outil distingue les informations en fonction de leur contenu et de leur situation dans le document : balise de début, balise de fin, etc. Plus généralement, un parseur peut être assimilé à un outil d'analyse syntaxique. C'est d'ailleurs le sens premier du terme anglais *parser*.

Un parseur XML validant vérifie également la validité des documents XML. Il vérifie la DTD ou le schéma du document et s'assure que ce dernier s'y conforme, alors qu'un parseur non-validant se contente de vérifier si le document est bien formé.

Dans certaines circonstances, un tel parseur n'est pas indispensable. Par exemple Internet Explorer par défaut n'est pas validant, ce qui lui permet d'afficher un document bien formé même s'il ne respecte pas son DTD ou schéma. Dans d'autres circonstances, comme l'échange de données entre deux systèmes informatiques, il est indispensable que les données échangées soient valides pour pouvoir être traitées au niveau du système destination.

4.2. xerces-c DOM Parser

Il existe deux grandes familles de parseur : DOM (*Document Object Model*) et SAX (*Simple API for XML*). SAX parcourt séquentiellement le document XML tandis que DOM crée un arbre permanent à partir du document XML de telle façon qu'on peut accéder à n'importe quel endroit de cette arborescence.

Xerces-C est un parseur XML validant de la compagnie *Appache* implémenté en C++. Une bibliothèque est mise à la disposition pour analyser, créer, manipuler et valider des documents XML en utilisant les APIs DOM, SAX, et SAX2. Pour accéder à la documentation de l'API de *Xerces-C*, référez vous à l'adresse suivante : http://xerces.apache.org/xerces-c/apiDocs/index.html .

Le cite web officiel de *Xerces* est http://xerces.apache.org/index.html .

Annexe B

Schéma XML détaillé

Ce document est généré par XmlSpy.

Schema **presence modified.xsd**

schema location: E:\analyse du projet\Nouveau dossier\presence modified.xsd
attribute form default: **unqualified**
element form default: **qualified**
targetNamespace: **urn:ietf:params:xml:ns:pidf**

Attributes	Elements	Complex types	Simple types
mustUnderstand	**presence**	**contact**	**PresenceState**
		note	**qvalue**
		presence	**status**
		tuple	

schema location: altova://ystream/xml.xsd
attribute form default:
element form default:
targetNamespace: **http://www.w3.org/XML/1998/namespace**

Attributes	Attr. groups
base	**specialAttrs**
id	
lang	
space	

attribute **mustUnderstand**

namespace: urn:ietf:params:xml:ns:pidf
type: **xs:boolean**
properties: default 0
annotation: documentation

This attribute may be used on any element within an optional PIDF extension to indicate that the corresponding element must be understood by the PIDF processor if the enclosing optional element is to be handled.

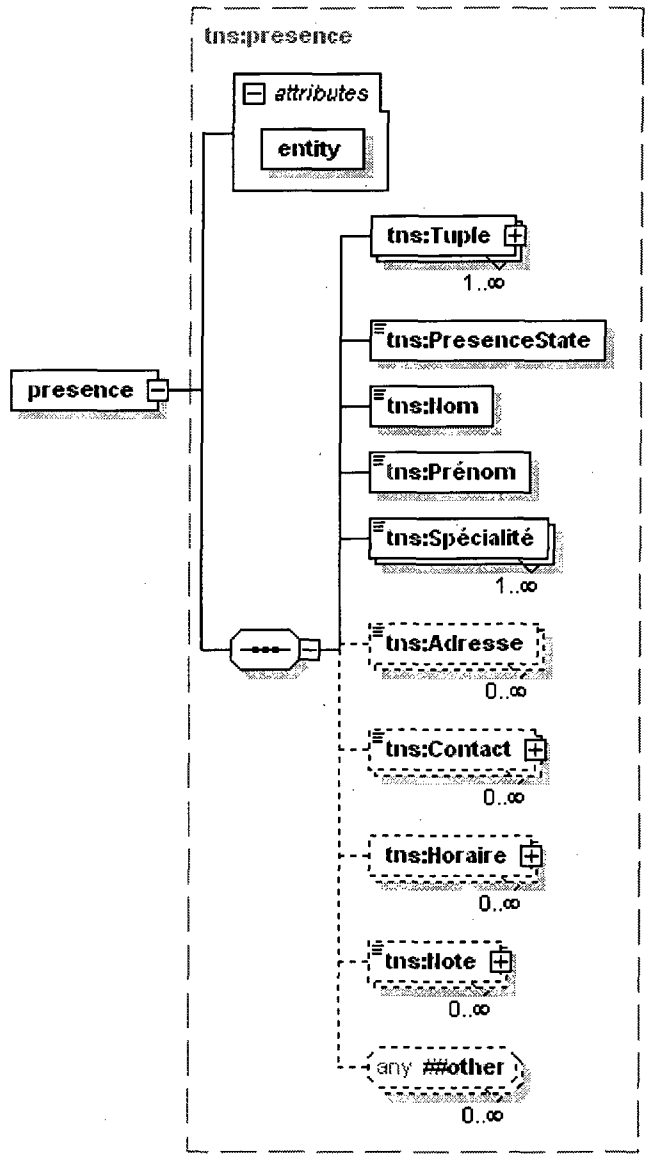
source:

```
<xs:attribute name="mustUnderstand" type="xs:boolean" default="0">
 <xs:annotation>
  <xs:documentation>
     This attribute may be used on any element within an optional
```

PIDF extension to indicate that the corresponding element must be understood by the PIDF processor if the enclosing optional element is to be handled.
</xs:documentation>
</xs:annotation>
</xs:attribute>

element **presence**

diagram



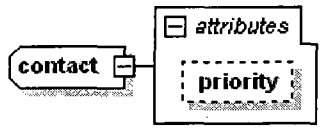
namespace	urn:ietf:params:xml:ns:pidf
type	**tns:presence**
properties	content **complex**
children	**tns:Tuple tns:PresenceState tns:Nom tns:Prénom tns:Spécialité tns:Adresse tns:Contact tns:Horaire tns:Note**

attributes	Name	Type	Use	Default	Fixed	Annotation
	entity	**xs:anyURI**	required			

source <xs:element name="presence" type="tns:presence"/>

complexType **contact**

diagram



namespace urn:ietf:params:xml:ns:pidf

type extension of **xs:anyURI**

properties base xs:anyURI

used by element **presence/Contact**

attributes

Name	Type	Use	Default	Fixed	Annotation
priority	tns:qvalue				

source
```
<xs:complexType name="contact">
 <xs:simpleContent>
  <xs:extension base="xs:anyURI">
   <xs:attribute name="priority" type="tns:qvalue"/>
  </xs:extension>
 </xs:simpleContent>
</xs:complexType>
```

attribute **contact/@priority**

type **tns:qvalue**

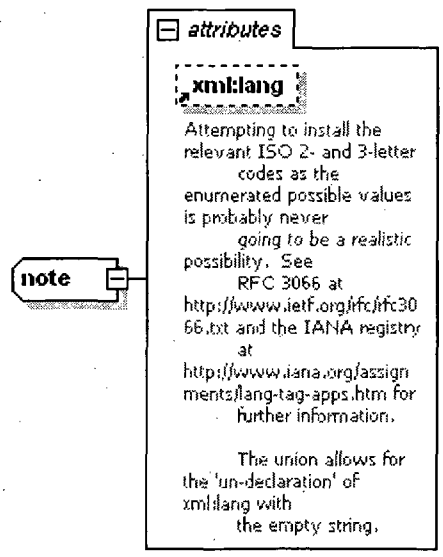
properties isRef 0

facets pattern 0(.[0-9]{0,3})?
pattern 1(.0{0,3})?

source <xs:attribute name="priority" type="tns:qvalue"/>

complexType **note**

diagram



namespace urn:ietf:params:xml:ns:pidf

type **extension of xs:string**

properties base xs:string

used by elements **presence/Note tuple/Note tuple/TypeDuTerminal**

attributes

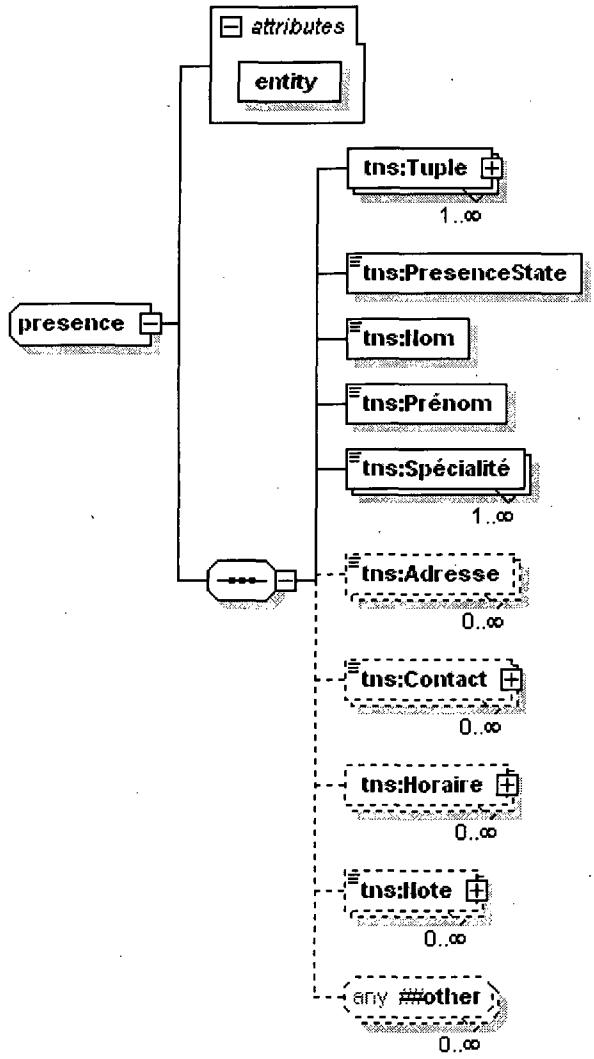
Name	Type	Use	Default	Fixed	Annotation
xml:lang					documentation Attempting to install the relevant ISO 2- and 3-letter codes as the enumerated possible values is probably never going to be a realistic possibility. See RFC 3066 at http://www.ietf.org/rfc/rfc3066.txt and the IANA registry at http://www.iana.org/assignments/lang-tag-apps.htm for further information. The union allows for the 'un-declaration' of xml:lang with the empty string.

source

```
<xs:complexType name="note">
 <xs:simpleContent>
  <xs:extension base="xs:string">
   <xs:attribute ref="xml:lang"/>
  </xs:extension>
 </xs:simpleContent>
</xs:complexType>
```

complexType **presence**

diagram



namespace urn:ietf:params:xml:ns:pidf

children **tns:Tuple tns:PresenceState tns:Nom tns:Prénom tns:Spécialité tns:Adresse tns:Contact tns:Horaire tns:Note**

used by element **presence**

attributes

Name	Type	Use	Default	Fixed	Annotation
entity	**xs:anyURI**	required			

source

```
<xs:complexType name="presence">
 <xs:sequence>
  <xs:element name="Tuple" type="tns:tuple" maxOccurs="unbounded"/>
  <xs:element name="PresenceState"/>
  <xs:element name="Nom"/>
  <xs:element name="Prénom"/>
  <xs:element name="Spécialité" maxOccurs="unbounded"/>
  <xs:element name="Adresse" minOccurs="0" maxOccurs="unbounded"/>
  <xs:element name="Contact" type="tns:contact" minOccurs="0" maxOccurs="unbounded"/>
  <xs:element name="Horaire" minOccurs="0" maxOccurs="unbounded">
   <xs:complexType>
```

```
        <xs:sequence>
          <xs:element name="Début" type="xs:time" minOccurs="0" maxOccurs="unbounded"/>
          <xs:element name="Fin" type="xs:time" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="Note" type="tns:note" minOccurs="0" maxOccurs="unbounded"/>
    <xs:any namespace="##other" processContents="lax" minOccurs="0"
maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="entity" type="xs:anyURI" use="required"/>
</xs:complexType>
```

attribute **presence/@entity**

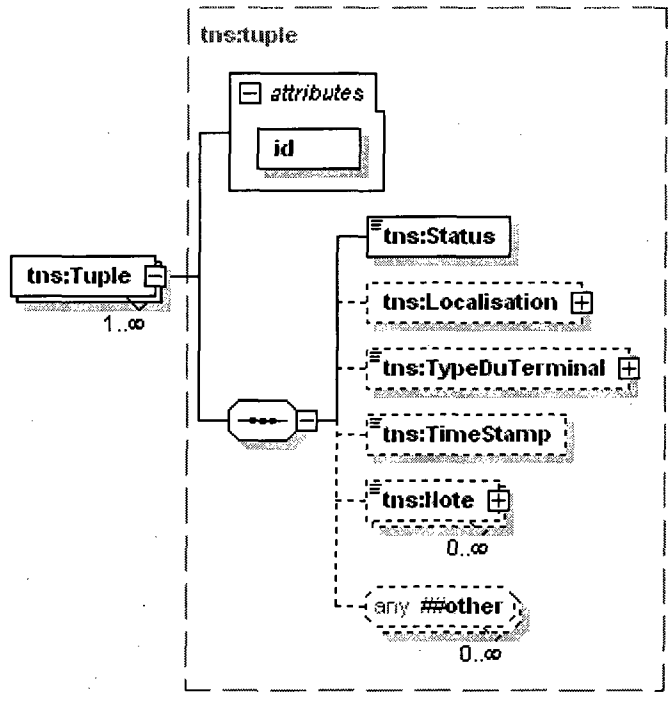
type **xs:anyURI**

properties isRef 0
use required

source <xs:attribute name="entity" type="xs:anyURI" use="required"/>

element **presence/Tuple**

diagram



namespace urn:ietf:params:xml:ns:pidf

type **tns:tuple**

properties isRef 0
minOcc 1
maxOcc unbounded
content complex

children **tns:Status tns:Localisation tns:TypeDuTerminal tns:TimeStamp tns:Note**

attributes

Name	Type	Use	Default	Fixed	Annotation
id	**xs:ID**	required			

source <xs:element name="Tuple" type="tns:tuple" maxOccurs="unbounded"/>

element **presence/PresenceState**

diagram 

namespace urn:ietf:params:xml:ns:pidf

properties isRef 0

source <xs:element name="PresenceState"/>

element **presence/Nom**

diagram 

namespace urn:ietf:params:xml:ns:pidf

properties isRef 0

source <xs:element name="Nom"/>

element **presence/Prénom**

diagram 

namespace urn:ietf:params:xml:ns:pidf

properties isRef 0

source <xs:element name="Prénom"/>

element **presence/Spécialité**

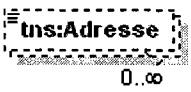
diagram 

namespace urn:ietf:params:xml:ns:pidf

properties isRef 0
minOcc 1
maxOcc unbounded

source <xs:element name="Spécialité" maxOccurs="unbounded"/>

element **presence/Adresse**

diagram 

namespace urn:ietf:params:xml:ns:pidf

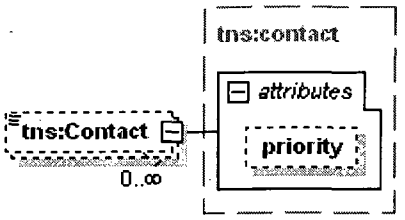
properties isRef 0
minOcc 0

maxOcc unbounded

source <xs:element name="Adresse" minOccurs="0" maxOccurs="unbounded"/>

element **presence/Contact**

diagram



namespace urn:ietf:params:xml:ns:pidf

type **tns:contact**

properties
isRef 0
minOcc 0
maxOcc unbounded
content complex

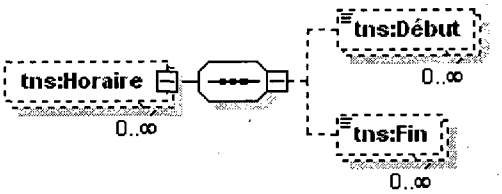
attributes

Name	Type	Use	Default	Fixed	Annotation
priority	tns:qvalue				

source <xs:element name="Contact" type="tns:contact" minOccurs="0" maxOccurs="unbounded"/>

element **presence/Horaire**

diagram



namespace urn:ietf:params:xml:ns:pidf

properties
isRef 0
minOcc 0
maxOcc unbounded
content complex

children **tns:Début tns:Fin**

source
```
<xs:element name="Horaire" minOccurs="0" maxOccurs="unbounded">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="Début" type="xs:time" minOccurs="0" maxOccurs="unbounded"/>
   <xs:element name="Fin" type="xs:time" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

element **presence/Horaire/Début**

diagram

namespace urn:ietf:params:xml:ns:pidf

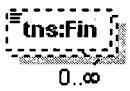
type **xs:time**

properties
- isRef 0
- minOcc 0
- maxOcc unbounded
- content simple

source <xs:element name="Début" type="xs:time" minOccurs="0" maxOccurs="unbounded"/>

element **presence/Horaire/Fin**

diagram



namespace urn:ietf:params:xml:ns:pidf

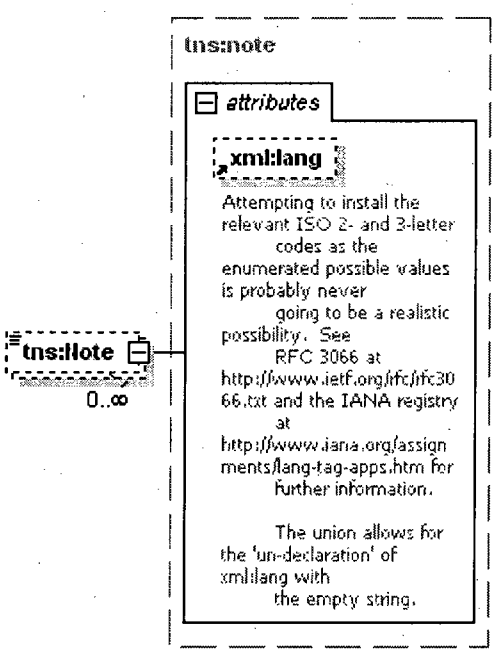
type **xs:time**

properties
- isRef 0
- minOcc 0
- maxOcc unbounded
- content simple

source <xs:element name="Fin" type="xs:time" minOccurs="0" maxOccurs="unbounded"/>

element **presence/Note**

diagram



namespace urn:ietf:params:xml:ns:pidf

type **tns:note**

properties
- isRef 0
- minOcc 0
- maxOcc unbounded
- content complex

attributes

Name	Type	Use	Default	Fixed	Annotation
xml:lang					documentation

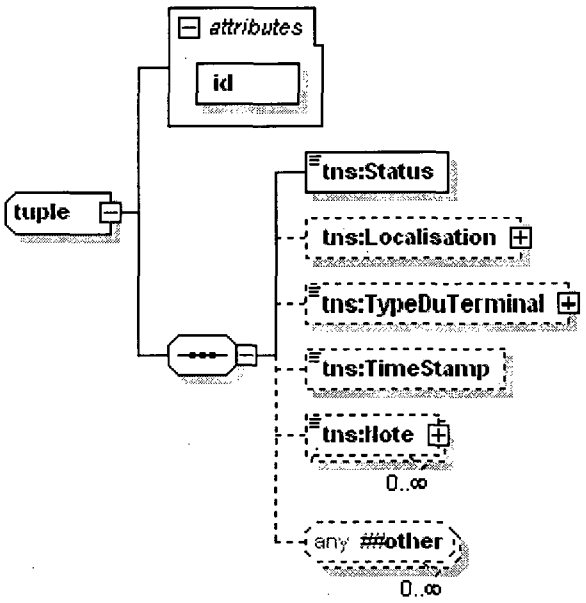
Attempting to install the relevant ISO 2- and 3-letter codes as the enumerated possible values is probably never going to be a realistic possibility. See RFC 3066 at http://www.ietf.org/rfc/rfc3066.txt and the IANA registry at http://www.iana.org/assignments/lang-tag-apps.htm for further information.

The union allows for the 'un-declaration' of xml:lang with the empty string.

source <xs:element name="Note" type="tns:note" minOccurs="0" maxOccurs="unbounded"/>

complexType **tuple**

diagram



namespace urn:ietf:params:xml:ns:pidf

children **tns:Status** **tns:Localisation** **tns:TypeDuTerminal** **tns:TimeStamp** **tns:Note**

used by element **presence/Tuple**

attributes	Name	Type	Use	Default	Fixed	Annotation
	id	**xs:ID**	required			

source
```
<xs:complexType name="tuple">
 <xs:sequence>
  <xs:element name="Status" type="tns:status"/>
  <xs:element name="Localisation" minOccurs="0">
   <xs:complexType>
    <xs:sequence>
     <xs:element name="Lieu" type="xs:string" minOccurs="0"/>
     <xs:element name="AdresseGéospaciale" minOccurs="0"/>
     <xs:element name="AdresseCivique" minOccurs="0"/>
     <xs:element name="AdresseIP" type="xs:hexBinary" minOccurs="0"/>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
  <xs:element name="TypeDuTerminal" type="tns:note" minOccurs="0"/>
  <xs:element name="TimeStamp" type="xs:dateTime" minOccurs="0"/>
  <xs:element name="Note" type="tns:note" minOccurs="0" maxOccurs="unbounded"/>
  <xs:any namespace="##other" processContents="lax" minOccurs="0"
maxOccurs="unbounded"/>
 </xs:sequence>
 <xs:attribute name="id" type="xs:ID" use="required"/>
</xs:complexType>
```

attribute **tuple/@id**

type **xs:ID**

properties
- isRef 0
- use required

source
```
<xs:attribute name="id" type="xs:ID" use="required"/>
```

element **tuple/Status**

diagram

namespace urn:ietf:params:xml:ns:pidf

type **tns:status**

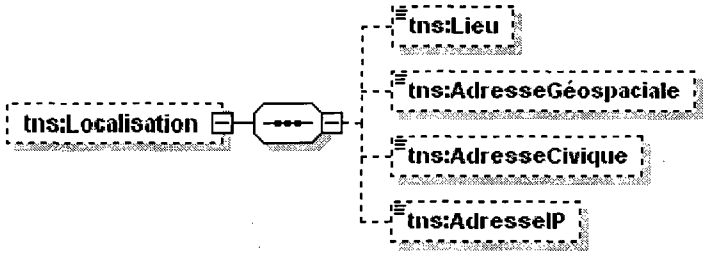
properties
- isRef 0
- content simple

facets
- enumeration connecté
- enumeration déconnecté
- enumeration hors service
- enumeration inanimé

source
```
<xs:element name="Status" type="tns:status"/>
```

element **tuple/Localisation**

diagram



namespace urn:ietf:params:xml:ns:pidf

properties
isRef 0
minOcc 0
maxOcc 1
content complex

children **tns:Lieu tns:AdresseGéospaciale tns:AdresseCivique tns:AdresseIP**

source
```
<xs:element name="Localisation" minOccurs="0">
 <xs:complexType>
  <xs:sequence>
   <xs:element name="Lieu" type="xs:string" minOccurs="0"/>
   <xs:element name="AdresseGéospaciale" minOccurs="0"/>
   <xs:element name="AdresseCivique" minOccurs="0"/>
   <xs:element name="AdresseIP" type="xs:hexBinary" minOccurs="0"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

element **tuple/Localisation/Lieu**

diagram

namespace urn:ietf:params:xml:ns:pidf

type **xs:string**

properties
isRef 0
minOcc 0
maxOcc 1
content simple

source
```
<xs:element name="Lieu" type="xs:string" minOccurs="0"/>
```

element **tuple/Localisation/AdresseGéospaciale**

diagram

namespace urn:ietf:params:xml:ns:pidf

properties
isRef 0
minOcc 0
maxOcc 1

source
```
<xs:element name="AdresseGéospaciale" minOccurs="0"/>
```

element **tuple/Localisation/AdresseCivique**

diagram

namespace urn:ietf:params:xml:ns:pidf

properties
isRef 0
minOcc 0
maxOcc 1

source <xs:element name="AdresseCivique" minOccurs="0"/>

element **tuple/Localisation/AdresseIP**

diagram

namespace urn:ietf:params:xml:ns:pidf

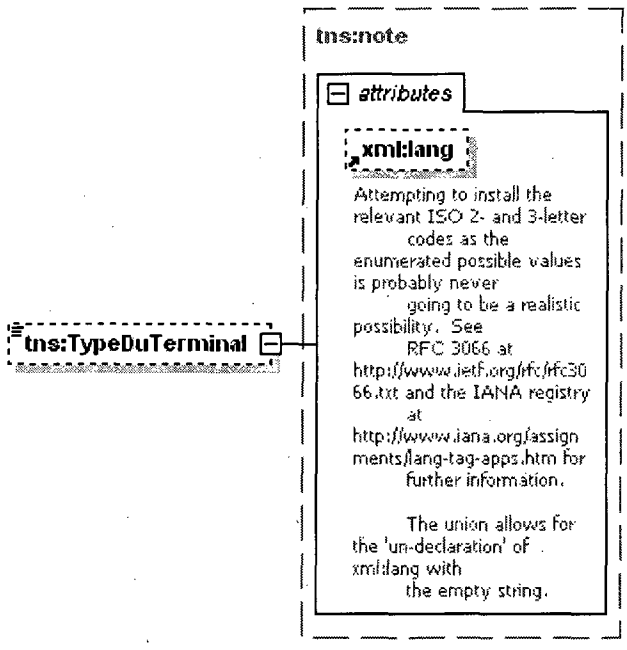
type **xs:hexBinary**

properties
isRef 0
minOcc 0
maxOcc 1
content simple

source <xs:element name="AdresseIP" type="xs:hexBinary" minOccurs="0"/>

element **tuple/TypeDuTerminal**

diagram



namespace urn:ietf:params:xml:ns:pidf

type **tns:note**

properties
isRef 0
minOcc 0
maxOcc 1

content **complex**

attributes

Name	Type	Use	Default	Fixed	Annotation
xml:lang					documentation Attempting to install the relevant ISO 2- and 3-letter codes as the enumerated possible values is probably never going to be a realistic possibility. See RFC 3066 at http://www.ietf.org/rfc/rfc3066.txt and the IANA registry at http://www.iana.org/assignments/lang-tag-apps.htm for further information. The union allows for the 'un-declaration' of xml:lang with the empty string.

source `<xs:element name="TypeDuTerminal" type="tns:note" minOccurs="0"/>`

element **tuple/TimeStamp**

diagram

namespace urn:ietf:params:xml:ns:pidf

type **xs:dateTime**

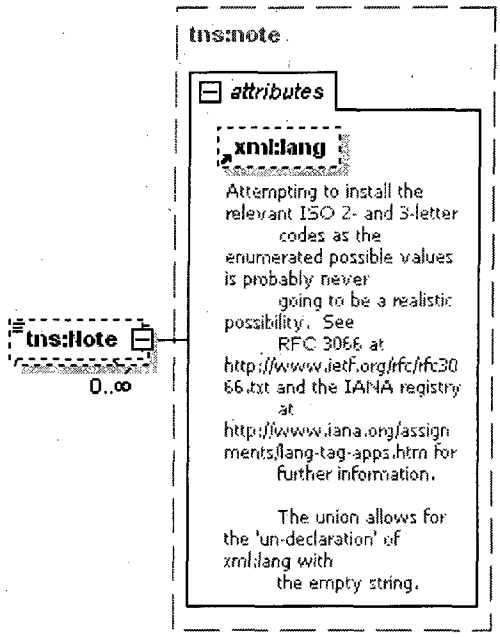
properties

isRef	0
minOcc	0
maxOcc	1
content	simple

source `<xs:element name="TimeStamp" type="xs:dateTime" minOccurs="0"/>`

element **tuple/Note**

diagram



namespace urn:ietf:params:xml:ns:pidf

type **tns:note**

properties
- isRef 0
- minOcc 0
- maxOcc unbounded
- content complex

attributes

Name	Type	Use	Default	Fixed	Annotation
xml:lang					documentation Attempting to install the relevant ISO 2- and 3-letter codes as the enumerated possible values is probably never going to be a realistic possibility. See RFC 3066 at http://www.ietf.org/rfc/rfc3066.txt and the IANA registry at http://www.iana.org/assignments/lang-tag-apps.htm for further information. The union allows for the

'un-declaration' of xml:lang with the empty string.

source <xs:element name="Note" type="tns:note" minOccurs="0" maxOccurs="unbounded"/>

simpleType **PresenceState**

namespace urn:ietf:params:xml:ns:pidf

type restriction of **xs:string**

facets
- enumeration **libre**
- enumeration **occupé**
- enumeration **hors service**
- enumeration **injoignable**
- enumeration **joignable**
- enumeration **en conférence**
- enumeration **sur appel**
- enumeration **de retour**

source
```
<xs:simpleType name="PresenceState">
 <xs:restriction base="xs:string">
  <xs:enumeration value="libre"/>
  <xs:enumeration value="occupé"/>
  <xs:enumeration value="hors service"/>
  <xs:enumeration value="injoignable"/>
  <xs:enumeration value="joignable"/>
  <xs:enumeration value="en conférence"/>
  <xs:enumeration value="sur appel"/>
  <xs:enumeration value="de retour"/>
 </xs:restriction>
</xs:simpleType>
```

simpleType **qvalue**

namespace urn:ietf:params:xml:ns:pidf

type restriction of **xs:decimal**

used by attribute **contact/@priority**

facets
- pattern 0(.[0-9]{0,3})?
- pattern 1(.0{0,3})?

source
```
<xs:simpleType name="qvalue">
 <xs:restriction base="xs:decimal">
  <xs:pattern value="0(.[0-9]{0,3})?"/>
  <xs:pattern value="1(.0{0,3})?"/>
 </xs:restriction>
</xs:simpleType>
```

simpleType **status**

namespace urn:ietf:params:xml:ns:pidf

type restriction of **xs:string**

used by element **tuple/Status**

facets
- enumeration **connecté**
- enumeration **déconnecté**
- enumeration **hors service**
- enumeration **inanimé**

source
```
<xs:simpleType name="status">
 <xs:restriction base="xs:string">
  <xs:enumeration value="connecté"/>
  <xs:enumeration value="déconnecté"/>
  <xs:enumeration value="hors service"/>
  <xs:enumeration value="inanimé"/>
 </xs:restriction>
</xs:simpleType>
```

attribute **xml:base**

namespace http://www.w3.org/XML/1998/namespace

type **xs:anyURI**

annotation documentation
See http://www.w3.org/TR/xmlbase/ for
information about this attribute.

source
```
<xs:attribute name="base" type="xs:anyURI">
 <xs:annotation>
  <xs:documentation>See http://www.w3.org/TR/xmlbase/ for
   information about this attribute.</xs:documentation>
 </xs:annotation>
</xs:attribute>
```

attribute **xml:id**

namespace http://www.w3.org/XML/1998/namespace

type **xs:ID**

annotation documentation
See http://www.w3.org/TR/xml-id/ for
information about this attribute.

source
```
<xs:attribute name="id" type="xs:ID">
 <xs:annotation>
  <xs:documentation>See http://www.w3.org/TR/xml-id/ for
   information about this attribute.</xs:documentation>
 </xs:annotation>
</xs:attribute>
```

attribute **xml:lang**

namespace http://www.w3.org/XML/1998/namespace

type union of (**xs:language**, restriction of **xs:string**)

annotation documentation
Attempting to install the relevant ISO 2- and 3-letter
codes as the enumerated possible values is probably never
going to be a realistic possibility. See
RFC 3066 at http://www.ietf.org/rfc/rfc3066.txt and the IANA registry
at http://www.iana.org/assignments/lang-tag-apps.htm for
further information.

The union allows for the 'un-declaration' of xml:lang with
the empty string.

source
```
<xs:attribute name="lang">
 <xs:annotation>
  <xs:documentation>Attempting to install the relevant ISO 2- and 3-letter
```

```
        codes as the enumerated possible values is probably never
        going to be a realistic possibility.  See
        RFC 3066 at http://www.ietf.org/rfc/rfc3066.txt and the IANA registry
        at http://www.iana.org/assignments/lang-tag-apps.htm for
        further information.

        The union allows for the 'un-declaration' of xml:lang with
        the empty string.</xs:documentation>
    </xs:annotation>
    <xs:simpleType>
     <xs:union memberTypes="xs:language">
      <xs:simpleType>
       <xs:restriction base="xs:string">
        <xs:enumeration value=""/>
       </xs:restriction>
      </xs:simpleType>
     </xs:union>
    </xs:simpleType>
  </xs:attribute>
```

attribute **xml:space**

namespace http://www.w3.org/XML/1998/namespace

type restriction of **xs:NCName**

facets enumeration **default**
enumeration **preserve**

source
```
<xs:attribute name="space">
 <xs:simpleType>
  <xs:restriction base="xs:NCName">
   <xs:enumeration value="default"/>
   <xs:enumeration value="preserve"/>
  </xs:restriction>
 </xs:simpleType>
</xs:attribute>
```

attributeGroup **xml:specialAttrs**

namespace http://www.w3.org/XML/1998/namespace

attributes

Name	Type	Use	Default	Fixed	Annotation
xml:base					documentation See http://www.w3.org/TR/xmlbase/ for information about this attribute.
xml:lang					documentation Attempting to install the relevant ISO 2- and 3-letter codes as the enumerated possible values is probably never

going to be a realistic possibility. See RFC 3066 at http://www.ietf.org/rfc/rfc3066.txt and the IANA registry at http://www.iana.org/assignments/lang-tag-apps.htm for further information.

The union allows for the 'un-declaration' of xml:lang with the empty string.

xml:space

source
```
<xs:attributeGroup name="specialAttrs">
  <xs:attribute ref="xml:base"/>
  <xs:attribute ref="xml:lang"/>
  <xs:attribute ref="xml:space"/>
</xs:attributeGroup>
```

Annexe C

Code source des classes et des interfaces principales

Dans cette partie, on va présenter le code source des classes et des interfaces principales qui sont schématisées dans le diagramme de classe du paragraphe 4.2.5.

1. L'interface ISipCoreUser

Cette interface est utilisée pour reporter la réception de paquets SIP entrants. Lors de la réception d'un paquet SIP, le *controler* peut choisir entre sa manipulation par un *user agent service* ou *proxy service*, généralement basé sur le *request-URI* et l'existence d'un dialogue pour ce paquet. Dans ce cas, le *controler* choisi le *user agent service* puisque le *proxy service* n'est pas activé. La classe CUAStackController qui représente le *controler* hérite cet interface et implémente ses méthodes de telle façon qu'elle réagit convenablement aux paquets reçus.

Deux méthodes sont implémentées dans l'application qui sont :

```
//  paramètre :
//    rPacket : Le paquet entrant.
//  Description :
//    Cet évènement est reporté au SipCoreUser lorsqu'un nouveau paquet
//    entrant est reçu et qui n'est pas manipulé par les couches plus
//    basse de la SIP Stack.
//-----------------------------------------
void EvOnPacketReceived(IN const CSipPacket& rPacket);

//  Description
//    Le processus de l'extinction de l'objet ISipCoreConfig est terminé.
//-----------------------------------------------------------------------
void EvShutdownCompleted();
```

2. ISipSubscriberMgr

L'interface par laquelle le service d'inscription reporte les événements au *subscriber*, cette interface est confidentielle, aucun code source ne peut être affiché. La classe CUASubscriber qui représente le *subscriber* hérite cette interface et implémente ses méthodes réagissant convenablement aux évènements détectés.

Plusieurs méthodes sont implémentées dans l'application qui sont :

```
//  Paramètres :
//    pSvc : L'interface sur laquelle la requete subscribe est envoyée. Ne
//     peut pas être NULL.
```

```
//    rResponse : Le paquet entrant.
//    PClientEventControl : L'interface de contrôle de l'évènement client
//     de cette transaction.
//    strId : Le paramètre id de l'entête Event de la requête subscribe.
//  Description :
//    Informer l'application qu'il y a une réponse provisoire qui est
//    reçue suite à une requête subscribe envoyée.
void EvProgress(IN ISipSubscriberSvc* pSvc,
                IN ISipClientEventControl* pClientEventControl,
                IN const CString& rstrId,
                IN const CSipPacket& rResponse);

//  Paramètres :
//    pSvc : L'interface sur laquelle la requête subscribe est envoyée. Ne
//     peut pas être NULL.
//    rResponse : Le paquet entrant.
//    PClientEventControl : L'interface de contrôle de l'évènement client
//     de cette transaction.
//    strId : Le paramètre id du l'entête Event de la requête subscribe.
//  Description :
//    Informer l'application qu'il y a une réponse de succès qui est
//    reçue suite à une requête subscribe envoyée.
void EvSuccess(IN ISipSubscriberSvc* pSvc,
               IN ISipClientEventControl* pClientEventControl,
               IN const CString& rstrId,
               IN const CSipPacket& rResponse);

//  Paramètres :
//    pSvc : L'interface sur laquelle la requête subscribe est envoyée.
//    rResponse : Le paquet entrant.
//    pClientEventControl : L'interface de contrôle de l'évènement client
//     de cette transaction. Ne peut pas être NULL.
//    strId : Le paramètre id du l'entête Event de la requête subscribe.
//    uMinExpirationSec : La valeur de l'entête Min-Expires trouvée dans //
la réponse 423, voir RFC3261 pour plus amples information.
//  Description :
//    Informer l'application qu'il y a une réponse 423 Interval Too Small
//    reçue suite à une requête subscribe envoyée.
void EvIntervalTooSmall(IN ISipSubscriberSvc* pSvc,
                        IN ISipClientEventControl* pClientEventControl,
                        IN unsigned int uMinExpirationSec,
                        IN const CString& rstrId,
                        IN const CSipPacket& rResponse);

//  Paramètres :
//    les même paramètres de ce qui précède.
//  Description :
//    Informer l'application qu'il y a une réponse d'échec (<300) reçue
//    suite à une requête subscribe envoyée.
void EvFailure(IN ISipSubscriberSvc* pSvc,
               IN ISipClientEventControl* pClientEventControl,
               IN const CString& rstrId,
               IN const CSipPacket& rResponse);

//  Paramètres :
//    pSvc : L'interface sur laquelle la requête subscribe est créée.
//    rNotify : La requête notify reçue.
//    pServerEventControl : L'interface de controle de l'évènement serveur
//     de cette transaction.
//    strId : Le paramètre id de l'inscription.
//  Description :
//    Informer l'application qu'il y a une requête notify valide reçue
//    suite à une requête subscribe envoyée.
void EvNotified(IN ISipSubscriberSvc* pSvc,
```

```
                IN ISipServerEventControl* pServerEventControl,
                IN const CString& rstrId,
                IN const CSipPacket& rNotify);

//  Paramètres :
//    pSvc : L'interface sur laquelle l'inscription est créée.
//    rNotify : La requête notify reçue.
//    pServerEventControl : L'interface de controle de l'évènement serveur
//    de cette transaction.
//    strId : Le paramètre id de l'inscription.
//  Description :
//    Informer l'application qu'il y a une requête notify valide reçue
//    suite à une requête subscribe envoyée. La valeur de l'entête
//    'subscription-state' est 'terminated'.
void EvTerminated(IN ISipSubscriberSvc* pSvc,
                  IN ISipServerEventControl* pServerEventControl,
                  IN const CString& rstrId,
                  IN const CSipPacket& rNotify);

//  Paramètres :
//    pSvc : L'interface qui manipule le notify.
//    rNotify : La requête notify reçue.
//    applicationData : Information configurée par un autre évènement pour
//     cette même requête.
//    reason : La raison pour laquelle le notify est invalide.
//  Description :
//    Informer l'application qu'il y a une requête notify non valide
//    reçue. L'application du système de présence analyse après le paquet,
//    si elle trouve que le notify est envoyé par un utilisateur connu
//    elle le considère un publish.
void EvInvalidNotify(IN ISipSubscriberSvc* pSvc,
                     IN mxt_opaque applicationData,
                     IN const CSipPacket& rNotify,
                     IN mxt_result reason);

//  Paramètres :
//    pSvc : L'interface de la subscribe qui va expirer.
//    strId : Le paramètre id de l'inscription qui va expirer.
//  Description :
//    Informer l'application qu'une inscription va expirer dans un laps de
//    temps configurable.
void EvExpiring(IN ISipSubscriberSvc* pSvc,
                IN const CString& rstrId);
//  paramètres :
//    pSvc : L'interface qui a une inscription expirée.
//    strId : Le paramètre id de l'inscription expirée.
//  Description :
//    Informer l'application qu'une inscription a expiré.
void EvExpired(IN ISipSubscriberSvc* pSvc,
               IN const CString& rstrId);
```

3. ISipNotifierMgr

L'interface par laquelle le service de notification reporte les évènements au *notifier*, cette interface est confidentielle, aucun code source ne peut être affiché. La classe CUANotifier qui représente le *notifier* hérite cette interface et implémente ses méthodes de telle façon qu'elles réagissent convenablement aux évènements détectés.

Plusieurs méthodes sont implémentées dans l'application qui sont :

```
//  Paramètres:
//    pSvc : L'interface sur laquelle la requête subscribe est reçue.
//    pServerEventControl : L'interface de contrôle de l'évènement du
//     serveur pour cette transaction.
//    rstrId : Le id de l'inscription.
//    uExpirationSec : Le temps d'expiration suggéré pour la demande
//     d'inscription.
//    rSubscribe : La requête subscribe reçue.
//  Description :
//    Cette méthode est appelée lorsqu'une requête subscribe valide est
//    reçue. Acceptant cette requête avec une réponse de succès va créer
//    une inscription sur la quelle l'application peut appeler Notify et
//    Terminate.
void EvSubscribed(IN ISipNotifierSvc* pSvc,
                  IN ISipServerEventControl* pServerEventControl,
                  IN const CString& rstrId,
                  IN unsigned int uExpirationSec,
                  IN const CSipPacket& rSubscribe);

//  Paramètres:
//    pSvc : L'interface sur laquelle la requête subscribe est reçue.
//    applicationData : L'information configurée par l'application dans un
//     autre évènement pour cette même requête.
//    reason : La raison pour laquelle le notify est invalide.
//    rSubscribe : La requête subscribe reçue.
//  Description :
//    Informer l'application qu'une requête subscribe invalide est reçue.
void EvInvalidSubscribe(IN ISipNotifierSvc* pSvc,
                        IN mxt_opaque applicationData,
                        IN const CSipPacket& rSubscribe,
                        IN mxt_result reason);

//  Paramètres:
//    pSvc : L'interface sur laquelle la requête subscribe est reçue.
//    pClientEventControl : L'interface de contrôle de l'évènement du
//     client pour cette transaction.
//    rstrId : Le id de l'entête Event da la requête notify.
//    uExpirationSec : Le temps d'expiration suggéré pour la demande
//     d'inscription.
//    rResponse : La réponse provisoire reçue.
//  Description :
//    Informer l'application qu'une réponse provisoire est reçue pour une
//    requête notify reçue.
void EvProgress(IN ISipNotifierSvc* pSvc,
                IN ISipClientEventControl* pClientEventControl,
                IN const CString& rstrId,
                IN const CSipPacket& rResponse);

//  Paramètres:
//    pSvc : L'interface sur laquelle la requête notify est envoyée.
//    pClientEventControl : L'interface de contrôle de l'évènement du
//     client pour cette transaction.
//    rstrId : Le id de l'entête Event da la requête notify.
//    uExpirationSec : Le temps d'expiration suggéré pour la demande
//     d'inscription.
//    rResponse : La réponse de succès reçue.
//  Description :
//    Informer l'application qu'une réponse de succès est reçue pour une
//    requête notify envoyée.
void EvSuccess(IN ISipNotifierSvc* pSvc,
               IN ISipClientEventControl* pClientEventControl,
```

```
                IN const CString& rstrId,
                IN const CSipPacket& rResponse);

//  Paramètres:
//    pSvc : L'interface sur laquelle la requête notify est envoyée.
//    pClientEventControl : L'interface de contrôle de l'évènement du
//     client pour cette transaction.
//    rstrId : Le id de l'entête Event da la requête notify.
//    uExpirationSec : Le temps d'expiration suggéré pour la demande
//     d'inscription.
//    rResponse : La réponse d'échec reçue.
//  Description :
//    Informer l'application qu'une réponse d'échec (<300) est reçue. Voir
//    RFC3261 pour plus amples informations.
void EvFailure(IN ISipNotifierSvc* pSvc,
                IN ISipClientEventControl* pClientEventControl,
                IN const CString& rstrId,
                IN const CSipPacket& rResponse);

//  Paramètres:
//    pSvc : L'interface qui a une inscription expirée.
//    rstrId : Le id de l'inscription expirée.
//  Description :
//    Informer l'application qu'une inscription est expirée.
void EvExpired(IN ISipNotifierSvc* pSvc,
                IN const CString& rstrId);

//  Paramètres:
//    pSvc : L'interface sur laquelle la requête subscribe est reçue.
//    pServerEventControl : L'interface de contrôle de l'évènement du
//     serveur pour cette transaction.
//    rstrId : Le id de l'inscription.
//    rSubscribe : La requête subscribe reçue.
//  Description :
//    Similaire à EvSubscribed mais avec le paramètre 'Expires' égale à 0.
void EvFetched(IN ISipNotifierSvc* pSvc,
                IN ISipServerEventControl* pServerEventControl,
                IN const CString& rstrId,
                IN const CSipPacket& rSubscribe);

//  Paramètres:
//    pSvc : L'interface sur laquelle la requête subscribe est reçue.
//    pServerEventControl : L'interface de contrôle de l'évènement du
//     serveur pour cette transaction.
//    rstrId : Le id de l'inscription.
//    rSubscribe : La requête subscribe reçue.
//  Description :
//    Cette méthode est appelée quand une requête subscribe valide est
//    reçue pour une inscription déjà existante.
void EvRefreshed(IN ISipNotifierSvc* pSvc,
                 IN ISipServerEventControl* pServerEventControl,
                 IN const CString& rstrId,
                 IN unsigned int uExpirationSec,
                 IN const CSipPacket& rSubscribe);

//  Paramètres:
//    pSvc : L'interface sur laquelle la requête subscribe est reçue.
//    pServerEventControl : L'interface de contrôle de l'évènement du
//     serveur pour cette transaction.
//    rstrId : Le id de l'inscription.
//    rSubscribe : La requête subscribe reçue.
//  Description :
//    Cette méthode est appelée quand une requête subscribe avec une
//    entête 'Expires' configurée à 0 est reçue pour une inscription
```

```
//    active ou en attente.
void EvTerminated(IN ISipNotifierSvc* pSvc,
                  IN ISipServerEventControl* pServerEventControl,
                  IN const CString& rstrId,
                  IN const CSipPacket& rSubscribe);
```

4. IUASubscriberMgr

L'interface par laquelle le notifier reporte les évènements à l'application, PresenceSubscriptionObject représente la classe principale de l'application, cette classe hérite l'IUASubscriberMgr et implémente ses méthodes de telle façon qu'elles réagissent convenablement aux évènements détectés.

```
class IUANotifierMgr
{
public:
    //  Paramètre:
    //    rNotifier:
    //      L'objet CUANotifier qui reporte cet évènement.
    //    pTo:
    //      L'entête To du paquet reçu.
    //    pFrom:
    //      L'entête From du paquet reçu.
    //    pContact:
    //      L'entête Contact du paquet reçu.
    //
    // Description:
    //      Cette évènement est reporté quand une demande d'inscription
    //     est reçue.
    virtual void EvSubscriptionA(IN CUANotifier& rNotifier,
                                 IN const CNameAddr& pTo,
                                 IN const CNameAddr& pFrom,
                                 IN const CNameAddr& pContact ) = 0;

    //  Paramètre:
    //    rNotifier:
    //      L'objet CUANotifier qui reporte cet évènement.
    //
    //
    //    Description:
    //      Cet évènement est reporté lorsque l'objet rNotifier a
    //      désincrit avec
    //      succés l'inscription du serveur.
    //
//=================================================================
    virtual void EvUnsubscriptionA(IN CUANotifier& rNotifier) = 0;


    //  Parameters:
    //    rNotifier:
    //      L'objet CUANotifier qui reporte cet évènement.
    //
    //    Description:
    //      Cet évènement est reporté lorsque l'objet CUASubscriber est
    //      Entrain d'envoyer un unsubscribe.
    //
    //=================================================================
virtual void EvSendingUnsubscriptionA(IN CUANotifier& rNotifier) = 0;
```

```
};
```

5. IUASubscriberMgr

L'interface par laquelle le *subscriber* reporte les évènements à l'application, PresenceSubscriptionObject représente la classe principale de l'application, cette dernière hérite cette interface et implémente ses méthodes de telle façon qu'elle réagisse convenablement aux évènements détectés.

```
class IUASubscriberMgr
{
public:

//  Paramètres:
//    rSubscriber:
//      L'objet CUASubscriber qui reporte cet évènement.
//    pMsgContent :
//    La charge utile de la trame reçue.
// Description:
//      Cet évènement est reporté lorsqu'un notify est reçu.
//
//=========================================================================
    virtual void EvNotifiedA(IN CUASubscriber& rSubscriber,
                             IN TO CString* pMsgContent) = 0;

//=========================================================================
//==
//==  EvSubscribedA
//==
//=========================================================================
//
//  Paramètre:
//    rSubscriber:
//      l'objet CUASubscriber qui reporte cet évènement.
//
//  Description:
//      Cet évènement est reporté lorsque l'objet CUASubscriber est
//    inscrit auprès du serveur de présence avec succès.
//=========================================================================
virtual void EvSubscribedA(IN CUASubscriber& rSubscriber) = 0;

//  Paramètre:
//    rSubscriber:
//      l'objet CUASubscriber qui reporte cet évènement.
//
//    Description:
//      Cet évènement est reporté lorsque l'objet est désinscrit auprès du
//    serveur
//    de présence avec succès.
//=========================================================================
    virtual void EvUnsubscribedA(IN CUASubscriber& rSubscriber) = 0;

//    Le même principe sauf que cet évènement est reporté lorsqu'une
//    inscription est refusé
    virtual void EvFailedA(IN CUASubscriber& rSubscriber) = 0;

//  Parameters:
//    CUASubscriber:
```

```
//      L'objet CUASubscriber qui reporte cet évènement.
//    pMsgContent :
//    La charge utile de la trame reçue.
//    pTo:
//      L'entête To du paquet reçu.
//    pFrom:
//      L'entête From du paquet reçu.
//    pContact:
//      L'entête Contact du paquet reçu.
//
//    Description:
//    Cet évènement est reporté lorsque un publish est reçu.
//
//=========================================================================
    virtual void EvPublishedA(IN CUASubscriber& rSubscriber,
                              IN TO CString* pMsgContent,
                              IN const CNameAddr& pTo,
                              IN const CNameAddr& pFrom,
                              IN const CNameAddr& pContact ) = 0;
};
```

6. CEventDriven

Cette classe du cœur du système dans laquelle les *threads* de l'application tournent. La classe qui hérite CEventDriven doit implémenter la méthode EvMessageServiceMgrAwaken(). Le CEventDriven avertit l'application alors qu'il y a un évènement à traiter. Les évènements sont insérés dans la file d'attente à travers la méthode PostMessage(..).

7. CUAStackController

Cette classe est responsable de :

- Être notifiée de requêtes entrantes qui ne sont pas associées à une transaction, les joindre à un dialogue existant, ou finalement créer un nouveau dialogue pour cette requête.
- Répondre à une requête insupportée ou illégale.
- Démarrer ou éteindre la SIP Stack.
- Ouvrir la ou les ports d'écoute du *User Agent.*

```
class CUAStackController : public ISipCoreUser,
                           public ISipTransportUser,
                           public CEventDriven
{
protected:
//    Des commandes passées de la méthode PostMessage(..) à la méthode
//    EvMessageServiceMgrAwaken(..) via CEventDriven.
```

```
//-------------------------------------------------------------------------
    enum EMessageId
    {
        eCREATESUBSCRIPTION_S,
        eCREATENOTIFIER_S,
        eGETADDR_S,
        eVIA_S,
        eCONTACT_S,
        eMGR_S,

        // ISipTransportUser.
        //--------------------
        eEVCOMMANDRESULT
    };

    // Des commandes passées à la couche transport.
    //---------------------------------------------
    enum ETransportCommand
    {
        eLISTEN,
        eSTOPLISTENING,
        eCONNECT
    };

public:
    //-- << Constructeurs / Destructeurs / Operateurrs >>
    //-------------------------------------------------

    //  Constructeur par défaut.
    //--------------------------
    CUAStackController();

    //   Créer une inscription.
    //-------------------------
    mxt_result CreateSubscriptionS(OUT GO CUASubscriber*& rpSubscription);

    //   Créer une notification.
    //-------------------------
    mxt_result CreateNotifierS(OUT GO CUANotifier*& rpNotifier);

    //   Configurer la valeur de l'entête local via du UA pour apparaitre
    //   dans le paquet.
    //-------------------------------------------------------------------
    mxt_result SetUALocalViaS(IN CSipHeader& rLocalVia);

    //   Configurer la valeur de l'entête local contact du UA pour
    //apparaitre dans le paquet.
    //-------------------------------------------------------------------
      mxt_result SetUALocalContactS(IN CSipHeader& rLocalContact);

    //   Configurer les managers qui vont être notifiés d'évènements
    //   spécifiques.
    //-------------------------------------------------------------
    mxt_result SetUAManagersS(IN IUASubscriberMgr* pBasicMgr,
                              IN IUANotifierMgr* pRegMgr);

    //   Ordonner au SIP Stack d'écouter l'adresse et le transport
    //donnés.
    //-------------------------------------------------------------------
    mxt_result Listen (IN ESipTransport eTransport,
                       IN const CSocketAddr& rAddr);

    //   Eteindre la SIP Stack.
    //-------------------------
```

```
    mxt_result Shutdown();

    //   Démarrer la SIP Stack.
    //-------------------------
    mxt_result StartUp(IN const CString& rstrApplicationId);

    //   Ordonner au SIP stack d'arrêter d'écouter l'adresse et le
    //   transport donnés.
    //-----------------------------------------------------------------
    mxt_result StopListening(IN ESipTransport eTransport,
                             IN const CSocketAddr& rAddr);

    //   Ordonner au SIP Stack de se connecter à l'adresse et le transport
    //   donnés.
    //-----------------------------------------------------------------
    mxt_result Connect(IN const CSocketAddr& rLocalAddr,
                               IN const CSocketAddr& rPeerAddr,
                               IN ESipTransport eTransport);
    //-- << CEventDriven >>.
    //----------------------
    void EvMessageServiceMgrAwaken(IN bool bWaitingCompletion,
                                   IN unsigned int uMessage,
                                   IN CMarshaler* pParameter);
    //-- << ISipCoreUser >>.
    //----------------------
    void EvOnPacketReceived(IN const CSipPacket& rPacket);

    void EvShutdownCompleted();

    //-- << ISipTransportUser >>.
    //---------------------------
    virtual void EvCommandResult(IN mxt_result res,
                                 IN mxt_opaque opq);
//    Les variables.
//-------------------
protected:

    // L'entête local Via du UA.
    //--------------------------
    CSipHeader m_localVia;

    // L'entête local contact du UA.
    //------------------------------
      CSipHeader m_localContact;

    // L'objet utilisé pour associer un dialogue externe à une
    // transaction.
    //-----------------------------------------------------------
    CSipDialogMatcherList m_dialogList;

    // Le thread qui est utilisé pour activer les objets event-driven
    // créés.
    //-----------------------------------------------------------------
    IActivationService* m_pCoreThread;

    // Le  subscriber mgr.
    //--------------------
    IUASubscriberMgr* m_pSubscriberMgr;

    // Le notifier mgr.
    //-----------------------
    IUANotifierMgr* m_pNotifierMgr;

};
```

8. CUASubscriber

Cette classe est responsable de la gestion des *subscribes*, hérite aussi le ISipSubscriberMgr et implémente ses méthodes. Le ISipSubscriberMgr est responsable de la réception des *notifys* associés concernant l'inscription encours.

```
class CUASubscriber : public ISipSubscriberMgr,
                              public CEventDriven
{

public :
    //   Attacher les service liés au subscriber au contexte. Configurer
    //   les paramètres nécessaires pour l'envoi de la requête
    //-------------------------------------------------------------
    mxt_result Initialize(IN IUASubscriberMgr* pMgr,
                          IN const CSipHeader& rLocalVia,
                          IN const CSipHeader& rLocalContact);
    //   Envoyer un subscribe à l'entité configurée dans l'objet
    //   CUASubscriber en utilisant les services attachés dans le
    //   contexte correspondant.
    //-------------------------------------------------------------
    void SubscribeA(unsigned int* id,
                    IN const CNameAddr& rFromAddr,
                    IN CSipHeader& rLocalContact,
                    IN const IUri* pRequestUri,
                    IN CNameAddr& rToAddr);

    //   Terminer l'inscription.
    //-------------------------------------------------------------
    void TerminateA();

    //   Notifier le contexte de la réception d'un
    //    paquet.
    //-------------------------------------------------------------------------
    virtual mxt_result OnPacketReceived(IN const CSipPacket& rSubscribe);

protected:

    //    Rafraichir l'inscription.
    //-----------------------------
    void Refresh();
    //-- << CEventDriven >>.
    //----------------------

    //-- << ISipSubscriberMgr >>.
    //----------------------------

protected:
    // Le contexte de l'inscription.
    //------------------------------
    ISipContext* m_pContext;

    // L'objet qui gère la transaction du client.
    //-------------------------------------------
    ISipClientTransaction* m_pClientTransaction;

    // Une référence au Subscriber mgr.
    //---------------------------------
    IUASubscriberMgr* m_pMgr;

    // L'information opaque du manager.
```

```
    //---------------------------------
    mxt_opaque m_opq;

};
```

9. CUANotifier

Cette classe est responsable de la gestion des *notifys*, elle hérite aussi le ISipNotifierMgr et implémente ses méthodes. Le ISipNotifierMgr est responsable de la réception et traitements des *subscribes*.

```
class CUANotifier : public ISipNotifierMgr,
                    public CEventDriven
{
    //   Attacher les service liés au notifier au contexte. Configurer
    //   les paramètres nécessaires pour l'envoi des notifications.
    //-------------------------------------------------------------
    mxt_result Initialize(IN IUANotifierMgr* pMgr,
                          IN const CSipHeader& rLocalVia );

    //   Envoyer un notify à l'entité configurée dans l'objet
    //   CUANotifier en utilisant les services attachés dans le
    //   contexte correspondant.
    //-------------------------------------------------------------
    void NotifyA(IN const CNameAddr& rFromAddr,
                         IN CSipHeader& rLocalContact,
                         IN const IUri* pRequestUri,
                         IN CNameAddr& rToAddr,
                         IN const char* pBufferToSend);
    //    Terminer l'inscription associé à l'objet CUANotifier.
    //-------------------------------------------------------
    void TerminateA();

    //    Créer une charge utile, généralement un document XML, afin de
    //    l'associer au notify à envoyer.
    //-------------------------------------------------------
      GO CSipContentInfo* CreatePresenceContent( const char* rBuffer );

    //   Notifier le contexte de la réception d'un
    //    paquet.
    //-------------------------------------------------------------------------
    virtual mxt_result OnPacketReceived(IN const CSipPacket& rPacket);

protected:

    //-- << CEventDriven >>.
    //-----------------------

    void Refresh( const char* rBuffer );

    //-- << ISipNotifierMgr >>.
    //--------------------------

protected:
    // Le contexte de la notification.
    //--------------------------------
    ISipContext* m_pContext;

    // L'objet qui gère la transaction du client.
    //-------------------------------------------
      ISipClientTransaction* m_pClientTransaction;
```

```
    // L'objet utilisé pour envoyer les réponses.
    //-------------------------------------------
    ISipServerEventControl* m_pServerEventCtrl;

    // La référence au notifier mgr.
    //------------------------------
    IUANotifierMgr* m_pMgr;

    // L'information opaque du mgr.
    //-----------------------------
    mxt_opaque m_opq;

};
```

10. PresenceSubscriptionObject

```
class PresenceSubscriptionObject : private IUASubscriberMgr,
                                   private IUANotifierMgr,
                                   public CEventDriven

{
protected:


    //  Structure de données qui tienne l'information d'inscription, elle
    //  est utilisée localement par la SIP Stack.
    //-------------------------------------------------------------------------
    struct SSubscriptionInfo
    {
            //  L'identificateur de cette inscription.
            //---------------------------------------
            unsigned int rstrId;

        // The username.
        //--------------
        CNameAddr m_user;

        // The subscriber address.
        //------------------------
        IUri* m_pRegistrar;

            // The address of record where to subscribe.
        //------------------------------------------
        CNameAddr m_addrOfRecord;

            // The contact address.
            //---------------------
            CNameAddr m_contact;

            SSubscriptionInfo()
        {
            };
        ~SSubscriptionInfo()
        {};
    };

public :
    //  Structure de données qui tienne l'information d'inscription, elle
    //  est exportée et utilisée par toute l'application.
    //-------------------------------------------------------------------------

struct SubscriptionTable
{
```

```
public:
      // L'état d'inscription.
      //----------------------
      bool m_subscribed;

      // Indique si cette inscription a reçu un publish.
      //------------------------------------------------
      bool m_publish;

      // Le notifier.
      //--------------
      CUANotifier* m_notifier;

      //L'identificateur de cette inscription.
      //---------------------------------------
      unsigned int m_notifierId;

    // Le SIP Uri de l'usager.
    //------------------------
    const char* m_userSipUri;

    // L'adresse SIP Uri de la cible.
    //-------------------------------
    const char* m_addrOfRecord;

      //Le charge utile.
      //-----------------
      const char* m_buffer;

      SubscriptionTable( const char* rUserSipUri)
    {
      //Initialisation des paramètres.
      //-------------------------------
            m_subscribed = false;
            m_publish = false;
            m_buffer = "";
            m_userSipUri = rUserSipUri;
      };
    ~SubscriptionTable()
    {};
};

      //    Initialiser les composantes SIP du User Agent et ses
      //    dépendances.
      //-----------------------------------------------------------
      mxt_result Initialize();

      //  Démarrer l'application.
      //-------------------------
      int start();

      //    Terminer l'application.
      //-----------------------------
      void Terminate();

      //    Initialiser le temps.
      //-------------------------
      void InitTime();

      //   Retourne l'adresse IP et le port destination.
      //------------------------------------------------
      CSocketAddr getPeerAddr();

      //   Configurer l'adresse IP destination et port.
```

```
        //-----------------------------------------------
        mxt_result setRemoteAddr( const char* remoteAddr );

        //   Retourne l'adresse IP locale et le port.
        //-----------------------------------------------
        CSocketAddr getLocalAddr();

        //   Configurer l'adresse IP locale et le port.
        //-----------------------------------------------
        mxt_result setLocalAddr( const char* hostAddr );

        //   Retourne le controleur de la SIP Stack.
        //-----------------------------------------------
        CUAStackController* getStackCtrl();

        //   Analyser le fichier de configuration. Extraire les paramètres
        //   de configuration et les enregistrer dans les variables locales.
        //-----------------------------------------------------------------
        mxt_result ParseConfigFile();

        //   Configurer le chemin du fichier de configuration.
        //-----------------------------------------------------
        void setConfigFile( const char * configFile );

        //   Retourner le chemin du fichier de configuration.
        //-----------------------------------------------------
        CString* getConfigFile();

        //   Afficher les valeurs des paramètres de configuration de la SIP
        //   Stack.
        //-----------------------------------------------------------------
        void PrintCurrentConfiguration();

//  Parameters:
//    rUser:
//      L'adresse SIP URI destination. S'il est null l'envoi sera
//      multicast.
//    rBuffer :
//      Le contenu utile du notify.
//
//    Description:
//    Envoyer un notify.
//
//=================================================================
        void HandleNotify( const char* rUser, const char* rBuffer);

      //   Retourne true si la configuration du Subscriber est complète.
      //-----------------------------------------------------------------
      bool IsSubscriptionConfigComplete();

      //    créer un objet CUASubscriber, initialiser les paramètres de
      //    configuration dans cet objet et envoyer un subscribe.
      //-----------------------------------------------------------------
      void HandleSubscription();

      //   Terminer l'inscription encours. Terminer toutes les inscriptions
      //encours s'il y a plusieurs inscription dans le cas d'un serveur.
      //-----------------------------------------------------------------
      void HandleTermination();

      //   Configurer les managers. Ils sont responsables de la détection des
      //   évènements de notification et d'inscription.
      //-----------------------------------------------------------------
      void SetManagers();
```

```
    //   Ajouter un utilisateur dans la liste des utilisateurs
    //   enregistrés.
    //-------------------------------------------------------------------
      void SetUser(SubscriptionTable* rUser);

    //   Enlever un utilisateur enregistré de la table.
    //-------------------------------------------------
      void RemoveUser(const char* rUser);

    //   Chercher un utilisateur enregistré dans la table par SIP URI.
    //-------------------------------------------------------------------
      SubscriptionTable* GetRegisteredUserStruct(const char* rUser);

    //   Chercher un utilisateur enregistré dans la table par numéro
    //   d'ordre.
    //-------------------------------------------------------------------
      SubscriptionTable* GetRegisteredUserStruct(unsigned int rUser);

    //   Retourner le nombre d'élément de la liste des utilisateurs
    //   enregistrés.
    //-------------------------------------------------------------------
      unsigned int GetRegisteredUserElementCount();

    //   Trouver le numéro d'ordre d'un utilisateur enregistré par SIP URI
    //   dans la liste.
    //-------------------------------------------------------------------
      unsigned int FindRegisteredUser(const char* rUser);

    //   Lorsqu'un utilisateur enregistré s'inscrit auprès du serveur de
    //   présence, ce dernier change l'état de la varible
    //   m_ subscribed de l'état false à l'état true.
    //-------------------------------------------------------------------
      void SetUserSubscriptionState(const char* rUser, bool rState);

    //   Lorsqu'un client reçoit un notify de la part du serveur de
    //   présence, il change la variable m_subscribed de l'état false à
    //   l'état true.
    //-------------------------------------------------------------------
      void SetUserNotifiedState(const char* rUser, bool rState);

//  Parameters:
//    rUser:
//      L'adresse SIP URI à configurer.
//    rBuffer :
//      Le contenu utile à modifier.
//
//    Description:
//      Remplacer la varible m_buffer de l'utilisateur rUser par rBuffer.
//====================================================================
      void SetUserBuffer(const char* rUser, const char* rBuffer);

    //   Retourner la variable m_buffer du l'utilisateur rUser.
    //-------------------------------------------------------
      const char* GetUserBuffer(const char* rUser);

      //-- << CEventDriven>>.
    //----------------------

      //-- << IUASubscrierMgr >>.
    //-----------------------------

      //-- << IUANotifierMgr >>.
    //-----------------------------
```

```
private:

    // Référence vers le controleur de la SIP stack.
    //----------------------------------------------
    CUAStackController* m_pStackCtrl;

    // Le paramètre local Via du UA.
    //------------------------------
    CSipHeader m_localVia;

    // La valeur de l'entête From du UA.
    //--------------------------
    CNameAddr m_fromAddr;

    // La valeur de l'entête contact.
    //-------------------------------
    CSipHeader m_localContact;

    // La valeur de l'entête Request-URI.
    //-----------------------------------
    IUri* m_pRequestUri;

    // La valeur de l'entête To.
    //--------------------------
    CNameAddr m_toAddr;

    // La variable qui indique si le fichier de configuration est lu ou
    // non.
    //-----------------------------------------------------
    bool m_bReadConfigFile;

    // La variable qui indique si l'inscription est active ou non.
    //------------------------------------------------
    bool m_bSubscribed;

    // La référence au subscriber.
    //----------------------------
    CUASubscriber* m_pSubscriber;

    // La liste des références vers les notifier.
    //-------------------------------------------
    CVector<CUANotifier*> m_vpActiveNotifiers;

    // La liste des utilisateurs enregistrés.
    //---------------------------------------
    CVector<SubscriptionTable*> m_RegisteredUsers;

    // Le publisher.
    //---------------
    CUANotifier* m_pPublisher;

    // Le chemin du fichier de configuration.
    //---------------------------------------
      CString* m_strConfigFilePath;
};
```

11. PersonnelChoice

La classe 'PersonnelChoice' représente un objet permettant la recherche d'un ou plusieurs personnels médicaux qui sont disponibles à accomplir une tâche précise. Il faut définir les critères de sélection que cette classe utilisera pour choisir les personnels désirés

```
class PersonnelChoice
{

public:
//  C'est les critères de la sélection et sont définis au moment de la
//   recherche.
std::vector<Criterium*> criteriumLooked;

//  Le document concerné par la recherche.
DOMDocument *documentQuested;

//  Nombre de critères considérés pour la recherche
static const int criteriumNumber = 3;

//  Les critères considérés pour la recherche.
static const char * const
consideredCriterium[PersonnelChoice::criteriumNumber];

//  Les constructeurs.
PersonnelChoice( DOMDocument* document );
PersonnelChoice();

//  Le destructeur.
~PersonnelChoice();

//  Configurer le document à chercher à cet objet
void setDocument( DOMDocument *document );

//  Vérifier si ce critère est pris en considération par la recherche.
static bool isAllowed( const XMLCh *criterium );

//  Remettre à zéro cet objet.
void renew();

//  Ajouter un critère de sélection au vecteur criteriumLooked.
int setCriterium( const XMLCh *criteriumName , const XMLCh *criteriumValue
, const bool _isStatic );

//  Supprime un critère de sélection.
int removeCriterium( const XMLCh* criteriumName );

//  Retourner le vector qui comprend tous les critères de sélection.
std::vector<char*> getCriteriums();

//  Retouner les personnels médicaux trouvés par le système de recherche.
std::vector<Presence*> getChosenPersonnel( int maxPersonnel );
};
```

12. Criterium

La classe 'Criterim' représente un objet qui est utilisé par la classe 'PersonnelChoice' pour stocker les critères de sélection.

```
class Criterium
{

private :
//  Le nom du critère.
const XMLCh *criteriumName;

//  La valeur du critère.
const XMLCh *criteriumValue;

//  Indique si ce critère change au cours du temps.
      bool changeState;
 public :
//  Le constructeur.
      Criterium( const XMLCh *name , const XMLCh *value , const bool
isStatic );
//  Le destructeur.
~Criterium();

//  Retourne le nom du critère.
const XMLCh * getCriteriumName();

//  Retourne la valeur de critère.
const XMLCh * getCriteriumValue();

//  Retourne la valeur de la variable 'changeState'
      const bool isStatic();
};
```

13. EventCatcher

La classe 'EventCatcher' représente un objet qui permet d'enregistrer les évènements détectés de chaque élément de présence d'un document. Tous les éléments sont listés dans un 'vector' et chaque élément qui subit un changement dans son information de présence sera marqué.

```
class EventCatcher
{
private :
//  La liste des éléments de présence du document.
std::vector<EventCatcherElement*> eventCatcherElementArray;

public :
//  Le constructeur.
EventCatcher();

//  Le destructeur.
~EventCatcher();

//  Retourner le vecteur des éléments contrôlés par cet objet.
```

```
std::vector<EventCatcherElement*> getElementArray();

//  Insérer un nouveau élément de présence dans le vecteur.
int setElement( const XMLCh * _SIPUser );

//  Retourner un élément spécifique.
EventCatcherElement * getElement( const XMLCh * _SIPUser ) ;

//  Vérifier si cet élément existe dans le vecteur.
bool isFound( const XMLCh * _SIPUser );

//  Supprimer un élément du vecteur.
      int removeElement( const XMLCh * _SIPUser );

//  Remettre à zéro cet objet.
      void clear();
};
```

14. EventCatcherElement

La classe 'EventCatcherElement' représente un objet utilisé par la classe 'EventCatcher' pour stocker des variables des éléments de présence.

```
class EventCatcherElement
{
private :
//  La valeur 'entity' de l'élément presence.
const XMLCh *SIPUser;

//  Indiquer si cet élément a subi un changement.
bool updatedState;

//  Varible utilisée par le programme principal, dont tous les éléments de
//  présence ont été notifiés du changement subi à cet élément.
bool refreshed;

public :
//  Les constructeurs.
EventCatcherElement( const XMLCh *_SIPUser );
EventCatcherElement();

//  Le destructeur.
~EventCatcherElement();

//  Fixe la valeur 'SIPUser' qui repésente un usager ou un élément de
//  présence dans le document principal.

//  Configurer l'adresse SIP de cet objet.
void setSIPUser( const XMLCh *_SIPUser );

//  Retourner la valeur de 'entity' de cet objet.
const XMLCh *getSIPUser();

// Fixe la variable updatedState.
void updated( bool _updated );

//  Retouner la valeur updatedState.
bool getUpdatedState();

//  Supprimer un élément.
void removeElement();
```

```
//  Remettre à zéro la liste.
void clear();

//  Retouner la valeur de la variable 'refreshed'.
bool isRefreshed();

//  Configurer la valeur de la variable 'refreshed'.
void SetRefreshedState( bool _refreshed );
};
```

15. Document

La classe 'Document' représente l'objet manipulant le document principal de présence, des pointeurs se créent pour localiser le document et ses composantes dans la mémoire.

```
class Document
{
public:
//  Le nom de la balise qui représente tout le document de présence.
static const char * const documentElementName;

//  Le fichier d'entrée.
const char * const inputXMLFileName;

//  Le fichier de sortie.
const char * const outputXMLFileName;

//  Le parseur du document de présence, le fichier d'entrée.
XercesDOMParser *parser;

//  Le détecteur d'évènement.
EventCatcher *eventCatcher;

//  Le pointeur de la racine du document dans la mémoire.
DOMDocument * const sourceDOMDocument;

//  Les constructeur. Au cas où le fichier lecture est introuvable ou vide,
ce dernier crée un nouveau document.
Document( char *inputFile , char *outputFile );
Document( const char *file );
Document( DOMNode* importedNode );
Document(const char* rBuffer, unsigned int docID);

//  Le destructeur.
~Document();

//  Retourner le pointeur de l'élément racine.
DOMDocument* getSourceDocument();

//  Créer un nouveau document.
DOMDocument* createNewDocument();

//  Anlyser le contenu du buffer et l'ajouter au document.
DOMDocument* SetBuffer( const char* rBuffer);

//  Retourner un objet 'Presence' qui représente un élément de présence
//  spécifique
Presence* getPresenceObjectBySpecifyElement( DOMElement* element );

//  Retourner l'élément racine de ce document.
```

```
DOMElement* getSourceElement();

//  Parser le document de présence.
DOMDocument * parseDocument();

//  Initialiser le parseur.
int initParser();

//  Terminer le parsage.
int terminateParser();

//  Insérer un nouveau élément de présence dans le document.
Presence* insertPresenceElement( const XMLCh *entityValue );
int insertPresenceElement( Presence* presence );
int insertPresenceElement( DOMElement* element );//insérer un élément

//  Enregistrement du document de présence dans le fichier de sortie
//  configurer.
int savePresenceDocument();
//  Supprimer un élément de présence spécifié par son paramètre 'entity'
int removePresenceElement( const XMLCh * entityValue );

//  Remplacer un élément de présence.
int replacePresenceElement( DOMNode* newElement, DOMNode* oldElement );

//  Configurer le détecteur d'évènement.
void setEventCatcher( EventCatcher* eventCatcher );
};
```

16. Presence

La classe 'Presence' représente un moyen permettant de manipuler un utilisateur.

```
class Presence
{
public:

//  Le document source de cet objet 'Presence'.
DOMDocument * const sourceDOMDocument;

//  L'élément racine de cet objet 'presence'.
DOMElement * const rootElement;

//  Le détecteur d'évènement associé à cet élément de présence.
EventCatcher *eventCatcher;

//  Le constructeur.
Presence( DOMDocument *dOMDocument , const XMLCh *attributeValue );
Presence( DOMDocument *dOMDocument , DOMElement *rootElement );

//  Le destructeur.
~Presence();

//  Insérer un élément dans cet élément 'presence'.
void insertSimpleElement( const XMLCh *elementName , const XMLCh
*elementValue );

//  Insérer un élément complexe 'Tuple' en spécifiant son paramètre 'id'.
//  Un 'Tuple' représente un moyen de communication pour un usager.
Tuple* insertTupleElement( const XMLCh *idValue );

//  Retourne la racine d'un élément spécifié par son nom et sa valeur.
```

```
DOMElement* getSpecifyElementByValue( const XMLCh *elementName , const
XMLCh *elementValue );

//  Retourne un 'Tuple' qui correspond à l' 'id' spécifié
Tuple* getTupleById( const XMLCh *attributeValue);

//  Retoune l'élément racine de cet Objet 'Presence'.
DOMElement* getRootElement();

//  Configure une valeur  d'un élément d'un utilisateur après avoir
//  vérifié l'existance de cette valeur et sa redondance.
int setElementValue(const XMLCh *elementName , const XMLCh *elementValue );

//  Supprime un élément 'presence' dans le document spécifié par son nom
//  et sa valeur.
int removeSimpleElement( const XMLCh * elementName , const XMLCh
*elementValue );

//  Supprime un élément 'tuple' avec ses sous éléments.
int removeTupleElement( const XMLCh * idValue );

//  Supprime tous les éléments qui vérifient le nom mentionné.
int removeSimpleElements( const XMLCh *elementName );

//  Configurer le détecteur d'évènement de cet utilisateur.
void setEventCatcher( EventCatcher *eventCatcher );

//  Enregistrer le document de présence.
int savePresenceElement( const XMLCh *fileName );

};
```

17. Tuple

La classe 'Tuple' représente un moyen permettant de manipuler un moyen de communication assigné à un utilisateur.

```
class Tuple
{
public:
//  Le document source de cet objet 'tuple'.
DOMDocument * const sourceDOMDocument;

//  L'élément racine de cet objet 'tuple'.
DOMElement * const rootElement;

//  Le détecteur d'évènement associé à cet objet 'tuple'.
EventCatcher *eventCatcher;

//  Le constructeur.
Tuple( DOMDocument *dOMDocument , DOMElement *rootPresenceElement , const
XMLCh *attributeValue );
Tuple( DOMDocument *dOMDocument , DOMElement *rootElement );

//  Le destructeur.
~Tuple();

//  Insérer un élément dans le tuple.
void insertSimpleElement( const XMLCh *elementName , const XMLCh
*elementValue );
```

```
//  Rechercher un élément spécifié dans un tuple et le retourner.
DOMElement* getSpecifyElementByValue( const XMLCh * , const XMLCh * );

//retoune l'élément racine de cet Objet 'Tuple'
DOMElement* getRootElement();

//  Configurer la valeur d'un élément dans un tuple après avoir vérifié
//  l'existance de la valeur et sa redondance.
int setElementValue(const XMLCh * , const XMLCh * );

//  Supprimer un élément d'un tuple.
void removeSimpleElement( const XMLCh * name, const XMLCh * value);

//  Supprime tous les éléments simples qui vérifient le nom mentionné.
void removeSimpleElements( const XMLCh *elementName );

//  Génère aléatoirement un 'id' unique d'un tuple.
static const XMLCh* generateIdTuple( DOMDocument* );

//  Met en place un détecteur d'évènement pour ce tuple
void setEventCatcher( EventCatcher *eventCatcher );
};
```

18. PresenceSystem

C'est la classe mère unissant toutes les autres classes, elle permet de démarrer un client de présence ou un serveur de présence ou une application gérant l'information de présence. Elle contient les principales méthodes suivantes :

```
public class PresenceSystem
{
private:

//  Le détecteur d'évènement associé à l'application.
EventCatcher *m_pEventCatcher;

//  La classe PresenceSubscriptionDLL représence la DLL de la classe
//  PresenceSubscriptionObject, elle représente la M5T SIP Stack.
PresenceSubscriptionDLL* m_pPresence;

//  Le pointeur vers le processus.
HANDLE m_pThread;

//  Le document de présence pricipal.
Document * m_pDocument;

public:

//  Le constructeur.
WINAPI PresenceSystem();

//  Le desctructeur.
~PresenceSystem();

static unsigned long WINAPI startEventCatcher(void* params)
{
      (reinterpret_cast<PresenceSystem*>(params))->ThreadProc();
      return 0;
};
```

```
//  Inialiser le parseur.
int initParser();

//  Terminer le parseur.
int terminateParser();

//  Parser un fichier XML de présence.
int parseFile( XercesDOMParser *parser,
                    const char *inputXMLFileName );

//  Démarrer l'application de gestion de l'information de présence.
void startPresenceAdministration();

//  Démarrer un serveur de présence.
void startPresenceServer();

//  Démarrer un client de présence.
void startPresenceClient();

//  La méthode responsable du contrôle et de la gestion des évènements.
void ThreadProc();

//  Démarrer le processus ou le thread. Il fait appel à la méthode
//  ThreadProc().
void RunEventCatcher();

//  Paramètres :
//    rUser : L'adresse SIP de l'utilisateur dont le publish va être
//     envoyé.
//    rBuffer : La charge utile du publish.
//  Description :
//    Utiliser par le client de présence. Envoyer un publish au serveur de
//     présence.
void SendNotify();

//  Paramètres :
//    rUser : L'adresse SIP de l'utilisateur dont le notify va être
//     configuré.
//    rBuffer : La charge utile du notify reçu.
//  Description :
//    Placer un notify reçu dans le document de présence.
void SetNotify(const char* rUser, const char* rBuffer);

//  Paramètres :
//    rUser : L'adresse SIP de l'utilisateur dont le notify va être
//     envoyé. Si rUser est NULL le notify s'envoie à tous les
//     utilisateur enregistrés.
//    rBuffer : La charge utile du notify.
//  Description :
//    Envoyer un ou plusieurs notifys.
void NotifyUsers( const XMLCh* rNotified);

//  Utiliser par le client de présence. Envoyer une demande de personnel
//   qualifié.
void SendPersonalRequest();

//  Paramètres :
//    rUser : L'adresse SIP de l'utilisateur cible.
//    rBuffer : Le document reçu du client de la demande de personnels
//     qualifiés.
//  Description :
//    Utiliser par le serveur de présenceRépondre à une demande de
//      personnels qualifiés. Chercher les utilisateurs, les enregistrer
//      dans un document XML et envoyer la liste.
```

```
void SendPersonalList( const char* rUser, const char* rBuffer);

//  Au cas du client, se désinscrire auprès du serveur de présence.
//  Au cas d'un serveur, désinscrire tous les clients.
void HandleUnsubscription();

//  Utiliser par le client, s'inscrire auprès du serveur de présence.
void HandleSubscription();

//  Paramètres :
//    rUser : L'utilisateur dont son état de présence va être notifié.
//  Description :
//    Envoyer un rafraîchissement de l'état de présence de rUser aux
//    autres utilisateurs enregistrés.
      void RefreshPresenceStates(const char* rUser);

//  Remettre à zéro le document de présence. Intitialiser les états de
//   présence de tous les utilisateurs sur l'état 'injoignable'.
      void ResetDocument();
};
```