MASTER EN SCIENCES INFORMATIQUES

Evaluation des performances d'une solution Presence distribuée

Delforge, Kristoffer

Award date:
2006

Facultés Universitaires Notre-Dame de la Paix
Institut d'informatique

Année académique 2005-2006

Evaluation des performances d'une solution *Presence* distribuée

Delforge Kristoffer

Mémoire présenté pour l'obtention du grade de
Licencié en informatique

Remerciements

Nous tenons dans un premier temps à remercier notre promoteur, Monsieur Laurent Schumacher, pour son soutien, son aide précieuse et sa compréhension dans les périodes surchargées. Ce mémoire ne pourrait être ce qu'il est sans l'aide de Jean-Louis Sacré, qui nous a consacré de nombreuses soirées afin de réinstaller le serveur et résoudre des problèmes de configuration. Merci également à Nicolas Zastavni qui a répondu aux questions que nous avions lors de la réinstallation du serveur, par l'intermédiaire de Jean-Louis Sacré.

Nous tenons également à remercier Quan Truc Nguyen pour ses conseils de lecture et ses réponses aux problèmes de configuration du serveur. Nous remercions aussi Hugues Van Peteghem pour son aide quant au fonctionnement et l'accès du labo.

Enfin, nous tenons à remercier notre entourage pour son soutien tout au long des études et plus particulièrement au cours de la réalisation de ce mémoire.

Résumé

Ce mémoire a pour but d'évaluer les performances d'une solution de *Presence* distribuée. Les concepts de base de *Presence* ainsi que l'architecture évaluée sont exposés. Une évaluation sur base de critères de qualité logicielle et de systèmes distribués ainsi qu'une comparaison d'algorithme de recherche dans un arbre sont étudiés.

Mots-clés : Indigo Software, SIP, *Presence*, performance, robustesse, systèmes distribués, recherches dans des arbres.

Abstract

The purpose of this thesis is to evaluate the performances of a *Presence* distributed solution. The basic concepts of *Presence* as well as evaluated architecture are exposed. An evaluation on the basis of software distributed system and quality standard and a comparison of algorithm of research in a tree are studied.

Keywords : Indigo Software, SIP, *Presence*, performance, robustness, distributed systems, trees search.

Table des matières

Table des figures

Liste des tableaux

Introduction

Internet s'est ouvert au public au milieu des années 90. Il évolue en permanence dans des domaines variés, aussi bien les télécommunications que l'e-business. Comme les spécialistes le disent, nous ne sommes qu'à son début et le potentiel de l'Internet n'est pas encore totalement exploité. Les services fournis aux utilisateurs se sont développés en masse, et régulièrement des solutions propriétaires sont mises sur le marché. Nous avons connu la mise en place des courriers électroniques, suivi par des messageries instantanées. Dans ce domaine, nous connaissons une panoplie de solutions fermées (Yahoo Messenger, MSN Messenger, ICQ, etc.).

Les utilisateurs de telles messageries souhaitent connaître l'état de présence de leurs contacts. Ceci est un cas particulier de ce que *Presence* peut offrir. Par exemple, un abonné de GSM (*Global System for Mobile Communication*) peut souhaiter être prévenu par un message lorsqu'il ne lui reste que dix minutes d'appel sur son crédit. Ceci est déjà disponible, mais il manque parfois une certaine instantanéité. En effet, Proximus propose ce type de service pour les abonnés, mais cette notification de crédit n'est reçue que deux ou trois heures après cet événement.

Nous avons évoqué l'esprit fermé de certaines solutions proposées par les fournisseurs de logiciels. Ceci empêche une interopérabilité et oblige les utilisateurs soit à faire un choix, soit à se multi-équiper. L'Internet Engineering Task Force (IETF), organisme normatif, développe différents protocoles comme Simple Mail Transfer Protocol (SMTP) pour les mails, HyperText Transfer Protocol (HTTP) pour le web, etc. Le protocole qui nous intéresse dans ce travail est Session Initiation Protocol (SIP) qui définit l'établissement de sessions.

Indigo Software fournit à ses clients les services d'un serveur SIP. Celui-ci a été développé en partie par Jean-Louis Sacré dans son mémoire [19] lors de

l'année académique 2004-2005. Quan Truc Nguyen a, quant à lui, développé des patches pour ce serveur lors de son stage [10] durant l'été 2005, effectué aux Facultés. Le premier, ayant participé aux premiers développements du serveur SIP de la société Indigo Software, a étudié le passage à grande échelle du service dans le cadre de son mémoire [19]. Dans ce travail, il a proposé une architecture pouvant répondre aux problèmes de dimensionnement du système. Le problème principal était le nombre d'utilisateurs qui va en grandissant. Ainsi, il a découpé le problème en deux niveaux : une distribution locale et une distribution globale. La première permet d'augmenter la capacité du service au sein d'une région. N'étant pas suffisant, cet élément est répété à travers le monde. Chaque élément possède une vue partielle du système, un mécanisme de communication entre eux a été précisé. Le second, lors de son stage, a principalement développé le mécanisme de localisation de contact. Il a également proposé des solutions pour le groupement des serveurs en cluster.

Le but du présent mémoire est d'évaluer les performances de la solution développée. Celle-ci doit permettre une distribution à grande échelle du service. Dans le cadre de ce travail, nous avons eu accès au laboratoire de développement du pôle "Réseaux et Sécurité" de l'Institut d'Informatique.

Le laboratoire ayant subi un relooking, il a fallu réinstaller le serveur et les applications que Quan Truc Nguyen a utilisées lors de son stage. Cependant, cette phase a pris beaucoup plus de temps que prévu. En effet, nous avons rencontré pas mal de problèmes de configuration dans un premier temps et, dans un second temps, des problèmes d'installation des outils supplémentaires. Cela a engendré un retard dans l'évaluation pratique qui n'a pu être réalisée. Dès lors, l'analyse des performances a principalement été réalisée théoriquement sur base de critères repris dans les cours de Conception des systèmes distribués et coopératifs [4] et d'Ingénierie du logiciel [6].

Ce mémoire se compose de quatre chapitres. Le premier est une présentation des protocoles SIP et *Presence*, indispensable pour la compréhension du système. Le second chapitre se concentre sur la compréhension du serveur d'Indigo Software. L'évaluation du système établi se compose des troisième et quatrième chapitres. Le premier expose l'évaluation théorique. Dans le quatrième chapitre, nous nous concentrons sur des tests pratiques. Cependant, par manque de temps suite à des retards dans l'installation du serveur, ceux-ci n'ont pu être entrepris. Nous les exposons puisque nous aurions souhaité les réaliser. Nous terminerons ce document par une conclusion, dans laquelle

nous exposerons des pistes pour d'éventuels travaux futurs dans lesquelles nous retrouverons les tests proposés au dernier chapitre.

Le premier chapitre étant un exposé de SIP et *Presence*, nous invitons le lecteur familier de ces concepts à passer ce premier chapitre et à se rendre directement au second chapitre qui présente l'architecture que nous évaluerons.

Le présent mémoire est soumis à un *Non Disclosure Agreement* afin de ne pas divulguer le contenu du travail. De ce fait, ce document ne peut être distribué sans l'accord de la société Indigo Software.

Liste d'acronymes

Cette première section reprend la liste des acronymes utilisés dans ce mémoire. Elle permet au lecteur de s'y référer au cours de sa lecture.

Acronymes	**Significations**
AAL	ATM Adaptation Layer
ACL	Access Control List
ADSL	Asymmetric Digital Subscriber Line
AoR	Address of Record
API	Application Programming Interface
ATM	Asynchronous Transfer Mode
BIND	Berkeley Internet Name Daemon
C-UA	Client User Agent
CIPID	Contact Information Presence Information Data format
CORBA	Common Object Request Broker Architecture
DNS	Domain Name System
ESC	Event State Compositor
FQDN	Fully Qualified Domain Name
GSM	Global System for Mobile communication
HTTP	HyperText Transfer Protocol
IETF	Internet Engineering Task Force
IP	Internet Protocol
ISO	International Organiszation for Standardization
LDAP	Lightweight Directory Access Protocol
LVS	Linux Virtual Server
MIME	Multipurpose Internet Mail Extensions
MSN	Microsoft Network
NAT	Network Address Translation

Acronymes	**Significations**
PA	Presence Agent
PDA	Personal Digital Assistant
PIDF	Presence Information Data Format
POA	Portable Object Adapter
PPP	Point to Point Protocol
PS	Presence Server
PSTN	Public Switched Telephone Network
PUA	Presence User Agent
RFC	Request For Comments
RLI	Resource Lists Information
RLS	Resource Lists Server
RPID	Rich Presence Information Data format
RTP	Real-Time Protocol
RTSP	Real-Time Streaming Protocol
S-UA	Server User Agent
SDP	Session Description Protocol
SIP	Session Initiation Protocol
SMTP	Simple Mail Transfer Protocol
SOA	Start Of a zone of Authority
SSH	Secure Shell
TCP	Transmission Control Protocol
TGV	Train à Grande Vitesse
TTL	Time To Live
UDP	User Datagram Protocol
UACE	User Agent Capability Extension
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
URN	Uniform Resource Name
WI	Watcher Info
XCAP	XML Configuration Access Protocol
XML	eXtensible Markup Language

Chapitre 1

Notions théoriques

Le sujet de ce mémoire est orienté vers *Presence* qui est une extension du protocole SIP. Afin de comprendre *Presence*, les bases de SIP seront exposées dans un premier temps et permettront, ainsi, de mieux cerner les principes de *Presence*. Nous allons donc présenter le protocole SIP avec les différentes entités et types de messages échangés ainsi que le principe de notification défini par le protocole. Certains aspects de SIP ne seront pas abordés et nous laissons le soin au lecteur de se référer à l'ouvrage de Camarillo Gonzalo et Garcia-Martin Miguel A. [1] qui traite le sujet plus amplement.

A la section 1.2, nous verrons *Presence*, extension au protocole SIP et thématique principale de ce mémoire. Nous commencerons cette section par l'introduction du principe de présence. Nous aborderons ensuite les différentes entités ainsi que le format d'échange de données de présence. Nous verrons également que ce format est minimal et qu'il nécessite certaines extensions afin de fournir une image de présence plus complète. Nous terminerons ce chapitre par l'exposé de l'interaction entre les différents services définis dans *Presence*.

1.1 Introduction à SIP

La première section consiste à introduire le protocole SIP afin d'avoir les bases pour comprendre le mécanisme de *Presence*.

SIP est un protocole développé par l'IETF et décrit dans le RFC 3261 [14] (*Request For Comment*). Deux documents sont venus ajouter des fonctionnalités à ce protocole. Il s'agit des RFC 3265 [12] et RFC 3428 [2]. Le premier définit un mécanisme de notification d'événements basé sur SIP, que nous aborderons à la section 1.1.3. Le second définit l'utilisation de SIP pour l'envoi de messages instantanés.

SIP est utilisé pour l'établissement de sessions entre deux entités. Il se base sur le paradigme *Client-Serveur*, le client joignant le serveur. Tout comme le protocole HTTP, qui se base sur le même principe, SIP met en place l'échange de requêtes et réponses directement lisibles par l'être humain, que nous appelons également des transactions. Celles-ci sont constituées d'une requête, de zéro ou plusieurs réponses provisoires et d'une réponse finale. Au cours d'une même session, plusieurs transactions seront échangées. Nous parlons dès lors d'un dialogue qui regroupe ces transactions au sein d'un même contexte.

Les utilisateurs sont identifiés par leur URI (*Uniform Resource Identifier*), équivalent à l'URL (*Uniform Resource Locator*) de HTTP. Le format de cet identifiant doit respecter la structure suivante :

`type:utilisateur@domaine`.

Des éléments optionnels peuvent être greffés comme le numéro de port et des paramètres. La structure devient alors :

`type:utilisateur@domaine:port;paramètres`.

Un exemple d'identifiant peut être :

`sip:kdelforg@info.fundp.ac.be:5060;transport=tcp`.

Dans cet exemple, nous indiquons un paramètre relatif au protocole de transport utilisé pour la session. SIP est un protocole de la couche *Application* dans le modèle TCP-IP. La figure 1.1 montre sa position dans le modèle en couche TCP/IP. Il se situe au même niveau que HTTP, SMTP, DNS, etc. A l'inverse de DNS, la couche transport utilisée par SIP peut être TCP ou UDP.

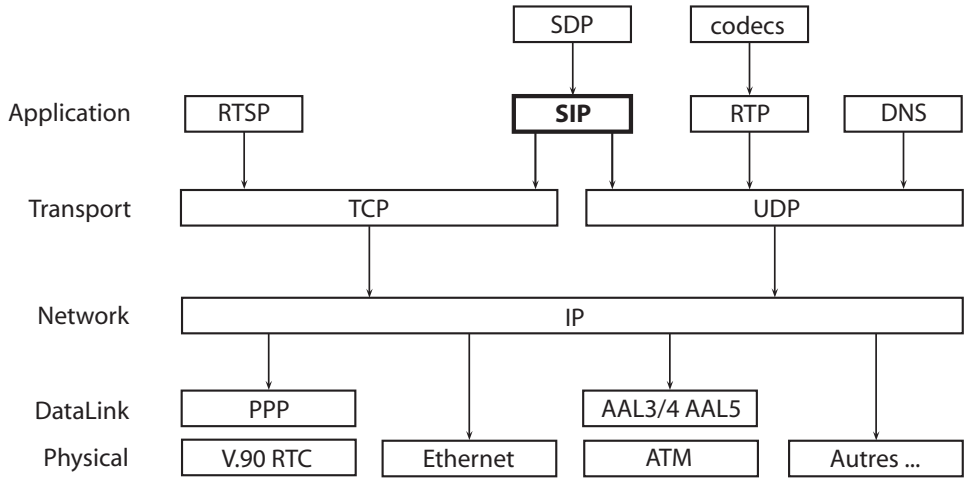


FIG. 1.1 – SIP dans le modèle en couche TCP-IP.

1.1.1 Entités SIP

Sur le réseau, certains équipements, comme des téléphones, sont capables de comprendre le protocole SIP. Ces entités sont classées en cinq catégories en fonction de leur rôle au sein des dialogues.

User Agent

Comme son nom l'indique, cette catégorie regroupe les terminaux des utilisateurs. Ceux-ci peuvent être des téléphones SIP, des ordinateurs de poche, des téléphones mobiles, des applications utilisées sur les ordinateurs (par exemple Windows Messenger), etc.

Nous y distinguons le C-UA (*Client User Agent*) et le S-UA (*Server User Agent*). Le client initie l'appel SIP et envoie sa requête au serveur. Ce dernier enverra sa réponse finale au client, éventuellement précédée de réponses provisoires.

Registrar Server

Le serveur d'enregistrement, ou *Registrar Server*, est contacté par le client afin de s'enregistrer sur le réseau SIP. Le protocole permet une mobilité

accrue de l'utilisateur grâce au principe de l'*Address of Record* (AoR).

Un utilisateur peut, en effet, posséder plusieurs URI : à son travail, à son domicile, auprès d'une organisation, etc. Par exemple, Lance Armstrong peut posséder les trois URI suivants :

`lance.armstrong@team.discovery.com`,
`lance.armstrong@uci.ch`,
`lance.armstrong@aol.com`.

Cependant, l'utilisateur peut souhaiter ne divulguer qu'une de ses URI, son AoR. Idéalement, son AoR sera, parmi ses différents URI, celui qui aura la durée de vie estimée la plus grande. On veillera en effet à ce que cette AoR reste inchangée, même si l'utilisateur change de travail ou de domicile. L'AoR est donc l'URI que l'utilisateur a décidé de communiquer au monde extérieur pour être contacté, son URI public. Les autres URI sont alors associés à cette AoR par le biais de l'enregistrement. Son principe en est simple : l'utilisateur s'enregistre auprès du serveur d'enregistrement gérant le domaine de son AoR avec l'URI qu'il utilise à ce moment-là, par exemple celui de son travail. Lorsqu'un contact tente de le joindre, ce dernier utilise l'AoR de son correspondant. Le serveur de localisation du domaine dont relève l'AoR détermine quel URI il faut utiliser pour joindre cet utilisateur. Ce mécanisme est illustré à la figure 1.2. Ainsi, les contacts ne connaissent qu'un seul URI de l'utilisateur, à savoir l'AoR, et ne doivent pas se soucier de l'URI qu'il utilise pour se connecter.

Dans notre exemple, ce mécanisme permet aux journalistes et aux fans de Lance Armstrong de ne connaître qu'un seul URI du coureur, l'AoR. Ainsi, s'il change d'équipe entre deux saisons, ceux-ci ne doivent pas modifier leur liste de contact.

L'utilisateur peut s'enregistrer simultanément avec plusieurs URI. Le serveur doit alors décider où router la requête. Pour ce faire, il dispose de données indiquant les priorités de routage comme des intervalles de temps, l'expéditeur, le type de sujet, etc.

Location Server

Le serveur de localisation (*Location Server*) est l'entité de stockage des correspondances entre l'AoR et les URI utilisés lors de l'enregistrement. Ré-

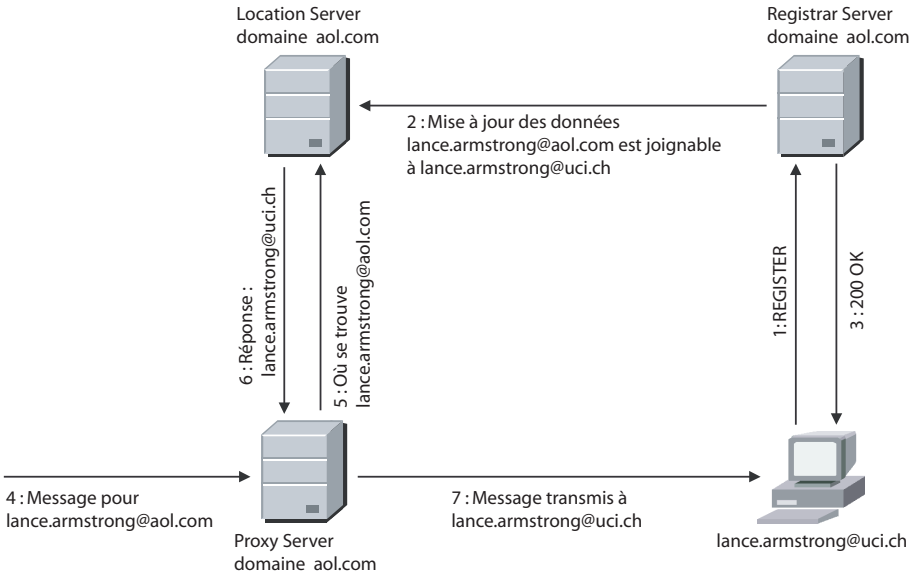


FIG. 1.2 – Illustration du principe de l'AoR.

gulièrement, il s'agit d'une base de données mise à jour par un serveur d'enregistrement et utilisée par un serveur proxy ou un serveur de redirection. De plus, il est généralement implémenté avec le serveur d'enregistrement.

Redirect Server

Le serveur de redirection (*Redirect Server*) n'est contacté que dans le but de connaître la destination réelle d'une requête. En effet, il connaît la correspondance entre l'AoR et les différents contacts possibles, grâce à une communication avec le Location Server. Dès la réception de la réponse, le client peut directement contacter la personne désirée en utilisant l'URI retournée par le serveur de redirection.

Proxy Server

Le serveur proxy (*Proxy Server*) permet de rediriger les messages SIP à la bonne destination. Pour y arriver, il communique avec le serveur de localisation qui connaît l'adresse courante de l'utilisateur ; celui-ci ayant effectué une

requête d'enregistrement de son adresse afin de pouvoir la faire correspondre avec l'adresse publique. Les proxies ont cependant d'autres fonctionnalités, comme la vérification des messages ou l'interprétation de leur destination. La figure 1.3 représente les interactions entre les différents services. La première étape est l'enregistrement de l'utilisateur contacté auprès de son *Registrar Server*. Ce dernier communique avec le *Location Server* afin de mettre à jour les informations. Lorsqu'un message SIP arrive au *Proxy Server*, celui-ci interroge le *Location Server* pour connaître la destination à joindre. La réponse lui permet d'envoyer le message au *User Agent* concerné. Il est à noté que dans ce scénario, les trois serveurs, *Registrar*, *Location* et *Proxy*, appartiennent tous au même domaine.

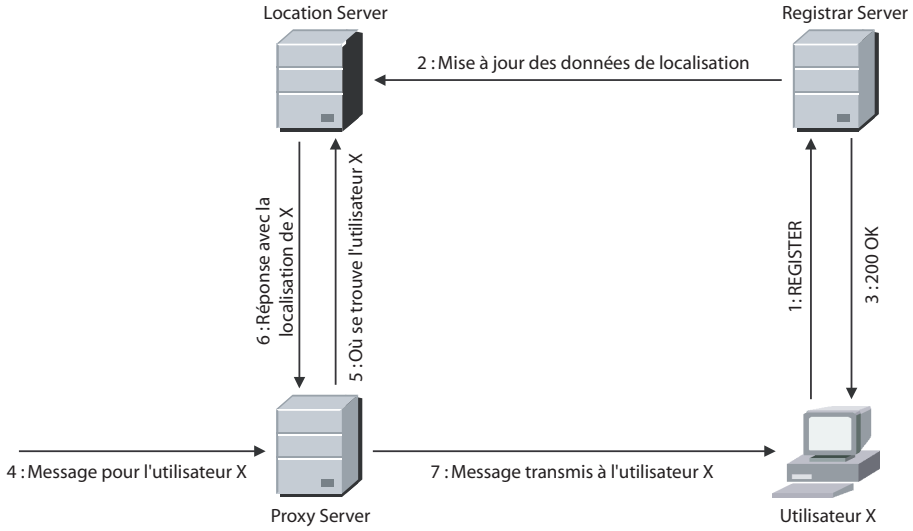


FIG. 1.3 – Interactions entre serveurs.

Parmi les proxies, nous devons distinguer ceux qui gardent de l'information (*stateful*) et ceux qui n'en gardent pas (*stateless*). Les premiers gèrent les transactions. Si une réponse n'est pas envoyée, ils peuvent en solliciter une. Les seconds ne gèrent pas ce principe de transaction. Une absence de réponse n'est pas détectée par ceux-ci étant donné qu'ils ne connaissent pas le contexte de transmission. Au niveau des ressources, les premiers en sont plus demandeurs, ce qui réduit leur capacité de traitement.

Certains proxies définis dans SIP permettent d'envoyer simultanément le message à plusieurs destinations. Ce principe est assuré par les *Proxy Servers*

muni de la fonctionnalité de *forking*, que nous appellerons *Forking Proxies*. Ceci est illustré sous forme de diagramme de séquence (cf. figure 1.4). Le *Forking Proxy* reçoit plusieurs adresses en réponse du *Location Server* et envoie le message aux différentes destinations. L'exemple montre un envoi séquentiel mais celui-ci pourrait également s'effectuer en parallèle.

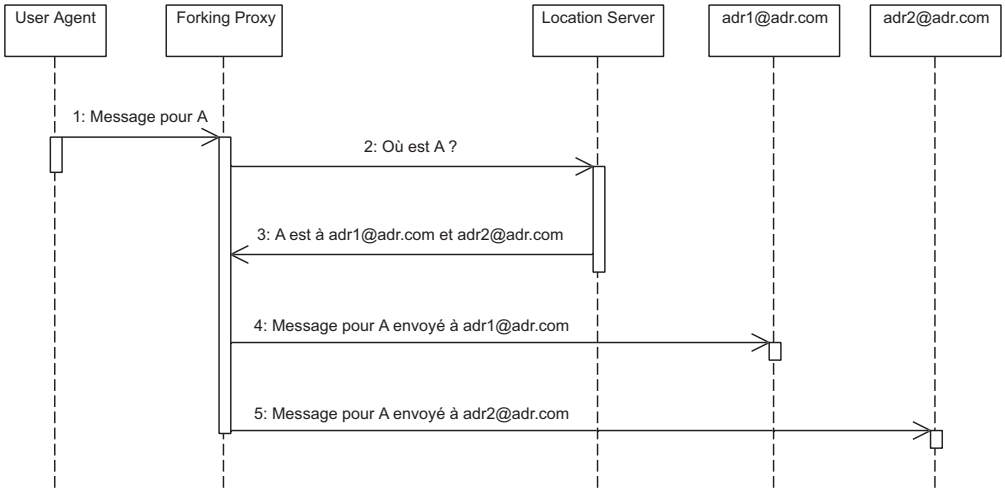


FIG. 1.4 – Interactions entre serveurs avec la fonctionnalité de *Forking*.

1.1.2 Messages SIP

Comme nous l'avons déjà mentionné précédemment, SIP se base sur le protocole HTTP. Il est donc un protocole textuel dont un message prend la forme générale suivante :

```
Start line
A number of headers
Empty line
Optional message body
```

Une transaction étant un échange de messages SIP, elle se compose d'une requête émise par le client, suivie de zéro ou plusieurs réponses provisoires et se termine par une réponse finale.

La première ligne du message est la *start line*. Dans le cas d'une requête, elle est appelée la *request line*, et pour une réponse, il s'agit de la *status line*. La distinction étant amorcée, nous allons les explorer séparément dans les deux sections suivantes. Nous évoquerons ensuite certains en-têtes des messages SIP afin d'avoir un aperçu général avant la notification d'événements et *Presence*.

Requête SIP

La ligne de requête contient la méthode, l'URI et la version de protocole. Les méthodes sont définies dans le tableau 1.1.

Nom de méthode	Signification
ACK	Reconnaît un message
BYE	Termine une session
CANCEL	Annule la requête en cours
INFO	Transporte de la signalisation pour le réseau téléphonique classique (PSTN)
INVITE	Initie une session
NOTIFY	Notifie un client d'un événement
OPTIONS	Interroge un serveur sur ses capacités
PRACK	Reconnaît une réponse provisoire
PUBLISH	Publie des informations sur un serveur
REGISTER	Enregistre la localisation actuelle pour correspondre avec l'URI publique
SUBSCRIBE	Demande à être notifier d'événements
UPDATE	Modifie des caractéristiques d'une session
MESSAGE	Transporte un message instantané
REFER	Demande au serveur d'envoyer une requête

TAB. 1.1 – Liste des méthodes SIP.

Réponse SIP

Une réponse commence par la ligne de statut qui reprend la version du protocole utilisé et le statut de la transaction. Ce dernier est constitué d'un code numérique et d'une phrase explicative qui est destinée à une lecture

humaine. Le tableau 3.1 reprend les différentes classes de réponses et leur code.

Classes	Interprétations
100 - 199	Provisoire
200 - 299	Succès
300 - 399	Redirection
400 - 499	Erreur chez le client
500 - 599	Erreur sur le serveur
600 - 699	Erreur globale

TAB. 1.2 – Liste des codes de réponses SIP par classes.

SIP se basant sur HTTP, la majorité des messages d'erreurs ont été récupérés. Ainsi, le traditionnel code d'erreur 404 correspond à un échec parce que le destinataire n'est pas connu sur le serveur.

Les en-têtes

Comme nous l'avons vu en introduction de cette section, un message est composé d'une ligne de début, suivi d'un certain nombre d'en-têtes et terminé par une ligne vide et un corps de message optionnel. Nous allons nous intéresser aux six en-têtes indispensables pour avoir un message SIP correct : *To*, *From*, *Cseq*, *Call-ID*, *Max-Forward* et *Via*.

Le premier en-tête, *To*, contient l'adresse publique du destinataire. L'en-tête *From* contient l'adresse publique de l'émetteur. Ces deux champs ne servent qu'à des fins de filtrage ou d'utilisation humaine. Ils ne servent donc pas à transporter le message au sein du réseau.

L'en-tête *Cseq* contient un numéro de séquence et le nom de la méthode contenue dans le message. Le groupement des deux permet d'identifier les requêtes et les réponses associées. Il permet donc de repérer les doublons dans une transaction.

L'en-tête *Call-ID* est l'identifiant unique de l'échange de messages. Il permet donc d'identifier un dialogue entre deux utilisateurs.

L'en-tête *Max-Forward* a le même rôle que le *Time To Live* (TTL) de la couche réseau du modèle TCP/IP, à savoir, éviter les boucles dans le réseau.

En effet, chaque proxy décrémente sa valeur d'une unité, comme chaque routeur diminue la valeur du TTL d'une unité également.

Le dernier en-tête obligatoire est *Via*. Celui-ci permet de mémoriser dans le message l'ensemble des proxies par lesquels la requête est passée. Il est utilisé par la réponse afin de parcourir le même chemin à l'envers.

D'autres en-têtes optionnels peuvent être ajoutés. C'est le cas de *Event* que nous décrirons brièvement dans la section suivante, relative à la notification avec SIP.

1.1.3 Notification d'événements avec SIP

SIP étant prévu initialement pour l'établissement de sessions, des extensions permettent d'ajouter des fonctionnalités au protocole. Parmi les extensions connues, nous allons nous pencher sur le principe de la notification d'événements, défini dans le RFC 3265 [12] et sur lequel se base *Presence*.

Ce principe permet aux utilisateurs de connaître une information sur une ressource, par exemple le statut de connexion. Il repose sur le paradigme *subscriber/notifier* et met en application les méthodes SUBSCRIBE et NOTIFY. Deux nouveaux rôles ont été définis : le *subscriber* et le *notifier*. Le premier est un *User Agent* qui émet une requête SUBSCRIBE à l'entité qui gère la ressource dont il souhaite obtenir l'information. La souscription à l'information de la ressource reste valable un certain temps, après lequel le *subscriber* devra émettre à nouveau un message SUBSCRIBE. Le *notifier*, quant à lui, réceptionne la requête SUBSCRIBE et envoie une requête NOTIFY avec l'information demandée. D'autres requêtes de ce type sont envoyées à chaque changement de l'information, tant que la souscription est valable. L'échange de messages est illustré à la figure 1.5.

Lors de la souscription, le *subscriber* spécifie dans l'en-tête *Event* quel type d'informations l'intéresse. Par exemple, si nous voulons connaître le statut de notre boîte de courrier électronique, nous enverrons un message ayant comme en-tête *Event*, la valeur `message-summary`. Tout type d'informations est potentiellement intéressant. Dès lors, en définissant de nouvelles valeurs d'en-têtes, nous pourrions ajouter la notification de l'information désirée. Ces nouvelles valeurs sont définies dans un *event package*.

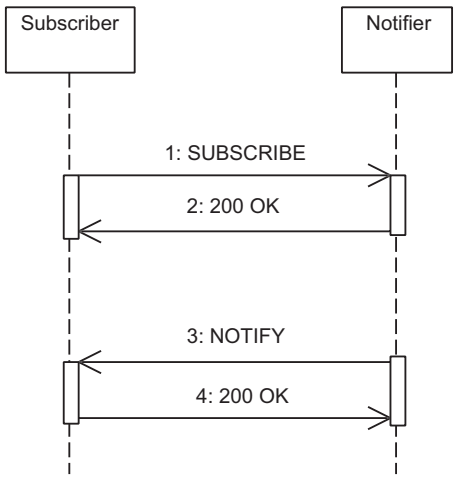


FIG. 1.5 – Principe général de la notification.

L'information qui nous intéresse, dans le cadre de ce travail, est la présence de l'utilisateur. Comme SIP supporte la mobilité des entités, l'information de présence peut varier au cours du temps. Supposons que l'information que nous désirons connaître est le domaine dans lequel l'utilisateur se trouve. Supposons également qu'il voyage en TGV et qu'il risque donc de changer régulièrement de réseau. Nous risquons dès lors d'obtenir un grand nombre de messages notifiant les changements alors que la fréquence de changement est peut-être trop importante pour l'usage que nous voulons en faire. L'extension de notification d'événements de SIP définit un mécanisme, l'*Event Throttling*, qui permet au *subscriber* de définir un intervalle de temps durant lequel il ne désire pas recevoir de notification. Le *notifier* enverra alors l'information à la fin de cet intervalle. Ceci permet, entre autres, d'éviter une utilisation trop importante de la bande passante.

1.2 *Presence* : extension de SIP

1.2.1 Notion de présence

La notion de présence est un principe utilisé inconsciemment par beaucoup de surfeurs. En effet, le fait de connaître le statut de nos contacts dans les messageries instantanées comme Windows Messenger, Skype ou d'autres, en est un exemple. Ce principe nous permet donc de savoir si notre contact est occupé, en communication téléphonique, absent, voire parti manger ; ceci étant le statut de connexion de nos contacts. Cependant, lorsque nous parlons de présence, il s'agit bien plus que de connaître l'état des contacts. En effet, il est possible de définir nos préférences, nos capacités de communication. C'est ainsi que nous pouvons connaître dans la plupart des logiciels de messagerie instantanée la possibilité d'établir une vidéoconférence avec nos contacts. Ceci est un exemple simple et courant de nos jours, mais d'autres éléments peuvent être connus grâce à la présence. Il est ainsi possible de définir quels sont les codecs audio que notre application supporte. Il n'est pas courant dans les messageries connues de définir une telle propriété. En effet, étant propriétaires, ces messageries ne sont pas interopérables et ne proposent pas de travailler avec d'autres codecs. Il est donc inutile dans ces cas de définir une telle information.

La présence, c'est également la possibilité d'être notifié à propos de changements. En introduisant ce mémoire, nous évoquions la possibilité d'être prévenu lorsque le solde de minutes sur un abonnement de GSM atteint la limite des dix minutes. Ceci est un exemple parmi d'autres. Nous pouvons ainsi être prévenus de la mise en communication d'un contact, ce qui nous permet de ne pas en initier une avec cette personne tant que sa communication n'est pas terminée.

Comme nous pouvons le remarquer, ce service fait appel à la notification d'événements que nous avons vue à la section 1.1.3. Les informations de présence étant de plus en plus sollicitées, l'IETF a défini une extension, *Presence*, au protocole SIP que nous avons introduit à la section 1.1. Le RFC 3856 [15] définit l'*event package presence* à spécifier dans l'en-tête *Event* d'un message SIP. Nous verrons dans un premier temps les différents rôles définis dans *Presence*. Nous verrons ensuite le format de données, *Presence Information Data Format* (PIDF), utilisé pour transmettre les informations de présence, de manière uniforme. Etant défini dans le RFC 3863 [23], ce

format est utilisable par les protocoles supportant l'échange de l'information de présence, permettant ainsi une interopérabilité de système. Le corps d'un message SIP utilisant l'encodage MIME (*Multipurpose Internet Mail Extensions*), le format XML (*eXtensible Markup Language*) est utilisé pour représenter l'information de présence. Nous verrons également des extensions du format PIDF.

1.2.2 Les rôles définis dans *Presence*

Avant d'expliquer les techniques utilisées pour transmettre les informations, nous allons voir les différents rôles définis dans le framework de *Presence* : le *presentity*, le *Presence User Agent* (PUA), le *Presence Agent* (PA), le *Presence Server* (PS) et le *watcher*.

Le *presentity*, terme abrégé de *presence entity*, est l'entité qui fournit l'information de présence. Le *presentity* dispose généralement de plusieurs périphériques, un téléphone, un ordinateur, un PDA (*Personal Digital Assistant*), etc., fournissant l'information désirée. Un tel périphérique est appelé PUA dans la terminologie *Presence*. Le PA reçoit les informations de présence de la part des différents PUA. Grâce à l'information collectée de tous les PUA d'un *presentity*, le PA possède une image complète de la présence du *presentity*. Le PS peut comprendre un PA ou servir de proxy pour les messages SUBSCRIBE.

Le dernier rôle est celui de *watcher*. Celui-ci est l'utilisateur qui récupère l'information de présence. Il en existe deux types : le *fetcher* et le *subscribed watcher*. Le premier type est celui qui demande de l'information à un instant donné. Le second type souhaite être informé des changements de l'information de présence du *presentity*.

Nous venons de voir chacun des rôles définis dans *Presence*. Cependant, certains rôles peuvent être regroupés au sein d'un même logiciel, comme par exemple, le PS qui regroupe les fonctions du PA. C'est ainsi que les applications clientes auront les fonctionnalités du *watcher* et du *presentity*. En effet, un logiciel comme Windows Messenger permet aux utilisateurs de connaître le statut des contacts et leur permet également de définir leur propre statut qui sera transmis aux contacts, par l'intermédiaire d'un serveur suivant le mode de travail de l'application.

Le *presentity* émet son information au moyen d'une transaction PUBLISH. Le PA reçoit ces données, les traite selon un mécanisme que nous ne voyons pas dans le cadre de cette étude. Il peut alors transmettre le résultat aux *watchers* par une transaction NOTIFY. Afin d'avoir un format uniforme au sein des applications, le format PIDF, que nous verrons dans la section suivante, est utilisé pour l'échange d'information entre les entités.

Nous terminons cette présentation des rôles par la figure 1.6. Nous voyons la présence d'un PA, d'un *presentity* et de deux *watchers*. Le premier *watcher* souscrit à l'information de présence du *presentity* (1). Le PA notifie alors avec l'état par défaut de présence du *presentity* (2). Le *presentity* modifie son état (3), par exemple lorsqu'il se connecte. Le PA avertit alors les *watchers* inscrits avec le nouvel état du *presentity* (4). Le second *watcher* souscrit à son tour à cette information (5) et reçoit un NOTIFY (6) reprenant le dernier état de présence connu par le PA au sujet du *presentity*. Lorsque le *presentity* modifie à nouveau son état (7), le PA notifie chaque *watcher* avec la nouvelle information (8 et 9).

1.2.3 Presence Information Data Format

La section qui nous occupe maintenant concerne le format des données échangées entre les différentes entités, le PIDF. Comme nous l'avons mentionné au début de la section, le format PIDF n'est pas défini pour un seul protocole. Il est donc possible de l'utiliser avec d'autres protocoles supportant l'échange de l'information de présence. Ce format est défini dans le RFC 3863 [23], ne reprenant que les caractéristiques de base de la présence. Nous verrons, dans la section 1.2.5, que des extensions à ce format ont été définies afin d'ajouter des caractéristiques et de permettre une image plus précise de la présence des *presentities*. De ce fait, le format PIDF ne définit que les statuts `open` et `close`, respectivement pour spécifier l'état en-ligne ou hors-ligne. Les extensions permettent de spécifier de nouveaux états de connexion, comme absent ou au téléphone.

Etant destiné à des transferts dans des transactions SIP, le format est transportable comme une application MIME. Ainsi, le format XML est compatible aux besoins. Afin de préciser le but des données transmises dans l'en-tête d'un message, un nouveau type MIME a été défini, il s'agit de `application/pidf+xml`. Les données sont donc transmises à l'aide de requêtes PUBLISH, SUBSCRIBE ou NOTIFY.

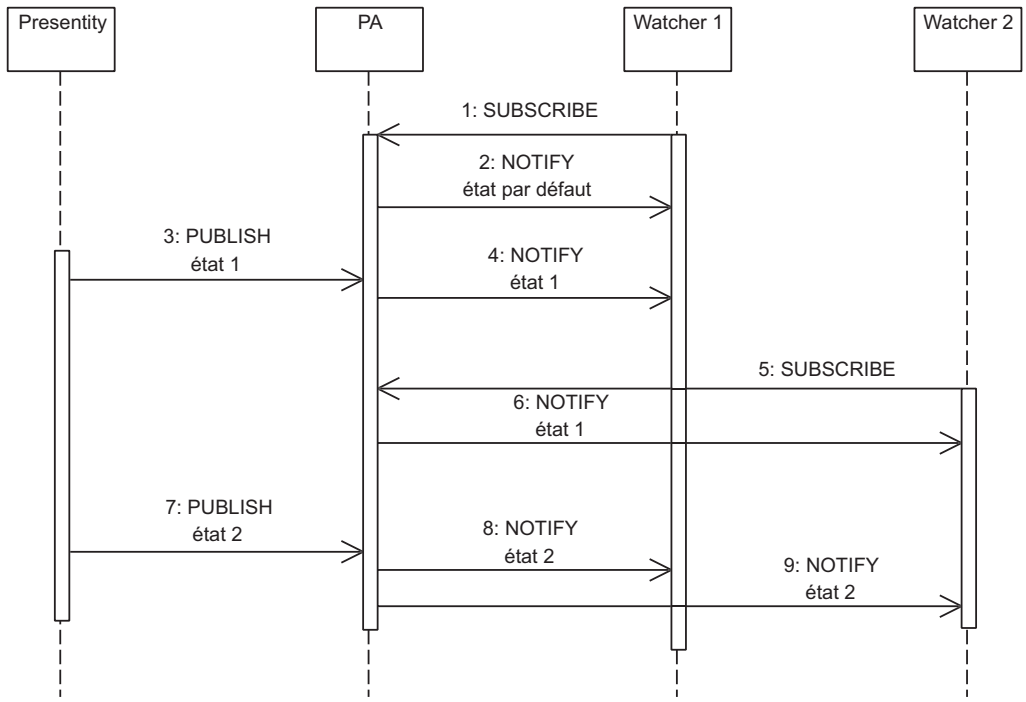


FIG. 1.6 – Illustration des méthodes PUBLISH et NOTIFY.

Chaque élément de l'information est décrit dans un champ `tuple` qui reprend le statut du *presentity* et des éléments optionnels comme `contact`, `note` et `timestamp`, ainsi que des éléments venant d'extensions au format PIDF.

La figure 1.7 montre un exemple de document PIDF dans lequel le *presentity* est identifié par l'adresse `eddy.merckx@uci.ch`. Celui-ci spécifie qu'il est en ligne et précise son numéro de téléphone. La note indique qu'il s'agit du téléphone familial.

1.2.4 SIP *Presence* Data Model

Le format PIDF est peu fourni en éléments et nous verrons des extensions de ce format à la section 1.2.5. Cependant, le draft "A Data Model for

```
<?xml version="1.0" encoding="UTF-8"?>
<presence xmlns="urn:ietf:params:xml:ns:pidf"
          entity="pres:eddy.merckx@uci.com">
  <tuple id="lauci56">
    <status>
      <basic>open</basic>
    </status>
    <contact priority="">tel:+3210439129</contact>
    <note>Téléphone familial</note>
  </tuple>
</presence>
```

FIG. 1.7 – Exemple de document PIDF.

Presence" [16] définit un modèle qui se concentre sur les trois caractéristiques de base d'un *presentity* : les services, les périphériques et la personne.

Le modèle décrit l'information de présence sous forme de quatre éléments : les trois caractéristiques énoncées ci-dessus et l'URI du *presentity*. Le dernier élément est le seul qui ne contient pas d'informations descriptives, à l'inverse des trois précédents qui possèdent l'information relative aux trois caractéristiques. En effet, l'élément URI ne contient que l'identifiant du *presentity*.

L'élément `person`, correspondant à la personne, contient les données et le statut de l'utilisateur. L'élément `service`, représentant un service, décrit les caractéristiques du service et également son statut.

Ces deux premiers éléments contiennent de l'information statique et de l'information dynamique. Les données statiques sont par exemple les caractéristiques de la personne ou du service, alors que les données dynamiques peuvent être le statut. Par exemple, le statut d'un service peut indiquer le souhait qu'a l'utilisateur d'utiliser ce service.

Le dernier type d'élément est `device` qui reprend les informations d'un périphérique. Par exemple, en donnée statique nous pourrons retrouver le type de codecs supportés et autres caractéristiques physiques. Une donnée dynamique sera, par exemple, l'avertissement que l'utilisateur est en communication à un instant donné.

Nous venons de donner les principes de bases de ce modèle et invitons le lecteur intéressé à consulter [1]. Ce modèle a été défini après le format PIDF. La correspondance entre le modèle que nous venons de présenter et le format PIDF est illustrée à la figure 1.8[1]. Le format PIDF a été complété. L'élément racine `presence` contient un attribut `entity` qui contient l'URI du *presentity*.

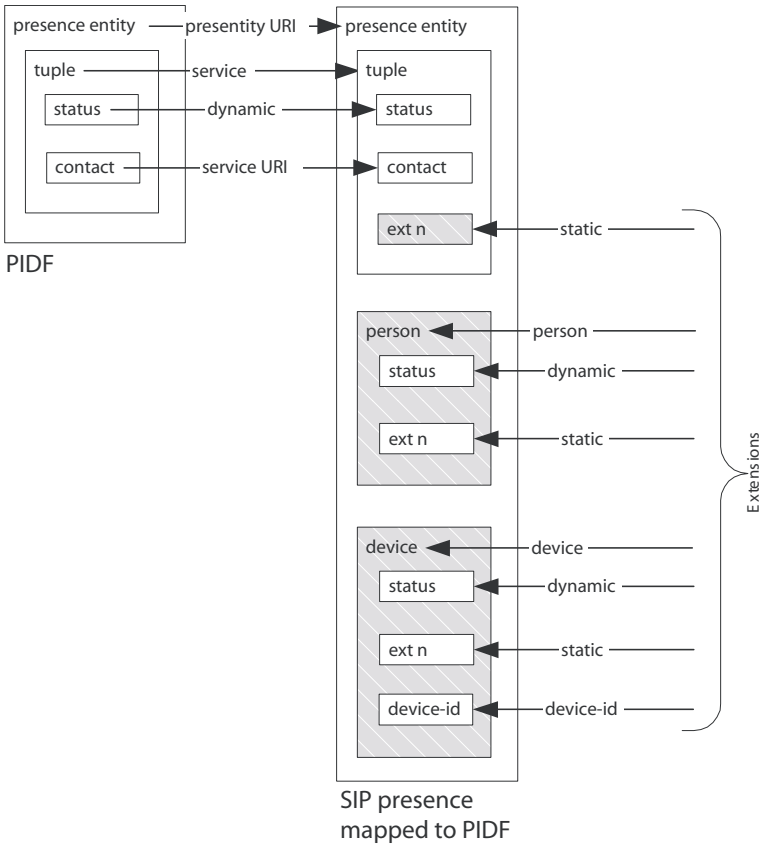


FIG. 1.8 – Correspondance entre le modèle de donnée Presence.

L'élément `tuple` du format PIDF reste présent dans le modèle et est associé à la notion de service que nous venons de présenter. Les notions de personne et périphériques nécessitent l'ajout des éléments `person` et `device`, respectivement.

[1]Figure tirée de l'ouvrage [1], à la page 302

Chaque élément contient des fils `status` et `ext n`, représentant respectivement ses données dynamiques et statiques. Ces dernières sont vues comme des extensions de l'élément.

L'élément `device`, reprenant des informations sur le périphérique, possède également un identifiant contenu dans un élément `device-id`. Celui-ci doit respecter un format *Uniform Resource Name* (URN) parmi ceux définis dans le RFC 2141 [8].

1.2.5 Extensions à PIDF

Nous avons vu que le format PIDF proposait les informations minimales de présence. Certaines extensions ont été développées afin de pouvoir compléter les données de présence. Les extensions que nous allons exposer sont : *Rich Presence Information Data Format* (RPID), *Contact Information in Presence Information Data Format* (CIPID), *Timed Presence Extension to the PIDF* et *User Agent Capability Extension to the Presence Information Data Format* (UACE). Chacune de ces extensions doit permettre une compatibilité arrière. En effet, un composant ne comprenant aucune de celles-ci doit être capable de comprendre l'information minimale définie dans PIDF. Chaque extension rajoute donc des éléments XML à la sémantique de PIDF.

RPID

Le RPID est une extension au format PIDF, qui permet d'enrichir l'information de présence. Cette extension est définie dans le draft "RPID : Rich Presence Extensions to the Presence Information Data Format (PIDF)" [20].

Le détail de l'information peut être encodé manuellement, mais RPID fournit un mécanisme de mise à jour automatique. Par exemple, si le calendrier indique une réunion de 14 heures à 16 heures, il enverra l'information `busy` à partir de 14 heures. Ce format permet d'étendre chacun des trois éléments vus précédemment.

Il permet de spécifier des informations comme les activités du *presentity*, son humeur, les caractéristiques du lieu dans lequel il se trouve, le type de relation qu'il entretient avec d'autres contacts. L'extension permet également de spécifier quel type de média peut être utilisé afin qu'il ne soit pas possible

d'intercepter la communication[2]. Ses possibilités et d'autres sont disponibles par les éléments suivants : `activities`, `class`, `mood`, `place-is`, `place-type`, `privacy`, `relationship`, `service-class`, `sphere`, `status-icon`, `time-offset` et `user-input`. Ce dernier a également un attribut qui est défini : `last-input`. Le premier élément cité, `activities`, peut contenir un ou plusieurs fils `activity`, indiquant l'activité en cours.

CIPID

Le CIPID, défini dans le draft "CIPID : Contact Information in Presence Information Data Format" [21], permet d'ajouter des informations sur le *presentity* ou l'élément `tuple`. Pour ce faire, il ajoute les six éléments suivants : `card`, `display-name`, `homepage`, `icon`, `map` et `sound`.

Ces éléments permettent de spécifier des informations comme une carte de visite, l'adresse de la page personnelle du *presentity*, l'image le représentant, le son à jouer lorsque nous recevons un appel de cette personne ou un plan d'accès au lieu où il se trouve au moment de la publication des données.

Timed Presence Extension

Les extensions, présentées ci-dessus, ne permettent qu'une connaissance actuelle de la présence d'un *presentity*. La connaissance d'actions futures des contacts peut nous amener à revoir notre planification de communication. Par exemple, si nous désirons contacter un ami qui publie une information sur sa prochaine conférence de travail, nous pouvons décider de le contacter immédiatement s'il s'agit d'un appel bref, sinon nous pouvons reporter notre communication.

Ce principe est actuellement défini dans le draft, "Timed Presence Extensions to the Presence Information Data Format (PIDF) to Indicate Status Information for Past and Future Time Intervals" [22]. Il permet ainsi aux *presentities* de spécifier les actions du passé proche ou du futur. Il définit un nouvel élément, `timed-status`, faisant partie du `tuple` définit dans le PIDF. Un attribut obligatoire est le `from`, représentant le début d'un événement, alors que la fin de celui-ci est encodée dans l'attribut `until`, qui est optionnel.

[2]La nature exacte de l'interception n'est pas précisée dans notre source [1]

UACE

Nous avons vu que l'extension RPID permet de spécifier quel type de média peut être utilisé sans qu'une tierce personne ne puisse intercepter la communication grâce à l'élément `privacy`. SIP permet également de préciser quelles sont les capacités des UA dans les messages transmis. Ainsi, il est possible d'annoncer le support de l'audio mais pas de la vidéo dans une requête SIP. Le principe de *Presence* permet également de transmettre des informations sur les services supportés ou les limites des périphériques. Deux éléments sont donc disponibles dans cette extension, à savoir `servcaps` et `devcaps`, respectivement pour les services et les périphériques.

Le premier élément, `servcaps`, doit être intégré dans l'élément `tuple` et peut lui-même contenir un certain nombre d'éléments. Nous ne les exposerons pas dans le cadre de cette introduction aux principes. Nous pouvons cependant citer les exemples de `audio`, `video`, `languages` et `event-packages`. Les trois premiers sont explicites et nous nous attardons quelque peu sur le dernier. Celui-ci répertorie les *Event Packages* supportés par le service concerné. Il est courant qu'une application ne supporte pas l'intégralité des packages existants. De plus, au sein d'une même application, certains services peuvent ne gérer qu'une partie des packages supportés par celle-ci. Il s'agit de ces derniers que nous trouverons présents dans l'élément `event-packages`.

Le second élément, `devcaps`, est rattaché à un élément `device`. Trois éléments peuvent y être définis : `mobility`, `priority` et `description`. Le premier indique le type de périphérique concerné (mobile ou fixe). Le second indique la priorité de l'appel qui sera géré par le périphérique. Le dernier permet de donner une description textuelle du périphérique.

1.2.6 Interactions entre entités

La dernière section de ce premier chapitre concerne les interactions entre les différentes entités. Nous avons vu à la section 1.2.2 le mécanisme de base de la publication et de la notification. Ceci indiquait déjà certaines interactions que nous allons approfondir dans cette section.

Un *presentity* envoie des requêtes PUBLISH au PA qui lui répond avec la réponse 200 OK. Ces requêtes permettent de spécifier l'état de présence du

presentity. La réponse permet au *presentity* de savoir que le PA a bien reçu l'information publiée.

A la réception d'une requête PUBLISH, outre la réponse 200 OK envoyée au *presentity*, le PA envoie une requête NOTIFY à chaque *watcher* ayant souscrit pour cette information. Il est donc indispensable que les *watchers* émettent un message SUBSCRIBE auquel le PA répond avec un 200 OK. Ainsi, le *watcher* est enregistré auprès du PA pour connaître l'information de présence du *presentity*. A chaque *notify* reçu, le *watcher* répond avec un 200 OK pour signaler qu'il a bien reçu les informations.

Nous venons de présenter les méthodes utilisées pour communiquer entre les différentes entités. Cependant, dans ce que nous avons décrit, le *presentity* ne connaît pas les *watchers* ayant souscrit à son information de présence. L'IETF a élaboré le RFC 3857 [17] définissant un *event package Watcher Info*, utilisable avec n'importe quel autre *event package*. Dans le cas de *Presence*, l'en-tête *event* d'un message SIP aura comme valeur `presence.winfo`.

Le *presentity* s'authentifie auprès du PA. Celui-ci lui enverra une requête NOTIFY qui contient la liste des *watchers* désirant connaître son information de présence. Une requête du même type sera envoyée à chaque changement de la liste. Ce mécanisme permet dès lors de gérer des autorisations de souscription. Ainsi, l'utilisateur peut empêcher certains de connaître son information de présence et en autoriser d'autres. Les données transmises dans les messages sont également décrites à l'aide du langage XML.

Le mécanisme de souscription du *watcher* auprès du PA s'avère être relativement faible en terme de performance. En effet, si le *watcher* souhaite obtenir l'information de présence d'une personne, il doit émettre une requête SUBSCRIBE et reçoit une requête NOTIFY. A ces deux messages, deux réponses 200 OK sont échangées pour confirmer la réception de la requête. Si le *watcher* désire connaître l'information d'un deuxième *presentity*, il devra répéter ces opérations. Dès lors, pour chaque *presentity* dont nous souhaitons connaître l'information de présence, nous devons échanger un minimum de quatre messages SIP. Au plus nous avons de contacts dont nous souhaitons connaître l'information de présence, au plus un grand nombre de messages sera transmis. Afin de résoudre ce problème, l'IETF a défini les *Resource Lists Information* [13] qui contiennent la liste des URI des contacts de la personne.

Pour gérer cette liste, une nouvelle entité est définie, le *Resource List Server* (RLS). Dans le concept de *Presence*, nous parlerons également de

presence list. Plus précisément, cette liste contient la liste des *presentities* dont nous souhaitons connaître l'information de présence. Par exemple, dans MSN Messenger, il s'agit de la liste d'amis que nous possédons.

Le *watcher* n'émet plus qu'une seule requête SUBSCRIBE pour tous les *presentities*. Le RLS réceptionne cette requête et répond avec un 200 OK. Il se charge lui-même de contacter les PA de l'ensemble des *presentities* présents dans la liste. Nous diminuons ainsi considérablement le nombre de messages échangés entre le réseau et l'utilisateur final. Dès que le RLS reçoit les requêtes NOTIFY des différents PA, il répond à chacun d'eux par un 200 OK et répond ensuite au *watcher* avec un NOTIFY reprenant l'ensemble des informations de présence des contacts présents dans la liste.

Cette gestion centralisée de la liste de contacts permet à l'utilisateur final d'être nomade et d'utiliser d'autres terminaux tout en gardant les mêmes informations.

Nous venons de voir la gestion centralisée des listes de présence. Le terme gestion implique ajout, suppression de contacts, etc. Cela peut s'effectuer à l'aide d'une page web. Cependant, le contenu des messages HTTP n'est pas standardisé, ce qui pousse les sociétés à développer des solutions propriétaires et à rendre, de ce fait, leurs solutions incompatibles, au niveau de la communication, avec d'autres. Afin de résoudre ce problème d'interopérabilité, l'IETF est en train de développer une solution : *XML Configuration Access Protocol* (XCAP), défini dans un draft [18].

Ce mécanisme permet la création, la suppression et la modification de fichiers sur un serveur, en faisant correspondre une action à des méthodes HTTP. Lorsque nous parlons de modifications, il s'agit d'un élément ou de son attribut. Il est également possible de récupérer, à un moment donné, un document dans sa totalité ou en partie.

Nous verrons dans le prochain chapitre où se situe XCAP dans l'architecture.

Chapitre 2

Architecture évaluée

Le premier chapitre nous a permis d'introduire le protocole SIP et le concept de *Presence*. Ceux-ci sont à la base d'un serveur que nous évaluons dans le cadre de ce mémoire. Celui-ci a été conçu par la société Indigo Software Inc. Jean-Louis Sacré, engagé par celle-ci, a participé au développement de ce système. Dans le cadre de son mémoire de fin d'étude en licence, il a proposé une architecture permettant la mise à l'échelle du système. Quan Truc Nguyen a participé au développement de cette solution dans le cadre de son stage effectué en 2005 au sein du pôle "Réseaux et Sécurité"de l'Institut d'Informatique.

Dans ce chapitre, nous allons exposer la problématique d'un système distribué à grande échelle. Après cette introduction, nous évoquerons l'architecture définie théoriquement. Cependant, par manque de temps, Jean-Louis Sacré n'a pas pu implémenter celle-ci. Les travaux de Quan Truc Nguyen seront présentés dans cette troisième section.

2.1 Problématique

Dans cette première section, nous allons exposer la problématique du serveur que nous évaluons dans le cadre de ce mémoire. Le problème, que l'auteur a dû résoudre dans le cadre des ses recherches [19], est la gestion d'un nombre grandissant d'utilisateurs de services en ligne. En effet, avec la démocratisation des prix et l'arrivée de l'Internet haut débit dans les foyers, de plus en plus de personnes ont accès à ces services. Les serveurs sollicités gèrent donc un grand nombre d'utilisateurs. L'article [25] évoque plus de 70 millions de connexions simultanées. L'article [3], écrit un an plus tard, évoque la présence de près de 83 millions d'utilisateurs. Le site MSN Advertising [26] évoque quant à lui une étude montrant une augmentation de 26% des utilisateurs de MSN Messenger entre 2004 et 2005, ce qui nous montre l'intérêt grandissant pour ce type de service, et nous laisse penser à une augmentation aussi significative à l'avenir. Tous ces articles sont la preuve qu'il faut étendre les serveurs pour supporter un plus grand nombre d'utilisateurs.

Les utilisateurs n'étant pas regroupés dans une même région, leur répartition à travers le monde demande également l'étude d'un déploiement à grande échelle des systèmes. Dans ce cadre, différentes stratégies peuvent être envisagées pour la gestion des données. Par exemple, le mode centralisé et le mode migré ont été approchés dans l'étude de Jean-Louis Sacré.

Le mode migré propose aux utilisateurs de gérer les informations. Nous le représentons à la figure 2.1, sous forme de diagramme de séquence. Ce paradigme nécessite plus de traitement au niveau de l'application de l'utilisateur et, par conséquent, demande moins de puissance de traitement au niveau du serveur.

L'exemple de la figure 2.1 représente deux *watchers* qui souscrivent au *presentity*. Ce dernier répond à chacun en notifiant son statut (messages 3 et 7). La notification pour le *watcher* B indique un état hors ligne étant donné un rejet de la souscription. Par après, le *presentity* accepte le *watcher* B et lui notifie son statut qui est connecté.

Le mode centralisé, également représenté par un diagramme de séquence à la figure 2.2, implique une gestion plus importante, et donc plus complexe, au niveau du serveur et permet aux utilisateurs de travailler avec des terminaux moins puissants.

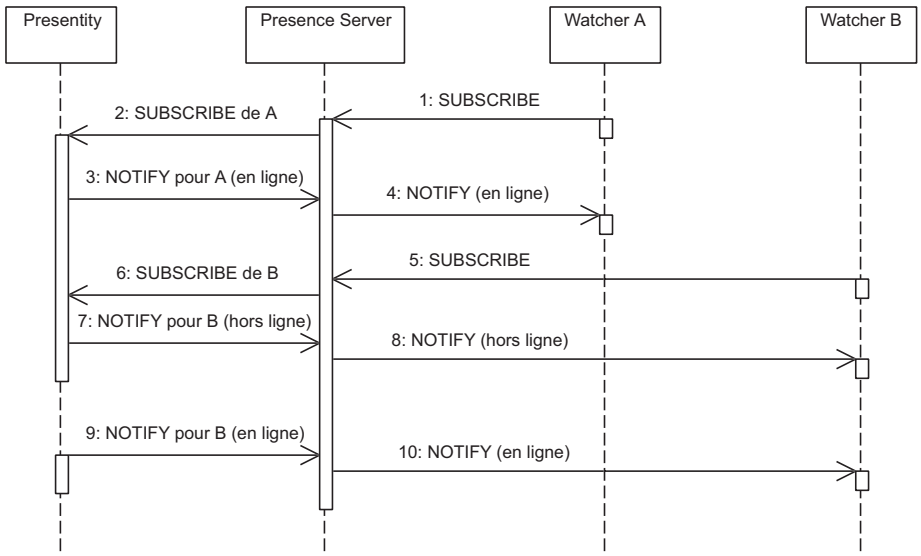


FIG. 2.1 – Représentation d'un échange de messages en mode migré.

Dans les deux cas, le serveur connaît la liste des contacts de la personne. Cependant, dans le mode centralisé, le serveur connaît également l'information de présence. En effet, l'exemple de la figure 2.2 montre l'échange de messages contenant l'information de présence du *presentity* (5 et 12). Nous apercevons également l'échange de messages de politique d'acceptation des souscriptions entre le *presentity* et le *Presence server* (3, 8 et 10). Lors de la réception du message 8 par le *Presence server*, ce dernier notifie le *watcher* avec l'état par défaut (*offline*). Lorsque le *presentity* accepte la souscription (message 10), le serveur notifie le *watcher* avec l'état actuel du *presentity*. Le message 12 représente la déconnexion du *presentity* et entraîne la notification à chaque *watcher* du nouvel état de présence (*offline*).

Dans cet exemple, nous observons que le *presentity* dialogue avec son serveur et ne communique plus directement avec les *watchers* ayant souscrit à son information de présence.

Nous pouvons relever deux différences majeures entre les deux modes : la complexité du serveur et la gestion de l'information de présence. Le serveur en mode centralisé gère plus d'informations. Il nécessite donc plus de ressources et est plus complexe que le serveur en mode migré. La gestion de l'information

en mode migré est réalisée par les utilisateurs. Cela requiert des machines plus puissantes que celles demandées en mode centralisé. De plus, l'utilisateur est maître de la diffusion de son information de présence dans le mode migré, alors qu'il en délègue la responsabilité au serveur dans le mode centralisé.

Dans [19], l'auteur a opté pour le second paradigme, le mode centralisé. En effet, il n'était pas envisageable de développer une solution qui demanderait aux utilisateurs d'adapter le comportement de leurs applications. Ceci relève de la transparence vis-à-vis du client, qui est l'une des contraintes imposées par la société Indigo Software, propriétaire du serveur. L'autre contrainte demande à la solution d'être évolutive, ce qui signifie que celle-ci doit être facilement modifiable afin d'intégrer de nouvelles fonctionnalités.

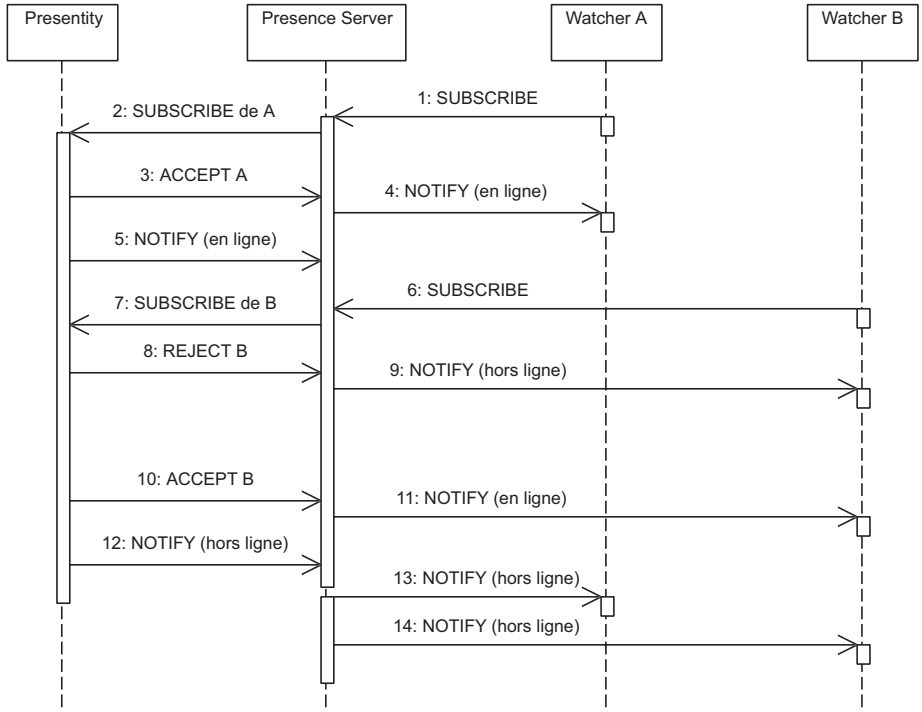


FIG. 2.2 – Représentation d'un échange de message en mode centralisé.

Le PS doit fournir les trois fonctionnalités suivantes : souscription, autorisation et notification. En effet, le serveur doit permettre aux *watchers* de souscrire à l'information de présence d'un *presentity* qui doit pouvoir les ac-

cepter ou les refuser. Ensuite, le *presentity* doit pouvoir prévenir ses contacts des changements de présence grâce à des messages de notification.

Les fonctionnalités sont développées à l'aide de différents services qui seront ajoutés au serveur existant à l'aide de modules. L'IETF a déjà défini certains services qui ont été repris dans le cadre de l'étude de Jean-Louis Sacré. Ceux-ci sont : le *Presence Agent* [15] (PA), l'*Event State Compositor* [11] (ESC), le *XCAP Server* [18] (XCAP) et le *Watcher Info* [17] (WI). D'autres services existent mais ceux cités précédemment forment un ensemble complet pour la gestion de l'état de présence.

Ces services et leurs interactions avec les utilisateurs sont déjà définis dans des RFC et des drafts. Cependant, l'échange de messages entre les services n'est pas standardisé. Un des buts du mémoire [19] était d'analyser ces transactions.

Dans la solution établie, tous les services ne communiquent pas avec les autres. Le PA envoie des messages au WI lorsqu'il reçoit une souscription pour un *presentity* et en reçoit de l'ESC et du serveur XCAP. L'ESC prévient le PA lorsque le *presentity* change son état de présence afin de prévenir les différents *watchers*. De même, le serveur XCAP reçoit du *presentity* les messages d'autorisation relatifs aux souscriptions reçues. Il peut alors les transmettre au PA qui sert d'intermédiaire avec les *watchers*.

La gestion optimale de ces échanges entre services serait d'exécuter chacun d'eux au sein d'une même application. Cependant, dans un souci d'extension du service, il est demandé d'être flexible en termes de taille afin de gérer un grand nombre d'utilisateurs. Dans ce cas, chaque service peut tourner sur des stations différentes et générer des échanges de paquets sur le réseau afin de communiquer. Il est alors indispensable de réduire ces transactions pour augmenter l'efficacité de la solution établie.

De plus, lorsque nous distribuons les services, il est possible d'en dédoubler, c'est-à-dire lancer plusieurs fois le même service afin d'augmenter la charge supportable. Dans ce cas, il est indispensable que les services aient connaissance de tous les autres services lancés et ne communiquent qu'avec celui connaissant le client concerné. Dans la pratique, nous dupliquerons plus souvent des services ayant une charge plus importante. Ainsi, il sera courant de rencontrer plusieurs PA relevant d'un seul et même serveur ESC et également d'un seul et même serveur XCAP.

Nous venons d'aborder l'instanciation multiple des services. Deux stratégies pour leur reconnaissance mutuelle nous sont proposées : la connaissance statique et la connaissance dynamique.

La configuration statique n'est pas conseillée. En effet, si le nombre d'utilisateurs grandit fortement, nous souhaiterions augmenter la capacité de traitement d'un service en ajoutant des instances de celui-ci. Dans ce cas, nous devrions prévoir toutes les possibilités. Cependant, dans une optique dynamique, nous pourrions rajouter des instances qui seront reconnues automatiquement. En effet, nous ne connaissons pas les charges potentielles des serveurs. Il est également possible de voir la suppression d'une instance pour différentes raisons. Dans ce cas, il est important que cette instance soit retirée de la liste des instances connues pour une meilleure performance. Il serait en effet pénalisant pour les temps de réponses qu'un service tente de contacter ceux qui ne sont plus en fonctionnement.

Nous venons d'aborder les problèmes d'ajout et de suppression d'instances. Ceci peut arriver lorsque l'administrateur du serveur désire augmenter la puissance de traitement ou lors de la panne d'un serveur. Cette dernière possibilité nous conduit à introduire la problématique liée à la robustesse du serveur.

SIP, grâce à la notion de dialogue, permet de récupérer de l'information en cas de crash. Ceci permet de récupérer une partie de l'information perdue suite à un crash système. Cependant, ceci nécessite l'implication de l'utilisateur qui doit lui-même rechercher un nouveau serveur. La solution développée devait être transparente pour l'utilisateur et lui permettre de se connecter à une adresse quelque soit le serveur qui s'y cache. Ainsi, un serveur ne répondant plus devra être remplacé automatiquement par un autre.

Nous avons présenté SIP dans le premier chapitre et l'abordons encore ici. L'importance de SIP dans la solution réside dans un besoin d'interopérabilité avec d'autres serveurs. En effet, la société Indigo Software n'est pas la seule sur le marché des services de présence et il existe donc des clients connectés à d'autres serveurs qui désirent converser avec des clients d'Indigo Software. Il est alors indispensable de pouvoir communiquer avec ces serveurs-là.

2.2 Architecture

Nous venons de présenter la problématique relative à l'extension du service *Presence* de la société Indigo Software. Dans la section qui nous occupe maintenant, nous allons présenter la solution proposée dans [19]. Nous allons dans un premier temps montrer un exemple d'interaction des services avec un *presentity* et deux *watchers*. Nous allons ensuite présenter la décomposition du problème avec la découpe locale et la découpe globale. Le dernier point de cette section abordera la robustesse de l'architecture.

2.2.1 Interactions entre services

L'exemple qui nous intéresse pour présenter l'interaction entre services reprend le scénario suivant : le *watcher* A souscrit à l'information de présence de l'utilisateur C, le *presentity*, qui n'est pas connecté au moment de cette opération ; le *presentity* se connecte ensuite et décide de refuser la souscription ; le *watcher* B se connecte à son tour et souscrit également à l'information du *presentity* ; à cet instant, ce dernier accepte les demandes des deux *watchers*. Ce scénario est représenté à la figure 2.3. Tant que le *presentity* ne se connecte pas, le premier *watcher* reçoit l'état par défaut, hors ligne. En effet, la souscription doit être rafraîchie pour être ainsi sûr que le *watcher* ne s'est pas déconnecté brusquement. Cette requête SUBSCRIBE génère une réponse du PA contenant l'état du *presentity*.

Lorsque le *watcher* B souscrit à l'information de présence (message 14), il reçoit immédiatement un message NOTIFY de la part du PA, en attendant que le *presentity* reçoive et traite la demande. Le *presentity* accepte ensuite le second *watcher*, ce qui provoque l'envoi de son état à ce dernier quand le PA reçoit la notification du changement d'autorisation.

Le *watcher* A se retire en émettant un message SUBSCRIBE (27) et dès ce moment, il ne reçoit plus les états de présence du *presentity*. Les messages 28, 29, 30 et 31 sont combinés pour indiquer la déconnexion du *presentity*. En effet, il transmet son état hors ligne à l'ESC qui peut ainsi prévenir le PA pour diffusion aux *watchers*. Le *presentity* prévient ensuite le WI qu'il se déconnecte afin de ne plus lui transmettre de souscription. Les messages 27 et 34 indiquent la déconnexion respective du *watcher* A et du *watcher* B. Ils préviennent ainsi le PA de ne plus leur envoyer l'information de présence du *presentity*.

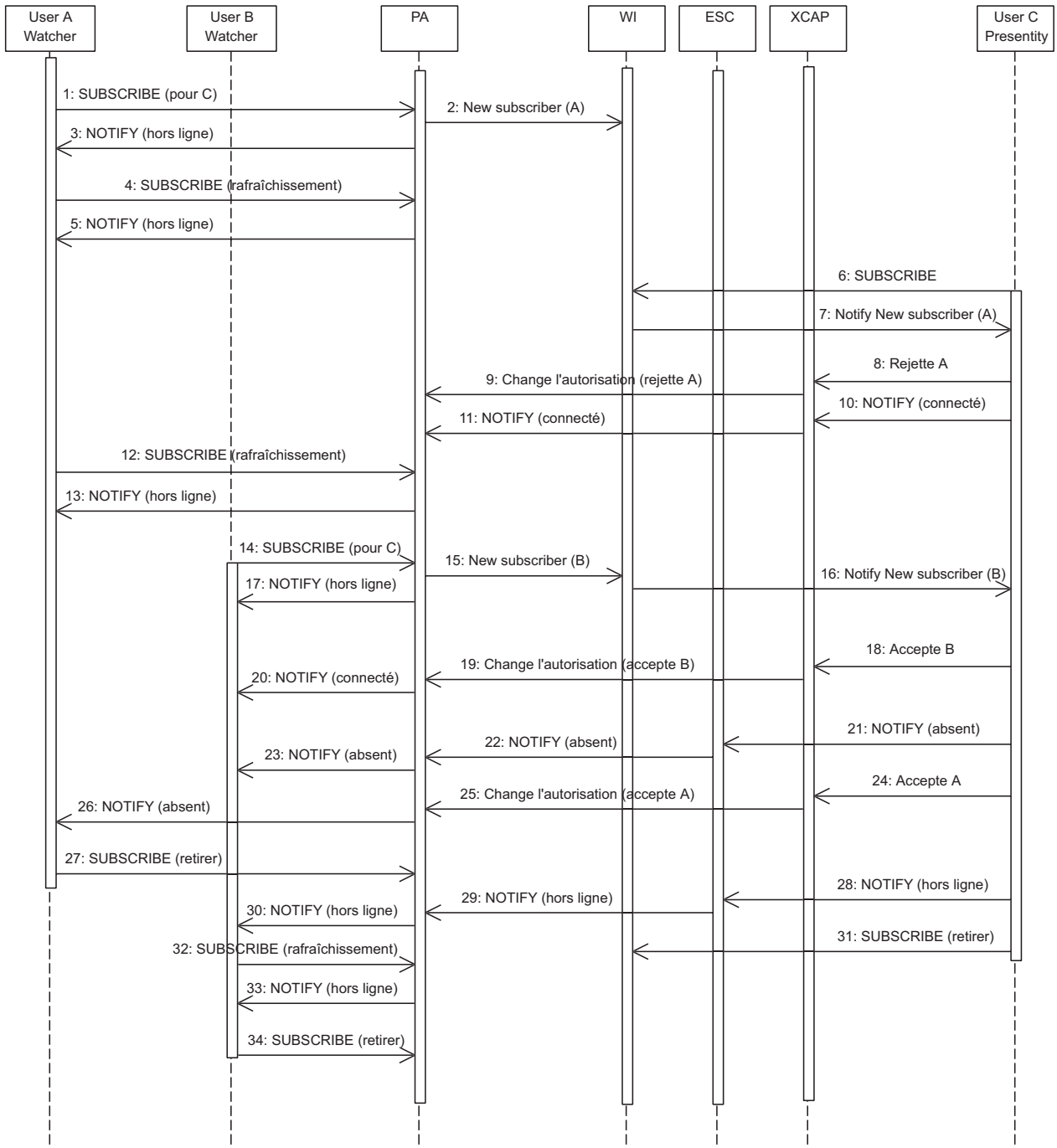


FIG. 2.3 – Exemple d'interactions entre les services.

Sur base de cet exemple, nous pouvons généraliser les échanges entre le *presentity* et les services, entre les services eux-mêmes et entre le PA et un *watcher*. Les premiers échanges sont représentés à la figure 2.4. Le *presentity* souscrit auprès du WI. Par la suite, il est prévenu d'une demande de souscription par ce service. Le rejet ou l'acceptation de celle-ci est envoyée au serveur XCAP. Quant aux changements de son information de présence, ils sont envoyés à l'ESC.

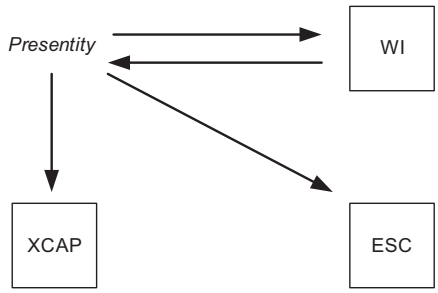


FIG. 2.4 – Interactions entre un *presentity* et les services.

Nous venons de voir comment le *presentity* interagit avec les différents services. Certains reçoivent des données et doivent les transférer vers le service adéquat, à savoir le PA, afin de notifier le *watcher*. Ceci est représenté à la figure 2.5. Le *watcher* émet son désir de connaître l'information de présence d'un *presentity* auprès du PA. Ce dernier envoie cette demande au WI. L'ESC et le serveur XCAP transfère au PA les données reçues du *presentity*, respectivement, l'information de présence et la décision par rapport à la souscription. Le PA peut alors transmettre l'information de présence au *watcher* en fonction des données reçues du serveur XCAP.

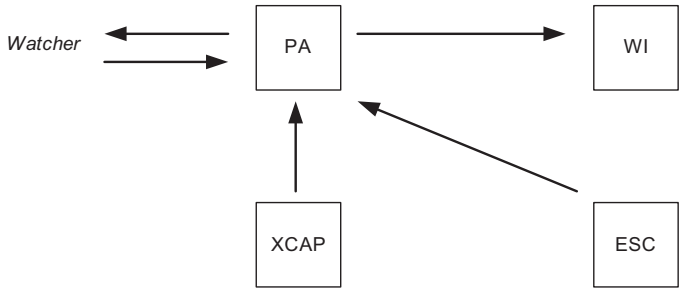


FIG. 2.5 – Interactions entre services avec un *watcher*.

2.2.2 Réflexion générale

Cette section nous permet d'introduire la réflexion générale qui a permis de conduire à la solution qui sera présentée dans les deux sections suivantes.

En partant d'un nombre réduit d'utilisateurs, il est tout à fait envisageable de proposer une architecture contenant un serveur centralisé, connecté à une base de données générales. Avec l'augmentation du nombre d'utilisateurs, le serveur risque vite d'être saturé. La réplication des instances du serveur derrière un *load balancer* (répartiteur de charge) est une solution, mais la base de données centralisée devient le seul guide des performances, c'est-à-dire le goulot d'étranglement du système. En effet, avec un grand nombre d'utilisateurs, de nombreuses requêtes sur la base de données seront effectuées, ce qui ralentirait plus que considérablement les performances.

Le clustering, qui est cette opération de réplication des serveurs derrière un *load balancer*, ne constitue pas la bonne solution, mais sert de base à la solution finale. En effet, il est possible de répliquer les clusters afin de réduire le nombre d'utilisateurs connectés à chacun. Chaque cluster possède sa propre base de données. Il reste à définir comment la répartition des utilisateurs au sein de la distribution de cluster a lieu. Nous verrons dans la section 2.2.4 que les utilisateurs sont répartis en fonction de leur situation géographique.

Nous venons d'évoquer l'optique générale de la recherche présentée dans [19]. Le problème est découpé en deux niveaux : le local et le global.

La section suivante (2.2.3) présente le premier niveau qui consiste au groupement de plusieurs instances des services au sein d'un même cluster. Un cluster doit fournir au moins une instance de chaque service utile dans la gestion de présence. Il est possible de retrouver des serveurs ne permettant qu'un service, d'autres une partie et d'autres encore nous proposant l'entièreté des services. Dans le cas de la société Indigo Software, un serveur fournit chaque service défini précédemment. Un cluster, dans ce cas, sera composé de plusieurs instances du serveur de la société, ce qui permet de gérer les crashs de certains services ou serveurs, sans que l'utilisateur ne s'en aperçoive.

La section 2.2.4 reprend la distribution globale de la solution. Cela consiste en une distribution de clusters, sous forme d'arbre. Les utilisateurs sont répartis en fonction de leur position géographique. Il arrive fréquemment que

des utilisateurs désirent contacter des amis présents dans une autre région géographique. Nous aborderons également la recherche de clusters qui gèrent les contacts.

2.2.3 Distribution locale

La distribution locale consiste à définir un cluster, composé de plusieurs instances d'un même serveur. Tous les services nécessaires à la gestion de présence sont repris dans un même serveur. Afin d'accroître les performances, nous avons des serveurs redondants et pour les utiliser tour à tour, un *load balancer* est utilisé à l'entrée du cluster.

Comme énoncé dans la section 2.1, la notion de dialogue permet de récupérer de l'information en cas de crash d'un serveur. Etant donné que chaque service SIP (PA, ESC, WI) nécessite l'utilisation de dialogues pour le suivi de l'information, nous allons présenter la gestion des dialogues au sein du cluster. Nous allons ensuite passer à la reconnaissance des autres services, pour terminer la section par le mécanisme d'échange de messages entre les services.

Gestion des dialogues

La notion de dialogue, présentée au premier chapitre, identifie un échange de messages entre deux entités SIP. Certaines données relatives au dialogue, comme le numéro de séquence, sont modifiées dès qu'un message transite par un service. Ces informations doivent donc être sauvegardées. Cependant, l'option d'une base de données est vite abandonnée puisque les dialogues sont régulièrement modifiés, ce qui générerait une masse de transaction énorme et ralentirait, voire bloquerait, le système.

L'approche retenue est donc la mise en mémoire des dialogues. Cela nécessite une redirection des messages vers le bon serveur. Il faut donc que le *load balancer* situé à l'entrée du cluster oriente correctement les messages. Ceci nécessite l'adaptation du *load balancer* afin qu'il transfère les messages vers le bon serveur. En effet, un serveur qui reçoit des messages n'appartenant pas aux dialogues se comporte différemment selon qu'il s'agisse d'une requête ou d'une réponse. Dans le premier cas, il créera un nouveau dialogue

et provoquera une nouvelle gestion d'un même utilisateur. Dans le second, il ignorera la réponse et provoquera une attente infinie du serveur qui attend la réponse. Ces deux cas sont inacceptables, le premier provoquant l'incohérence du système alors que le second est inconsistant.

Afin d'éviter de remonter dans le message SIP et donc au niveau applicatif du modèle TCP-IP, le *load balancing* s'effectue au niveau de la couche réseau. Ceci permet de réduire le temps de traitement.

Reconnaissance des autres services

Comme nous l'avons présenté dans la section 2.1, la reconnaissance des autres services peut être configurée de manière statique ou être réalisée dynamiquement. Nous avons, à ce moment, introduit les inconvénients de la première approche. Le principal défaut est l'obligation de prévoir toutes les possibilités de configuration. Or, il se peut qu'à une période donnée, nous connaissions un pic d'utilisation qui demande d'augmenter la capacité de traitement au delà de la prévision, ce qui nous obligerait à reconfigurer l'entièreté du cluster et à relancer les serveurs.

La reconnaissance automatique s'impose donc de fait à la solution et accroît l'efficacité de la gestion du cluster.

Dans son mémoire [19], l'auteur propose deux mécanismes pour la reconnaissance automatique des autres services. Le premier repose sur le paradigme de souscription et notification. Chaque serveur comprend un module qui permet de s'enregistrer auprès d'un manager de services, qui informe tous les services inscrits à chaque changement de topologie.

Le deuxième mécanisme repose sur un serveur d'annuaire *Lightweight Directory Access Protocol* (LDAP). Chaque service s'y enregistre et regarde à intervalle de temps régulier quels sont les changements de structures.

Notification entre les services

La notification d'événements entre les services repose également sur le principe de souscription et de notification. L'auteur de [19] s'est également basé sur le principe des RLI que nous avons présenté à la section 1.2.6. Ce

principe permet de regrouper les messages SUBSCRIBE et NOTIFY dans le but d'en échanger moins sur le réseau. Sans l'intégration des RLI dans le mécanisme, un grand nombre de messages est transmis. En effet, le principe de base définit un message SUBSCRIBE par utilisateur dont nous souhaitons connaître l'information de présence, auquel un NOTIFY est envoyé en guise de réponse. Le fait de regrouper les messages en deux, permet de réduire considérablement le trafic sur le réseau.

La différence entre le mécanisme de base et celui défini par les RLI réside dans le contenu du message SUBSCRIBE : l'utilisateur ne souscrit plus à un seul contact, mais à une liste d'amis. Cette liste est mise à jour en interagissant avec le serveur XCAP. De même, le message NOTIFY est modifié pour transmettre l'état de présence de plusieurs *presentities*.

Nous avons présenté à la section 2.2.1 les interactions entre les services. Nous y avons découvert que le PA envoyait des messages au WI et en recevait de l'ESC et du XCAP. Dès lors, le WI doit souscrire au PA, qui doit lui-même souscrire à l'ESC et au XCAP. Etant donné que plusieurs instances des serveurs sont présentes au sein d'un cluster, le WI devra souscrire à chaque PA et le PA souscrira à tous les ESC et XCAP du cluster. Ce principe est illustré à la figure 2.6. Nous y apercevons les souscriptions entre les services d'un serveur et aussi entre les services de différents serveurs d'un même cluster. Pour une meilleure lisibilité, nous n'y avons représenté que les souscriptions des services du premier serveur. Il est évident qu'elles sont également présentes au sein des services du second serveur.

Cependant, si ceux-ci souscrivent pour chaque utilisateur géré par le domaine, cela engendrerait un grand nombre de messages. Les souscriptions entre services reprennent en partie le principe des RLI, étant donné qu'il n'est pas envisageable que chaque service travaille avec sa propre liste de contacts. En effet, chaque service est potentiellement intéressé par l'ensemble des utilisateurs du cluster. Dès lors, le principe, mis en place dans [19], repose sur une liste générique, ayant pour syntaxe : `*@cluster-address`. Cette liste contient implicitement tous les utilisateurs présents et futurs du cluster.

Ce principe permet d'éviter la mise à jour des listes pour lesquelles les services souscrivent. La liste générique permet à ceux-ci de souscrire directement à tous les utilisateurs connectés au cluster. Cela diminue la charge de travail, mais peut générer des notifications inutiles. En effet, un serveur peut recevoir des informations qui ne l'intéressent pas. Dans ce cas, nous par-

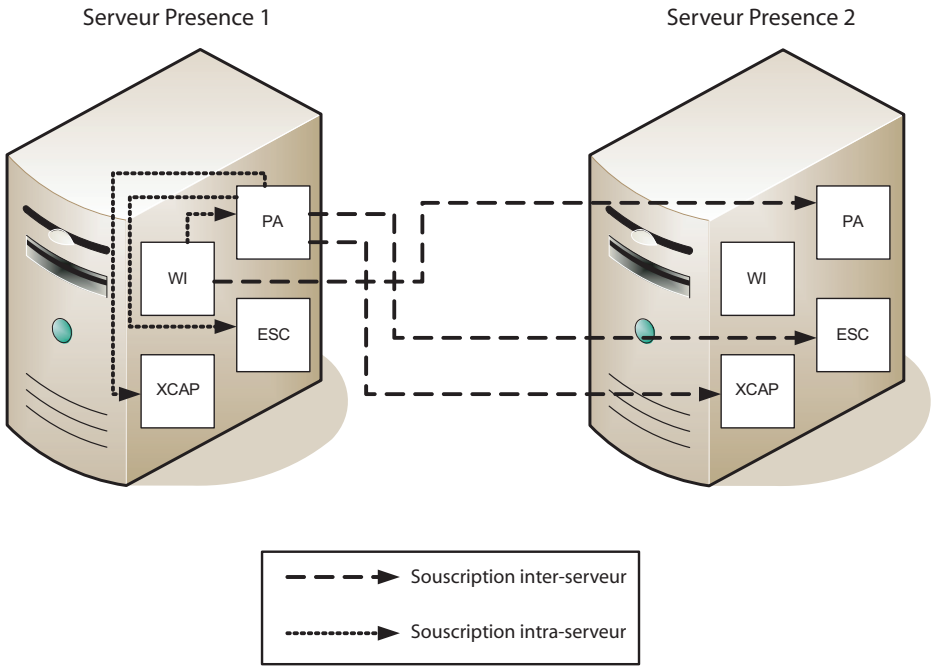


FIG. 2.6 – Souscriptions entre les services.

lons de trafic inutile, mais il s'agit du prix à payer pour avoir une solution facilement extensible.

Les services distinguent les notifications qui les intéressent et gardent en mémoire les données utiles. Chaque notification reçue n'est qu'une mise à jour de l'information et engendre une adaptation des données gardées en mémoire auprès du service.

2.2.4 Distribution globale

Nous venons de présenter la distribution locale. Nous allons, dans cette section, parler de la distribution globale. Dans un premier temps, nous évoquerons la localisation géographique, sur laquelle se base la répartition des utilisateurs pour leur cluster. Après cette introduction du principe de localisation, nous présenterons la recherche de contacts, en évoquant dans un pre-

mier temps l'arborescence des clusters et dans un second temps la recherche à proprement dite. Nous aborderons ensuite l'utilisation d'un *outbound proxy*.

Géolocalisation

La géolocalisation IP permet de récupérer la localisation géographique sur base de l'adresse IP. En effet, chaque fournisseur d'accès possède une fourchette d'adresses, dont l'une est donnée à l'utilisateur qui se connecte au service de présence. Ces fourchettes sont donc connues et il suffit de maintenir à jour une base de données faisant correspondre les adresses IP et leurs régions. Ce mécanisme est utilisé, par exemple, pour effectuer de la publicité géolocalisée.

Pour l'architecture proposée, il ne s'agit pas de publicité mais d'optimiser les temps de transfert des messages. En effet, il est préférable qu'un utilisateur se connecte à un cluster proche plutôt qu'à un serveur situé à l'autre bout du monde.

Le serveur DNS est modifié pour répondre différement en fonction de la localisation des utilisateurs et est appelé géoDNS. Le mécanisme défini ci-dessus permet donc de retrouver la région en fonction de l'adresse IP. Il nous est possible d'associer un ou plusieurs clusters à une même région.

Ce principe permet donc de réduire le coût en termes de chemin internet pour la connexion d'un utilisateur à son cluster. De plus, la plupart des contacts d'une personne se situe généralement dans une même région, voire une région proche. Les clusters sont donc géographiquement proches, ce qui limite aussi la distance parcourue par les messages.

Arborescence de clusters

Nous venons d'aborder la possibilité d'avoir des contacts gérés par un autre cluster. Un cluster ne gère que certaines souscriptions : celles destinées aux *presentities* qu'il connaît. Dès lors, une souscription à destination d'un *presentity* inconnu doit être redirigée vers le cluster concerné. L'opération de recherche du cluster est appelée localisation de contact. Avant de pouvoir l'aborder dans la prochaine section (2.2.4), nous devons présenter l'arborescence des clusters.

Comme le nom l'indique, la structure est organisée en arbre bidirectionnel. Il s'agit d'une structure similaire à la hiérarchie des *Portable Object Adapters* (POA), définis dans *Common Object Request Broker Architecture* (CORBA), la différence étant que chaque noeud n'est pas une structure de POA, mais un cluster de serveurs. L'architecture définie est également semblable à la structure DNS. Tout comme les hiérarchies de DNS et POA, un cluster peut avoir plusieurs fils. En effet, bien que rien dans les documents de Quan Truc Nguyen [10] et Jean-Louis Sacré [19] n'indique s'il s'agit d'un arbre binaire, après un entretien avec l'auteur de [19], rien n'indique et ne nous oblige d'utiliser un tel type d'arbre.

La figure 2.7 représente un exemple d'arborescence des clusters. Chaque cluster est identifié par son nom suivi de celui de son père. Par exemple, si le père est connu par le nom `be.society.com` et le fils porte le nom de `wallonie`, le cluster fils aura le nom `wallonie.be.society.com` comme identifiant dans l'arbre. Ces identifiants respectent la structure d'un *Fully Qualified Domain Name* (FQDN).

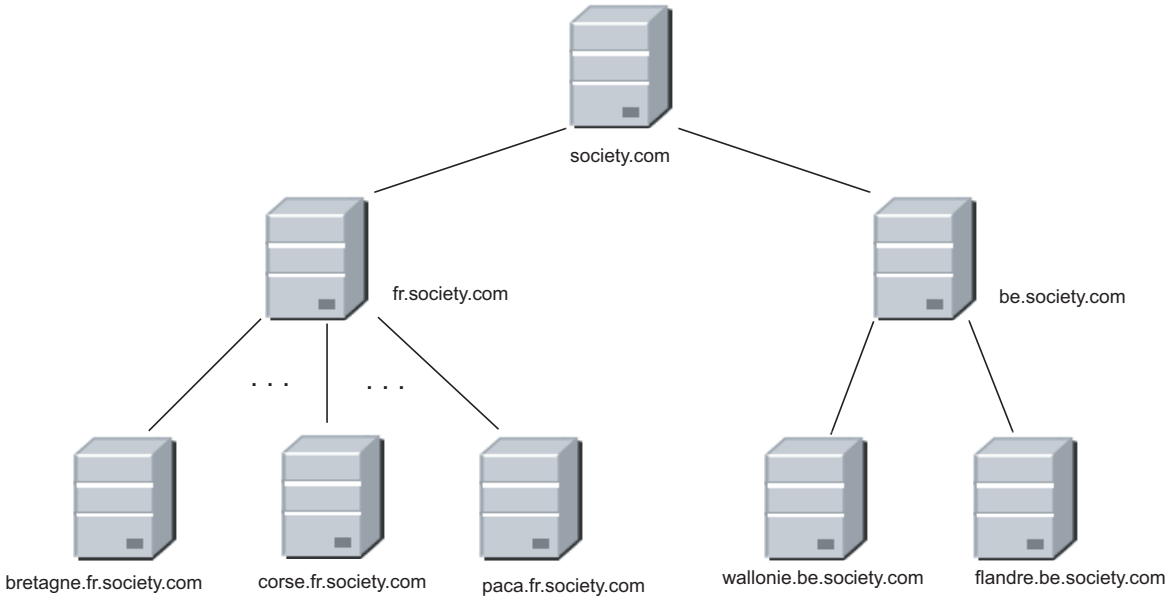


FIG. 2.7 – Arborescence des clusters.

Il est dès lors facile d'implémenter la connaissance du père. En effet, il suffit de soustraire son nom au FQDN, ce qui nous laisse le FQDN du père.

La connaissance dans l'autre sens, du fils par le père, est plus délicate. Le mécanisme d'enregistrement est envisagé. Il suffit au cluster qui démarre de s'enregistrer auprès de son père. Afin d'éviter un noeud mort dans l'arbre, le fils doit réactiver son enregistrement auprès de son père. Ceci permet en effet de détecter la panne d'un cluster. Si le père ne reçoit pas de rafraîchissement d'enregistrement, il considère que son fils est inaccessible.

Localisation de contact

Maintenant que nous connaissons l'architecture mise en place avec l'arborescence des clusters, nous pouvons évoquer le mécanisme de localisation de contact dans la structure. Lors d'une souscription, l'utilisateur envoie sa requête à son point d'entrée dans le réseau SIP, l'*outbound proxy*. Ce dernier interroge le cluster de l'utilisateur, que nous appellerons *home cluster*. Dans le cas le plus favorable, son cluster connaît le contact et la recherche se termine normalement.

Si le *home cluster* ne connaît pas le contact, il interroge les clusters fils. Deux cas de figures sont alors envisageables : l'un des fils le trouve ou aucun ne le connaît.

Dans le premier cas, la réponse est remontée au cluster père qui peut, par l'intermédiaire de l'*outbound proxy*, la transférer vers l'utilisateur. Nous verrons à la section 2.2.4 comment l'*outbound proxy* oblige les messages SIP à passer par lui. La réponse contient l'adresse du cluster ayant trouvé le contact dans sa base de données afin d'éviter une répétition du mécanisme à chaque message. Nous évitons ainsi d'alourdir la charge de signalisation.

Dans le second cas, une recherche récursive sur les fils des fils est exécutée afin de parcourir toutes les branches de l'arbre. Si l'un des descendants du *home cluster* trouve le contact, nous effectuons les mêmes opérations que dans le cas précédent, à savoir remonter la réponse, contenant l'adresse du cluster concerné, vers le *home cluster*, qui se charge de la transférer à l'utilisateur en passant par l'*outbound proxy*.

Si aucun descendant ne trouve le contact, le *home cluster* va répondre à l'*outbound proxy* avec un message REDIRECT afin de transférer la demande vers un autre cluster. Cet autre cluster est en réalité le père du *home cluster* et permet ainsi de remonter logiquement dans l'arbre.

Le cluster père reçoit la requête et interroge chacun de ses fils à tour de rôle. Il va donc interroger le *home cluster* qui ne connaît pas le contact lui-même et qui sait qu'aucun de ses descendants ne peut le joindre. La solution apportée à ce problème dans [19] est celle de la mise en cache de la réponse relative à cette recherche. Ceci permet d'éviter d'interroger à nouveau l'ensemble des descendants pour obtenir une réponse que le cluster connaît déjà : l'impossibilité de joindre la personne. Le cache en question est présent au niveau du *home cluster* afin de répondre immédiatement. Nous verrons à la section 3.3 comment nous pouvons améliorer la solution et à la section 2.3.2 comment l'auteur de [10] a développé ce cache. Pour parler du cache évoqué ci-dessus, il utilise le terme de mémoire afin de la différencier avec le cache mis en place au sein de l'*outbound proxy* et que nous évoquerons également à la section 2.3.2.

Si l'un des clusters descendant du père connaît le contact, la transmission de la réponse se passe de manière similaire aux cas précédents. Cependant, il est possible de parcourir toutes les branches de l'arbre sans que le contact ne soit trouvé. Dans ce cas, une réponse du type 404 NOT FOUND est retournée à l'utilisateur.

La recherche peut paraître laborieuse, mais elle permet d'étendre le service de présence plus facilement. En effet, il est possible d'ajouter des clusters dans l'arborescence sans que cette recherche ne doive être modifiée. De plus, celle-ci n'est effectuée qu'à la première souscription reçue puisque l'*outbound proxy*, qui force le passage des messages, connaît les résultats et permet de dialoguer directement avec le cluster connaissant le contact.

Outbound proxy

Nous avons parlé de l'*outbound proxy* dans la section précédente. Celui-ci s'est imposé lors de l'étude [19] afin de résoudre un problème dû à l'autorisation des souscripteurs. En effet, la solution est que le cluster ne gère que les souscriptions des *presentities* qu'il connaît. Ceci nécessite des échanges de messages entre clusters lorsqu'une souscription porte sur des contacts connectés à d'autres clusters. Pour des raisons de sécurité et d'interopérabilité, cette solution doit être adaptée.

C'est là qu'entre en jeu l'*outbound proxy*. En effet, dialoguer avec un autre domaine géré par une autre société ne permet pas de développer une

solution basée sur un échange de messages propriétaires. En effet, une telle solution n'autoriserait pas le dialogue avec d'autres domaines.

Etant donné que l'*outbound proxy* est un point centralisé, seuls des messages de localisation sont échangés entre clusters. La gestion des autorisations se faisant au niveau des clusters connaissant les *presentities*, l'interopérabilité est vérifiée car les changements d'autorisation sont gérés dans les clusters et aucun échange de message de ce type ne doit avoir lieu sur le réseau. La gestion de présence s'effectue également sur les clusters connaissant les *presentities*. Un domaine d'une autre société peut dès lors s'apparenter à un ensemble d'utilisateurs qui communiquent leur information de présence à l'*outbound proxy*.

Dans la section précédente, nous avons parlé d'un mécanisme qui permet à l'*outbound proxy* de forcer le passage des messages en son sein. Ce principe repose sur le champ d'en-tête, *Record Route*, défini dans le protocole SIP. Cet attribut est modifié par certaines entités afin de forcer le passage des messages suivants du dialogue par elles.

Dans notre cas, l'*outbound proxy* force le chemin des messages et oblige les utilisateurs à communiquer avec lui afin d'envoyer les requêtes et réponses vers les clusters. Dans la solution d'Indigo Software, seul l'*outbound proxy* possède la propriété de forcer le chemin des messages. Les serveurs de présence n'ont pas la possibilité d'effectuer cette opération.

2.2.5 Robustesse

La robustesse repose principalement sur la caractéristique de SIP qui permet de récupérer l'information de dialogues. En effet, si une instance d'un service venait à disparaître, une autre devrait reprendre la main. Pour y arriver, elle devrait recouvrer certaines informations. Cette opération est coûteuse en ressources mais est indispensable pour que le système retrouve sa stabilité.

Si une instance d'un service peut manquer, il est possible que toutes les instances de celui-ci viennent à manquer. Dans ce cas, le cluster ne peut plus garantir le service minimum pour la gestion de présence. Dès lors, la solution est de retirer le cluster de l'arborescence et de rediriger tous les messages arrivés à l'*outbound proxy* vers un autre cluster. Ce nouveau cluster doit être proche et peut être trouvé par le principe de géolocalisation.

2.3 Développement

Après avoir abordé la problématique, nous avons présenté l'architecture proposée par l'auteur de [19]. Cependant, par manque de temps lors de ses travaux, il n'a pas eu le temps de développer celle-ci. Dans le cadre de son stage [10] passé aux Facultés Universitaires Notre-Dame de la Paix, l'auteur a pour sa part développé certaines parties de l'architecture. Il n'a pas eu le temps non plus de construire l'entièreté de l'architecture, ne disposant que de treize semaines au sein du laboratoire.

Nous allons présenter les parties développées lors de ce stage : le résolveur de nom de domaine, la localisation des utilisateurs et la collaboration de serveurs.

2.3.1 Résolveur de nom de domaine

La première partie que l'auteur nous présente dans son rapport [10] est la mise en place d'un résolveur de nom de domaine. Dans le cadre d'une simulation, nous pourrions nous passer d'une telle entité puisque nous pouvons utiliser les adresses IP pour identifier les serveurs. Cependant, l'architecture présentée à la section 2.2 repose sur un mécanisme de cluster et de FQDN. Ces noms sont choisis de manière hiérarchique afin de faciliter la gestion du réseau.

La documentation du serveur [24] nous présente un mécanisme pour répartir la charge entre serveurs, il s'agit des enregistrements DNS SRV définis dans le RFC 2782 [5]. Ceux-ci respectent la syntaxe suivante :

```
_Service._Proto.Name [TTL] [Class] SRV Priority Weight Port Target
```

Les deux champs entre crochets sont facultatifs et ont leur signification définie dans le RFC 1035 [9] concernant les noms de domaines. `Service` représente le type de service qui est concerné par l'enregistrement. `Proto` indique le protocole de la couche de transport (TCP ou UDP). `Name` indique le nom de domaine auquel l'enregistrement SRV se réfère. `Priority` est comme son nom l'indique l'élément tenant compte de l'enregistrement dans le choix du serveur à atteindre. `Weight` permet de départager les destinations qui ont la même priorité. `Port` est le numéro du port sur lequel la destination écoute

et `Target` est le nom de cette cible, sous forme FQDN. Dans le cadre de cette étude, certains champs ont des valeurs spécifiques. Ainsi `Service` sera toujours remplacé par `sip` et le port sera 5060.

Afin de visualiser le concept, nous reprenons l'exemple donné dans [24]. Il s'agit de deux serveurs SIP et d'un serveur de secours. Le serveur DNS est configuré comme suit :

```
_sip._udp SRV 0 3 5060 server1.society.com
_sip._udp SRV 0 3 5060 server2.society.com
_sip._udp SRV 1 0 5060 backup0.society.com
```

Les deux premiers serveurs seront choisis avec la même probabilité et le même poids. Ceci indique donc que le choix du serveur dans la répartition de charge n'est pas déterministe. Par contre, le troisième serveur (`backup0.society.com`) n'est présent qu'en cas de panne des deux premiers serveurs simultanément.

2.3.2 Localisation des utilisateurs

Nous avons évoqué la localisation de contact dans la section 2.2.4. Nous y avons expliqué le mécanisme de recherche dans lequel nous avons relevé le problème de localisation répétitive. En effet, lorsqu'un cluster ne trouve pas le contact ni chez lui ni parmi ses clusters fils et ses descendants, la requête est transmise au cluster père. Celui-ci dans la logique d'un algorithme va interroger l'entièreté de ses fils, y compris le cluster ayant provoqué la remontée dans l'arbre. La solution évoquée est la mise en cache et a été exploitée par l'auteur de [10].

Ce cache mémoire est mis en place au niveau des *outbound proxies*. Une seconde optimisation, au niveau du cluster cette fois, est la mise à disposition d'une mémoire afin d'éviter de redescendre dans l'arborescence à chaque requête qui se répète. Nous y reviendrons par la suite, mais nous allons, dans un premier temps, aborder la première optimisation évoquée.

Celle-ci consiste donc en l'ajout d'un cache au niveau des *outbound proxies*. Ce cache contient une table de routage mémorisant les utilisateurs et l'adresse de leur home cluster. Ceci évite de répéter la recherche dès l'arrivée d'une nouvelle requête. La table ainsi créé doit être limitée en taille pour des raisons

d'efficacité. En effet, une table comportant un trop grand nombre d'entrées serait pénalisant étant donné que son parcours serait fastidieux dans la moitié des cas. Une table trop petite, quant à elle, ne peut être envisagée puisqu'il est fort probable que la requête soit à destination d'une personne inconnue et engendre une nouvelle recherche. Dès lors, si la table est complète, le résultat sera soit enregistré au détriment d'un autre soit oublié volontairement. Dans les deux cas, une table pleine demanderait à chaque nouvelle requête d'effectuer une nouvelle recherche si le contact ne s'y trouve pas déjà. Le choix de l'auteur lors de l'implémentation est une table de hashage avec la fonctionnalité de suppression de l'enregistrement le plus ancien de la table si celle-ci est complète. Tous les utilisateurs ne doivent cependant pas figurer dans la table. En effet, l'*outbound proxy* envoie, par défaut, la requête au cluster le plus proche géographiquement. Ainsi, les utilisateurs étant gérés par ce cluster ne doivent pas figurer dans la table.

Lorsque l'utilisateur n'est pas connu de la table de routage, soit il est connu du cluster par défaut, soit il s'agit de la première requête à sa destination. Quel que soit le cas, la requête est transférée au cluster par défaut. Si l'utilisateur n'est pas connu par ce cluster, il entame le processus de recherche auprès de ses clusters fils. Comme la solution le nécessite, les réponses doivent retourner au cluster initiateur qui répond à l'*outbound proxy*. Dès lors, ce dernier ne peut pas se baser sur la source de la réponse puisque rien d'indique qu'il s'agisse du cluster chargé de gérer cet utilisateur. Le choix de l'auteur a été de se baser sur les requêtes SUBSCRIBE du serveur vers le *presentity*.

La technique utilisée pour l'implémentation de la table est donc une table de hashage. Celle-ci peut être programmée en Java à l'aide de l'objet *Hashtable*. Celle-ci n'est pas de taille fixée comme un tableau et grandit à chaque nouvelle entrée. Une file permet de borner la table afin d'éviter un dépassement de mémoire et une perte d'efficacité. Comme nous l'avons dit ci-dessus, la taille maximale ne doit pas être trop grande ni trop petite. Le calcul de cette borne doit tenir compte du nombre d'utilisateurs gérés par les autres clusters, mais aussi de la proportion de ceux-ci qui pourraient éventuellement se connecter depuis une autre région géographique, dépendant d'un autre cluster.

La deuxième optimisation proposée par l'auteur de [10] consiste à ajouter une mémoire au niveau du cluster. La première optimisation évitait de répéter le processus de recherche pour un contact dont on connaît le résultat, car une recherche a eu lieu précédemment. La seconde évite, quant à elle,

de propager une recherche pour laquelle le cluster a déjà eu le résultat et a transmis la recherche auprès de son père. En effet, il est inutile d'effectuer une recherche dans sa descendance si nous avons transféré la requête à notre père. Sans cette optimisation, la recherche aurait une complexité plus importante car elle parcourt plus de noeuds dans l'arbre. Cependant, le cluster ayant initié la redirection reçoit encore une requête l'interrogeant. Nous proposerons à la section 3.3 une méthode permettant d'éviter ce message et par la même occasion le système de mémoire mis en place.

2.3.3 Collaboration de serveurs : Clustering

La mise en cluster est réalisée au moyen de *Linux Virtual Server* (LVS), dont la configuration est expliquée à l'annexe C. Cette application permet de répartir la charge en travaillant sur la couche transport du modèle TCP/IP. Comme son nom l'indique, il s'agit de créer un serveur virtuel, composé de plusieurs serveurs réels. Le répartiteur de charge reçoit les messages et les oriente vers l'un des serveurs réels.

LVS utilise l'une des trois techniques de répartition de charge : le *Network Address Translation* (NAT), le *Direct Routing* et l'*IP Masquerading*.

La première méthode consiste à modifier le paquet pour les transférer vers les serveurs. Le message de réponse doit obligatoirement passer par le répartiteur de charge et effectuer le changement inverse.

La seconde technique transfert le paquet sans modification à l'un des serveurs qui répondra directement à l'utilisateur. Ce dernier s'attend à recevoir une réponse de l'entité qu'il a contactée, à savoir le répartiteur de charge.

La contrainte des dialogues SIP imposée dans l'architecture ne peut donc être vérifiée. La dernière méthode consiste à créer un tunnel entre le répartiteur de charge et le serveur sélectionné. La réponse repasse dès lors par le répartiteur de charge qui la transmet à l'utilisateur. Cette technique permet donc de respecter la notion de dialogue.

Les deux méthodes, NAT et *IP Masquerading* se ressemblent et permettent de respecter la notion de dialogue. La technique choisie pour le développement de la solution est la dernière exposée ci-dessus. Celle-ci permet de disposer les serveurs réels sur différents domaines, ce que le NAT

n'autorise pas. La méthode d'*IP Masquerading* propose aux serveurs d'utiliser l'adresse virtuelle du répartiteur de charge. Ainsi, tous les serveurs du cluster possèdent la même adresse.

LVS fournit également une résistance aux pannes. En effet, des processus sont exécutés sur le répartiteur de charge afin de vérifier la présence des serveurs. Ceux-ci effectuent des requêtes *Internet Control Message Protocol* (ICMP) *ECHO_REQUEST*, plus connues sous le nom de requête *ping*. Si un serveur n'y répond plus, il est considéré comme inaccessible et est retiré de la liste des serveurs disponibles. De plus, la maintenance est facilitée puisqu'un serveur qui démarre sera reconnu par le répartiteur de charge et ajouté à la liste.

Une dernière caractéristique de LVS est de fournir un mécanisme de récupération en cas de pannes. Nous pouvons, en effet, configurer LVS pour qu'un serveur ne soit utilisé que lorsqu'il n'y en a plus en activité.

Chapitre 3

Evaluation théorique

Ce troisième chapitre nous conduit à évaluer théoriquement, l'architecture proposée à la section 2.2. Nous avons basé notre étude sur les cours de "Conception des systèmes distribués et coopératifs" [4] et d'"Ingénierie du logiciel" [6]. Ce second cours [6] s'oriente principalement sur les processus de développement des logiciels mais présente également la qualité des produits. Les performances d'un système en font partie. Nous entamons donc ce chapitre par une analyse des qualités du système. Cette analyse est complétée par la vérification du respect des exigences d'un système distribué.

Certains composants de l'architecture sont supposés fiables. Cependant, cette hypothèse nous semble trop forte et nous aborderons les conséquences qu'une panne des composants peut amener. Pour terminer le chapitre, nous évaluerons le développement de la localisation de contact.

Des éléments de performance, comme la charge supportée par les serveurs, sont difficiles à évaluer théoriquement et feront l'objet de tests que nous exposerons dans le chapitre 4.

3.1 Analyse de la qualité et des exigences

Cette première section nous mène à analyser la qualité du produit et à vérifier que celui-ci respecte les exigences d'un système distribué. Comme nous l'avons évoqué dans l'introduction de ce chapitre, notre analyse se base sur les grilles proposées dans les cours [6] et [4].

Le but de ce travail est d'évaluer les performances du système. La performance fait partie des qualités. Afin de définir quelles sont les qualités d'un produit logiciel, l'*International Organization for Standardization* (ISO) a établi la norme *ISO 9126*. Parmi la liste définie par cette norme, le cours [6] nous propose certains critères parmi lesquels nous allons étudier les suivants : la justesse, la sûreté du fonctionnement, l'efficacité, la portabilité et l'interopérabilité.

D'autres critères existent comme la facilité d'utilisation. Cependant, lors du développement d'un système, certains critères peuvent être contradictoires et il est alors indispensable d'en privilégier l'un ou l'autre. Par exemple, dans le cas d'un serveur, il est préférable d'optimiser l'efficacité quitte à réduire la facilité d'utilisation. C'est pour cette raison que tous les critères ne sont pas étudiés.

Les cinq critères énoncés ci-dessus forment l'ensemble de base pour ce que nous appelons les performances du système. La justesse vérifie que le logiciel, dans notre cas le serveur, effectue correctement ce que nous en attendons. Il est en effet indispensable que les opérations se déroulent sans erreur pour que l'utilisateur ait conscience d'une certaine performance. La sûreté de fonctionnement reprend également la justesse dans le cadre d'un fonctionnement correct. Lors d'un fonctionnement poussé aux limites, l'application doit toujours effectuer ce que nous attendons d'elle ou ne rien exécuter. Ce deuxième critère reprend les aspects de robustesse. L'efficacité reprend l'utilisation performante des ressources matérielles comme le processeur ou la mémoire. La portabilité est également signe de performance. En effet, si l'application est portable, nous pouvons la distribuer sur des machines hétérogènes. Enfin, l'interopérabilité est une grande qualité puisque le logiciel n'empêche pas l'utilisateur de travailler avec des personnes n'utilisant pas la même application que lui.

La seconde grille à la base de notre analyse est celle des exigences d'un système distribué. Le cours [4] en expose quatre principales : l'évolution,

l'hétérogénéité, l'accès et partage des ressources et la tolérance aux pannes. Cette grille comprend également des niveaux de transparence. Un système distribué doit paraître comme étant une entité unique et intégrée pour l'utilisateur. Dès lors les mécanismes internes ne doivent pas être visibles de l'extérieur. Différents niveaux de transparence sont donc définis et nous n'en verrons qu'une partie. Nous évoquerons les transparences de la localisation, de la migration et des échecs. D'autres niveaux ne seront pas exploités.

Le système ne doit pas uniquement respecter les exigences d'un système distribué. La société Indigo Software a également mis ses contraintes pour le développement de la mise à l'échelle de son service. Celles-ci portent sur l'évolutivité du système et sa transparence aux utilisateurs. En effet, le système doit être extensible. De nouvelles fonctionnalités peuvent être définies dans le futur et il est inconcevable qu'une nouvelle fonctionnalité ne demande une adaptation structurelle du système pour être intégrée. La transparence vise principalement à imposer que la solution développée s'adapte au comportement des applications existantes.

Ces trois éléments peuvent être regroupés en une seule grille d'analyse que nous exposons ci-dessous. En effet, la justesse du produit demande à ce que les contraintes de la société Indigo Software soient vérifiées. L'une de ces contraintes peut être vue comme un niveau de transparence d'un système distribué. De plus, certaines qualités sont proches de certaines exigences des systèmes distribués. Lorsque nous parlons de sûreté du fonctionnement, nous devons vérifier que le système réagit bien en conditions normales mais également en conditions extrêmes. Ceci se rapproche de l'exigence de la tolérance aux pannes. L'exigence relative à l'évolution du système peut être reprise au sein de la première contrainte imposée par la société Indigo Software.

Notre grille d'analyse suit donc le schéma suivant : la justesse, la robustesse, l'efficacité, l'hétérogénéité, l'accès et partage des ressources et les transparences d'un système distribué.

3.1.1 Justesse

Le premier critère nous mène à nous poser la question suivante : le serveur répond-il aux besoins fonctionnels ? En ce qui concerne le serveur pris isolément, nous supposons qu'il a vérifié une batterie de test pour pouvoir être distribué par la société Indigo Software. Il doit donc correspondre aux

besoins exprimés lors de l'analyse du serveur. Les travaux [19] et [10] ont pour but d'étendre le service à grande échelle. Dans ce cadre, une architecture (cf. section 2.2) a été proposée selon certaines exigences. La solution développée doit être évolutive pour pouvoir ajouter facilement de nouvelles fonctionnalités et doit être transparente pour l'utilisateur final afin que celui-ci ne doive pas modifier le comportement de son application de messagerie. Nous allons également répondre aux deux questions suivantes : l'architecture permet-elle de gérer un plus grand nombre d'utilisateurs ? Pouvons-nous facilement étendre la solution ?

Extension à grande échelle

Dans un premier temps, nous allons répondre aux deux dernières questions avant de nous pencher sur les contraintes imposées par la société Indigo Software. Les réponses sont positives aux deux questions. En effet, le nombre d'utilisateurs gérables par l'architecture est augmenté par rapport à un serveur isolé et la solution permet d'étendre le service très facilement.

La première réponse apportée est vérifiée par le double niveau de distribution apporté. Dans un premier temps, la distribution locale permet de gérer plus d'utilisateurs car il y a plus de serveurs qui s'exécutent simultanément, mais les performances restent liées à la seule base de données présente au sein du cluster. Ce goulot d'étranglement est en partie résolu grâce à la répétition de clusters, à savoir la distribution globale, sur base d'une arborescence et regroupant les utilisateurs sur base de leur zone géographique. Ainsi tous les utilisateurs d'une même région se connectent à un cluster. Les utilisateurs sont donc répartis uniformément sur l'ensemble de l'arborescence. De ce fait, au plus nous créons de clusters, au plus nous avons de bases de données. Nous multiplions donc par un facteur correspondant au nombre de cluster que nous avons lançons le nombre d'utilisateurs que nous pouvions gérer avec un seul serveur.

La deuxième réponse est vérifiée par l'aspect dynamique de l'architecture. En effet, nous savons que les utilisateurs sont répartis en fonction de leur situation géographique. Imaginons que la France soit découpée en autant de clusters qu'il y a de régions administratives, à savoir 26. Cependant, nous pouvons croire que la région Ile-de-France regroupe un nombre d'utilisateurs trop important pour un cluster. Dans ce cas, nous pouvons créer un cluster par département présent dans cette région, il y en aurait ainsi 8. Chacun

serait fils du cluster représentant l'Ile-de-France, fils lui-même du cluster associé à la France. Par exemple, si le nom du cluster de l'Ile-de-France était `ile-de-france.fr.eu.society.com`, nous aurions, pour le département de Paris, le nom de cluster suivant : `paris.ile-de-france.fr.eu.societe.com`.

Respect des exigences d'Indigo Software

Maintenant que les deux principales questions sont vérifiées, nous allons examiner les contraintes imposées par la société Indigo Software.

La transparence est vérifiée grâce à la présence de l'*outbound proxy* qui permet de camoufler certains mécanismes introduits par le dimensionnement du service, comme la localisation de contact.

L'évolutivité du système est également assurée. En effet, il suffit de développer les nouvelles fonctionnalités en modules et de les intégrer à la solution existante. C'est ainsi que le développement de la localisation de contact a été intégrée dans [10].

3.1.2 Robustesse

L'architecture est donc correcte et nous pouvons évaluer sa sûreté au fonctionnement. Comme il est expliqué dans [6], la sûreté de fonctionnement se découpe en deux niveaux. Le premier vérifie le comportement dans des conditions normales d'utilisation, le second teste dans des conditions extrêmes. Théoriquement, le premier niveau est difficilement vérifiable. D'une manière générale, un produit correct peut ne pas être fiable. Par exemple, le système effectue correctement les actions, mais peut être instable. Cependant, dans notre cas, étant donné la justesse théorique du système, nous pouvons estimer que le service est correct lors d'une utilisation normale. Afin de s'en assurer, des tests de validation doivent être réalisés. La robustesse du système lors de pannes et autres conditions anormales demande également la mise à l'épreuve du système.

La robustesse demande au système d'être tolérant aux pannes. En effet, un système distribué est plus facilement sujet aux pannes d'un composant. Il faut dans ce cas prévoir d'autres composants qui peuvent reprendre le traitement afin de garantir une continuité dans le service. Une panne peut

être due à l'instabilité du composant, l'atteinte de la capacité maximale de traitement ou à la défectuosité d'une carte réseau. D'autres causes peuvent subvenir, mais quelles qu'elles soient, il faut être capable de fournir les mêmes services par l'intermédiaire d'un autre composant. Nous avons déjà vu que les services peuvent tomber en panne sans pour autant priver l'utilisateur des possibilités des serveurs. En effet, plusieurs serveurs sont présents au sein d'un cluster, ce qui permet de récupérer le traitement sur un autre serveur en cas de panne de l'un deux.

Nous avons évoqué le besoin de tests pour évaluer la robustesse du système. Il est néanmoins possible d'estimer cette robustesse dans certains cas d'utilisation. En effet, les points fragiles du système sont le serveur lui-même, la base de données d'un cluster, l'*outbound proxy*, l'entrée d'un cluster (le *load balancer*) et le cluster lui-même. Nous exposons ici les problèmes que ces composants peuvent engendrer. Nous verrons à la section 3.2 comment nous pouvons essayer de remédier à ces faiblesses.

Si un serveur vient à défaillir au sein d'un cluster, les autres serveurs peuvent récupérer ses opérations et ainsi poursuivre le traitement en cours. Ceci s'effectue de manière transparente pour l'utilisateur. Cependant, s'il s'agit du dernier serveur en activité au sein du cluster, le service pour ces utilisateurs-là est indisponible, à moins de mettre en place un serveur de secours.

La base de données d'un cluster est également un point névralgique du cluster. Si celle-ci devait être indisponible, le cluster ne pourrait plus garantir la continuité du service pour les utilisateurs se connectant à ce cluster. Dans ce cas, il serait nécessaire de relancer une base de données ou d'en disposer d'une de secours. Dans cette deuxième option, un impératif est de mise : la synchronisation des bases de données. En effet, il est indispensable dans ce cas que les deux bases de données soient mises à jour simultanément. De plus, une transaction échouant sur une base de données ne peut réussir sur l'autre, il faut dès lors créer une transaction qui consiste à écrire dans plusieurs bases de données. Cela impliquerait plus de traitement et entraînerait une perte de performances.

Dans [19], l'auteur nous fait part de la faiblesse de la solution au niveau de l'*outbound proxy*. En effet, si celui-ci ne répond plus, l'utilisateur est obligé de réinitialiser ses connexions avec un autre *outbound proxy*. Il s'agit d'une perte de temps pour tous les utilisateurs connectés à cet *outbound proxy*. Ceux-ci

génèrent alors une masse de transactions pour réinitialiser leurs connexions. La solution proposée par l'auteur est de démultiplier les *outbound proxies*. Ceci permettrait à chaque *outbound proxy* de gérer moins de connexions. Une panne d'un composant engendrerait moins de connexions perdues et donc moins de transactions de régénération.

L'entrée du cluster est également une faiblesse de l'architecture puisque sans elle, le cluster n'est plus accessible. Aucune solution n'est apportée dans les travaux [19] et [10] et nous tenterons d'en apporter une à la section 3.2.

A un niveau d'abstraction supérieur, la perte d'un cluster est critique pour le système. En effet, un cluster possède une partie de la connaissance du système. Il s'agit principalement des utilisateurs qu'il gère. Si le cluster tombe en panne, le système perd une partie de ses utilisateurs. De plus, aucun mécanisme n'est défini dans ce cas, la conséquence peut être encore plus grave. Nous approfondirons ces problèmes à la section 3.2.3 et tenterons d'y apporter une solution.

3.1.3 Efficacité

Nous venons d'aborder la robustesse du système que nous approfondirons à la section 3.2. Le troisième critère de qualité est l'efficacité. Celle-ci nécessite impérativement la mise en place de scénarios de tests afin d'évaluer les performances sur les machines. Nous verrons au chapitre 4 les tests que nous prévoyons de réaliser.

Ces scénarios auront pour but de définir un temps de réponse moyen et ainsi pouvoir définir une limite maximale du nombre d'utilisateurs que peut gérer un serveur afin de rester efficace.

Nous avons vu à la section 3.1.1 que la solution proposée permet de gérer un plus grand nombre d'utilisateurs. Cependant, il est difficile de donner une valeur à ce nombre théoriquement. Afin de préciser ce nombre, des tests doivent être effectués. Ceux-ci permettront alors de donner un rapport de gain de performances entre un serveur unique et l'arborescence de clusters définie.

3.1.4 Hétérogénéité

L'hétérogénéité apparaît à trois niveaux : interne à l'application, au sein de l'architecture et en déploiement dans un environnement de production. Le deuxième niveau est apparenté à la portabilité et le dernier à l'interopérabilité.

L'hétérogénéité interne repose sur le principe de l'utilisation possible d'éléments dont nous n'avons pas le contrôle en plus de ceux développés spécifiquement pour le système. En effet, l'hétérogénéité est également l'exigence de pouvoir utiliser des composants externes. C'est ainsi que LVS a été intégré à la solution pour réaliser le clustering.

L'hétérogénéité au sein de l'architecture est assurée par la portabilité du serveur. Développé en langage Java, un serveur est facilement installable grâce aux propriétés de ce langage de programmation. Il est dès lors possible d'utiliser des machines exécutant des systèmes d'exploitation différents munis de la machine virtuelle Java. Cependant, l'installation du serveur s'effectue à l'aide d'un fichier d'installation, disponible pour Linux ou Windows. Ces deux systèmes étant les plus répandus, il est possible de trouver une machine pouvant héberger un serveur qui exécute l'un des deux. Si nous devons installer le serveur sur une station différente, nous devrions créer le programme d'installation ou réaliser manuellement cette installation.

De plus, les clusters étant répartis à travers le monde, les communications s'effectuent par l'Internet, qui est un réseau hétérogène puisqu'il utilise différents supports physiques (fibre optique, câble coaxial, etc.).

Ce deuxième niveau d'hétérogénéité est soumis à certaines limites. En effet, l'extension du service nécessite la mise en cluster des serveurs. Dans la solution développée lors de son stage [10], l'auteur propose de travailler avec LVS, qui est spécifique à Linux. Nous perdons ici la portabilité obtenue par le langage de développement. Une solution serait l'intégration du système LVS au sein de l'application ou trouver une application similaire offrant les mêmes fonctionnalités et étant portable.

Le dernier niveau d'hétérogénéité, l'interopérabilité, est la possibilité de communiquer avec d'autres systèmes. Des tests d'interopérabilité ont été effectués l'année dernière en utilisant les serveurs iptel SER et NIST JAIN-SIP Presence Proxy. Ceux-ci ont permis à l'auteur de [10] de vérifier que

les serveurs de la société Indigo Software peuvent communiquer avec des serveurs appartenant à d'autres sociétés. Ainsi des utilisateurs gérés par le domaine d'Indigo Software peuvent communiquer avec d'autres utilisateurs gérés par d'autres domaines.

3.1.5 Accès et partage des ressources

Chaque cluster est composé d'une base de données et de plusieurs serveurs qui se partagent celle-ci. Ils y accèdent en concurrence afin de traiter dans un délai assez bref une requête. L'accès est régulé suivant la politique du gestionnaire de bases de données. Le partage de ressources a également lieu au sein d'un serveur. En effet, l'architecture propose l'isolement des services. Un serveur de la société Indigo regroupe tous les services. Il est cependant possible d'exécuter chaque service séparément sur des machines différentes. Etant donné que chaque service est encapsulé dans une application dite serveur, chacun se partage les ressources physiques de la machine. Dans ce cas, les accès sont gérés par le système d'exploitation.

3.1.6 Transparences d'un système distribué

Le dernier point de notre grille d'analyse concerne les niveaux de transparence d'un système distribué.

La première transparence que nous examinons est celle de la localisation. Il s'agit de pouvoir atteindre une partie du système sans savoir où elle se trouve exactement. Dans l'architecture proposée à la section 2.2, cette transparence est vérifiée puisque l'utilisateur ne doit pas connaître les serveurs sur lesquels il peut se connecter. Ce mécanisme est automatique avec le nom de domaine. De plus les serveurs sont situés derrière un *load balancer* et ils ne sont pas directement accessibles.

La transparence de la migration est également présente étant donné l'aspect dynamique de la découverte des services et de l'arborescence de clusters. En effet, il est possible de déplacer un composant sans que les utilisateurs ne s'en aperçoivent. Pour cela, il suffit de l'arrêter et de le relancer ailleurs. Au niveau des clusters, la transparence est également vérifiée. En effet, la connaissance des clusters se fait par le mécanisme de souscription. Ainsi, un

cluster venant à être déplacé, il suffit également de l'arrêter et de le relancer à son nouvel emplacement.

La transparence des échecs demande que le système cache l'échec de certains composants. De plus, deux propriétés systèmes, *liveness* et *safeness*, demandent respectivement que ce que nous demandons se produise à un moment ou un autre et correctement. Par exemple, lors de la localisation de contact, il faut parfois remonter dans l'arborescence afin de trouver celui-ci. La recherche est donc infructueuse dans les clusters de niveaux inférieurs, nécessitant une redirection. Dans ce cas, l'utilisateur ne doit pas être averti des échecs intermédiaires. Tôt ou tard, il recevra une réponse indiquant la présence ou non de l'utilisateur sur le réseau. Il faut également que cette réponse soit correcte. En effet, annoncer qu'un contact est inconnu alors qu'il est présent sur le système (faux positif) est regrettable, mais il est encore plus déplorable d'annoncer qu'un contact est connu alors qu'aucun cluster n'a pu le trouver dans sa base de données (faux négatif).

Certains niveaux de transparence ne sont pas examinés. En effet, dans la solution établie, la relocalisation n'est pas exploitée, il nous est donc pas possible de l'évaluer. Nous venons de montrer que l'architecture cache les mécanismes de distribution aux utilisateurs.

3.2 Problèmes de robustesse

Comme nous l'avons expliqué aux sections 3.1 et 3.1, la robustesse n'est pas respectée à tous les niveaux d'abstraction. En effet, les services sont robustes puisqu'ils sont répliqués en plusieurs instances. De ce fait, si l'un ne répond plus, un autre peut prendre le relais afin de garantir le service offert. Cependant, au niveau du *load balancer* et de l'*outbound proxy*, il existe une faiblesse.

Nous allons dans un premier temps explorer les problèmes de robustesse du *load balancer*, ensuite ceux de l'*outbound proxy*. Une troisième partie de cette section sera consacrée à la robustesse de l'arborescence. En effet, l'arbre est défini ainsi que les interactions verticales entre clusters. Cependant, si un cluster venait à disparaître, l'arbre doit s'adapter afin de garantir le service. Cette dernière partie sera donc exclusivement consacrée aux causes et aux conséquences de la disparition d'un noeud de l'arbre, et nous tenterons d'apporter une solution au problème.

3.2.1 Le *load balancer*

Le *load balancer* constitue le seul point d'entrée d'un cluster. Si celui-ci vient à tomber en panne, l'ensemble du cluster devient inaccessible et provoque la perte du cluster dans l'arborescence. Outre l'inaccessibilité des utilisateurs gérés par ce cluster, la conséquence la plus importante est la scission de l'arbre en plusieurs sous-arbres. Nous détaillerons les conséquences de la perte d'un cluster à la section 3.2.3.

Le *load balancer* est donc une charnière importante et nous ne pouvons pas concevoir le système sans lui. En effet, les serveurs d'un cluster sont accessibles par le *load balancer* et ne sont pas visibles de l'extérieur. La figure 3.1 montre les interactions entre les différentes entités d'un cluster ainsi que l'interaction entre le cluster et le monde extérieur.

Nous y observons que le *load balancer* est le seul point d'entrée du cluster. En effet, il est le seul élément du cluster à communiquer avec le monde extérieur. Ceci explique pourquoi une panne du *load balancer* serait critique.

Afin d'éviter la perte du cluster, nous proposons de répliquer le *load balancer*. Dans ce cas de figure, plusieurs machines ayant la même adresse IP sont disponibles. Seule l'une d'entre elles effectue la répartition de la charge, les autres sont présentes en cas de problème, pour prendre la relève.

La réplication est donc intéressante, mais suscite des questions relatives à la cohérence des données de gestion des transactions. Nous avons vu à la section 2.2.3 que le *load balancer* est intelligent puisqu'il redirige les messages vers celui des serveurs qui gère déjà le dialogue concerné. La solution apportée se situe au niveau de la couche IP du modèle TCP/IP et se base sur une clé, composée de la source et de la destination du message. Ceci évite de remonter dans les couches et d'effectuer des opérations plus coûteuses en temps.

Afin de garder une cohérence entre les *load balancers*, une synchronisation des données doit être réalisée. La première solution est de gérer ces informations à l'aide d'une base de données. Cependant, comme nous l'indique [19], cette solution devient vite le goulot d'étranglement étant donné la fréquence élevée des modifications des dialogues. La gestion de cette information a donc lieu en mémoire sur la machine effectuant le *load balancing*. La solution que nous apportons est la mise à jour des informations mémorisées par les *load*

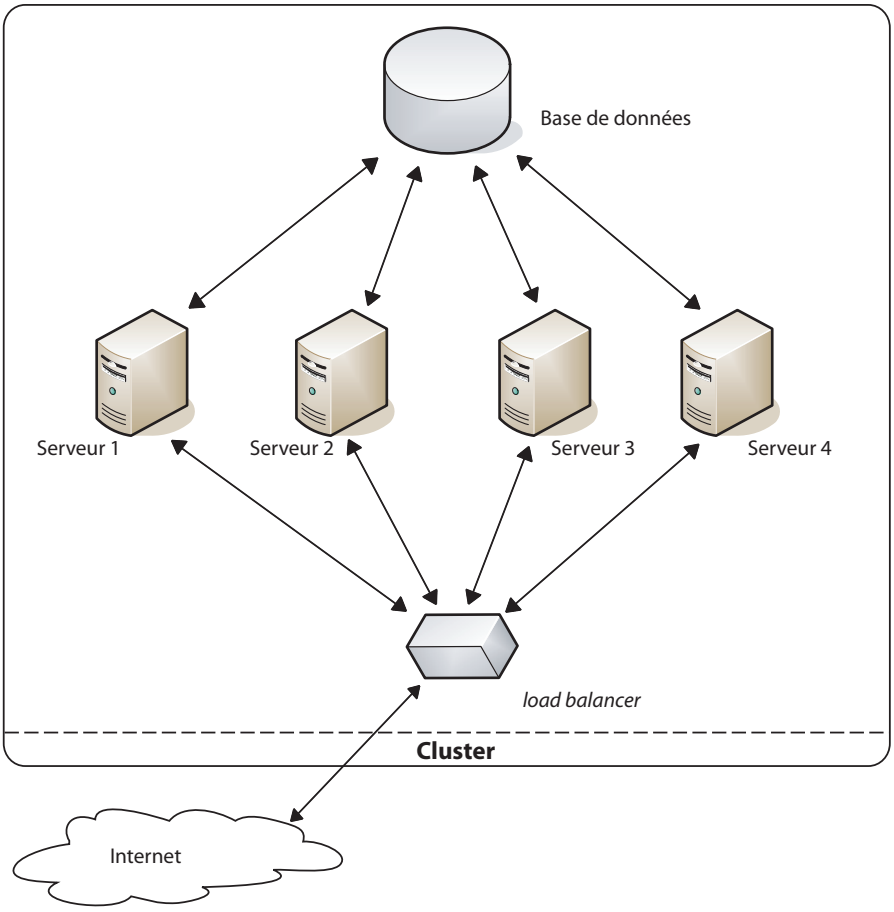


FIG. 3.1 – Exemple de structure d'un cluster.

balancers de secours dès qu'une modification est réalisée sur le *load balancer* actif. Afin de connaître les différentes destinations, chaque *load balancer* s'enregistre auprès de tous les autres.

Grâce à ce mécanisme de souscription, chaque *load balancer* a une connaissance complète et l'actif peut envoyer à chacun de ses substituts les changements de données. Dès lors, à tout instant, le *load balancer* actif peut être arrêté et un autre prendra la main.

Cette solution risque cependant d'inonder le réseau. En effet, si nous envoyons chaque modification de dialogue, étant donné que cela se produit

régulièrement, nous risquons d'émettre un nombre élevé de messages sur le réseau. Ce phénomène est assimilé à de l'inondation de réseau. Afin d'éviter ce problème, nous pouvons agréger les modifications. Un tel agrégat ne contiendrait que la dernière donnée de chaque dialogue, si celle-ci a été modifiée pendant l'intervalle de temps défini entre deux agrégations. Ceci peut engendrer une perte de données. En effet, si aucune modification de dialogue n'a lieu entre l'envoi du dernier agrégat et la panne, la reprise par un autre *load balancer* se fera avec un contexte cohérent. Cependant, si une ou plusieurs modifications ont lieu après l'envoi du dernier agrégat, les autres *load balancers* n'auront pas le même contexte lors de la panne. Le système est dans ce cas incohérent.

L'inondation du réseau peut également être évitée en utilisant une interface réseau différente, par exemple une connexion dédiée à la communication entre les *load balancers*. La figure 3.2 montre un exemple de conception physique résolvant le problème d'inondation du réseau par l'utilisation de cette interface supplémentaire.

Cet exemple est composé de trois *load balancers* accessibles depuis le monde extérieur et connectés aux serveurs par un réseau local, représenté par les traits fins. Chaque *load balancer* dispose également d'une troisième interface réseau, utilisée pour le dialogue des *load balancers* et constituant un second réseau local, représenté par les traits pointillés. Les connexions vers le monde extérieur sont quant à elles représentées à l'aide de traits plus épais.

Une autre possibilité serait de placer les serveurs et les *load balancers* conjointement sur un même réseau local, relié au monde extérieur par l'intermédiaire d'un routeur. Ce dernier ne laisserait transiter des messages qu'à destination des *load balancers*. D'autres réalisations physiques peuvent être imaginées mais nous nous limiterons à ces deux exemples donnés dans un but de compréhension.

L'exemple de la figure 3.2 expose une solution physique au problème de l'inondation et impose que chaque *load balancer* dispose de trois interfaces afin de connecter trois réseaux différents. Notre but n'est pas de fournir des solutions physiques au problème de l'inondation, mais nous offre l'opportunité de sensibiliser le lecteur aux possibilités pour l'éviter.

L'inondation n'est pas catastrophique étant donné les technologies de réseau que nous connaissons à l'heure actuelle. De plus, les messages échangés

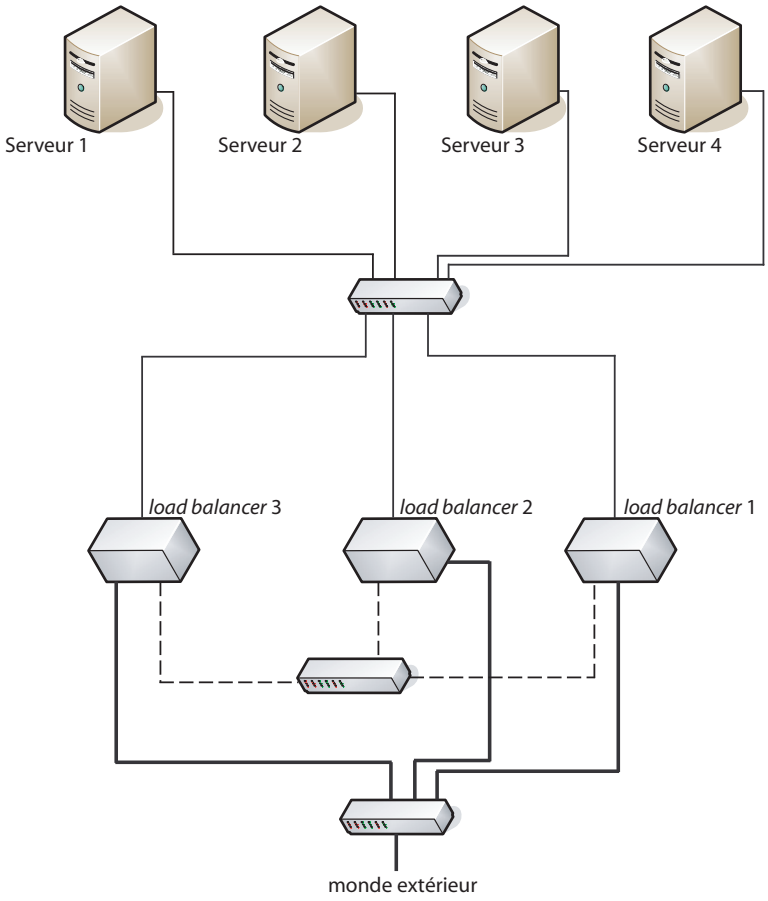


FIG. 3.2 – Exemple de conception physique.

entre les *load balancers* sont de très petite taille, ce qui ne provoque pas trop d'embouteillage. Cependant, si les serveurs hébergeant les services de la société Indigo Software ne leur sont pas dédiés, le réseau peut alors être utilisé pour des transferts de multimédia. Dans ce cas, l'inondation peut devenir pénalisante et l'exemple de la figure 3.2 peut être une bonne solution à envisager par l'administrateur du réseau.

Le *load balancer* doit pouvoir être configuré afin de travailler sur différentes interfaces. En effet, selon la topologie du réseau sur lequel se situent les *load balancers*, l'interface à utiliser pour communiquer avec les serveurs peut être différente de celle utilisée pour la communication entre les *load*

balancers. Il est dès lors important de permettre la configuration des *load balancers* afin d'être modulable pour chaque topologie.

3.2.2 L'*outbound proxy*

Nous venons de parler du problème de robustesse lié à la panne d'un *load balancer*. La solution était la réplication de celui-ci. Dans cette section, nous allons explorer les problèmes liés à la panne de l'*outbound proxy*. Nous verrons que la solution se base également sur la réplication de l'élément.

L'*outbound proxy* est le point d'entrée dans le réseau SIP pour les utilisateurs. Afin de préserver les performances obtenues avec l'architecture définie au chapitre 2, plusieurs *outbound proxies* sont mis en place. En effet, il est inconcevable de faire entrer tous les utilisateurs du système par une seule porte alors que le système défini permet une gestion bien plus efficace. Nous comparons cela à ouvrir une seule porte d'un mètre vingt de large pour les quatre-vingt mille visiteurs du Stade de France. Comme pour le stade de France plusieurs portes sont utilisées, plusieurs *outbound proxies* sont déployés.

Les *outbound proxies* sont répartis géographiquement pour en améliorer la gestion. Chacun renverra les utilisateurs s'y connectant à un cluster proche géographiquement. Si l'un d'eux tombe en panne, les utilisateurs doivent se réorienter vers d'autres. Ainsi, un utilisateur africain pourrait être redirigé vers un *outbound proxy* aux Etats-Unis. Ceci amènerait des délais de transmissions supplémentaires.

Cette fragilité a déjà été pointée dans la conclusion de [19]. L'auteur y donne une solution qui est de multiplier les *outbound proxies* gravitant autour des clusters. Il est ainsi possible de mettre en place plusieurs *outbound proxies*, rapprochés géographiquement et pouvant accepter les connexions des utilisateurs gérés par plusieurs clusters.

Nous savons que les utilisateurs entrant dans le système par l'*outbound proxy* qui tombe en panne devront réinitialiser leurs transactions avec un autre *outbound proxy*. La multiplication d'*outbound proxies* autour des clusters permet de répartir les utilisateurs uniformément sur ces points d'entrées. Une solution pratique pour cette répartition est l'utilisation d'enregistrements DNS de type SRV, que nous avons présentés à la section 2.3.1.

Ceci permet donc de réduire la charge de travail des *outbound proxies*. Cependant, nous ne devons pas réduire la capacité de traitement des machines puisque chaque *outbound proxy* doit être capable de récupérer les utilisateurs d'une entrée défaillante, ce qui engendrerait une augmentation de la charge.

Comme nous l'avons vu à la section 2.3.2, chaque *outbound proxy* garde en mémoire cache de l'information relative aux résultats des localisations de contacts. A première vue, cette information ne sera pas identique sur chacun des *outbound proxies* et nous pouvons imaginer un mécanisme de communication entre eux afin de les synchroniser pour qu'ils aient la même information.

La solution proposée pour mettre à jour le contexte de répartition de charge à la section 3.2.1 peut être répétée entre les *outbound proxies*. Ceci engendrerait beaucoup de messages au lancement du système et si la mémoire n'a pas besoin d'être modifiée régulièrement (mémoire saturée), le nombre de messages transmis diminuera fortement.

Dès qu'un *outbound proxy* tombe en panne, les utilisateurs n'arrivent plus à le joindre et doivent en contacter un autre. Celui-ci, ayant une connaissance identique à celui qui ne répond plus, grâce à la synchronisation des *outbound proxies*, peut reprendre directement la gestion des connexions de ces utilisateurs. Nous évitons ainsi de réinitialiser des recherches ayant déjà été exécutées.

Cependant, dans une même région géographique, il est fort probable que plusieurs utilisateurs aient en commun une liste d'amis. Il est également probable que ces utilisateurs soient redirigés vers des *outbound proxies* différents lors de leur connexion. Ces différents serveurs d'entrée initieront une localisation des contacts à chaque souscription reçue. Ils auront donc tous une vue quasi similaire des utilisateurs dans leur mémoire. Nous parlons de vue quasi similaire puisqu'il est rare de trouver deux utilisateurs ayant une liste d'amis entièrement identique.

Pour ces raisons, nous proposons de ne pas mettre en place un mécanisme de diffusion de mémoire afin de ne pas alourdir le traitement des *outbound proxies*. La récupération en cas de panne d'un *outbound proxy* s'effectue assez rapidement car la majorité des contacts est déjà localisée par les autres *outbound proxies*.

3.2.3 Robustesse de l'arborescence

Nous venons d'aborder les problèmes de robustesse engendrés par la perte du *load balancer* ou de l'*outbound proxy*. Si nous remontons d'un niveau d'abstraction, nous pouvons nous demander ce qu'il se produit lorsqu'un noeud de l'arbre est inaccessible. Nous allons étudier, dans cette section, les conséquences et dysfonctionnements au sein de l'arbre.

L'arborescence des clusters peut être l'objet de dysfonctionnements, pouvant provoquer la perte d'un cluster complet. Dans ce cas, aucun des travaux [19] et [10] à la base de notre étude n'explique comment le système réagit à ce type de panne. Afin d'illustrer le principal problème causé par une perte de cluster dans l'arborescence, considérons la figure 3.3. Supposons donc que le cluster C ne réponde plus. Le cluster A, dans ce cas, ne connaît plus que deux fils : les clusters B et D. Dès lors, la recherche d'un contact géré par le cluster G, initiée par l'un des clusters connus de A (B, D, E, F, I et J), échouera toujours. De même, si la recherche est générée par H, le contact ne sera pas trouvé. Cela pose d'énormes problèmes puisque l'utilisateur est connu du système dans son entièreté. En effet, suite à la perte d'un cluster, le système n'est plus vu comme une seule entité, mais comme deux voire plus selon les cas.

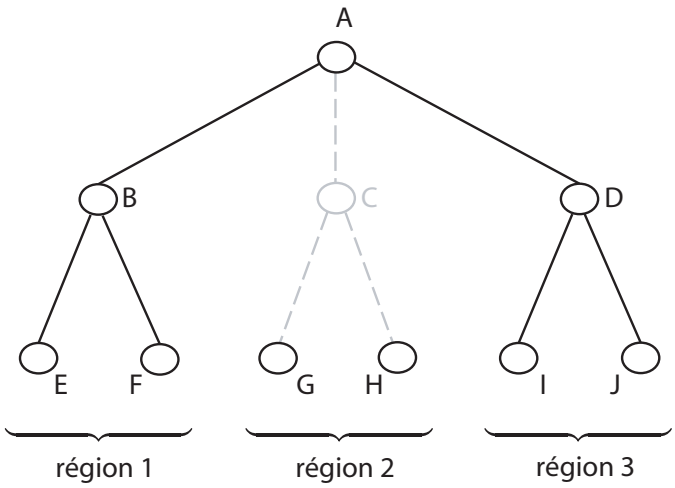


FIG. 3.3 – Exemple de la perte d'un cluster.

Le meilleur des cas reste celui de la perte d'un cluster situé à l'extrémité

d'une branche et représenté par une feuille dans l'arbre. En effet, dans ce cas, le système constitue toujours une seule entité mais a perdu une partie de sa connaissance des utilisateurs, ceux gérés par le cluster défaillant. Le seul cas dans lequel le système est vu comme deux entités distinctes est celui où la racine de l'arbre ne fonctionne pas correctement et qu'elle ne possède que deux fils. Dans ce cas, l'arbre est coupé en deux et chaque sous-arbre possède comme racine l'un des fils du cluster ayant provoqué la coupure. D'une manière générale, nous pouvons déterminer le nombre d'entités engendrées en fonction du nombre de fils que possède le cluster défaillant. La correspondance se fait grâce au tableau **??**.

Type de noeuds	**Nombre d'entités**
racine à f fils	f
intermédiaire à f fils	$f+1$
feuille	1

TAB. 3.1 – Liste des codes de réponses SIP par classes.

Dans l'exemple de la figure 3.3, le cluster C ne répond plus et le système est coupé en trois entités. Nous pouvons vérifier aisément le nombre de sous-arbres générés. La formule nous indique 3 systèmes. En effet, les deux premiers systèmes sont engendrés par les fils du cluster C, au nombre de deux. Le troisième est généré par le père et le reste de l'arborescence. Les trois systèmes sont représentés à la figure 3.4. La région 2 du système initial, composée des clusters C, G et H, n'est plus connue des régions 1 et 3. Nous remarquons donc un problème de connaissance du système et allons proposer une solution pour reconstruire l'arbre.

Afin de donner une solution au problème évoqué, nous devons cibler la source ou les sources de celui-ci. Dans le problème qui nous occupe, qui est la perte d'un cluster dans l'arborescence, nous relevons trois sources : l'entrée du cluster qui ne répond plus, l'ensemble des serveurs d'un cluster qui ne répondent plus ou un problème de réseau.

La première source de problème évoquée a été abordée précédemment au point 3.2.1. Nous rappelons la solution qui est de répliquer les *load balancers* en utilisant une seule adresse IP. Ainsi, lorsque l'un ne répond plus, un autre reçoit automatiquement les messages et le service reste disponible.

Si l'ensemble des serveurs du cluster ne répond plus, celui-ci est inaccessible. Dans certains cas, la cause est un mauvais dimensionnement du clus-

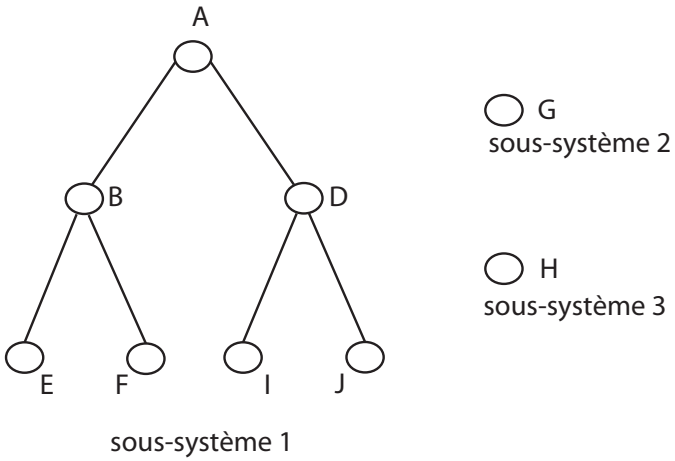


FIG. 3.4 – Exemple de la perte d'un cluster

ter. En effet, si chaque serveur atteint sa capacité maximale de traitement, il risque de saturer et de ne plus répondre aux différentes requêtes. Dans ce cas, un redimensionnement du cluster est nécessaire. Si plusieurs serveurs sont interrompus simultanément, il ne suffit pas toujours de lancer un voire plusieurs serveurs pour résoudre le problème, mais il est indispensable de diagnostiquer correctement la cause de cet échec collectif. Par exemple, il se peut qu'une coupure de courant provoque l'arrêt du cluster. Si celle-ci est provoquée par une utilisation excessive de la ligne électrique, il faut alors renforcer la ligne afin d'offrir une plus grande capacité électrique pour le cluster.

Si la panne du cluster est provoquée par une panne au niveau du réseau, il faut éventuellement prévoir plus d'entrées sur le réseau, mettre en place des liaisons de secours, afin que plusieurs chemins d'accès soient disponibles.

Quelles que soient la cause et la durée de la panne, nous devons trouver une solution pour que le système demeure une seule entité. Cette solution dépend du cluster qui est perdu : un cluster feuille, le cluster racine ou un cluster intermédiaire (nous entendons par cluster intermédiaire tout cluster qui n'est ni racine, ni feuille).

Dans le premier cas, le cluster est perdu et nous ne pouvons qu'attendre que celui-ci soit remis en marche. En effet, la perte d'une feuille dans l'arbre

ne scinde pas le système en plusieurs entités. Ce cas est donc facile et ne demande aucune modification du mécanisme de souscription établi pour la connaissance des clusters, mécanisme que nous avons expliqué à la section 2.2.4.

La perte de la racine de l'arbre est plus complexe que la perte d'un cluster intermédiaire. En effet, la coupure provoquée engendre plusieurs sous-systèmes qui n'ont aucune connaissance l'un de l'autre. Leur seul point commun est celui du nom FQDN de la racine présent dans leur propre FQDN.

Dans un premier temps, nous exposons la solution à la perte d'un cluster intermédiaire. Nous exposerons ensuite une solution à la perte de la racine.

Perte d'un cluster intermédiaire

La perte d'un cluster intermédiaire est résolue par le principe de souscription et se base sur les noms des clusters. La solution se décompose en deux parties, suivant le moment où la perte a été détectée. En effet, la panne du cluster peut être détectée soit lors d'un rafraîchissement de souscription d'un cluster fils, soit encore lors de l'envoi d'un message du cluster père, soit lors de l'envoi d'une réponse d'un cluster fils.

Le premier cas de figure, lors du rafraîchissement de la souscription est le plus simple car aucune transaction n'est en cours entre les clusters. Dès lors, tous les fils détecteront la panne après un temps maximal équivalent au temps qu'il y a entre deux messages SUBSCRIBE. Le mécanisme proposé dans ce cas est la souscription des clusters fils auprès de leur grand-père. En effet, avec le FQDN, ils connaissent le nom de leur grand-père et peuvent très facilement prendre contact avec lui afin de se faire connaître. Afin d'y arriver, ils retirent de leur FQDN, leur nom et celui de leur père. Par exemple, si le cluster `ile-de-france.fr.eu.society.com` détecte la perte de son père (`fr.eu.society.com`), il souscrira auprès du cluster `eu.society.com`. Le cluster grand-père obtient ainsi de nouveaux fils et le cluster défectueux sera retiré de cette liste après l'intervalle de temps entre deux messages SUBSCRIBE. En effet, après cet intervalle, si le cluster ne reçoit pas ce renouvellement de souscription, il considère que le cluster est absent dans l'arborescence et le retire de sa liste.

Dans le second cas traité, la perte est détectée lorsqu'un message doit être transmis au cluster défectueux. Dans des conditions normales, le grand-père doit transmettre à tous ses fils ce message, qui est généralement une requête de localisation. Au cas où le fils ne répondrait pas, il doit garder cette requête afin de la transmettre à ses petits-fils qui souscriraient auprès de lui. La durée, pendant laquelle il doit conserver cette requête, est définie par rapport à l'intervalle de temps entre deux messages SUBSCRIBE, soit t_{SUB} cet intervalle. Le temps maximal d'attente est d'une fois cet intervalle. La figure 3.6 permet de visualiser plus clairement ce calcul. Il se base sur la petite arborescence définie à la figure 3.5, dans laquelle les clusters 3 et 4 sont fils du cluster 2 qui est lui-même le fils du cluster 1. Notre exemple se base sur la panne du cluster 2 qui est détectée lorsque le cluster 1 émet un message à destination de son fils.

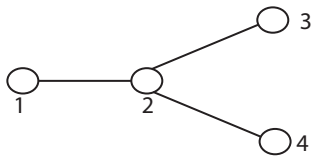


FIG. 3.5 – Mini arborescence de quatre clusters.

Afin de faciliter la compréhension du diagramme (figure 3.6), le lecteur doit interpréter la croix présente sur la ligne de vie du cluster 2 comme étant l'instant où la panne se produit et les ronds noirs à l'extrémité fléchée de certains messages comme étant la détection de la perte du message. En effet, lorsqu'une destination n'est plus disponible, le message se perd et une réponse d'inaccessibilité de la destination est soit réceptionnée, soit générée par un *timeout*.

N'obtenant pas de réponse de la part du cluster 2, la panne est détectée. Le cluster 1 doit attendre au maximum le temps d'un intervalle entre deux messages SUBSCRIBE. En effet, lorsque le cluster 1 s'aperçoit de la panne du cluster 2, après l'envoi du message 2, il attend au maximum le même temps que l'intervalle défini entre deux messages de renouvellement de souscription. Durant cet intervalle, à chaque nouvelle souscription identifiée comme venant d'un fils du cluster défaillant, il peut lui envoyer la requête à exécuter. Ainsi, l'exemple montre la détection de la perte par les clusters 3 et 4, respectivement, après les messages 6 et 3. Suite à cette découverte, ils peuvent souscrire auprès du cluster grand-père (messages 7 et 4). Comme

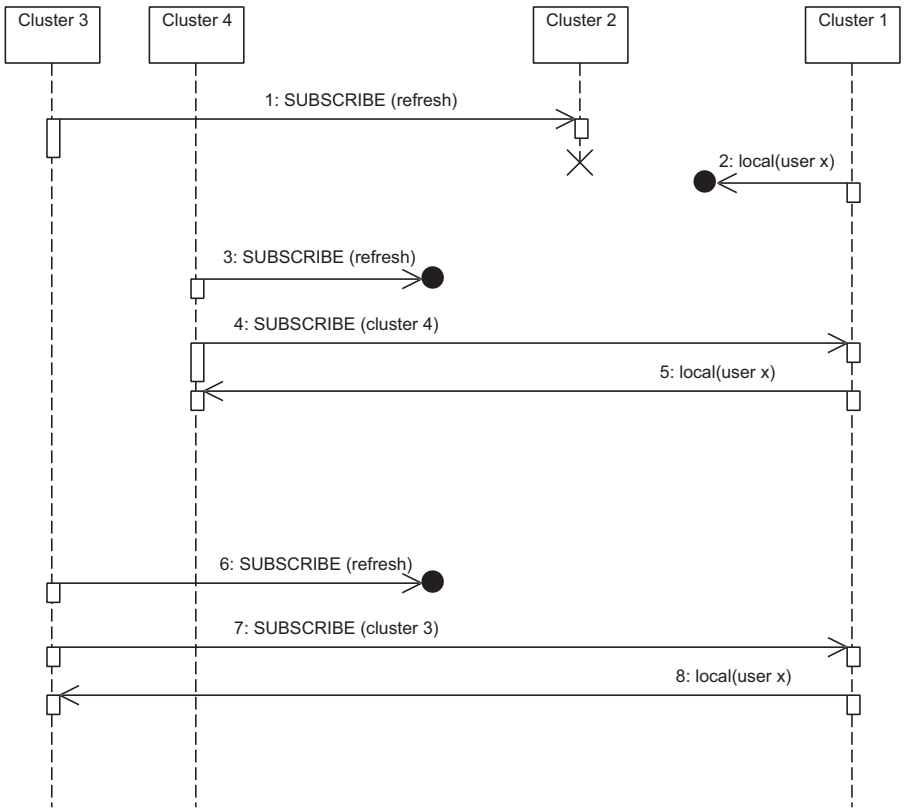


FIG. 3.6 – Diagramme de séquence illustrant la découverte de la perte du cluster 3.

nous l'avons dit, à la réception de ces messages, le cluster 1 émet les messages 8 et 5 vers, respectivement, les clusters 3 et 4.

Le troisième cas où la perte d'un cluster est détectée est celui où le fils répond à une requête et que ce message n'aboutit pas. Le fils détecte ainsi la perte et souscrit auprès de son grand-père. Si la requête concerne une localisation, le grand-père est lui-même intéressé par la réponse et le petit-fils peut la transmettre directement auprès du grand-père.

Quand la perte est découverte par le fils, quel que soit le cas traité (le premier ou le troisième), dès que le grand-père reçoit une souscription venant d'un petit fils, il peut en déduire la panne de son fils dont le nom est

obtenu en retirant le premier nom du FQDN du petit-fils. Il peut ainsi avoir une connaissance plus rapide du problème et prévenir ses autres fils de la suppression d'un cluster.

Perte du cluster racine

Le cas de figure où la racine de l'arbre vient à être instable demande plus de réflexion. En effet, les fils n'ont pas de grand-père dans l'arborescence et le principe que nous venons d'exposer ne peut pas être appliqué.

La première solution qui nous est venue à l'esprit est celle de changer la connaissance dans l'arbre et d'imposer aux fils de se connaître l'un l'autre. Nous proposons donc d'introduire une relation de fraternité en plus de la filiation préalablement établie. Celle-ci demande un changement de philosophie dans l'arborescence pour introduire également une connaissance horizontale partielle.

Lors d'un entretien avec l'auteur de [19], une solution a été trouvée : la mise en place de plusieurs clusters connus comme un seul en racine de l'arbre. Cette solution vise donc à renforcer la robustesse de la racine. Ainsi, si l'un de ces clusters venait à tomber en panne, un autre cluster peut reprendre la main sans devoir modifier le reste de l'arbre.

Cependant, visant à garantir une robustesse dans la majorité des cas de figure possibles, nous sommes revenus à la première solution que nous avons évoquée, la connaissance des frères dans l'arbre. En effet, comment définir le nombre de clusters utiles pour garantir la robustesse de la racine ? De plus, si tous les clusters présents en racine de l'arbre viennent à défaillir simultanément, nous nous retrouvons dans le même cas de figure.

La solution que nous proposons demande plus de ressources au sein des clusters afin de connaître l'ensemble des clusters frères. De plus, afin de trouver une nouvelle racine à l'arbre, une élection doit avoir lieu entre les héritiers, ce qui nécessite du temps pour stabiliser la structure. Différentes politiques d'élection peuvent être établies. Parmi celles-ci, nous pouvons citer : le premier cluster dans l'ordre alphabétique, le cluster qui peut trouver le plus grand nombre d'utilisateurs dans sa descendance et sur base d'un indice de priorité défini par les administrateurs. Le plus simple et le plus rapide se base sur le critère alphabétique. Chaque cluster maintient une liste

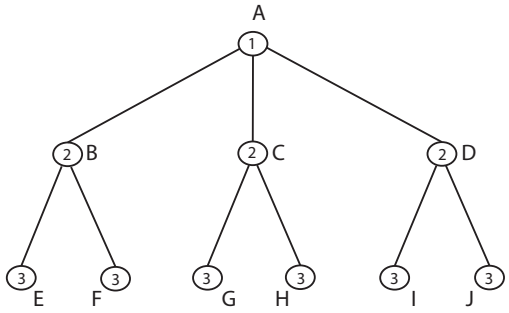
des frères triée dans l'ordre alphabétique. Dès la découverte de la perte du cluster racine, ils peuvent souscrire auprès du premier dans la liste.

Cette liste doit être tenue à jour et demande des échanges de message entre les clusters. En effet, dès qu'un cluster s'identifie auprès de son père pour la première fois, ce dernier doit prévenir l'ensemble de ses fils afin qu'ils connaissent l'ensemble des clusters de leur niveau. De même, dès que le père détecte la perte d'un fils, que ce soit suite à un message qui n'aboutit pas ou par la souscription d'un cluster identifié comme étant un petit-fils, il doit répercuter la nouvelle de la perte de celui-ci afin qu'il ne soit pas pris en compte lors d'une éventuelle élection future.

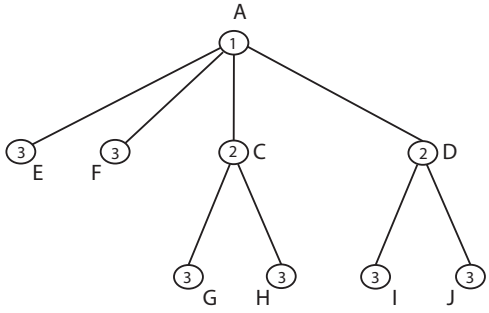
Enfin, la nouvelle racine peut à nouveau échouer et provoquer une nouvelle élection parmi tous les frères. Cependant, si nous comparons cette structure à un arbre généalogique, nous trouvons des neveux qui montent au niveau de leurs oncles. En effet, les fils du cluster élu comme racine sont montés d'un niveau dans l'arborescence et côtoient ainsi les frères initiaux de leur père. Si la nouvelle racine tombe en panne, une nouvelle élection a lieu et ne doit prendre en compte que les clusters dont le FQDN est de profondeur minimale. Nous entendons par là, les clusters dont le nom comporte le moins de niveau. Par exemple, le cluster `fr.eu.society.com` est de profondeur 4 alors que celle de `wallonie.be.eu.society.com` est de 5.

La figure 3.7(a) représente l'arbre initial. Les valeurs contenues dans les cercles symbolisant les clusters indiquent la profondeur du cluster. Cette donnée n'est pas modifiée suite à une restructuration de l'arbre et dépend du nom du cluster. La figure 3.7(b) montre l'arbre recomposé après une élection engendrée par la panne du cluster A et le choix de B pour nouvelle racine. Les clusters E et F deviennent les frères de C et D. Lorsque le cluster B n'est plus disponible, seuls les clusters C et D peuvent prétendre à la place de racine. En effet, ils ont une profondeur de 2 contre 3 pour les clusters E et F.

Ce mécanisme résout la panne du cluster racine. Celle-ci peut se produire en même temps que celle d'un cluster fils de la racine. Ce cas de figure est suffisamment rare pour être jugé exceptionnel. De plus, l'effort à fournir pour résoudre ce cas de figure demande la mise en place de beaucoup de ressources. La probabilité de cet évènement étant très faible, il n'est pas intéressant de mettre en place tout ce mécanisme coûteux en temps et en ressource pour un cas de figure qui pourrait bien ne jamais se produire.



(a) Arborescence initiale.



(b) Arborescence après une élection.

FIG. 3.7 – Exemple de niveau et profondeur pour l'élection.

Restauration de l'arborescence dans son état initial

Nous venons de voir comment résoudre un problème de panne d'un cluster. Dès que celle-ci est détectée, il est important de remettre le cluster en fonctionnement afin de rééquilibrer l'arbre. Le mécanisme décrit ci-dessus met en place une solution temporaire. Il est donc évident que cette solution doit être adaptée pour mettre à jour automatiquement l'arborescence suite au retour d'un cluster.

Lorsque le cluster se reconnecte, il souscrit à son père. Dans l'exemple de la petite arborescence de la figure 3.5, le cluster 2 est absent et les clusters 3 et 4 sont connectés auprès du cluster 1. Lorsque le cluster 2 revient dans le système, il souscrit auprès de son père. Celui-ci doit prévenir ses petits fils. Afin d'éviter une surcharge de messages, nous proposons que le cluster attende que les petits-fils renouvellent leur souscription. Nous illustrons ce

mécanisme à la figure 3.8, dans laquelle, le trait horizontal situé au début de la ligne de vie du cluster 2 indique le moment où celui-ci est revenu. Le temps nécessaire au rétablissement de l'arbre est donc égal à l'intervalle de temps entre deux messages SUBSCRIBE.

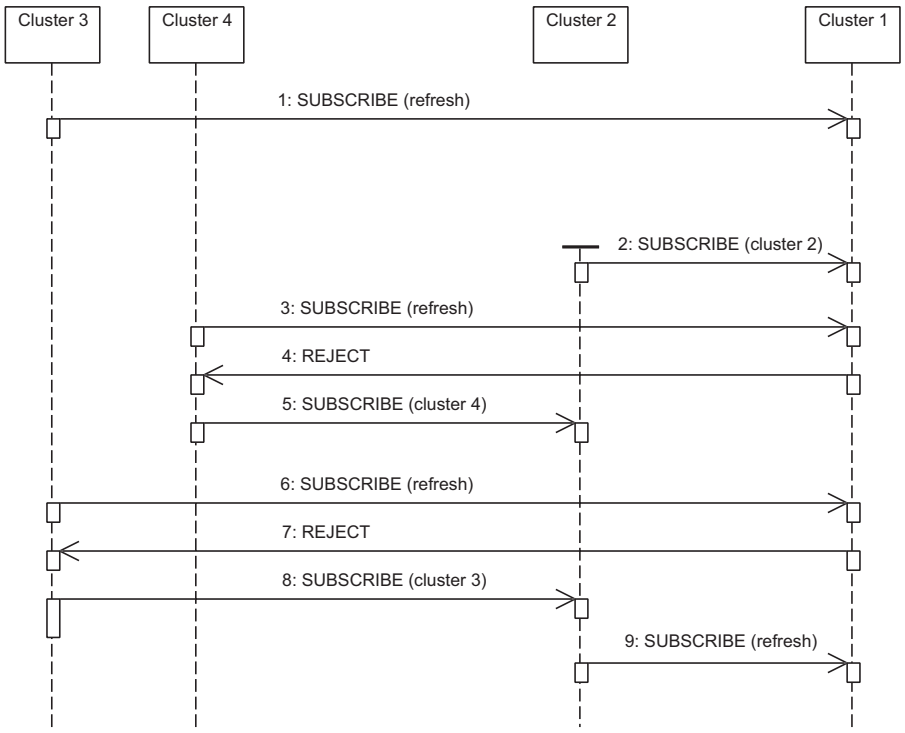


FIG. 3.8 – Diagramme de séquence pour le retour du cluster défaillant au sein de l'arborescence.

La figure 3.8 représente le retour du cluster 2 qui souscrit au cluster 1 (message 2). Les clusters 3 et 4 sont enregistrés auprès du cluster 2. Lorsqu'ils renouvellent leur souscription auprès du cluster 1 (messages 6 et 3, respectivement pour les clusters 3 et 4), ils reçoivent un refus (messages 7 et 4). A cet instant, ils savent que leur père est de retour et peuvent s'y souscrire (messages 8 et 5). Lors de la réception des messages 6 et 3 auprès du cluster 1, ces deux clusters peuvent être retirés de la liste des fils. Ce retrait entraine la modification de la liste des frères auprès des autres fils du cluster 1. A la fin d'un temps maximal égal au temps de rafraîchissement de la souscription, l'arborescence redevient ce qu'elle était (cf. figure 3.5).

Nous venons d'expliquer le principe et allons montrer que celui-ci ne nuit pas à la localisation d'un contact. En effet, lorsque le cluster 1 reçoit une requête pour localiser l'utilisateur X, il la transmet à ses fils s'il ne le connaît pas lui-même. Supposons que le cluster 4 s'est vu rejeté puisque son père (le cluster 2) est de retour et que le cluster 3 n'a pas encore renouvelé sa souscription auprès du cluster 1. L'arborescence est à ce moment identique à la figure 3.9.

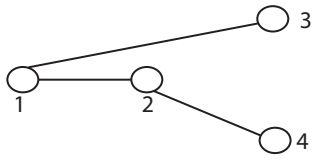


FIG. 3.9 – Arborescence intermédiaire dans le processus de rétablissement de l'arbre.

La requête est donc transmise aux clusters 2 et 3. Le cluster 2 possède un fils à cet instant, le cluster 4, qui recevra la requête et pourra l'exploiter. Le cluster 3 renverra sa réponse au cluster 1 tout comme le 2. La réponse du 2 sera composée du résultat de la recherche au sein de ce cluster et de celui du cluster 4. Ainsi la connaissance des utilisateurs est correcte.

Une autre solution, se basant sur le même principe, peut être développée. Cependant, celle-ci risque de poser des problèmes de cohérence du système et génère deux fois plus de messages. Pour que le lecteur en ait un aperçu et comprenne le choix proposé ci-dessus, nous l'expliquons dans les grandes lignes.

Au lieu de demander aux clusters petits-fils de renouveler leur souscription et d'éventuellement recevoir un refus, nous pouvons leur demander de vérifier si leur père ne revient pas. Il faut dès lors déterminer l'instant auquel nous effectuons cette vérification. Deux possibilités sont envisagées : à la fin de l'intervalle de rafraichissement, à savoir avant le renouvellement de souscription chez le grand-père, ou au milieu de cet intervalle.

Etant donné que nous ne pouvons pas être enregistrés auprès de deux clusters simultanément, nous ne pouvons pas souscrire simultanément au père, pour vérifier qu'il n'est pas relancé, et au grand-père, pour rafraichir l'enregistrement. Dans ce cas, si le père est à nouveau actif, nous nous retrouverions avec deux souscriptions actives. Nous devons donc attendre le

résultat de la vérification avant d'éventuellement envoyer le rafraichissement auprès du grand-père. Cette solution n'est pas performante : le temps nécessaire à la découverte de l'absence du père provoque le dépassement de l'intervalle et entraine auprès du grand-père la perte du cluster. Cette perte est temporaire puisque le cluster doit renouveler sa souscription. Cependant, s'apercevant du dépassement de délai, le grand-père modifie sa liste de fils et prévient chacun d'eux de cette modification. Il reçoit ensuite la nouvelle souscription et doit à nouveau modifier cette liste et avertir ses fils du changement.

La seconde option est d'effectuer la vérification du père au milieu de l'intervalle de temps défini entre deux messages SUBSCRIBE. Cette solution semble bonne mais le temps d'indisponibilité du cluster est invariable et durant cette période, les messages de souscription sont doublés, un vers le père pour tester l'absence de celui-ci et l'autre vers le grand-père pour un rafraîchissement. En effet, si l'intervalle de temps entre deux messages vers le grand-père est de t_{SUB} secondes, après $\frac{t_{SUB}}{2}$ secondes, le cluster effectue une vérification.

Pour la comparaison des deux solutions, supposons que la durée d'indisponibilité est t_{ind} et que l'intervalle de temps entre deux souscriptions est t_{SUB}. La première solution propose d'envoyer un message toutes les t_{SUB} secondes. Ainsi, en tout il y aura $\frac{t_{ind}}{t_{SUB}}$ messages envoyés au grand père, soit y ce nombre. L'ensemble du mécanisme demande donc l'envoi de $y + 1$ messages SUBSCRIBE. La deuxième solution demande l'envoi d'un message toutes les t_{SUB} secondes vers le grand-père, mais également un message vers le père toutes les t_{SUB} secondes à partir de la moitié de l'intervalle t_{SUB}, c'est-à-dire en $\frac{t_{SUB}}{2}$, $\frac{3t_{SUB}}{2}$, $\frac{5t_{SUB}}{2}$, etc. Ainsi durant le temps t_{ind}, deux fois plus de messages sont générés, à savoir $2y$.

Or la période d'indisponibilité peut être considérée comme bien plus grande que le temps de l'intervalle ($t_{ind} \gg t_{SUB}$). La première solution présentée est donc préférable.

3.3 Evaluation de la localisation de contact

Nous avons recensé des problèmes de robustesse. Ceux-ci jouent sur les performances du système. En étudiant l'architecture, nous avons remarqué

d'autres points susceptibles de jouer sur les performances du système. Dans un premier temps, nous allons exposer une méthode qui permet d'améliorer la localisation de contact. Ensuite, l'algorithme effectue ce que nous appelons du *forking*, à savoir la demande à tous les noeuds en parallèle. Nous verrons les conséquences que cela peut engendrer sur le système. Nous évaluerons ensuite la méthode utilisée par rapport à un autre type d'algorithme.

3.3.1 Optimisation de l'algorithme

Le premier point que nous abordons est donc celui de l'optimisation de l'algorithme de localisation. Celui-ci semble performant, mais il est possible de ne pas utiliser de mémoire au sein des clusters comme proposé à la section 2.3.2. De plus, le mécanisme que nous proposons ci-après évite également la transmission de deux messages SIP entre le cluster père et le cluster qui initie la redirection.

Le mécanisme que nous proposons afin d'optimiser le processus de localisation de contact est l'utilisation du corps de message REDIRECT envoyé par le *home cluster* vers son père par l'intermédiaire de l'*outbound proxy*. En indiquant dans le corps de ce message le nom du cluster initiant la redirection, le cluster père sait qu'il ne doit plus interroger ce fils-là puisque la recherche y a échoué. Nous évitons ainsi l'utilisation de la mémoire car le cluster ne sera plus interrogé lorsqu'il redirige une recherche. Nous générons également moins de trafic entre les clusters puisqu'à chaque indirection, le cluster père n'envoie plus un message de recherche au fils qui l'a prévenu de l'impossibilité à trouver le contact.

Nous pouvons discuter si l'utilisation du message REDIRECT pour optimiser la recherche est légitime. Nous nous posons la question de savoir si un tel message peut être envoyé à des domaines qui ne sont pas gérés par l'infrastructure de la société Indigo Software. La réponse est négative puisque ce mécanisme de localisation de contact est mis en place entre les clusters afin de trouver des contacts gérés par les clusters du système. Un contact qui serait géré par un autre domaine est directement détecté et renvoyé par l'*outbound proxy* vers son domaine.

Nous pouvons également évaluer l'optimalité de ce mécanisme. Afin d'y répondre, nous allons examiner les étapes de la solution proposée à la section 2.3.2 et ensuite celles que nous proposons pour améliorer le système. La méthode proposée reprend les étapes suivantes :

1. écriture dans la mémoire,
2. génération du message REDIRECT,
3. envoi à l'*outbound proxy*,
4. l'*outbound proxy* l'envoie au cluster père,
5. traitement par le cluster père,
6. envoi de la recherche au *home cluster*,
7. consultation de la mémoire au *home cluster*,
8. réponse 404 - NOT FOUND,
9. interrogation des autres clusters ...

La méthode que nous suggérons est la suivante :

1. génération du message REDIRECT,
2. envoi à l'*outbound proxy*,
3. l'*outbound proxy* l'envoie au cluster père,
4. traitement par le cluster père,
5. interrogation des autres clusters ...

Nous observons déjà un nombre réduit d'étapes. La génération du message REDIRECT est légèrement plus lourde puisqu'il y a un corps de message d'une ligne à écrire avec les en-têtes. Cela reste néanmoins plus rapide que l'écriture dans la mémoire et la génération du message dans la première optimisation proposée puisque nous travaillons sur un seul élément, le message.

Lors du traitement, le cluster père récupère le nom du cluster qui sait que l'utilisateur ne figure pas dans sa descendance. Il lui suffit dès lors de tester les noms pour savoir s'il peut transmettre la recherche ou non. Le traitement demande donc une lecture supplémentaire dans le message et l'exécution d'un test. Cependant, le cluster père n'interroge plus le cluster à l'origine de la redirection et nous économisons ainsi l'envoi de deux messages sur le réseau et la recherche dans la mémoire.

Ce mécanisme supprime la mémoire au niveau des clusters et évite ainsi le problème de la saturation. Bien qu'une durée de vie maximale soit définie pour chaque entrée dans la mémoire, il se peut, par une utilisation normale ou une tentative de mise à mal du système, que les localisations demandées trouvent le contact dans des clusters situés au-dessus dans l'arborescence et que leur nombre soit suffisamment grand pour saturer la mémoire. Dans ce cas, nous risquons d'effectuer à deux reprises certaines recherches, ce qui engendrerait une perte de performance.

3.3.2 Evaluation du *forking*

Le second point d'analyse de la localisation de contact est celui du *forking*. Comme nous l'avons énoncé ci-dessus, le *forking* consiste à envoyer les données simultanément à différentes destinations. Dans notre cas, il s'agit de la requête de localisation de contact qui est envoyée à l'ensemble des clusters fils.

La solution théorique proposée dans [19] ne définit pas l'implémentation de la recherche, ni la technique à utiliser (en profondeur d'abord, en largeur d'abord, etc.). Le choix effectué lors du développement est celui du *forking*, à savoir la transmission à toutes les branches de l'arbre. Nous pouvons évaluer cette technique sur base des ressources utilisées et du temps nécessaire pour obtenir une réponse. L'évaluation se base sur une comparaison avec un autre type d'algorithme, la recherche en profondeur d'abord. Nous n'aborderons donc pas la recherche en largeur d'abord.

En effet, celle-ci demande à chaque cluster fils s'il connaît l'utilisateur. Dans le cas d'une réponse positive, la recherche est terminée rapidement. Dans le cas contraire, il faut demander à chaque cluster d'effectuer la même opération, ce qui demande un envoi de messages vers les mêmes clusters dans le but d'approfondir la recherche. Il est donc évident que la double génération des messages est pénalisante en termes de performance.

Notre analyse se base sur deux cas différents : le contact se situe sur la première branche ou il se situe sur une autre branche.

Quelle que soit, dans l'arbre, la position du cluster qui connaît l'utilisateur, la recherche telle qu'elle est implémentée se terminera après un temps maximal déterminé par le temps de parcours d'une branche, le temps de traitement par un cluster et la profondeur maximale de l'arbre. Supposons que x est le temps de parcours d'une branche et de traitement par un cluster. Le temps maximal (t_{max}) est donc :

$$t_{max} = n \times x$$

où n est la profondeur maximale de l'arbre.

Dans l'exemple représenté à la figure 3.10, la profondeur vaut 2 ; il faudra donc au maximum un temps de $2x$ pour obtenir une réponse. Cependant,

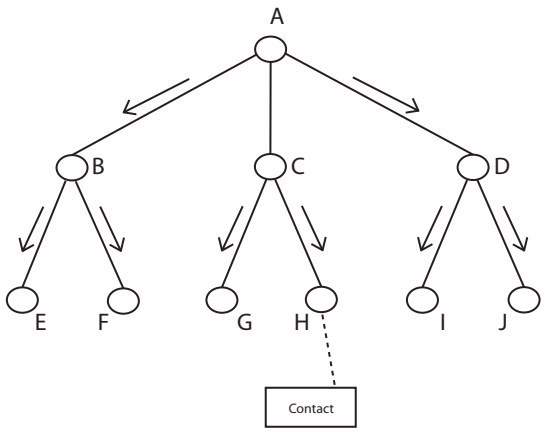


FIG. 3.10 – Mécanisme de recherche par le mécanisme du *forking*.

chaque cluster est sollicité et fait appel à des ressources qui ne seraient peut-être pas utilisées si un autre type d'algorithme était mis en oeuvre.

Les figures 3.11, 3.12 et 3.13 représentent la même structure que nous explorons avec l'algorithme en profondeur d'abord. La première figure (3.11) situe le contact à l'extrême gauche. Dans ce cas, le temps nécessaire pour obtenir la réponse est identique à celui que nous avons obtenu avec la méthode du *forking*. De plus, par rapport à cette méthode, les ressources utilisées sont minimales étant donné que seulement deux clusters (B et E) sont sollicités par le cluster A.

Dans le cas de la figure 3.12, le contact est connecté au deuxième cluster du dernier niveau (F). La recherche en profondeur d'abord nécessite un petit peu plus de temps mais ne sollicite qu'un minimum de clusters. En effet, dans l'exemple précédent, le cluster A appelait les clusters B et E. Dans ce deuxième exemple, les clusters B, E et F sont appelés. Etant donné que trois clusters sont sollicités, le temps nécessaire pour obtenir une réponse est de $3x$.

La figure 3.13 nous donne un temps de réponse et une utilisation des ressources maximaux puisqu'il faut interroger neuf clusters avant de recevoir la localisation du contact. Les clusters étant interrogés en série, les temps s'additionnent pour donner un temps de réponse de $9x$. De plus, l'entièreté des clusters est sollicitée et l'utilisation des ressources est donc maximale.

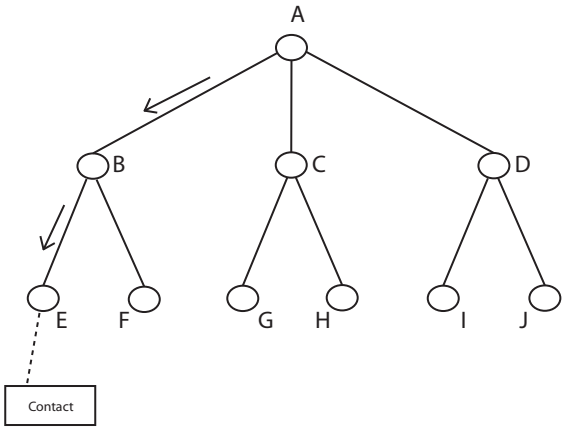


FIG. 3.11 – Recherche en profondeur d'abord avec un contact situé à l'extrême gauche.

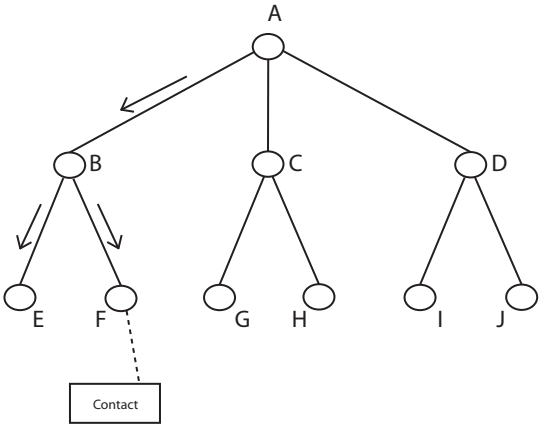


FIG. 3.12 – Recherche en profondeur d'abord avec un contact ne se situant pas à l'extrême gauche.

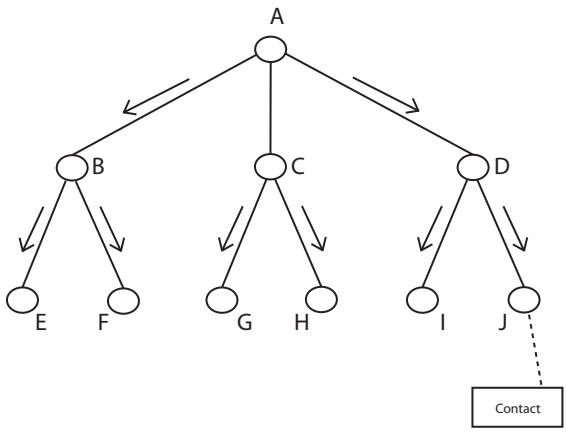


FIG. 3.13 – Recherche en profondeur d'abord avec un contact situé à l'extrême droite.

Le tableau 3.2 reprend les deux techniques pour l'exemple d'arbre exposé ci-dessus. La dernière colonne (Moyenne) est la moyenne de la seconde méthode, basée sur une distribution normale des utilisateurs.

	Forking	Profondeur d'abord			
Critère	Fig. 3.10	Fig. 3.11	Fig. 3.12	Fig. 3.13	Moyenne
Temps	$2x$	$2x$	$3x$	$9x$	$5.5x$
Ressources	9 clusters	2 clusters	3 clusters	9 clusters	5.5 clusters

TAB. 3.2 – Tableau comparatif des deux méthodes.

Bien que la recherche demande plus de ressources auprès des clusters, la technique développée semble la plus rapide dans la majorité des cas. Le tableau 3.2 confirme cette conclusion. En effet, nous y voyons un temps moyen de $5.5x$ pour la recherche en profondeur d'abord alors que le *forking* n'en demande que $2x$. L'algorithme développé est pratiquement trois fois plus rapide. Cependant, il sollicite un peu moins de deux fois plus de ressources.

Dans un cas où le temps de réponse n'est pas critique, la deuxième solution serait préférable pour réduire les ressources nécessaires. Dans ce cas, les machines hébergeant les serveurs devraient être moins puissantes pour effectuer un travail similaire.

Chapitre 4

Evaluation pratique

Le chapitre 3 nous a permis de relever des failles dans le système sur base de réflexions théoriques. Le présent chapitre se concentre sur la mise en place de tests pratiques du système. Ceux-ci constituaient initialement la base du mémoire. Cependant, suite à une remise à neuf du système d'exploitation sur les machines du laboratoire d'expérimentation, une réinstallation et une reconfiguration du système ont été entreprises (cf. annexes A, B et C). Cette phase, qui n'était pas prévue initialement dans le plan de travail, a pris plus de temps que prévu suite à de multiples problèmes d'installation et un besoin de formation dans certains outils. Dès lors, l'analyse théorique a été approfondie pour palier le manque de tests pratiques.

Nous présentons donc dans un premier temps le laboratoire de tests. La deuxième section présente le test de validation de la configuration du serveur Indigo. La dernière section nous permet de présenter les tests que nous aurions effectués si nous avions pu disposer de davantage de temps.

4.1 Environnement de test

Le laboratoire se compose de douze ordinateurs et est représenté à la figure 4.1. Nous l'expliquons en le découpant en trois niveaux : les passerelles, le diamant et les unités de travail. Les passerelles sont deux ordinateurs connectés au réseau des Facultés, `pc05` et `pc06`. Les quatre machines situées au centre du laboratoire, `pc01`, `pc02`, `pc03` et `pc04`, forment ce que nous appelons le diamant. Les unités de travail sont six ordinateurs, de `pc07` à `pc12`, et deux téléphones SIP de la société Grandstream Networks. Parmi les douze ordinateurs, dix tournent sous la distribution Linux de Fedora Core (*release 5*) alors que deux sont pourvus de Microsoft Windows 2000. Ces deux dernières machines font partie des unités de travail.

L'accès au laboratoire s'effectue donc par l'intermédiaire des deux passerelles. Chacune dispose de trois interfaces réseaux. La première effectue la connexion sur le réseau des Facultés avec une adresse IP publique, permettant ainsi de se connecter sur le laboratoire depuis l'extérieur. La seconde relie la machine au réseau central (`10.0.0.0/24`), communiquant ainsi avec les machines du diamant. La dernière appartient au réseau interconnectant les ordinateurs faisant partie des unités de travail.

Les quatre stations du diamant hébergent chacune un serveur de la société Indigo Software. Elles disposent toutes de quatre interfaces réseaux. La première relie la machine au réseau central. La seconde interface permet de connecter chaque machine avec un petit réseau local sur lequel se connectent deux unités de travail. Les troisième et quatrième interfaces sont des connexions de secours qui relient chaque machine avec deux autres du diamant. Ainsi, par exemple, `pc01` est mis en réseau avec `pc02` et également avec `pc04`. Il s'agit de deux réseaux distincts : `10.0.12.0/24` et `10.0.14.0/24`.

De plus, la machine `pc01` dispose d'un serveur géoDNS (cf. annexe A) et l'ordinateur `pc03` dispose de l'application de clustering, LVS, effectuant également la répartition de charge.

Les unités de travail possèdent deux interfaces, l'une pour le réseau avec les machines du diamant, l'autre pour le réseau des unités de travail (`10.0.5.0/24`).

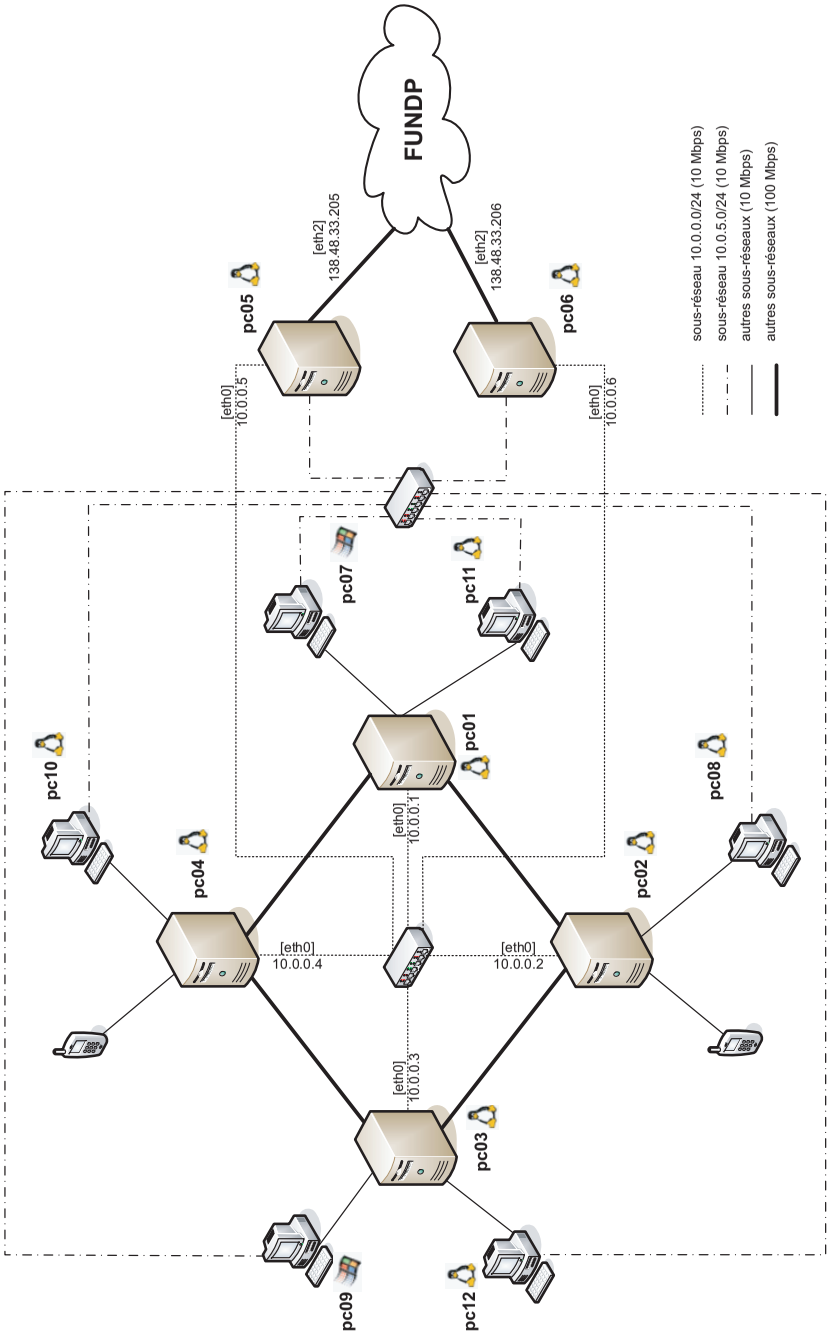


FIG. 4.1 – Plan du laboratoire d'expérimentation.

4.2 Tests de configuration

Le test de configuration effectué cherchait à vérifier que lorsque deux utilisateurs, gérés par des serveurs différents, communiquent, les messages transitent bien par leur serveur.

Le souhait étant que chaque serveur gère son propre domaine, il doit pouvoir transmettre une requête, qui est destinée à un autre domaine, au serveur adéquat. Nous avons donc effectué un test simple : deux serveurs et deux clients. Chaque serveur gère son domaine et doit transférer les requêtes à destination d'un autre domaine. Ces deux serveurs sont exécutés sur `pc01` et `pc02`. Le premier gère le domaine `10.0.0.1` et le second gère `10.0.0.2`.

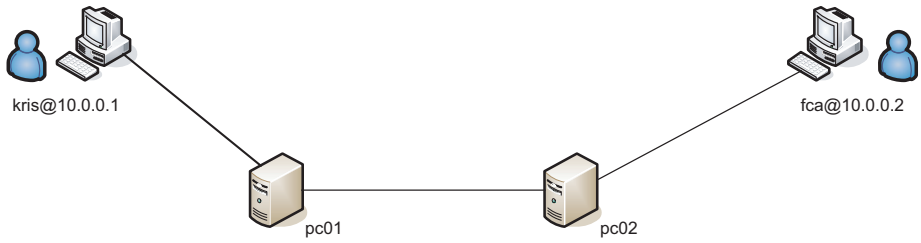


FIG. 4.2 – Déploiement pour le test de configuration.

Comme le montre la figure 4.2, le premier utilisateur, `kris@10.0.0.1`, se connecte au premier serveur, `pc01` et le second utilisateur, `fca@10.0.0.2`, se connecte au second serveur, `pc02`. Les utilisateurs se voient dans la fenêtre du logiciel Kphone et décident d'initier une conversation par message. `kris@10.0.0.1` émet un message qui génère l'ouverture d'une fenêtre de dialogue chez `fca@10.0.0.2`. L'échange de message se fait donc correctement.

Cependant, ceci ne garantit pas que les serveurs communiquent entre eux pour effectuer la tâche. Afin de vérifier que tout s'est produit comme attendu, nous avons analysé les fichiers log. Ceux-ci sont contenus dans le répertoire `$SERVER_DIR$/log/` où `$SERVER_DIR$` est le nom du répertoire d'installation du serveur. Le fichier `messages0.log` nous permet de vérifier que la communication s'effectue bien entre les serveurs. Nous avons mis en évidence les premiers échanges de messages dans un extrait de ce fichier, que vous trouverez à l'annexe D. Sur base de cet extrait, nous avons réalisé le diagramme de séquence (figure 4.3). Nous n'y avons représenté que trois

acteurs étant donné qu'il s'agit d'un fichier du premier serveur qui ne contient que des informations relatives aux échanges de messages entrants et sortants du serveur. Nous supposons que les messages arrivant au second serveur sont correctement transmis à l'utilisateur qui peut lui-même envoyer des messages sur le réseau par l'intermédiaire de son serveur.

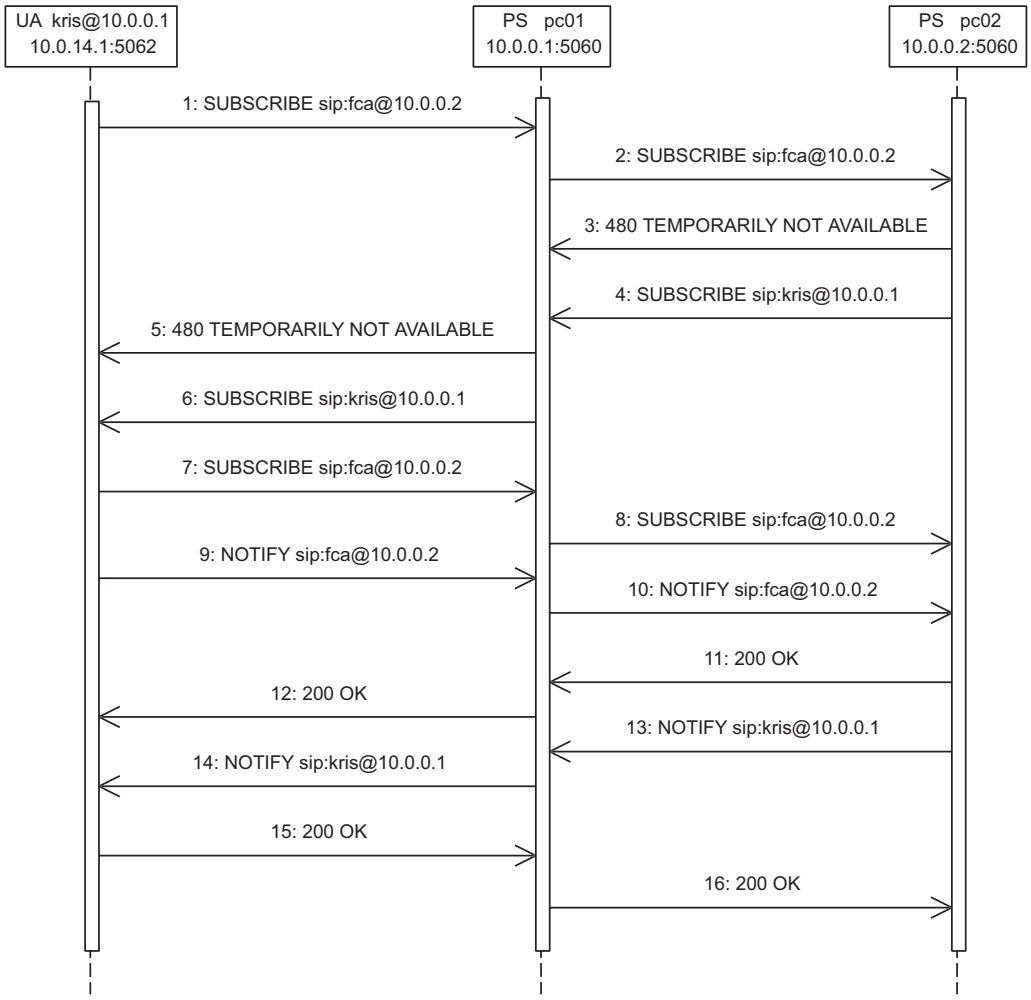


FIG. 4.3 – Diagramme de séquence du test de validation de la configuration.

La figure 4.3 montre bien l'interaction entre les deux serveurs. Comme nous nous y attendions, les messages SUBSCRIBE, NOTIFY ainsi que les ré-

ponses transitent bien par les deux serveurs.

4.3 Exposé des tests envisagés

Nous venons de montrer que la configuration actuelle des serveurs répondait aux attentes. Il est dès lors possible d'envisager les tests à entreprendre pour évaluer les performances du système. Nous allons présenter les différents tests au sein de cette section. Les premiers tests concernent la robustesse du système. Ensuite, des tests relatifs à la charge des serveurs seront exposés. Pour terminer le chapitre, nous évoquerons des tests de performances temporelles.

4.3.1 Robustesse

Dans le chapitre 3, nous avons souligné des problèmes de robustesse liés aux clusters, aux *load balancers* et aux *outbound proxies*. La présente section propose des scénarios de tests visant à montrer les problèmes que nous avons mis en évidence précédemment. Nous supposons la configuration présentée à la figure 4.4.

Deux clusters sont opérationnels avec une relation de père-fils. Le père est composé de deux serveurs hébergés sur `pc01` et `pc02`. Le cluster fils est également composé de deux serveurs sur `pc03` et `pc04`. LVS étant installé sur `pc03`, nous proposons de lancer deux instances de celui-ci qui jouerait le rôle de *load balancer* pour les clusters. Plus tard, lorsque nous évaluerons les performances afin d'établir une limite pour le nombre d'utilisateurs, nous pourrons installer LVS sur une seconde machine afin de distinguer physiquement les *load balancers*. Deux *outbound proxies* sont également en fonctionnement, `pc02` et `pc04`, chacun renvoyant les requêtes à l'un des clusters. De plus, deux utilisateurs, `pc08` et `pc10`, se connectent par l'intermédiaire de leur *outbound proxy*. D'autres utilisateurs peuvent se connecter au système à partir des `pc11` et `pc12`.

Le premier test propose d'arrêter l'un des deux *outbound proxies* afin de voir comment se comporte le système. En théorie, l'utilisateur doit réinitialiser les connexions.

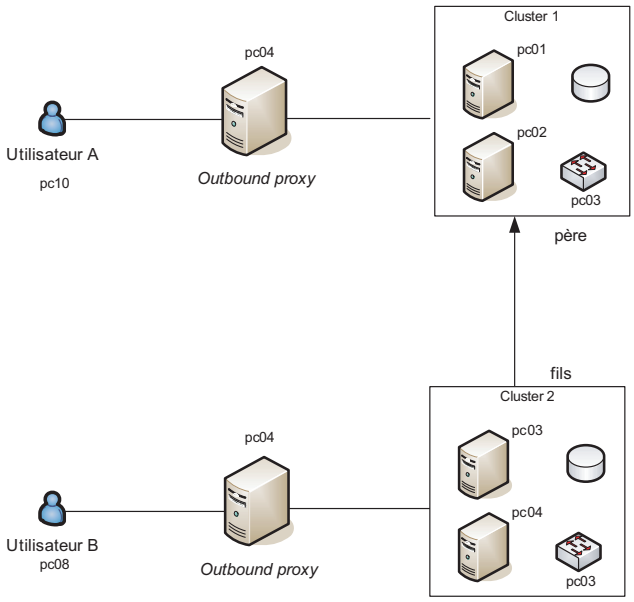


FIG. 4.4 – Configuration pour les tests.

Le deuxième test vise à retirer un cluster de l'arborescence. Travaillant avec deux clusters uniquement, le résultat sera simple : l'utilisateur géré par le cluster interrompu n'aura plus accès au service. Le second utilisateur, quant à lui, ne pourra plus joindre son contact étant donné que le cluster est arrêté.

Afin de prouver les conséquences de la perte d'un cluster que nous avons exposées à la section 3.2.3, nous devrions effectuer ce test sur une arborescence plus fournie. Etant donné qu'aucun mécanisme de récupération en cas de problèmes dans l'arborescence n'a été défini, le système devrait être incohérent si un cluster intermédiaire venait à tomber en panne. La mise en place du mécanisme que nous avons défini pour résoudre le problème devrait y porter remède. Afin de le vérifier, nous devrons répéter ce test qui permettrait de prouver le concept.

Un test supplémentaire de l'arborescence peut être mis en place sur une architecture similaire à celle présentée à la figure 3.5. Cette petite arborescence peut être réalisée sur le diamant, par des clusters composé d'un seul serveur, en observant les correspondances suivantes : `pc01` est le cluster 1, `pc02` le cluster 2, `pc03` le cluster 3 et `pc04` le cluster 4. Nous pouvons ainsi

vérifier le comportement du système en cas de perte de n'importe quel cluster dans l'arbre. En effet, l'arrêt du premier cluster permet de constater le comportement en cas de perte de la racine. Les clusters 3 et 4 constituant des feuilles de l'arbre, nous pouvons vérifier comment réagit le système à la perte d'un cluster feuille. La panne d'un cluster intermédiaire peut être observée par l'arrêt du cluster 2.

Après avoir effectué les tests sur les *outbound proxies* et les clusters, nous pouvons également tester les *load balancers*. Nous avons remarqué précédemment que si le *load balancer* tombe en panne, c'est le cluster complet qui sera injoignable. Les conséquences seront les mêmes que le test précédent. Cependant, nous devons le vérifier à l'aide d'un troisième test de robustesse qui propose d'arrêter le *load balancer* d'un cluster.

A la section 3.2.1, nous avons donné une solution à ce problème qui était de répliquer les *load balancers* du cluster. Ce troisième test devra à nouveau être effectué après le développement de la solution.

4.3.2 Charge des serveurs

Nous venons d'exposer les différents tests de robustesse. Nous allons maintenant nous intéresser aux tests relatifs à la charge des serveurs. L'auteur de [10] a déjà effectué un petit test de charge. En effet, lors de son stage en 2005 aux Facultés, il a pu observer que l'utilisation du processeur diminuait sur les serveurs d'un cluster par rapport à un serveur isolé. Pour effectuer ce test, il a généré du trafic de messagerie instantanée avec un outil développé par la société Indigo Software, MSNSimulator.

Le test que nous proposons dans cette section vise à trouver la limite du nombre d'utilisateurs que le serveur peut supporter. Cette limite peut être atteinte à cause de l'utilisation maximale des ressources des serveurs. Cependant, la mise en cluster des serveurs permet d'augmenter le nombre d'utilisateurs supportés.

Le test est donc de simuler un certain nombre d'utilisateurs et de visualiser l'utilisation des ressources sur un serveur. En augmentant le nombre d'utilisateurs, nous allons atteindre une limite qui met à mal le serveur, c'est-à-dire que le temps de réponse dépasse un certain pourcentage de la moyenne. La première phase consiste donc à établir cette moyenne de temps

de réponse. La suite du test est l'exécution de plusieurs serveurs sur le cluster et l'augmentation parallèle des utilisateurs. Tout en surveillant les ressources du serveur de base de données, nous pourrons établir une limite.

La limite sera majorée par celle du serveur de base de données. En effet, la base de données est, comme nous l'avons déjà mentionné, le goulot d'étranglement d'un cluster. Prenons l'exemple fictif de serveurs supportant 1200 connexions simultanées et d'une base de données n'en supportant que 5000. Si cinq serveurs sont exécutés au sein du cluster, la capacité est de 6000 connexions. Cependant, la base de données en supporte moins et sera donc la limite du cluster. Dès lors, les serveurs seront moins sollicités. Cela diminuera le risque de pannes.

Nous verrons à la section 4.3.3 que ce test sera couplé avec d'autres afin de solliciter le système à plus grand échelle. En effet, les tests proposés jusqu'à présent mettent en oeuvre un voire deux clusters. Ceci permet de tester le système à petite échelle. Nous allons donc proposer des tests sur une arborescence de cluster plus volumineuse.

4.3.3 Performances temporelles

Par performances temporelles, nous entendons le temps nécessaire pour la transmission d'une requête. Les utilisateurs souhaitent que leurs messages instantanés arrivent à leurs correspondants dans les plus brefs délais. Il est évident que dix minutes pour cet échange de message est une solution inconcevable. Le meilleur temps de traitement est le plus proche de l'instantanéité. Lors des petits tests qui ont permis de valider la configuration des serveurs Indigo, nous avons pu remarquer une réception quasi simultanée à l'envoi. Comme le montre l'annexe D, le premier message de souscription vers `fca@10.0.0.2` est réceptionné par le premier serveur à `11:10:19.083`. Il est envoyé au second serveur à `11:10:19.268`. La réponse à cette requête, qui indique que l'utilisateur est connu mais qu'il n'est pas joignable, est reçue par le premier serveur à `11:10:19:648`. Ce serveur renvoie la réponse à l'utilisateur à `11:10:19.717`. Nous voyons donc qu'à partir du moment où le serveur reçoit la requête SUBSCRIBE et celui où il envoie la réponse, 634 millisecondes se sont écoulées. A ce temps, nous devons ajouter deux fois le temps de transmission entre le serveur et l'utilisateur.

Ce temps est raisonnable. Cependant nous sommes dans des conditions de charge minimale. En effet, nous n'avons exécuté que deux serveurs avec

chacun un seul utilisateur connecté. A la section 4.3.2, nous avons parlé des tests relatifs à la charge des serveurs.

Nous proposons également d'effectuer des tests de performances temporelles couplés à l'évaluation de la charge. En effet, le test précédent implique deux utilisateurs et il est prévisible que les délais de réponse augmentent avec le nombre d'utilisateurs. Il est alors intéressant d'étudier l'influence de ce paramètre sur les temps de réponse. En étudiant les temps de réponse nous pourrons établir une relation entre l'augmentation du temps de réponse et celle du nombre d'utilisateurs. Le but de ce test permettrait de fixer une limite du nombre d'utilisateurs que peut gérer un cluster en ayant des temps de réponses admissibles.

Le troisième test de performances temporelles viserait à évaluer un grand nombre d'utilisateurs sur une arborescence plus consistante et répartie sur une zone géographique plus importante. En effet, les temps de connexions au sein du laboratoire sont faibles parce qu'il y a peu de trafic et qu'il s'agit également de faibles distances.

Imaginons que certains de nos contacts se situent outre-Atlantique. Afin de joindre leurs clusters, nous devrons effectuer une connexion transocéanique qui peut augmenter les temps de réponses de deux voire trois centaines de millisecondes. Il est important que les temps de traitement au niveau des serveurs ne soient pas trop grands afin que les temps de réponse cumulés ne soit pas trop grands par rapport à l'attente moyenne.

Ces tests permettront de donner une limite au nombre d'utilisateurs qu'un cluster peut supporter.

Conclusion

Notre travail s'est basé sur le mémoire de Jean Louis Sacré et sur le rapport de stage de Quan Truc Nguyen. Ceux-ci ont abordé la mise à l'échelle d'une implémentation *Presence* confrontée à l'augmentation du nombre d'utilisateurs. Nous avons présenté leurs travaux dans le chapitre 2. Nous avons pu ensuite nous concentrer sur l'évaluation de l'architecture.

Des tests étaient prévus sur les machines du laboratoire. Cependant, par manque de temps, ceux-ci n'ont pu être réalisés. Dès lors, nous avons approfondi l'évaluation théorique. Une panoplie de tests pratiques est néanmoins présentée à la section 4.3.

Lors de la réflexion théorique, nous avons principalement mis en évidence des problèmes de robustesse. Comme l'auteur de [19], nous avons souligné la vulnérabilité de l'architecture au niveau de l'*outbound proxy*. Afin d'y remédier, une augmentation d'instance de celui-ci est proposée afin de limiter les pertes des données de transaction.

Le *load balancer* d'un cluster est également un point fragile de l'architecture. Nous avons vu que celui-ci pouvait provoquer la perte d'un cluster. Pour éviter ceci, nous avons proposé d'augmenter le nombre d'instances au sein de chaque cluster. Cependant, la répartition devant respecter la gestion des dialogues, nous devons synchroniser les données entre le *load balancer* actif et ceux de secours.

Nous venons d'évoquer la perte d'un cluster. Celle-ci peut avoir des conséquences catastrophiques pour le système. Bien qu'un cluster inactif provoque généralement la scission du système en plusieurs entités, une conséquence inéluctable est la perte d'une partie de ses connaissances. En effet, tous les utilisateurs connectés au cluster défaillant ne sont plus joignables. Afin d'y

remédier, nous avons proposé deux modifications dans la construction de l'arbre. La première consiste à une souscription du cluster auprès de son grand-père si son père n'est plus accessible. La seconde propose aux clusters d'avoir une connaissance de leurs frères. Cette seconde adaptation permet de résoudre la panne du cluster père.

Nous avons également analysé l'architecture selon des exigences de qualité et de systèmes distribués. Le résultat est convaincant même si l'utilisation de LVS nous force à travailler avec des machines Linux, ce qui limite la portabilité globale de l'implémentation réalisée.

Il serait intéressant d'appliquer les tests proposés dans ce mémoire afin de valider l'évaluation effectuée. Dans la présentation des tests, nous proposons également d'appliquer les tests une seconde fois après les modifications proposées. Afin de tester plus efficacement l'architecture, il serait également intéressant d'effectuer un déploiement plus large. En effet, le laboratoire propose une structure de test relativement petite et des liaisons rapides entre les serveurs. Un test dans un environnement plus proche de la réalité permettrait de renforcer la validation de l'architecture.

Un exemple serait d'effectuer une distribution mondiale avec des partenariats entre universités. Cependant, ceci serait soumis à la société Indigo Software pour accord. Nous pourrions ainsi tester plus efficacement les temps de réponse entre utilisateurs intercontinentaux. De plus, le trafic actuel de l'Internet peut ralentir les messages entre les entités du système.

Il faudra donc prendre les résultats des tests en laboratoire avec précaution. En effet, les temps de réponses risquent d'augmenter lors de tests dans un environnement de production.

Bibliographie

[1] Camarillo Gonzalo et Garcia-Martin Miguel A., "The 3G IP Multimedia Subsystem (IMS)", Edition John Wiley & Sons Ltd, England, 2004

[2] Campbell B., Rosenberg J., Schulzrinne H., Huitema C. et Gurle D., "Session Initiation Protocol (SIP)-Extension for Instant Messaging", Internet Engineering Task Force, RFC3428, Décembre 2002

[3] Cormier B, "MSN Messenger : la messagerie la plus utilisée ", `http://www.pcinpact.com/actu/news/27989-MSN-Messenger-la-messagerie-la-plus-utilisee.htm`, Avril 2006

[4] Englebert V., "Conception des systèmes distribués et coopératifs", Facultés Universitaires Notre-Dame de la Paix, Namur, 2005 - 2006

[5] Gulbrandsen A., Vixie P., Esibov L., "A DNS RR for specifying the location of services (DNS SRV)", Internet Engineering Task Force, RFC 2782, Février 2000

[6] Habra N., "Ingénierie du logiciel", Facultés Universitaires Notre-Dame de la Paix, Namur, 2005 - 2006

[7] Kadionik P., "Le projet HomeSIP : la domotique avec le protocole SIP", *Linux Magazine Hors Série*, numéro 25, pages 34–43, avril 2006

[8] Moats R., "URN Syntax", Internet Engineering Task Force, RFC 2141, Mai 1997

[9] Mockapetris P., "Domain Names - Implementation and Specification", Internet Engineering Task Force, RFC 1035, Novembre 1987

[10] Nguyen Quan Truc, "Passage à l'échelle d'un Service de Présence SIP", *Rapport de stage*, Facultés Universitaires Notre-Dame de la Paix / ENSEIRB, Namur / Bordeaux, Octobre 2005

[11] Niemi A., "Session Initiation Protocol (SIP) Extension for Event State Publication ", Internet Engineering Task Force, RFC 3903, Octobre 2004

[12] Roach A., "Session Initiation Protocol (SIP)-Specific Event Notification", Internet Engineering Task Force, RFC3265, Juin 2002

[13] Roach A. B., Campbell B., Rosenberg J., "A Session Initiation Protocol (SIP) Event Notification Extension for Resource Lists ", Internet Engineering Task Force, draft-ietf-simple-event-list-07, Décembre 2004, Travail en cours

[14] Rosenberg J., Schulzrinne H., Camarillo G., Johnston A., Peterson J., Sparks R., Handley M., Schooler E., "SIP : Session Initiation Protocol", Internet Engineering Task Force, RFC 3261, Juin 2002

[15] Rosenberg J., "A Presence Event Package for the Session Initiation Protocol (SIP)", Internet Engineering Task Force, RFC 3856, Août 2004

[16] Rosenberg J., "A Data Model for Presence", Internet Engineering Task Force, draft-ietf-simple-presence-data-model-07, Janvier 2006, Travail en cours

[17] Rosenberg J., "A Watcher Information Event Template-Package for the Session Initiation Protocol (SIP)", Internet Engineering Task Force, RFC 3857, Août 2004

[18] Rosenberg J., "The Extensible Markup Language (XML) Configuration Access Protocol (XCAP)", Internet Engineering Task Force, draft-ietf-simple-xcap-11, Mai 2006

[19] Sacré Jean-Louis, "Presence Scalability", *Mémoire de fin d'étude en licence informatique à horaire décalé*, Facultés Universitaires Notre-Dame de la Paix, Namur, Septembre 2005

[20] Schulzrinne H., Gurbani V., Kyzivat P., Rosenberg J., "RPID : Rich Presence Extensions to the Presence Information Data Format (PIDF)", Internet Engineering Task Force, Internet-Draft draft-ietf-simple-rpid-10, Décembre 2005, Travail en cours

[21] Schulzrinne H., "CIPID : Contact Information in Presence Information Data Format", Internet Engineering Task Force, Internet-Draft draft-ietf-simple-cipid-07, Décembre 2005, Travail en cours

[22] Schulzrinne H., "Timed Presence Extensions to the Presence Information Data Format (PIDF) to Indicate Status Information for Past and Future Time Intervals", Internet Engineering Task Force, Internet-Draft draft-ietf-simple-future-05, Décembre 2005, Travail en cours

[23] Sugano H., Fujimoto S., Klyne G., Bateman A., Carr W., Peterson J., "Presence Information Data Format (PIDF)", Internet Engineering Task Force, RFC 3863, Août 2004

[24] "Indigo Server Administration Guide", Indigo Software, Février 2004

[25] "How many servers does the MSN Messenger service run on ? ", `http://www.mess.be/msnmessengerfaq/idx/18/130/Basic_Various/article/How_many_servers_does_the_MSN_Messenger_service_run_on.html`, Avril 2005, online, dernière visite : 29 août 2006

[26] `http://advertising.fr.msn.be/Accueil/Services`, online, dernière visite : 29 août 2006

[27] `http://ernest.tuxicity.net/index.php?choix=Reseau&tuto=bind9`, online, dernière visite : 29 août 2006

[28] `http://www.tu-chemnitz.de/docs/lindocs/RH73/RH-DOCS/rhl-rg-fr-7.3/s1-bind-configuration.html`, online, dernière visite : 29 août 2006

[29] `http://www.linuxvirtualserver.org/`, online, dernière visite : 29 août 2006

[30] `http://www.ultramonkey.org/papers/lvs_tutorial/html/`, online, dernière visite : 29 août 2006

[31] `http://www.ultramonkey.org/`, online, dernière visite : 29 août 2006

[32] `http://www.linuxvirtualserver.org/docs/ha/ultramonkey.html`, online, dernière visite : 29 août 2006

[33] `http://openacs.org/forums/message-view?message_id=88831`, online, dernière visite : 29 août 2006

Annexes

Annexe A

Installation du serveur géoDNS

Le serveur géoDNS est comme nous l'avons expliqué dans la section 2.2.4 une adaptation d'un serveur de nom de domaine. Le serveur DNS est indispensable pour transformer un nom en adresse IP. Ce serveur est adapté dans le cadre du serveur Indigo pour récupérer la région géographique et ainsi attribuer un utilisateur à un cluster. Cette annexe se compose de trois sections. La première liste les fichiers utiles ainsi que les sources où nous pouvons nous les procurer. La seconde présente l'installation et la dernière aborde la configuration du serveur et son exécution.

A.1 Fichiers requis

Pour l'installation du serveur GeoDNS, nous avons besoin, dans un premier temps, de l'API GeoIP (`http://www.maxmind.com/app/c`) :

GeoIP-1.3.15.tar.gz

Un serveur DNS doit être installé (`http://www.isc.org/sw/bind`) :

bind-9.2.4.tar.gz

Les sources du serveur DNS doivent être modifiées avant l'installation par le patch (`http://www.caraytech.com/geodns`) :

bind-9.2.4-geodns-patch.tar.gz

A.2 Installations

La première étape consiste à décompresser chaque archive dans le répertoire contenant les sources des applications : `/usr/local/src`.

```
tar zxvf GeoIP-1.3.15.tar.gz
tar zxvf bind-9.2.4.tar.gz
tar zxvf bind-9.2.4-geodns-patch.tar.gz
```

En suivant les instructions présentées par Quan Truc dans son rapport de stage [10], nous avons rencontré quelques erreurs d'installation. La première étape est l'installation de l'API GeoIP :

```
[root@pc01 root]# cd /usr/local/src/GeoIP-1.3.15
[root@pc01 GeoIP-1.3.15]# ./configure --prefix=/usr/local/geoip
[root@pc01 GeoIP-1.3.15]# make install
```

Lors de cette dernière instruction, nous avons rencontré une erreur de paramètre à la compilation : `/usr/bin/ld: cannot find -lz`. Cette erreur nous étant incomprise, nous avons cherché sur Internet et un forum [33] nous a permis de comprendre le problème. L'option `-l` indique au compilateur qu'il doit prendre en compte la libraire dont le nom commence par `lib` suivi nom spécifié après le paramètre `l` et terminé, comme toute librairie, par l'extension `.h`. Dans notre cas, le compilateur cherchait la librairie `libz.h` qu'il n'a pu trouvé. La page qui nous a permis de comprendre ceci et de résoudre le problème est `http://openacs.org/forums/message-view?message_id=88831`. Ayant trouvé un fichier correspondant mais avec une extension particuliaire (`.so.1` au lieu .so), nous avons créé le lien vers le fichier, comme indiqué dans l'article :

```
ln -s /usr/lib/libz.so.1 /usr/lib/libz.so
```

Suite à la résolution du problème énoncé ci-dessus, une autre erreur s'est produite, à savoir une librairie non installée : le fichier `zlib.h` n'a pas été trouvé. Pour tenter de résoudre le problèmes, nous avons copié le fichier depuis les sources :

```
cp /usr/src/kernels/2.6.17-1.2145_FC5-smp-i686/include/linux/zlib.h
   /usr/include/.
```

Après cette copie du fichier, nous avons à nouveau exécuté l'installation qui s'est également terminée par une erreur : le fichier `zconf.h` n'a pas été trouvé. Pour résoudre l'erreur, nous avons effectué la commande :

```
cp /usr/src/kernels/2.6.17-1.2145_FC5-smp-i586/include/linux/zconf.h
   /usr/include/linux/.
```

Nous pensions avoir résolu le problème, cependant l'installation a à nouveau échoué à cause d'un type non défini (`gzFile`). Après quelques recherches sur Internet, nous nous sommes aperçus que la version des fichiers présents sur les machines du laboratoire n'était pas la plus récente. La récupération de fichiers datant de 2003 et 2004 sur le site `http://wigner.cped.ornl.gov/hpac/4.1/zlib/` a permis de compiler correctement l'API.

L'installation du serveur geoDNS peut donc se poursuivre en patchant les sources de BIND :

```
# cd /usr/local/src
# patch -p0 < bind-9.2.4-geodns-patch/patch.diff
```

Après avoir modifié les sources du serveur BIND, nous pouvons lancer l'installation de celui-ci :

```
# cd /usr/local/src/bind-9.2.4
# export LDFLAGS="$LDFLAGS -lGeoIP"
# cp /usr/local/geoip/lib/* /usr/lib/
# ./configure --includedir=/usr/local/geoip/include
              --libdir=/usr/local/geoip/lib
              --prefix=/usr/local/bind
# make install
```

Lorsque l'installation est terminée, nous devons créer des répertoires et un fichier indispensables pour le fonctionnement de BIND :

```
mkdir /usr/local/bind/var
mkdir /usr/local/bind/var/run
touch /usr/local/bind/var/run/named.pid
```

A.3 Configuration du serveur

Le principal fichier de configuration est `/etc/named.conf`, lequel reprend une liste de règles qui seront examinées lorsque le serveur DNS sera interrogé.

La configuration du serveur géoDNS se base sur le concept de *view*. Celui-ci permet de créer une vue pour des utilisateurs correspondants à un type défini d'adresse IP. La syntaxe d'une vue est la suivante :

```
view "nomdelavue" {
  match-clients { <classe-d'adresse> | <une_acl> | "any" | "localnets" };
  <options>;
}
```

La règle `match-clients` permet de définir les adresses qui correspondent à la vue de nom `nomdelavue`. La figure A.1 reprend l'exemple de la vue *belgium* définie dans le cadre du travail.

```
view "belgium" {
  match-clients { country_BE; };
  recursion no;
  zone "indigosw.com" {
    type master;
    file "zone/indigosw.com-be";
  };
};
```

FIG. A.1 – Exemple de vue *belgium*.

La règle de correspondance est une *Access Control List* (ACL). Celle-ci permet de définir un ensemble de sous-réseaux de manière plus souple. Ainsi, si plusieurs règles doivent gérer les mêmes sous-réseaux, il sera plus simple d'utiliser une ACL. En effet, en cas de changement de la liste des sous-réseaux appartenant à une même vue, un changement de la liste permet de modifier l'ensemble des règles. La syntaxe d'une règle ACL est la suivante :

```
acl "nomdelaliste" {
  {sous-réseau};
};
```

La figure A.2 illustre la syntaxe définie ci-dessus.

```
acl "country_BE" {
  { 10.0.1.0/24; };
};
```

FIG. A.2 – Exemple d'ACL.

Le fichier de la zone `indigosw.com` spécifié à la figure A.1 reprend l'ensemble des règles applicables pour la zone. Etant donné que nous travaillons

avec des vues représentant la situation géographique des utilisateurs, nous pouvons définir différentes valeurs de règles. Ainsi nous pouvons rediriger les clients Belges sur un serveur alors que les Français seront envoyés sur un autre.

La figure A.3 reprend un exemple de configuration de zone. Le premier élément (TTL) indique que les serveurs DNS qui récupèrent cette information ne peuvent pas la stocker plus longtemps que la durée spécifiée. Au delà de cette durée, ils devront effectuer une nouvelle requête afin d'obtenir la dernière donnée. Le premier élément contenu dans le *Start Of a zone of Authority* est le numéro de série qui indique la date de la dernière modification ainsi que le nombre de modifications effectuées à cette date. Ensuite, nous définissons l'intervalle de temps avant un rafraîchissement. Le `retry` définit le temps d'attente avant de réessayer une requête. L'élément suivant définit la durée après laquelle l'enregistrement n'est plus valable. Le dernier élément du SOA est un TTL. Il définit la durée minimale d'un enregistrement.

```
$TTL 3D
indigosw.com. IN SOA 10.0.0.1 (
  2006082601 ; serial
  86400      ; refresh
  300        ; retry
  2592000    ; expire
  86400      ; ttl
)

indigosw.com. IN NS 10.0.0.1
proxy IN A 10.0.0.3
```

FIG. A.3 – Exemple de fichier de zone.

Les fichiers de zone se trouvent généralement dans le répertoire /var/named.

A.4 Exécution

Le lancement du serveur géoDNS s'effectue sous linux à l'aide de la commande :

```
[root@pc01 ~]# /etc/init.d/named start
```

Si le fichier de configuration est correct, l'exécution se produit sans erreur et le serveur tourne, sinon le lancement s'interrompt et indique l'erreur qui en est responsable. Nous pouvons ainsi facilement vérifié si la configuration est correcte et également la déboguer. Cependant, pour éviter de lancer le serveur et de s'apercevoir du problème à ce moment-là, une commande permet de vérifier la configuration. Il s'agit de `named-checkconf`.

Il est possible de relancer le serveur en indiquant `restart` à la place de `start`. Afin de stopper le serveur, la commande est :

```
[root@pc01 ~]# /etc/init.d/named stop
```

Annexe B

Installation du serveur Indigo

Cette annexe présente l'installation du serveur Indigo, que nous avons dû exécuter suite à la mise à niveau du laboratoire

B.1 Fichiers requis

Pour l'installation du serveur Indigo, le fichier suivant est nécessaire :

install-ICS.bin

Afin d' utiliser le serveur, nous devons être en possession de licence. Les fichiers nécessaires sont pour la station 10.0.0.1 :

Indigo_Software_RSA2048.asc
license.dat

Une application permet de visualiser graphiquement certaines informations relatives au serveur. Pour l'installer, il est indispensable de posséder le fichier suivant :

install-IUM.bin

Ces fichiers sont fournis par la société Indigo Software dans le cadre d'un partenariat avec les Facultés et ne sont pas en téléchargement libre sur internet.

B.2 Installations

B.2.1 Serveur Indigo

Le listing B.1 présente la concole complète de l'installation du serveur Indigo. La plupart des options ont été configurée par défaut en appuyant sur ENTER.

Listing B.1 – Console de l'installation du serveur Indigo.

```
[root@pc04 indigo]# ./install-ICS.bin
Preparing to install...
Extracting the JRE from the installer archive...
Unpacking the JRE...
Extracting the installation resources from the installer
archive...
Configuring the installer for this system's environment...

Launching installer...

Preparing CONSOLE Mode Installation...

===============================================================
Indigo Server 4.7       (created with InstallAnywhere by Zero G)
---------------------------------------------------------------



===============================================================
Introduction
------------

InstallAnywhere will guide you through the installation of
Indigo Server 4.7.

It is strongly recommended that you quit all programs before
continuing with this installation.

Respond to each prompt to proceed to the next step in the
installation.  If you want to change something on a previous
step, type 'back'.

You may cancel this installation at any time by typing 'quit'.

If you have a graphical environment running (e.g. X-Window),
you may prefer to quit this installation now and then run it
```

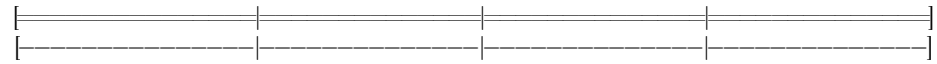
```
again with the "-i gui" command line parameter.

PRESS <ENTER> TO CONTINUE:


================================================================================
License Agreement
---------------------

Installation and use of Indigo Server 4.7 requires acceptance
of the following License Agreement:

This software is protected by copyright laws and international
treaties.

The use of this software is limited to the type of license
bought by the licensee.

License Exclusions. Licensee shall not:
- copy, in whole or in part, the software;
- reverse compile, reverse engineer or reverse assemble all
or any portion of the software;
- distribute, disclose, market, rent, lease or transfer the
software to third parties;
- modify the software;
- assign or sublicense the software.

This software includes software provided by Sun, Apache,
Cryptix and other third party sources.
These are distributed under the licenses included.
Indigo Software and these third parties do not provide
any warranties regarding this software. See the LICENSE files
in the ../docs directory for more legal information.


PRESS <ENTER> TO CONTINUE:

DO YOU ACCEPT THE TERMS OF THIS LICENSE AGREEMENT? (Y/N): Y

================================================================================
Choose Install Folder
---------------------
```

```
Where would you like to install?

  Default Install Folder: /usr/Indigo/Indigo_Server_4_7

ENTER AN ABSOLUTE PATH, OR PRESS <ENTER> TO ACCEPT THE DEFAULT
      :
```

```
Set Application Memory
----------------------

You can here set the initial and maximum size of the JVM heap
that will be used by the Indigo Server 4.7.
Initial size MUST be less than or equal to maximum size. It is
recommended to set initial and maximum size to the same value.

Initial heap size (in Mb) (DEFAULT: 256):

Maximum heap size (in Mb) (DEFAULT: 256):
```

```
Pre-Installation Summary
------------------------

Please Review the Following Before Continuing:

Product Name:
    Indigo Server 4.7

Install Folder:
    /usr/Indigo/Indigo_Server_4_7

Disk Space Information (for Installation Target):
    Required:  81,369,990 bytes
    Available: 2,431,574,016 bytes

PRESS <ENTER> TO CONTINUE:
```

```
Installing...
```



```
Server Configuration - DNS Server

Please enter here the IP address of your DNS Server. This will
be used by the location service of the Indigo Server 4.7 to
perform DNS lookup or SRV queries.

DNS Server IP Address (DEFAULT: ): 10.0.0.1
```

```
Server Configuration - Database Type

Please choose the type of database the Indigo Server 4.7 will
use. A HSQL database server is supplied with this software,
but you may prefer to use MySQL or Oracle. However, these
servers are NOT supplied. If you want to use one of them,
you will have to download and install it by yourself. For more
information, you can go to www.mysql.org or www.oracle.com.

    1- MySQL
  ->2- HSQL Server
    3- Oracle

ENTER THE NUMBER FOR YOUR CHOICE, OR PRESS <ENTER>
TO ACCEPT THE DEFAULT:
```

```
Indigo Server 4.7 will now be configured for your system. This
may take a moment...

PRESS <ENTER> TO CONTINUE:
```

```
===============================================================================
Installation Complete
-------------------------

Congratulations. Indigo Server 4.7 has been successfully
installed to:

/usr/Indigo/Indigo_Server_4_7


By default, the server will listen on 10.0.0.4, port 5060, but
you can change these values via the config files or the Web
Admin interface.


PRESS <ENTER> TO EXIT THE INSTALLER:
[root@pc04 indigo]#
```

B.2.2 User Management

Le listing B.2 présente également la console de l'installation du serveur User Management. Lors de cette installation nous avons également accepté la plupart des choix par défaut.

Listing B.2 – Console de l'installation du serveur User Management.

```
[root@pc04 indigo]# ./install-IUM.bin
Preparing to install...
Extracting the JRE from the installer archive...
Unpacking the JRE...
Extracting the installation resources from the installer
archive...
Configuring the installer for this system's environment...

Launching installer...

Preparing CONSOLE Mode Installation...

===============================================================================
Indigo User Management 4.7
                         (created with InstallAnywhere by Zero G)
-------------------------------------------------------------------------------
```

```
Introduction
------------

InstallAnywhere will guide you through the installation of
Indigo User Management 4.7.

It is strongly recommended that you quit all programs before
continuing with this installation.

Respond to each prompt to proceed to the next step in the
installation.  If you want to change something on a previous
step, type 'back'.

You may cancel this installation at any time by typing 'quit'.

If you have a graphical environment running (e.g. X-Window),
you may prefer to quit this installation now and then run it
again with the "-i gui" command line parameter.

PRESS <ENTER> TO CONTINUE:
```

```
License Agreement
-----------------

Installation and use of Indigo User Management 4.7 requires
acceptance of the following License Agreement:

This software is protected by copyright laws and international
treaties.

The use of this software is limited to the type of license
bought by the licensee.

License Exclusions. Licensee shall not:
- copy, in whole or in part, the software;
- reverse compile, reverse engineer or reverse assemble all or
any portion of the software;
- distribute, disclose, market, rent, lease or transfer the
software to third parties;
- modify the software;
- assign or sublicense the software.

This software includes software provided by Sun, Apache,
```

```
Cryptix and other third party sources.
These are distributed under the licenses included.
Indigo Software and these third parties do not provide any
warranties regarding this software. See the LICENSE files in
the ../docs directory for more legal information.


PRESS <ENTER> TO CONTINUE:


DO YOU ACCEPT THE TERMS OF THIS LICENSE AGREEMENT? (Y/N): Y


================================================================================
Choose Install Folder
---------------------

Where would you like to install?

  Default Install Folder: /usr/Indigo/Indigo_UserManagement_4_7

ENTER AN ABSOLUTE PATH, OR PRESS <ENTER> TO ACCEPT THE DEFAULT
      :

================================================================================
Set Application Memory
----------------------

You can here set the initial and maximum size of the JVM heap
that will be used by the Indigo User Management 4.7.
Initial size MUST be less than or equal to maximum size. It is
recommended to set initial and maximum size to the same value.

Initial heap size (in Mb) (DEFAULT: 256):

Maximum heap size (in Mb) (DEFAULT: 256):


================================================================================
Pre-Installation Summary
------------------------

Please Review the Following Before Continuing:
```

```
Product Name:
    Indigo User Management 4.7

Install Folder:
    /usr/Indigo/Indigo_UserManagement_4_7

Disk Space Information (for Installation Target):
    Required:  70,764,739 bytes
    Available: 2,437,156,864 bytes

PRESS <ENTER> TO CONTINUE:
```

```
Installing...
-------------

|=================|=================|=================|=================|
|-----------------|-----------------|-----------------|-----------------|
```

```
Server Configuration - Database Type
------------------------------------

Please choose the type of database the Indigo User Management
4.7 will use. A HSQL database server is supplied with this
software, but you may prefer to use MySQL or Oracle. However,
these servers are NOT supplied. If you want to use one of them,
you will have to download and install it by yourself. For more
information, you can go to www.mysql.org or www.oracle.com.

    1- MySQL
  ->2- HSQL Server
    3- Oracle

ENTER THE NUMBER FOR YOUR CHOICE, OR PRESS <ENTER>
TO ACCEPT THE DEFAULT:
```

```
Server Configuration - Database Server Information
--------------------------------------------------
```

```
Please enter here the requested information concerning the HSQL
database server you will use.

The database server address can be a FQDN or an IP address.

Database server address (DEFAULT: localhost):
```

```
Indigo User Management 4.7 will now be configured for your
system. This may take a moment...

PRESS <ENTER> TO CONTINUE:
```

```
Installation Complete
---------------------

Congratulations. Indigo User Management 4.7 has been
successfully installed to:

/usr/Indigo/Indigo_UserManagement_4_7

By default, the server will listen on 10.0.0.4, port 4763,
but you can change these values via the config file
(#HOME_DIR#/web/conf/server.xml).

PRESS <ENTER> TO EXIT THE INSTALLER:
[root@pc04 indigo]#
```

B.3 Configuration des serveurs

Pour la configuration, nous avons d'abord créer la base de données grâce au script présent dans le répertoire `$SERVER_DIR$/bin`, où `$SERVER_DIR$` représente le chemin du répertoire d'installation du serveur. Cependant, avant d'exécuter ce script, nous devons lancer la base de données :

```
cd /usr/Indigo/Indigo_Server_4_7/bin
./HSQLServer &
```

L'étape suivante est la configuration de la base de données :

```
./DBPatcher &
```

Nous devons modifier un attribut de configuration afin de limiter un serveur à ne travailler que pour son domaine, c'est-à-dire à traiter uniquement les requêtes à destination de son domaine et transférer toutes les autres. Nous avons donc travaillé sur le fichier de configuration

`$SERVER_DIR$/etc/service/registrer/sipservice.ini`.

Nous y trouvons l'élément de configuration ACCEPTED_HOST, lequel a été mis à la valeur `10.0.0.1` pour la machine `pc01`. Ainsi, le serveur rejette tout sauf ce qui est à destination de son domaine.

Le serveur doit donc transférer vers un autre serveur lorsqu'il ne s'agit pas de son domaine. Nous avons cependant connu des problèmes lorsque le serveur 2 recevait une requête à destination d'un utilisateur du domaine 1.

Pour résoudre cette erreur, nous avons ajouté un élément dans l'élément `<dns>` à l'élément `<dnsdb>` du fichier de configuration

`$SERVER_DIR$/etc/db/usersdb.xml`.

L'ajout correspond à la ligne suivante :

```
<dns from="10.0.0.2" to="10.0.0.2">
```

Ceci correspond à une configuration statique et DEVRAIT ÊTRE RÉSOLU PAR L'UTILISATION D'UN DNS.

Pour la mise en cluster de plusieurs serveurs, nous avons groupé l'utilisation des enregistrements DNS SRV et de *Linux Virtual Server*. Ces deux techniques nous permettent d'effectuer un *load balancing* entre plusieurs adresses.

La seconde permet également de donner une adresse virtuelle à l'ensemble de serveurs.

Le serveur de domaine est donc configuré comme expliqué à l'annexe **??** pour supporter le *load balancing* au moyen des enregistrements DNS SRV. L'installation et la configuration de *Linux Virtual Server* est expliquée à l'annexe C

La seule configuration du serveur *User Management* est celle du fichier `DSConfig.ini` dans le répertoire `etc/db/` du serveur Indigo. Nous avons retiré la mise en commentaire de la ligne :

```
--UM_INIT_DB="#HOME_DIR#/etc/db/usersdb.xml"
```

B.4 Lancement des serveurs

B.4.1 Serveur Indigo

Pour exécuter le serveur, nous devons lancer des scripts présents dans le répertoire :

```
cd /usr/Indigo/Indigo_Server_4_7/bin
```

Il nous faut ensuite lancer la base de données :

```
./HSQLServer &
```

Nous pouvons maintenant exécuter le script du serveur :

```
./ServerStartScript server start
```

B.4.2 Serveur User Management

Le lancement du serveur de gestion doit être précédé du lancement de la base de données utilisée par le serveur Indigo :

```
cd /usr/Indigo/Indigo_Server_4_7/bin
./HSQLServer &
```

Pour exécuter le serveur lui-même, il suffit d'aller dans le répertoire bin correspondant par la première commande ci-dessous et puis exécuter le script de démarrage (seconde commande) :

```
cd /usr/Indigo/Indigo_UserManagement_4_7/bin
./ServerStartScript um start
```

Annexe C

Installation de Linux Virtual Server

Cette annexe présente l'installation de *Linux Virtual Server*, qui nous a permis de clusterisé la solution ainsi que de répartir la charge entre les serveurs. Nous commençons par présenter les fichiers nécessaires. La section suivante nous permet d'aborder l'installation du système et la dernière section parle de la configuration comme Quan Truc l'a fait dans son rapport [10]

C.1 Fichiers requis

Pour l'installation de Linux Virtual Server, nous avons téléchargé le fichier suivant (`http://www.linuxvirtualserver.org/software/ipvs.html#kernel-2.6`) :

ipvsadm-1.24.tar.gz

C.2 Installation

La première étape de l'installation est la compilation des sources de `upvsadm-1.24` à l'aide la commande `make all`, comme expliqué à l'adresse `http://www.ultramonkey.org/papers/lvs_tutorial/html/`. Lors de cette instruction, l'erreur suivante s'est produite : `net/ip_vs.h not found`. Nous avons recherché le fichier manquant sur internet, comme lors de l'installation du serveur géoDNS (cf. annexe A). Celui-ci fut trouvé à l'adresse :
`http://cvs.mandriva.com/cgi-bin/viewvc.cgi/SPECS/ipvsadm/ip_vs.h?revision=1.4`

La commande suivante que nous exécutons est `make install`. Celle-ci s'est déroulée correctement et l'installation s'est terminée. La commande `ipvsadm` est donc disponible et nous pouvons passer à la configuration.

C.3 Configuration

La configuration du système s'écrit dans le fichier `ldirectord.cf` que nous pouvons trouver dans le répertoire `/etc/ha.d/`. Pour connaître toutes les options de configuration, nous vous conseillons de vous reporter au manuel du démon (`man lidrectord`). Cependant, nous donnons l'exemple (figure C.1) donné par Quan Truc Nguyen dans [10] ainsi que les explications des éléments principaux.

La ligne `virtual` définit l'adresse du cluster. La ligne suivante (`fallback`) définit le serveur de secours. C'est celui-là qui sera utilisé si aucun des serveurs réels n'est disponible. Les lignes `real` spécifient les adresses de serveurs réels ainsi que la technique de transfert. Dans ce cas, nous travaillons avec la technique d'*IP Masquerading*.

Aucun service n'est spécifié étant donné que le cluster est prévu pour SIP qui n'est pas défini dans l'implémentation actuelle d'UltraMonkey. Le `scheduler` définit l'algorithme qui sera utilisé pour répartir la charge. L'attribut `persistent` définit la durée d'activité d'une connexion. Ceci permet à l'utilisateur de travailler avec le même serveur réel et ainsi garantir la gestion des dialogues.

```
checktimeout=3
checkinterval=1
autoreload=yes
logfile="/var/log/ldirectord.log"
quiescent=yes

virtual=10.0.0.3:5060
fallback=127.0.0.1:5060
   real=10.0.0.1:5060 masq
   real=10.0.0.2:5060 masq
   real=10.0.0.4:5060 masq
   service=none
   scheduler=wlc
   persistent=600
   protocol=udp
   checkport=5060
```

FIG. C.1 – Exemple de configuration LVS.

C.4 Exécution

Le lancement de LVS s'effectue à l'aide de la commande :
`[root@pc01 ~]# /etc/init.d/ldirectord start`

Il est possible de relancer le système en indiquant `restart` à la place de `start`. Afin de stopper le serveur, la commande est :
`[root@pc01 ~]# /etc/init.d/ldirectord stop`

Annexe D

Fichier log d'un serveur Indigo

Le listing D.1 reprend un extrait du fichier `log` du serveur `pc01`, fichier que nous avons récupéré après les tests de configuration.

Listing D.1 – Extrait du fichier `message0.log`.

```
------
FR 2006-08-24 11:10:19.083
UDP recv from [10.0.0.1:5062] to [10.0.0.1:5060]
--
SUBSCRIBE sip:fca@10.0.0.2 SIP/2.0
Via: SIP/2.0/UDP 10.0.14.1:5062;branch=z9hG4bK3771A082
CSeq: 5809 SUBSCRIBE
To: <sip:fca@10.0.0.2>
Expires: 600
From: "Kris" <sip:kris@10.0.0.1>;tag=2C9DAAAE
Call-ID: 576113497@10.0.14.1
Content-Length: 0
User-Agent: kphone/4.2
Event: presence
Accept: application/xpidf+xml
Contact: "Kris" <sip:kris@10.0.14.1:5062;transport=udp>

------
SE 2006-08-24 11:10:19.268
UDP send to [10.0.0.2:5060] from [10.0.0.1:5060]
--
SUBSCRIBE sip:fca@10.0.0.2 SIP/2.0
Via: SIP/2.0/UDP 10.0.0.1;
  branch=z9hG4bKb02cb7e8e17c862effcf58774eeb7cdf0
Via: SIP/2.0/UDP 10.0.14.1:5062;
  branch=z9hG4bK3771A082;received=10.0.0.1
```

```
CSeq: 5809 SUBSCRIBE
To: <sip:fca@10.0.0.2>
Expires: 600
From: "Kris" <sip:kris@10.0.0.1>;tag=2C9DAAAE
Call-ID: 576113497@10.0.14.1
Content-Length: 0
User-Agent: kphone/4.2
Event: presence
Accept: application/xpidf+xml
Contact: "Kris" <sip:kris@10.0.14.1:5062>
Max-Forwards: 70
Record-Route: <sip:10.0.0.1;
  ForkingID=8577e94426b79ace76b3a75db42b25;lr=lr>

 ----
FR 2006-08-24 11:10:19.648
UDP recv from [10.0.0.2:5060] to [10.0.0.1:5060]
 --
SIP/2.0 480 Temporarily not available
Via: SIP/2.0/UDP 10.0.0.1;
  branch=z9hG4bKb02cb7e8e17c862effcf58774eeb7cdf0
Via: SIP/2.0/UDP 10.0.14.1:5062;
  branch=z9hG4bK3771A082;received=10.0.0.1
From: "Kris" <sip:kris@10.0.0.1>;tag=2C9DAAAE
To: <sip:fca@10.0.0.2>;tag=679837c467a5679a-679837c467a5679a
Call-ID: 576113497@10.0.14.1
CSeq: 5809 SUBSCRIBE
Content-Length: 120
Content-Type: text/plain
Server: Indigo_Server_4_7/89

The server knows the user identified by the Request-URI,
but does not currently have a valid forwarding location for it.

 ----
SE 2006-08-24 11:10:19.717
UDP send to [10.0.0.1:5062] from [10.0.0.1:5060]
 --
SIP/2.0 480 Temporarily not available
Via: SIP/2.0/UDP 10.0.14.1:5062;
  branch=z9hG4bK3771A082;received=10.0.0.1
From: "Kris" <sip:kris@10.0.0.1>;tag=2C9DAAAE
To: <sip:fca@10.0.0.2>;tag=679837c467a5679a-679837c467a5679a
Call-ID: 576113497@10.0.14.1
CSeq: 5809 SUBSCRIBE
Content-Length: 120
Content-Type: text/plain
Server: Indigo_Server_4_7/89
```

```
The server knows the user identified by the Request-URI,
but does not currently have a valid forwarding location for it.

 ----
FR 2006-08-24 11:10:26.176
UDP recv from [10.0.0.2:5060] to [10.0.0.1:5060]
 --
SUBSCRIBE sip:kris@10.0.0.1 SIP/2.0
Via: SIP/2.0/UDP 10.0.0.2;
  branch=z9hG4bK97e0168ff5fb2cbc351f9234c9fa790
Via: SIP/2.0/UDP 10.0.12.2:5062;
  branch=z9hG4bK27131ED3;received=10.0.0.2
CSeq: 2934 SUBSCRIBE
To: <sip:kris@10.0.0.1>
Expires: 600
From: "Fabrice Calay" <sip:fca@10.0.0.2>;tag=1F469A21
Call-ID: 1759130460@10.0.12.2
Content-Length: 0
User-Agent: kphone/4.2
Event: presence
Accept: application/xpidf+xml
Contact: "Fabrice Calay" <sip:fca@10.0.12.2:5062>
Max-Forwards: 70
Record-Route: <sip:10.0.0.2;
  ForkingID=ad1b1ccdc09cb6db326c3f12c486af6;lr=lr>

 ----
SE 2006-08-24 11:10:26.248
UDP send to [10.0.14.1:5062] from [10.0.0.1:5060]
 --
SUBSCRIBE sip:kris@10.0.14.1:5062 SIP/2.0
Via: SIP/2.0/UDP 10.0.0.1;
  branch=z9hG4bKd386f4218a527489def3af96155630
Via: SIP/2.0/UDP 10.0.0.2;
  branch=z9hG4bK97e0168ff5fb2cbc351f9234c9fa790
Via: SIP/2.0/UDP 10.0.12.2:5062;
  branch=z9hG4bK27131ED3;received=10.0.0.2
CSeq: 2934 SUBSCRIBE
To: <sip:kris@10.0.0.1>
Expires: 600
From: "Fabrice Calay" <sip:fca@10.0.0.2>;tag=1F469A21
Call-ID: 1759130460@10.0.12.2
Content-Length: 0
User-Agent: kphone/4.2
Event: presence
Accept: application/xpidf+xml
Contact: "Fabrice Calay" <sip:fca@10.0.12.2:5062>
Max-Forwards: 69
Record-Route: <sip:10.0.0.1;
```

```
   ForkingID=f4ddbeb67b6dbf131fd121eac48e512;lr=lr>
Record-Route: <sip:10.0.0.2;
   ForkingID=ad1b1ccdc09cb6db326c3f12c486af6;lr=lr>

 ----
FR 2006-08-24 11:10:26.358
UDP recv from [10.0.0.1:5062] to [10.0.0.1:5060]
 --
NOTIFY sip:fca@10.0.12.2:5062 SIP/2.0
Via: SIP/2.0/UDP 10.0.14.1:5062;branch=z9hG4bK4C9560F7
CSeq: 433 NOTIFY
To: "Fabrice Calay" <sip:fca@10.0.0.2>;tag=1F469A21
Content-Type: application/xpidf+xml
From: "Kris" <sip:kris@10.0.0.1>;tag=530F9099
Call-ID: 1759130460@10.0.12.2
Route: <sip:10.0.0.1;lr=lr;f
   orkingid=f4ddbeb67b6dbf131fd121eac48e512>,
<sip:10.0.0.2;lr=lr;
   forkingid=ad1b1ccdc09cb6db326c3f12c486af6>
Content-Length: 353
User-Agent: kphone/4.2
Event: presence
Contact: "Kris" <sip:kris@10.0.14.1:5062;transport=udp>

<?xml version="1.0"?>
<!DOCTYPE presence
PUBLIC "-//IETF//DTD RFCxxxx XPIDF 1.0//EN" "xpidf.dtd">
<presence>
<presentity uri="sip:fca@10.0.0.2;method=SUBSCRIBE"/>
<atom id="1000">
<address uri="sip:kris@10.0.14.1:5062;transport=udp;user=ip"
   priority="0,800000">
<status status="open"/>
<msnsubstatus substatus="online" />
</address>
</atom>
</presence>

 ----
FR 2006-08-24 11:10:26.381
UDP recv from [10.0.0.1:5062] to [10.0.0.1:5060]
 --
SUBSCRIBE sip:fca@10.0.0.2 SIP/2.0
Via: SIP/2.0/UDP 10.0.14.1:5062;branch=z9hG4bK1A290013
CSeq: 2361 SUBSCRIBE
To: <sip:fca@10.0.0.2>
Expires: 600
From: "Kris" <sip:kris@10.0.0.1>;tag=B6CB061
Call-ID: 1946286114@10.0.14.1
```

```
Content-Length: 0
User-Agent: kphone/4.2
Event: presence
Accept: application/xpidf+xml
Contact: "Kris" <sip:kris@10.0.14.1:5062;transport=udp>

 ----
SE 2006-08-24 11:10:26.425
UDP send to [10.0.0.2:5060] from [10.0.0.1:5060]
 --
SUBSCRIBE sip:fca@10.0.0.2 SIP/2.0
Via: SIP/2.0/UDP 10.0.0.1;
  branch=z9hG4bK64e17b2c7f725a268865f2cdb93a7a40
Via: SIP/2.0/UDP 10.0.14.1:5062;
  branch=z9hG4bK1A290013;received=10.0.0.1
CSeq: 2361 SUBSCRIBE
To: <sip:fca@10.0.0.2>
Expires: 600
From: "Kris" <sip:kris@10.0.0.1>;tag=B6CB061
Call-ID: 1946286114@10.0.14.1
Content-Length: 0
User-Agent: kphone/4.2
Event: presence
Accept: application/xpidf+xml
Contact: "Kris" <sip:kris@10.0.14.1:5062>
Max-Forwards: 70
Record-Route: <sip:10.0.0.1;
  ForkingID=5855a9c87b56a53e1d197129c78f1496;lr=lr>

 ----
SE 2006-08-24 11:10:26.539
UDP send to [10.0.0.2:5060] from [10.0.0.1:5060]
 --
NOTIFY sip:fca@10.0.12.2:5062 SIP/2.0
Via: SIP/2.0/UDP 10.0.0.1;
  branch=z9hG4bK3268292529682e7d643c716f2de975d00
Via: SIP/2.0/UDP 10.0.14.1:5062;
  branch=z9hG4bK4C9560F7;received=10.0.0.1
CSeq: 433 NOTIFY
To: "Fabrice Calay" <sip:fca@10.0.0.2>;tag=1F469A21
Content-Type: application/xpidf+xml
From: "Kris" <sip:kris@10.0.0.1>;tag=530F9099
Call-ID: 1759130460@10.0.12.2
Route: <sip:10.0.0.2;lr=lr;
  forkingid=ad1b1ccdc09cb6db326c3f12c486af6>
Content-Length: 353
User-Agent: kphone/4.2
Event: presence
Contact: "Kris" <sip:kris@10.0.14.1:5062>
```

```
Max-Forwards: 70
Record-Route: <sip:10.0.0.1;
  ForkingID=c7ee091aa1ce0bcc4fcd47ddeff3ac;lr=lr>

<?xml version="1.0"?>
<!DOCTYPE presence
PUBLIC "-//IETF//DTD RFCxxxx XPIDF 1.0//EN" "xpidf.dtd">
<presence>
<presentity uri="sip:fca@10.0.0.2;method=SUBSCRIBE"/>
<atom id="1000">
<address uri="sip:kris@10.0.14.1:5062;transport=udp;user=ip"
  priority="0,800000">
<status status="open"/>
<msnsubstatus substatus="online" />
</address>
</atom>
</presence>

 ----
FR 2006-08-24 11:10:26.971
UDP recv from [10.0.0.2:5060] to [10.0.0.1:5060]
 --
SIP/2.0 200 OK
Via: SIP/2.0/UDP 10.0.0.1;
  branch=z9hG4bK3268292529682e7d643c716f2de975d00
Via: SIP/2.0/UDP 10.0.14.1:5062;
  received=10.0.0.1;branch=z9hG4bK4C9560F7
From: "Kris" <sip:kris@10.0.0.1>;tag=530F9099
CSeq: 433 NOTIFY
Call-ID: 1759130460@10.0.12.2
To: "Fabrice Calay" <sip:fca@10.0.0.2>;tag=1F469A21
Content-Length: 0
User-Agent: kphone/4.2
Contact: "Fabrice Calay" <sip:fca@10.0.12.2:5062>
Record-Route: <sip:10.0.0.2;
  ForkingID=dcc08b6111a73903d3a59426096b650;lr=lr>,
<sip:10.0.0.1;lr=lr;forkingid=c7ee091aa1ce0bcc4fcd47ddeff3ac>

 ----
FR 2006-08-24 11:10:26.981
UDP recv from [10.0.0.2:5060] to [10.0.0.1:5060]
 --
NOTIFY sip:kris@10.0.14.1:5062 SIP/2.0
Via: SIP/2.0/UDP 10.0.0.2;
  branch=z9hG4bKe55a5b80887a2c44818e39bf12de7da40
Via: SIP/2.0/UDP 10.0.12.2:5062;
  branch=z9hG4bK2AEC4FD9;received=10.0.0.2
CSeq: 3273 NOTIFY
To: "Kris" <sip:kris@10.0.0.1>;tag=B6CB061
```

```
Content-Type: application/xpidf+xml
From: "Fabrice Calay" <sip:fca@10.0.0.2>;tag=1A95499B
Call-ID: 1946286114@10.0.14.1
Route: <sip:10.0.0.1;lr=lr;
  forkingid=5855a9c87b56a53e1d197129c78f1496>
Content-Length: 353
User-Agent: kphone/4.2
Event: presence
Contact: "Fabrice Calay" <sip:fca@10.0.12.2:5062>
Max-Forwards: 70
Record-Route: <sip:10.0.0.2;
  ForkingID=c14da73287d1c7a4b11134f2541ed45f;lr=lr>

<?xml version="1.0"?>
<!DOCTYPE presence
PUBLIC "-//IETF//DTD RFCxxxx XPIDF 1.0//EN" "xpidf.dtd">
<presence>
<presentity uri="sip:kris@10.0.0.1;method=SUBSCRIBE"/>
<atom id="1000">
<address uri="sip:fca@10.0.12.2:5062;transport=udp;user=ip"
  priority="0,800000">
<status status="open"/>
<msnsubstatus substatus="online" />
</address>
</atom>
</presence>

 ----
SE 2006-08-24 11:10:27.023
UDP send to [10.0.0.1:5062] from [10.0.0.1:5060]
 --
SIP/2.0 200 OK
Via: SIP/2.0/UDP 10.0.14.1:5062;received=10.0.0.1;
  branch=z9hG4bK4C9560F7
From: "Kris" <sip:kris@10.0.0.1>;tag=530F9099
CSeq: 433 NOTIFY
Call-ID: 1759130460@10.0.12.2
To: "Fabrice Calay" <sip:fca@10.0.0.2>;tag=1F469A21
Content-Length: 0
User-Agent: kphone/4.2
Contact: "Fabrice Calay" <sip:fca@10.0.12.2:5062>
Record-Route: <sip:10.0.0.2;
  ForkingID=dcc08b6111a73903d3a59426096b650;lr=lr>,
<sip:10.0.0.1;
  ForkingID=c7ee091aa1ce0bcc4fcd47ddeff3ac;lr=lr>

 ----
SE 2006-08-24 11:10:27.043
UDP send to [10.0.14.1:5062] from [10.0.0.1:5060]
```

```
 --
NOTIFY sip:kris@10.0.14.1:5062 SIP/2.0
Via: SIP/2.0/UDP 10.0.0.1;
  branch=z9hG4bK3b5c7c5b53b3775727aeebe56cd7e940
Via: SIP/2.0/UDP 10.0.0.2;
  branch=z9hG4bKe55a5b80887a2c44818e39bf12de7da40
Via: SIP/2.0/UDP 10.0.12.2:5062;
  branch=z9hG4bK2AEC4FD9;received=10.0.0.2
CSeq: 3273 NOTIFY
To: "Kris" <sip:kris@10.0.0.1>;tag=B6CB061
Content-Type: application/xpidf+xml
From: "Fabrice Calay" <sip:fca@10.0.0.2>;tag=1A95499B
Call-ID: 1946286114@10.0.14.1
Content-Length: 353
User-Agent: kphone/4.2
Event: presence
Contact: "Fabrice Calay" <sip:fca@10.0.12.2:5062>
Max-Forwards: 69
Record-Route: <sip:10.0.0.1;
  ForkingID=cfa07851a796844a38e2f29819c4e;lr=lr>
Record-Route: <sip:10.0.0.2;
  ForkingID=c14da73287d1c7a4b11134f2541ed45f;lr=lr>

<?xml version="1.0"?>
<!DOCTYPE presence
PUBLIC "-//IETF//DTD RFCxxxx XPIDF 1.0//EN" "xpidf.dtd">
<presence>
<presentity uri="sip:kris@10.0.0.1;method=SUBSCRIBE"/>
<atom id="1000">
<address uri="sip:fca@10.0.12.2:5062;transport=udp;user=ip"
  priority="0,800000">
<status status="open"/>
<msnsubstatus substatus="online" />
</address>
</atom>
</presence>

 ----
FR 2006-08-24 11:10:27.053
UDP recv from [10.0.0.1:5062] to [10.0.0.1:5060]
 --
SIP/2.0 200 OK
Via: SIP/2.0/UDP 10.0.0.1;
  branch=z9hG4bK3b5c7c5b53b3775727aeebe56cd7e940
Via: SIP/2.0/UDP 10.0.0.2;
  branch=z9hG4bKe55a5b80887a2c44818e39bf12de7da40
Via: SIP/2.0/UDP 10.0.12.2:5062;received=10.0.0.2;
  branch=z9hG4bK2AEC4FD9
From: "Fabrice Calay" <sip:fca@10.0.0.2>;tag=1A95499B
```

```
CSeq: 3273 NOTIFY
Call-ID: 1946286114@10.0.14.1
To: "Kris" <sip:kris@10.0.0.1>;tag=B6CB061
Content-Length: 0
User-Agent: kphone/4.2
Contact: "Kris" <sip:kris@10.0.14.1:5062;transport=udp>
Record-Route: <sip:10.0.0.1;lr=lr;
   forkingid=cfa07851a796844a38e2f29819c4e>,
<sip:10.0.0.2;lr=lr;
   forkingid=c14da73287d1c7a4b11134f2541ed45f>

 ----
SE 2006-08-24 11:10:27.090
UDP send to [10.0.0.2:5060] from [10.0.0.1:5060]
 --
SIP/2.0 200 OK
Via: SIP/2.0/UDP 10.0.0.2;
   branch=z9hG4bKe55a5b80887a2c44818e39bf12de7da40
Via: SIP/2.0/UDP 10.0.12.2:5062;received=10.0.0.2;
   branch=z9hG4bK2AEC4FD9
From: "Fabrice Calay" <sip:fca@10.0.0.2>;tag=1A95499B
CSeq: 3273 NOTIFY
Call-ID: 1946286114@10.0.14.1
To: "Kris" <sip:kris@10.0.0.1>;tag=B6CB061
Content-Length: 0
User-Agent: kphone/4.2
Contact: "Kris" <sip:kris@10.0.14.1:5062>
Record-Route: <sip:10.0.0.1;
   ForkingID=cfa07851a796844a38e2f29819c4e;lr=lr>,
<sip:10.0.0.2;lr=lr;
   forkingid=c14da73287d1c7a4b11134f2541ed45f>
```