UNIVERSITÉ DE
SHERBROOKE
Faculté de génie
Génie électrique et génie informatique

MISE EN OEUVRE DES PROTOCOLES SIP et RTP SUR SYSTÈME EMBARQUÉ

Mémoire de maîtrise ès sciences appliquées
Spécialité : génie électrique

Composition du jury

Philippe Mabilleau
Ahmed Khoumsi
Alain Houle

Eduardo Luis ROMERO

Sherbrooke (Québec) Canada Mai 2009

Your file Votre référence
ISBN: 978-0-494-53424-3
Our file Notre référence
ISBN: 978-0-494-53424-3

À la mèmoire de mes parents

RÉSUMÉ

L'avènement de la VoIP *(Voice over IP)* a déclenché une période de profonds changements dans le marché des télécommunications. En particulier, dans le secteur de la téléphonie résidentielle, cette technologie s'est consolidée, rapidement et pour de nombreuses raisons, comme l'évolution de la téléphonie traditionnelle. Dès les tous débuts, et afin d'établir une base de compatibilité permettant l'interconnexion de plusieurs réseaux téléphoniques et la convergence entre les systèmes traditionnels analogiques et leur évolution numérique, l'industrie a demandé l'établissement de cadres normatifs. En réponse à ces besoins, plusieurs standards et protocoles, avec de successives modifications et corrections, ont été publiés dans une période relativement brève. Parmi les plus populaires, SIP *(Session Initiation Protocol)*, un protocole de signalisation, et RTP *(Real-Time Transport Protocol)*, un protocole de transport de flots temps réel, se démarquent et ils sont au cœur de la majorité des applications conçues actuellement.

Bien que, aujourd'hui, SIP et RTP sont liés fortement à la téléphonie sur IP, leur portée et leurs possibilités sont beaucoup plus vastes, ce qui déclenche un grand intérêt et justifie l'effort mis dans la conception des implémentations plus performantes et orientées plus spécifiquement à divers serveurs mandataires UA *(User Agent)*.

Dans ce contexte, le but du présent projet de maîtrise est de concevoir des piles de protocoles SIP et RTP orientées vers des applications de téléphonie sur IP, dans un environnement embarqué. Des conditions additionnelles sont que les piles doivent être codées en langage C et s'appuyer sur le système d'exploitation en temps réel MicroC/OS-II. Afin de faciliter la portabilité, il doit se prévoir des couches d'abstraction du matériel et du système d'exploitation. Même si les applications ciblées pour le projet sont, principalement, celles de VoIP, la pile SIP doit viser d'autres domaines, notamment des applications de domotique et de contrôle à distance. Cette dernière condition impose, de façon indirecte, d'autres conditions sur la taille du code et la puissance de calcul demandée, car le matériel pour ces types d'applications est d'habitude plus simple et moins puissant que les ordinateurs qui sont souvent utilisés dans les applications professionnelles de communication.

Ce mémoire, qui décrit le travail effectué, est organisé en deux parties. La première fait une introduction théorique à la téléphonie sur IP, et sert de fondement à la deuxième partie, où la mise en œuvre des protocoles SIP et RTP est décrite en détail. L'accent a été mis sur les justifications des décisions prises pendant toute la conception afin d'aider à mieux comprendre la logique appliquée et de permettre sa reconsidération et analyse dans de futures itérations.

Comme résultat des contraintes et limitations imposées dans le cadre de ce projet, les piles de protocoles conçues se sont révélées très compactes et performantes, ce que justifie pleinement la continuité du travail dans l'avenir.

Mots clés : SIP, RTP, Systèmes Embarqués, RTOS.

REMERCIEMENTS

Je tiens à exprimer mes sincères remerciements, avant tout, à mon directeur de recherche, Monsieur Philippe Mabilleau, pour ses conseils, ses enseignements, pour la confiance qu'il a déposé sur moi et pour m'avoir suggéré un sujet de recherche qui m'a passionné.

Mes remerciements vont aussi à Micrium inc., qui avec son support et sa décision de rendre disponibles aux étudiants plusieurs de ses produits, fait une grande contribution à l'apprentissage et à la recherche. Mon épouse, Monica, mérite une mention spéciale, pour son encouragement et sa patience.

J'exprime enfin ma gratitude à Madame Alexandra Ferrer pour son aide énorme avec la correction orthographique et grammaticale, et à Monsieur Mohamed Ali Ibrahim, pour ses conseils et ses suggestions.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

LISTE D'ACRONYMES

ABNF : Augmented Backus-Naur Form
ADPCM : Adaptative Differential Pulse Code Modulation
ADSL : Asymmetric Digital Suscriber Line
API : Application Programming Interface
BNF : Backus-Naur Form
CELP : Code Excited Linear Prediction
CNG : Confort Noise Generation
DMA : Direct Memory Access
DNS : Domain Name Ssystem
DTMF : Dual Tone Multi-Frecuency
GDB : GNU Debugger
HAL : Hardware Abstraction Layer
HTTP : HyperText Transfert Protocol
ID : Identification
IETF : Internet Engineering Task Force
IP : Internet Protocol
ITU : International Telecommunication Union
LPC : Linear Predictive Coding
LS : Location Server
M2M : Machine to Machine
MEF : Machine à États Finis
MTU : Maximum Transmission Unit
NAT : Network Address Translation
OSI : Open System Interconnection
PC : Personal Computer
PCM : Pulse Code Modulation
PDA : Personal Digital Assistant
PS : Proxy Server
PSTN : Public Switched Telephone Network
RFC : Request For Comments

RG : Registrar Server
RS : Redirect Server
RTOS : Real Time Operating System
RTP : Real-time Transfert Protocol
RTCP : Real-Time Control Protocol
SDP : Session Description Protocol
SIP : Session Initation Protocol
SSRC : Synchronization Source
SMTP : Simple Mail Transfer Protocol
STUN : Simple Traversal UDP through NATs
TCP : Transmission Control Protocol
TU : Transmission Unit
UA : User Agent
UAC : User Agent Client
UAS : User Agent Server
UDP : User Datagram Protocol
URI : Uniform Resource Identifier
URL : Uniform Resource Locator
VAD : Voice Activity Detector
VoIP : Voice over IP

PREMIÈRE PARTIE

L'ÉTAT DE L'ART

CHAPITRE 1
INTRODUCTION

L'apparition de l'Internet, consolidée pendant la dernière décennie du vingtième siècle, a généré de grands changements dans notre mode de vie. Internet a modifié la façon d'étudier, de travailler, de commercer, de partager et d'accéder à l'information, pour ne citer que quelques exemples.

Mais, une des transformations les plus frappantes est dans la communication. En effet, les courriels électroniques et la messagerie instantanée, aujourd'hui omniprésents, n'existaient pas avant l'avènement d'Internet et sont seulement possibles grâce à lui. La mise au point des technologies de codecs, la démocratisation des accès haut débit et la consolidation des réseaux IP *(Internet Protocol)*, ont rendu possible l'échange d'information audio et vidéo en temps réel sur Internet, et ce, avec des niveaux de qualité très acceptables. Dans cette conjoncture, la téléphonie sur Internet a déclenché une attention particulière, étant donné les importants intérêts et opportunités commerciales qui lui sont liés.

Plusieurs standards donnent un encadrement normatif aux sessions multimédias en temps réel sur des réseaux IP, mais SIP *(Session Initiation Protocol)*, un protocole de signalisation, et RTP *(Real-Time Transport Protocol)*, un protocole de transport de flots en temps réel sont, sans doute, les plus étendus. Tous les deux ont été publiés par l'IETF *(Internet Engineering Task Force)*, et même s'ils ont été conçus avec une portée beaucoup plus large que celle de la téléphonie sur IP, le fait est que, à nos jours, une grande majorité des implémentations de SIP et de RTP ciblent ce type d'applications.

Même si SIP et RTP sont qualifiés des protocoles simples, cette considération n'est que relative. La problématique de la téléphonie sur Internet est complexe et pourtant SIP et RTP sont des protocoles complexes. Évidemment, leur implémentation présente aussi un niveau de complexité important. Par ailleurs, les systèmes ciblés pour les applications qui utilisent SIP et RTP sont, de plus en plus, des systèmes embarqués. Ceci incorpore de fortes contraintes additionnelles aux implémentations et augmente encore plus la complexité globale du projet, mais, en même temps, il sert à justifier tout l'effort additionnel mis en jeu afin d'optimiser la performance.

Dans ce contexte, le but du présent projet de maîtrise est de concevoir des piles de protocoles SIP et RTP orientées vers des applications de téléphonie sur IP, dans un environnement embarqué. Pour des raisons de portabilité et de commodité de la mise en œuvre, la codification est faite en langage «C» et s'appuie sur le système d'exploitation en temps réel Micro OS-II. Afin de faciliter la portabilité, il doit se prévoir des couches d'abstraction du matériel et du système d'exploitation. Même si les applications ciblées pour le projet sont, principalement, celles de VoIP *(Voice over IP)*, la pile SIP doit viser d'autres domaines, notamment des applications de domotique et de contrôle à distance. Cette dernière condition impose, de façon indirecte, d'autres conditions sur la taille du code et la puissance de calcul demandée, car les plateformes pour ces types d'applications sont habituellement des ressources limitées.

Ce document, qui décrit la conception des piles de protocoles SIP et RTP dans le cadre de la maîtrise en Génie électrique à l'Université de Sherbrooke, est organisé en deux parties. La première, incluant cinq chapitres, fait une mise en contexte sur les principaux sujets reliés à la VoIP et sert d'introduction à la deuxième, où la mise en œuvre est décrite en profondeur. Ainsi, dans la première partie, le premier chapitre fait une introduction générale au projet. Le deuxième chapitre introduit les concepts principaux de la téléphonie sur IP. Les troisième et quatrième chapitres présentent les fondements des protocoles SIP et RTP respectivement. Le cinquième et dernier chapitre de la première partie, fait une brève référence aux systèmes embarqués et aux conditions additionnelles qu'ils imposent sur la conception. Dans la deuxième partie, le sixième chapitre sert d'introduction générale à la mise en œuvre. Le septième chapitre décrit en détail le processus de conception du protocole SIP et le huitième explique l'implantation de RTP. Finalement, ce mémoire inclut les conclusions sur le travail effectué et une exploration de ses possibilités futures.

CHAPITRE 2

TÉLÉPHONIE SUR IP

La téléphonie sur IP, aussi appelée VoIP, est une technique qui permet d'établir des communications téléphoniques sur un réseau de commutation par paquets comme l'Internet. Elle s'embarque dans un contexte plus général connu comme «convergence des services» et qui consiste à intégrer dans un même réseau différents services utilisant auparavant des réseaux indépendants. Un exemple est la fusion des services de télévision par câble, d'accès à l'Internet, de téléphonie et de sécurité sur un réseau IP haut débit. Cette convergence, qui permet une utilisation beaucoup plus efficace des ressources, donne lieu à de grands changements dans la manière dont ces services sont offerts avec des conséquences importantes pour le marché des fournisseurs.

Différents facteurs ont servi de motivation pour l'évolution vers la téléphonie sur IP. Du point de vue des opérateurs, l'intérêt est centré sur un modèle d'affaires, tout à fait différent de celui de la téléphonie traditionnelle et ouvert à la concurrence. Cette possibilité a déclenché l'intérêt des entreprises voulant rentrer dans le monde des affaires de la téléphonie. C'est dans ce cadre que plusieurs protocoles tels que SIP et RTP ont été développés.

L'optimisation des ressources que l'on vient de mentionner et la possibilité d'un marché concurrentiel se traduisent, pour l'usager, par une importante réduction des coûts de communication et par la disponibilité de nouveaux services avec des coûts marginaux réduits.

Mais, malgré tous les avantages de la téléphonie IP, il y a encore des problèmes à résoudre, principalement liés à la qualité de service, à la sécurité et à la prestation de services d'urgence. Parmi ces aspects, c'est ce dernier qui présente les difficultés les plus grandes à contourner et qui empêche encore la diffusion à grande échelle de la VoIP (Rosenberg, 2007; Newport-Networks, 2008).

De nombreux aspects différencient nettement la téléphonie sur IP de la téléphonie traditionnelle et ces différences se traduisent en une problématique particulière. Le but de ce chapitre est d'identifier et d'analyser les éléments qui définissent les caractéristiques de

la VoIP afin d'arriver, à la fin du chapitre, à élaborer un modèle d'architecture pour des applications logicielles de téléphonie sur Internet tout en considérant le point de vue des systèmes embarqués.

2.1 Téléphonie IP

La téléphonie sur IP est une technologie qui permet d'établir des communications téléphoniques sur des réseaux de commutation par paquets tels que l'Internet. Son histoire est très récente. Le premier système propriétaire de téléphonie est implémenté sur l'Internet, vers l'année 1995 (Hallock, 2004). Des standards pour normaliser cette technologie ont été publiés peu de temps après. En 1996 l'ITU (*International Telecommunication Union*) met en effet la norme H.323 (ITU-T, 1996) et en 1999, l'IETF (*Internet Engineering Task Force*) rend disponible la première version du protocole SIP : la recommandation RFC 2543 (Handley et al., 1999).

La disponibilité des réseaux de données de grande capacité, l'utilisation massive des ordinateurs avec des connexions haut débit, l'évolution de la technologie des codecs et l'expansion explosive de l'Internet sont, parmi d'autres, les éléments déclencheurs de la convergence des services. D'ailleurs, la dérégulation de la téléphonie de base pour permettre l'accès à un marché d'offre ouvert à la concurrence, contrairement au modèle monopolistique ou oligopolistique propre de la téléphonie conventionnelle, a été une des motivations les plus fortes qui ont fait de la téléphonie IP une solution d'affaires importante pour la plupart des entreprises liées au secteur des télécommunications (F.Groom and Groom, 2004; Brandl et al., 2004).

Par ailleurs, la contribution des utilisateurs a été, jusqu'à maintenant, indirecte et liée aux coûts. La préférence que les utilisateurs ont manifestée vers les fournisseurs de services internationaux et des communications de longue distance qui utilisent la téléphonie sur IP (*calling-card*, par exemple) a obligé les entreprises de téléphonie publique à réduire leurs tarifs et indirectement à adopter cette nouvelle technologie (Goralski and Kolon, 1999; F.Groom and Groom, 2004; Brandl et al., 2004).

L'apparition de la téléphonie sur Internet a introduit de profonds changements de paradigmes en matière de communication. Pour mieux comprendre son ampleur, il faut men-

tionner quelques aspects distinctifs de l'ancien modèle et de celui proposé à partir de la VoIP.

Le schéma de réseau téléphonique public commuté traditionnel (PSTN) s'appuie sur le modèle des réseaux de commutation par circuits. Ainsi, pour chaque communication établie, le système réserve l'utilisation exclusive d'un circuit physique pendant toute sa durée. Ce modèle est constitué des centraux de commutation, du côté de la compagnie de téléphone et des postes téléphoniques du côté de l'utilisateur. Presque toutes les opérations de signalisation sont exécutées par les centraux qui sont les points où se concentre l'intelligence du réseau. La philosophie de conception est telle que les éléments les plus complexes et dispendieux se retrouvent au central tandis que les éléments les plus simples et les moins dispendieux se retrouvent du côté des utilisateurs. L'opérateur de télécommunications doit constituer les liens entre les usagers. Normalement, les liens entre les centraux téléphoniques et les postes d'abonnés sont analogiques, alors que ceux entre les centraux sont numériques. En bref, il s'agit d'un système avec des postes d'abonnés très simples et peu coûteux, avec des installations complexes du côté des opérateurs, mais qu'a permis l'introduction et le développement de la téléphonie à un coût très abordable pour les abonnés. Ceci, clairement, ne favorise pas la concurrence et donne naissance à des «monopoles naturels». Jusqu'à peu d'années en arrière, ces monopoles étaient aux mains des gouvernements dans la plupart des pays. Les fondements de ce modèle sont restés invariables pendant presque un siècle et c'est l'apparition de la téléphonie mobile qui a marqué le début du changement : les téléphones cellulaires, qui sont devenus de plus en plus sophistiqués et moins coûteux, ont été acceptés par les utilisateurs en focalisant leur attention sur les coûts du service.

Le cadre actuel de la téléphonie traditionnelle montre des usagers bien adaptés et satisfaits, une qualité de service très bonne, des frais de service raisonnables, même gratuits pour les communications locales, et des technologies matures et stables. Par contre, les services de valeur ajoutée et les communications longue distance ont toujours des prix assez élevés et le développement de nouveaux services reste un processus compliqué et coûteux. D'ailleurs, l'addition des technologies, telles que l'ADSL *(Asymmetric Digital Subscriber Line)*, a permis d'améliorer cette situation, en permettant aux utilisateurs du service de téléphonie traditionnelle d'avoir un accès large bande à l'Internet.

C'est clair que la proposition d'un service de téléphonie alternatif doit se faire dans le cadre que l'on vient de décrire. L'avènement et la popularisation de l'Internet ont permis de conce-

voir ce nouveau modèle : la téléphonie sur Internet. Ceci est bien différent de l'architecture traditionnelle. Tout d'abord, la voix est numérisée à niveau des appareils téléphoniques des usagers et est ensuite transmise sur le réseau IP. Les tâches relatives à la signalisation sont aussi déléguées aux postes. De cette façon, et par opposition à la téléphonie commutée, le rôle des postes devient de première importance puisqu'ils concentrent presque toute l'intelligence du système. Il ne reste qu'un nombre limité de services localisés dans le réseau. Autrement dit, ce réseau a une architecture client-serveur distribuée. La capacité de pouvoir s'adapter aux préférences des usagers, soit l'appelant ou le destinataire de l'appel, et la possibilité de maîtriser le concept de mobilité de façon ample et d'incorporer les services de présence et de messagerie instantanée sont introduites d'une façon naturelle avec la VoIP.

En analysant la proposition de la téléphonie sur IP du côté des usagers, on trouve que les frais de service significativement moins élevés, spécialement pour les communications interurbaines, et la portée globale sont les facteurs les plus attrayants. De plus, l'intégration de nouveaux services, très utiles et appréciés, est un autre facteur déterminant, surtout lorsque les utilisateurs sont des entreprises. En contrepartie, la qualité de service offerte est inférieure à celle de la téléphonie traditionnelle et le coût des postes, même avec des fonctionnalités minimales, est assez significatif.

L'utilisation d'un protocole en commun, le protocole IP, permet aux opérateurs d'utiliser l'infrastructure dédiée à la transmission de données, pour la transmission de la voix. C'est ce que l'on appelle «convergence entre réseaux informatiques et téléphoniques». Ceci, en addition aux coûts d'implantation et d'exploitation inférieurs, montre qu'il s'agit d'un modèle totalement ouvert à la concurrence et plein d'opportunités. Ces deux faits ont déclenché un vif intérêt de la part des entreprises du secteur des télécommunications.

La VoIP est, donc, un grand pas dans l'évolution de la téléphonie dans le processus plus général de convergence de réseaux. Cependant, il y a encore d'importants problèmes à résoudre, notamment ceux liés à la qualité de service, à la sécurité et à la prestation de services d'urgence, ceux-ci étant les points principaux qui permettent la résistance de la téléphonie traditionnelle vis-à-vis l'avancement de la VoIP.

2.2 Défis de la téléphonie sur IP

Dans une communication téléphonique, il est possible de distinguer deux phases. D'un côté, il y a un ensemble de tâches responsables de l'établissement, de la finalisation de la communication et de l'échange des différentes informations, comme par exemple la tarification. Il s'agit de la signalisation. D'un autre côté, on trouve l'échange d'informations de média, soit la voix. Les défis pour chacune d'elles sont bien différents. Ainsi, le protocole SIP de l'IETF, se charge de la signalisation, pendant que le protocole RTP règle l'échange d'informations en temps réel.

Les sections suivantes sont destinées à l'analyse des particularités que présentent la signalisation et la transmission de la parole sur des réseaux IP.

2.3 Particularités de la transmission de la parole sur un réseau IP

La figure 2.1 montre un exemple typique de la densité spectrale de la parole. Son analyse révèle que la majorité de l'énergie est concentrée dans la zone des basses fréquences, et que la quantité d'énergie qui se trouve dans les fréquences plus hautes que 3400 Hz et inférieures à 300 Hz est peu significative. C'est pour cette raison que la téléphonie traditionnelle prend comme bande passante la plage de 300 à 3400 Hz.

Le standard G.711 (*Pulse Code Modulation -PCM- of Voice Frequencies*) (ITU-T, 1988) établit que la voix, dans les systèmes de téléphonie numérique, doit être numérisée en utilisant la codification PCM (*Pulse Code Modulation*) avec une fréquence d'échantillonnage de 8 kHz et une codification logarithmique à 8 bits/échantillon ce qu'implique un débit de 64 kb/s. La fréquence d'échantillonnage choisie surgit de l'application du théorème de Nyquist. En conséquence, dans l'architecture correspondant à la téléphonie classique, les liens numériques entre centraux utilisent la technique de multiplexage dans le temps (TDM) dont le débit est un multiple de 64 kb/s (voix numérisée et signalisation).

Dans le cas de la téléphonie sur Internet, le flot de données généré pour la source (la voix, codifiée en PCM) est coupé en morceaux qui sont encapsulés en paquets et envoyés à intervalles réguliers. Ces paquets sont reçus par le destinataire et reconvertis en un flot

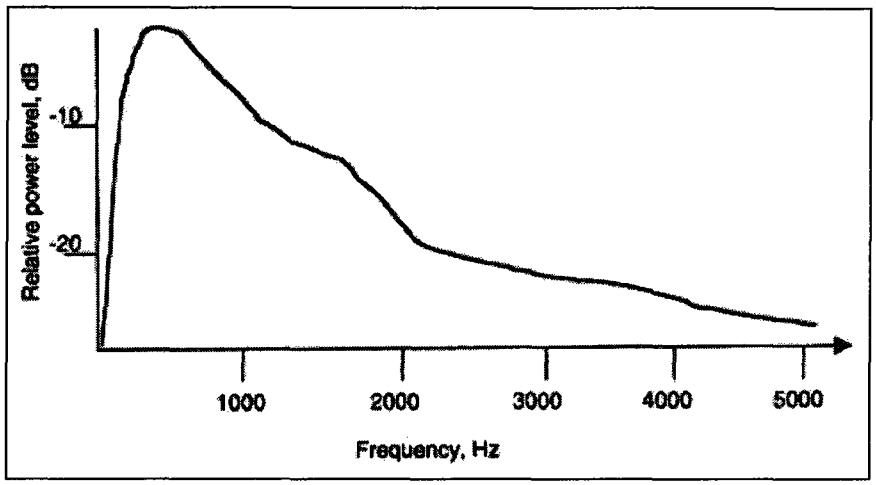

Figure 2.1: La densité spectrale de la parole (Goralski and Kolon, 1999)

continu pour alimenter les décodeurs. Dans cette situation, les trames ont typiquement une durée de 20 millisecondes (160 échantillons). Par conséquence, un poste qui établit une communication téléphonique en utilisant PCM sur l'Internet, émettra et recevra des paquets à chaque 20 millisecondes approximativement. Une première limitation qui se présente est, plutôt, d'ordre pratique : le débit requis pour des communications avec des codecs PCM est trop élevé pour des réseaux IP, surtout lorsqu'on considère la possibilité de plusieurs communications simultanées.

Des études des caractéristiques de la parole ont permis de contourner le problème. Par exemple, un codeur hybride, tel que celui spécifié par la recommandation G.729 (ITU-T, 2007), permet de transmettre la parole, avec une qualité raisonnablement bonne, avec un débit de seulement 8 kb/s. Plus encore, des techniques de suppression des silences, permettent d'atteindre des réductions additionnelles du débit de 30% à 50%.

En profitant de ces particularités, on peut transformer le flot continu et bidirectionnel simultané à 64 kb/s en un flot d'information de débit substantiellement inférieur, plus susceptible d'être transmis par un réseau de commutation par paquets.

Le débit requis n'est qu'un des problèmes reliés à la VoIP. Internet est un réseau qui n'a pas été conçu pour transporter des informations en temps réel et la conséquence immédiate est une série de difficultés qui se présentent dans ce type de service. Un ensemble de techniques permet de contourner, ou au moins de mitiger ces problèmes. Le coût à payer est une

complexité grandissante des applications de VoIP. Les sections suivantes sont destinées à présenter ces problèmes, leurs niveaux d'incidence et les solutions proposées afin d'arriver finalement à déterminer l'architecture de base d'une application de VoIP.

2.4 Facteurs incidents sur la qualité de la parole

Pour les applications de VoIP, on trouve quatre facteurs principaux, attribués à la nature du réseau, qui affectent la qualité de la voix : le délai de transit, la fluctuation de l'intervalle d'arrivée entre deux paquets successifs, ou gigue, l'écho et la perte de paquets.

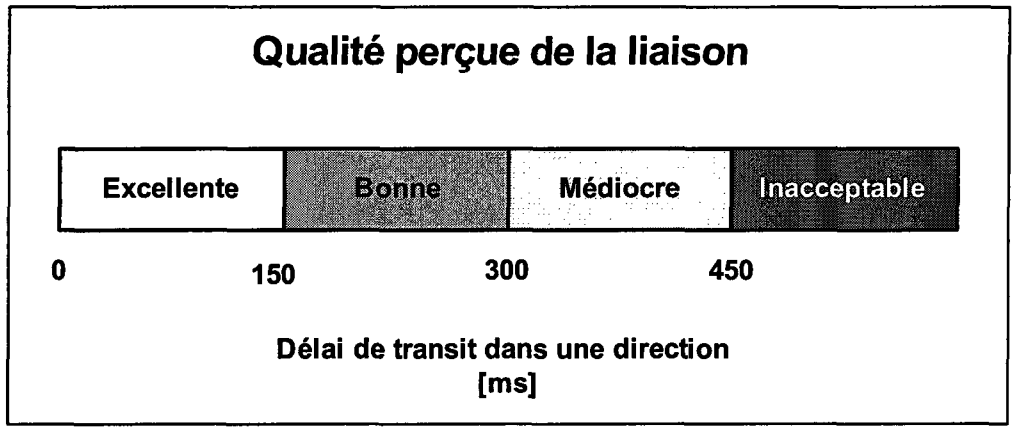

Figure 2.2: Qualité perçue de la liaison vs le délai de transit dans une direction (Percy, 2005)

2.4.1 Délai de transit

Le délai de transit (*latency*) est le temps qui s'écoule entre la production de la voix à une extrémité d'une communication, et sa reproduction à l'autre extrémité. Ce délai, qui n'est pas perçu quand la communication est unidirectionnelle, devient dérangeant pendant une conversation, car sa dynamique donne un cadre de référence. Autrement dit, lorsqu'on parle et qu'on fait une pause pour écouter notre interlocuteur, on attend pendant un certain temps, dépendant du rythme de la conversation. Nous sommes particulièrement sensibles à ce retard et à ses variations. Lorsqu'il dépasse une certaine valeur, cela devient incommodant jusqu'à perturber sérieusement la conversation.

La figure 2.2 illustre, de façon simplifiée, la qualité perçue en fonction du délai de transit. Des délais inférieurs à 150 ms ne sont pas perceptibles. Entre 150 et 300 ms, ils sont tolérables, mais des délais supérieurs à 300 ms ne sont pas acceptables.

L'ITU établit une valeur maximum de 150 ms pour le délai de transit dans une direction pour le trafic de voix. Pour les PSTNs les délais sont toujours inférieurs à 150 ms.

Délais de transit	Dans le téléphone	1. Délais de trame
		2. Délais de traitement
		3. Délais de paquétisation (regroupement de plusieurs trames dans un paquet)
		4. Délai dû au tampon de réception anti-gigue
	Dans le réseau	5. Délai dû aux passerelles (gateways)
		6. Délais de routage
		7. Délai dû aux coupe-feux (firewalls) et serveurs proxy
		8. Délais de transmission et de propagation

TABLEAU 2.1: Sources des délais de transit

Dans le cas de VoIP, certains délais sont inhérents aux principes de fonctionnement des téléphones IP et d'autres sont attribuables au réseau. Le tableau 2.1 résume les principales sources des délais pour la téléphonie IP.

Délais de trame

Comme on a déjà mentionné, la fréquence d'échantillonnage de la parole est de 8 kHz selon le standard utilisé par la téléphonie traditionnelle à partir des critères de qualité. Ceci signifie qu'un échantillon est disponible à chaque 125 μs. Les échantillons sont regroupés afin de permettre leur traitement sous forme de bloc (*batch processing*). La taille de ces groupes d'échantillons consécutifs, appelés trames, est variable, mais des considérations d'ordre pratique situent sa durée équivalente entre 10 et 30 ms. Certains codecs travaillent avec des longueurs de trame fixes. Ainsi, on peut identifier l'introduction d'un délai correspondant à l'intervalle qui s'écoule entre le moment de prise du premier échantillon d'une trame jusqu'à la prise du dernier, moment auquel la trame est rendue disponible. La table 2.2 donne quelques exemples.

. Codec .	Bande Passante [kb/s]	Durée de la trame [ms]	Taille de la trame [octets]
G.711 (ITU-T, 1988)	64	20	160
G.723.1 (ITU-T, 2006a)	5.3-6.3	30	20/24
G.729A (ITU-T, 2007)	8.0	10	10

TABLEAU 2.2: Caractéristiques des trames pour différents protocoles (Percy, 2005)

Délais de traitement

Lorsqu'une trame est disponible, elle est traitée par le détecteur d'activité vocale (*Voice Activity Detector*) et par le codec qui applique les algorithmes de compression. L'opération du VAD et du codec introduit un deuxième délai, qui dépendra de la complexité des algorithmes et de la puissance de calcul du processeur. La valeur supérieure pour ce délai est égale à la durée de la trame, car le processeur doit traiter les trames au minimum au même rythme qu'elles sont générées. Dans la pratique, le délai de traitement doit être suffisamment inférieur à la durée de la trame afin de permettre l'exécution d'autres tâches tout en conservant une certaine marge de sécurité.

Délais de paquétisation

Dans le but d'être transmise sur un réseau IP, une trame doit être encapsulée dans plusieurs protocoles. Le tableau 2.3 illustre cette situation dans le cas d'une trame G.723.1 (ITU-T, 2006a), dont la longueur est de 24 octets et sa durée équivalente est de 30 ms. Les entêtes des protocoles RTP, UDP *(User Datagram Protocol)* et IP utilisés pour le transport totalisent 40 octets ce qui représente une surcharge significative et, en conséquence, une efficacité du canal réduite.

En-tête Link Layer	En-tête IP 20 octets	En-tête UDP 8 octets	En-tête RTP 12 octets	Charge utile (trame G.723.1)
xx octets	40 octets			(24 octets)

TABLEAU 2.3: Encapsulation d'un paquet RTP G.723.1 (Percy, 2005)

Une manière de réduire cette inefficacité est de regrouper deux trames ou plus dans le même paquet. Mais, le coût à payer pour cet incrément de l'efficacité est l'introduction des délais additionnels égaux à la durée équivalente des trames ajoutées.

Délais dus au tampon de réception anti-gigue

En plus des délais de transit des paquets, il y a un autre effet qui affecte la qualité perçue de la voix. C'est la fluctuation de l'intervalle d'arrivée des deux paquets successifs, ou gigue (*jitter*).

Un réseau IP ne peut pas garantir le temps de transit des paquets étant donné que chaque paquet est acheminé individuellement et que le délai de transit dépend des conditions du réseau au moment de la transmission et du chemin à parcourir. En conséquence, des paquets émis à un rythme régulier arriveront de façon irrégulière ce qui entraîne le phénomène de gigue. Par ailleurs, la restitution de la parole est un processus synchrone, qui n'admet pas de variations dans le rythme de conversion sans qu'elles ne soient perçues clairement par l'utilisateur, en devenant ainsi gênantes. Le problème s'améliore en introduisant, dans les premières étapes du récepteur, un délai capable d'absorber de telles variations. Ce délai est matérialisé en utilisant un tampon de réception, qui, en raison de sa fonction, est appelé tampon anti-gigue. Évidemment, le retard introduit s'ajoute au délai de transit des paquets. Ainsi, il faut le maintenir le plus faible possible.

La capacité d'absorber les fluctuations des instants d'arrivée des paquets dépend de la longueur du tampon, ou d'un autre point de vue, du temps équivalent à la durée des trames stockées dans le tampon. D'habitude, le point d'opération de ces tampons découle d'un compromis entre les délais tolérés et la gigue admissible. Afin de maintenir ce délai le plus bas possible, les systèmes utilisent des mécanismes adaptatifs pour la détermination du point d'opération.

Passerelles

Chaque passage des paquets d'un média physique à un autre introduit des délais. Lorsqu'il s'agit de réseaux à vitesse modérée, ces délais peuvent devenir significatifs. Ils ont, fréquemment, leur origine au niveau des tampons employés par les procédés de conversion de média dans les passerelles (*gateways*).

Délai de routage

Dans le protocole IP, chaque paquet IP possède dans son en-tête l'adresse de destination

et l'adresse d'origine. Pour atteindre sa destination, les paquets sont acheminés par des routeurs qui doivent examiner les en-têtes. Avant d'être traités, les paquets sont mis dans des files d'attente. La logique de contrôle de ces files d'attente est conçue, en général, sans considérer les besoins des services temps réel et elle peut introduire des délais qui ne sont pas négligeables.

Pare-feux et serveurs mandataires (proxy servers)

Plusieurs réseaux utilisent des pare-feu (*firewalls*) et des serveurs mandataires afin d'assurer une meilleure sécurité. Comme ces dispositifs examinent chaque paquet entrant et sortant, ils peuvent introduire des délais considérables.

Délai de transmission et délai de propagation

Dans un réseau comme Internet, les paquets sont traités dans la modalité «premier arrivé, premier servi». Un paquet doit, alors, attendre que l'antérieur soit complètement traité avant son propre traitement. Ce temps d'attente est nommé «délai de transmission». Sa valeur est donnée pour le rapport entre la taille du paquet (en bits) et le débit (en b/s), en conséquence, le délai de transmission est de l'ordre des microsecondes pour des réseaux à haute vitesse avec des débits de 1 Mb/s ou supérieur.

Le délai de propagation d'un paquet est le temps qu'il prend pour parcourir le trajet entre l'entrée du circuit logique et la sortie. Il est calculé par le rapport entre la longueur du lien et la vitesse de propagation dans le milieu physique de transmission. Sachant que cette dernière est, pour un lien de cuivre, de $2x10^8$ m/s, le résultat sera aussi, dans l'ordre des microsecondes.

On observe alors que la contribution de ces deux phénomènes au délai de transit est négligeable par rapport au reste des sources de délai analysées.

2.4.2 Écho

L'écho est un phénomène lié plutôt à la téléphonie traditionnelle. Il est causé par les réflexions qui se produisent à cause de la désadaptation d'impédances quand le signal analogique est converti de deux à quatre fils ou vice-versa en utilisant des circuits (transformateurs) hybrides.

Dans le contexte de la téléphonie sur IP, l'écho se produit lorsque dans une communication de VoIP il y a des segments où le signal voyage sur une ligne analogique à deux fils et

des transformateurs hybrides deviennent nécessaires. Dans ces conditions, les délais toujours associés à la téléphonie IP font que le problème d'écho devient important. L'approche utilisée pour résoudre ce problème consiste à employer des dispositifs appelés annulateurs d'écho (*echo cancellers*) qui utilisent des techniques numériques. Lorsqu'il s'agit d'une communication VoIP entièrement numérique, l'écho ne constitue pas un problème. C'est le cas de deux ordinateurs ou téléphones numériques connectés via IP.

2.4.3 Perte de paquets

Dans une communication avec de l'information audio en temps réel, l'intervalle qui s'écoule entre la capture et la restitution des signaux est très court. Comme on peut voir dans la figure 2.2, une communication de bonne qualité montrera un délai de transit inférieur à 300 ms. En conséquence, la marge de manœuvre pour la correction d'erreurs est étroite. Si un paquet n'est pas arrivé et n'est pas disponible au moment où le codec le réclame, il est considéré perdu, même s'il arrive légèrement plus tard. Il s'ensuit qu'un protocole de transport fiable, tel que TCP (*Transmission Control Protocol*), ne serait pas avantageux, car si le protocole demande une retransmission, il est fort probable que le paquet retransmis arrive trop tard et qu'il ne soit plus utile. Finalement, on conclut que la complexité et la surcharge additionnelles associées à un protocole fiable ne contribuent pas à une meilleure performance. Ceci est la base de la justification de l'utilisation de UDP, face à TCP, pour le transport de flux en temps réel.

Étant donné que UDP est un protocole non fiable qui ne garantit ni l'arrivée, ni l'ordre des paquets, c'est l'application qui doit gérer ces problèmes. Ainsi, les applications avec des tampons anti-gigue adaptatifs ajustent ces points de travail afin de réduire le plus possible le taux de paquets perdus. Mais, malgré tous les efforts, il y aura un pourcentage de trames qui ne seront pas disponibles lorsqu'elles sont requises par le décodeur. Ainsi, le module de restitution de la parole doit essayer de compenser les informations manquantes. Une approche possible est de répéter le dernier segment de signal valide, une autre possibilité est de remplir le trou avec du bruit blanc. D'habitude, le choix surgit d'un compromis entre la complexité de la solution et la qualité désirée. Pour la conception de la pile SIP concernant à ce projet, on a sélectionné la première stratégie. Un mécanisme d'extrapolation des trames manquantes peut également être intégré au codec (Perkins, 2006).

Quelle que soit la stratégie de récupération employée, l'ouïe humaine est très sensible à ce type de problèmes. Un niveau de perte de paquets de 1% est normalement indétectable, tandis que des niveaux supérieurs à 10% ne sont pas tolérables.

2.5 Traitement de la parole

L'évolution des techniques de compression de la parole a permis de surmonter le problème du haut débit requis pour transmettre la parole numérisée avec une qualité adéquate en rendant disponibles des codecs très efficaces.

Dans une communication IP, la parole numérisée est soumise à différents traitements. La compression des trames, la détection de silences, l'annulation de l'écho, la détection des tonalités DTMF (*Dual Tone Multi-Frecuency*), la compensation des paquets perdus et l'ajout de bruit de confort sont des exemples. Le but principal des traitements faits avant la transmission est de réduire le débit requis tandis que les objectifs des traitements faits dans la réception sont de compenser les défauts introduits pendant le transit des paquets sur le réseau et de restituer le signal numérisé original le plus fidèlement possible.

2.5.1 Le codec et la compression de la parole

Le codeur fondamental de la parole en format numérique, et aussi le plus ancien, est le PCM. Comme on a déjà expliqué, il consiste à échantillonner la bande téléphonique, de 300 à 3400 Hz, à une fréquence de 8000 Hz en utilisant des convertisseurs de 8 bits. Afin d'améliorer le rapport signal sur bruit, les quantificateurs ont une échelle logarithmique. Il y a deux variantes pour la courbe de conversion : la loi mu, utilisée en Amérique du Nord et au Japon, et la loi A, utilisée dans le reste du monde. Il reste encore comme le standard qui établit les niveaux de référence pour la qualité de la parole numérisée en communications téléphoniques (ITU-T, 1988).

En prenant le PCM comme référence, différentes technologies ont été développées afin de réduire le débit requis pour une communication téléphonique. Les techniques utilisées peuvent être regroupées dans trois catégories :

- ***Les techniques paramétriques*** (LPC, *Linear Predictive Coding*) : elles s'appuient sur un modèle de production de la parole afin d'en extraire les paramètres significatifs, qui sont transmis au décodeur.

- ***Les techniques temporelles*** (PCM, ADPCM *Adaptive Delta Pulse Code Modulation*) : elles font une approximation de la parole à partir de sa forme d'onde.
- ***Les techniques hybrides*** (analyse par synthèse, codage CELP *Code Excited Linear Prediction*) : elles font une combinaison des techniques précédentes et permettent d'obtenir de hauts niveaux de compression avec une qualité acceptable.

Le tableau 2.4 résume les caractéristiques des codecs correspondant aux standards de l'ITU les plus utilisés.

Standard - ITU	CODEC Algorithme conversion	Débit [kb/s]
G.711	PCM	64
G.721	ADPCM	16 / 24 / 32 / 40
G.721	Sub band ADPCM	48 / 56 / 64
G.723.1	Multirate CELP	5,3 / 6,3
G.726	ADPCM	16 / 24 / 32 / 40
G.727	EADPCM	16 / 24 / 32 / 40
G.728	LD-CELP	16
G.729	CS-ACELP	8

TABLEAU 2.4: Standards de codification établis pour ITU

2.5.2 La suppression des silences

Pendant une conversation typique, on ne parle pas tout le temps. Des études démontrent que le 40 à 60% du temps correspond à des silences. Il s'ensuit qu'un pourcentage proche du 50% des trames transmises correspond seulement à du silence. La détection et l'écartement de ces trames permettent d'avoir une réduction importante du débit requis. Ainsi, les techniques de suppression des silences ne codent pas ces périodes de silence. Elles sont remplacées à l'autre extrémité par du bruit de fond. Un problème qui se présente est la détection efficace des silences. Cette tâche est effectuée par le VAD.

L'utilisation conjointe des techniques de compression de la parole et de suppression des silences permet d'atteindre de débits relativement réduits avec une utilisation très efficace du réseau.

2.5.3 La génération de bruit de confort

La suppression des silences produit, du côté récepteur, des trous dans la réception qui sont gênants. La génération de bruit de confort est utilisée pour compenser ce défaut et pour maintenir l'illusion d'un niveau constant de bruit pendant les intervalles supprimés. Ce processus est appelé CNG (*Confort Noise Generation*).

2.6 Transport en temps réel et le protocole RTP

Une analyse de la problématique de la téléphonie sur IP nous amène à identifier deux volets. D'un côté, on a l'ensemble des éléments en relation avec la signalisation, avec SIP, qui est le protocole associé. D'un autre côté, on trouve les aspects liés à la transmission de la parole numérisée. C'est le domaine du protocole RTP, spécifié par la recommandation RFC 3550 (Schulzrinne et al., 2003).

RTP est un protocole simple, mais qui incorpore les éléments nécessaires pour la transmission en temps réel d'information audio ou vidéo sur des réseaux IP. Conçu originalement comme un protocole de multidiffusion (*multicast*), il est si bien adapté à des applications à diffusion individuelle (*unicast*) qu'il est devenu une des pièces essentielles de la téléphonie sur IP.

Les services fournis par RTP incluent l'identification de la source et du type de charge utile, la numérotation séquentielle des paquets, des marques de référence temporelles (*timestamping*), une rétroaction sur la qualité de la réception et un mécanisme de synchronisation. Dans le chapitre 4, on reprendra le sujet du protocole RTP avec le but d'analyser plus en profondeur sa structure et ses caractéristiques les plus importantes.

2.7 Signalisation et le protocole SIP

Dans un réseau téléphonique traditionnel, la signalisation est très complexe et joue un rôle crucial puisqu'elle est chargée de l'établissement, la terminaison et la supervision d'une communication. La majorité des opérations sont réalisées par les centraux téléphoniques, du côté des opérateurs. Seule une partie infime de ces opérations est assurée par les postes téléphoniques.

Dans le cas de la téléphonie sur IP, l'importance de la signalisation est également décisive, mais les défis rencontrés sont différents. Parmi les aspects qui définissent le scénario auquel la VoIP doit faire face, on trouve que, contrairement au cas du PSTN, la responsabilité de gérer la signalisation incombe aux terminaux. De plus, les conditions continuellement changeantes des réseaux, et la différence entre les capacités des postes imposent, comme préalable à l'établissement d'une communication, la négociation des paramètres entre les postes participants. Un autre point à considérer est la dilution du concept de réseaux physiquement délimité, fortement rattaché à la téléphonie traditionnelle. En conséquence, dans l'hypothèse de faire face à un trafic de portée globale, les postes doivent répondre à des standards, même s'ils appartiennent à des réseaux privés.

Les besoins en matière de signalisation de la téléphonie sur IP ont donné lieu à une nouvelle génération de protocoles, dont H.323 (ITU-T, 2006b), de l'ITU, et SIP (Rosenberg et al., 2002), de l'IETF, sont les plus notables. Pour le présent travail, l'intérêt est centré sur le protocole SIP.

SIP est un protocole de signalisation qui peut établir, modifier et terminer des sessions multimédias sur des réseaux IP. Il est décrit par le RFC 3261 (Rosenberg et al., 2002) de l'IETF.

SIP est défini d'une façon flexible, utilisant une codification textuelle. Ceci le rend très extensible et facile à interpréter, mais à la fois, son implémentation devient plus complexe. Dès son apparition, SIP a profité d'une grande popularité auprès des concepteurs. Sa proximité avec HTTP *(HyperText Transfer Protocol)* et le style de spécification utilisé par l'IETF, avec lequel les informaticiens sont plus à l'aise, sont parmi les principales raisons.

Même si SIP permet d'utiliser UDP ou TCP pour le transport, UDP est considéré comme le premier choix, limitant TCP pour les cas où UDP n'est pas applicable. Le protocole SIP est couvert avec détail dans le chapitre 3.

2.8 Couches d'abstraction

Lorsqu'une application s'exécute sur un ordinateur, elle doit interagir avec un ensemble complexe de matériel et de logiciel. Dans le cas des systèmes embarqués, la situation est plus compliquée encore, car les différences dans les environnements d'exécution sont généralement très importantes. Ces différences peuvent inclure le microprocesseur, le type

et la taille de la mémoire, les périphériques et le système d'exploitation. La portabilité d'une application est en relation directe avec le niveau d'isolation qu'elle peut obtenir de tous ces éléments.

La manière d'isoler une application des couches sous-jacentes consiste à concevoir un ensemble de fonctions qui concentrent les points de dépendance et qui soient vues par l'application d'une façon uniforme. Ces ensembles des fonctions d'isolation sont appelés couches d'abstraction. Ainsi, le transport d'une application vers un autre environnement, idéalement, ne demanderait que de modifier ces couches.

Dans le cadre de ce projet, deux couches d'abstraction sont présentes : la couche d'abstraction matérielle et celle d'abstraction du système d'exploitation.

2.8.1 Services du système d'exploitation

Une application typique de téléphonie IP utilise un certain nombre de services du système d'exploitation. On peut citer, comme exemples, les suivants :

- Services de temporisation
- Gestion de la mémoire
- Gestion des systèmes de fichiers
- Gestion de processus et de tâches.
- Communication interprocessus et des mécanismes de synchronisation.
- Gestion de périphériques
- Information de la date / heure
- Événements
- Gestion de la sécurité
- Gestion du réseau
- Accès aux ressources matérielles

Ces services sont fournis en utilisant des APIs *(Application Programming Interface)* dont la complexité variera beaucoup selon le système d'exploitation avec lequel on travaille. Dans le cas des systèmes embarqués, il faut que les APIs soient limitées strictement au minimum, très simplifiées et performantes.

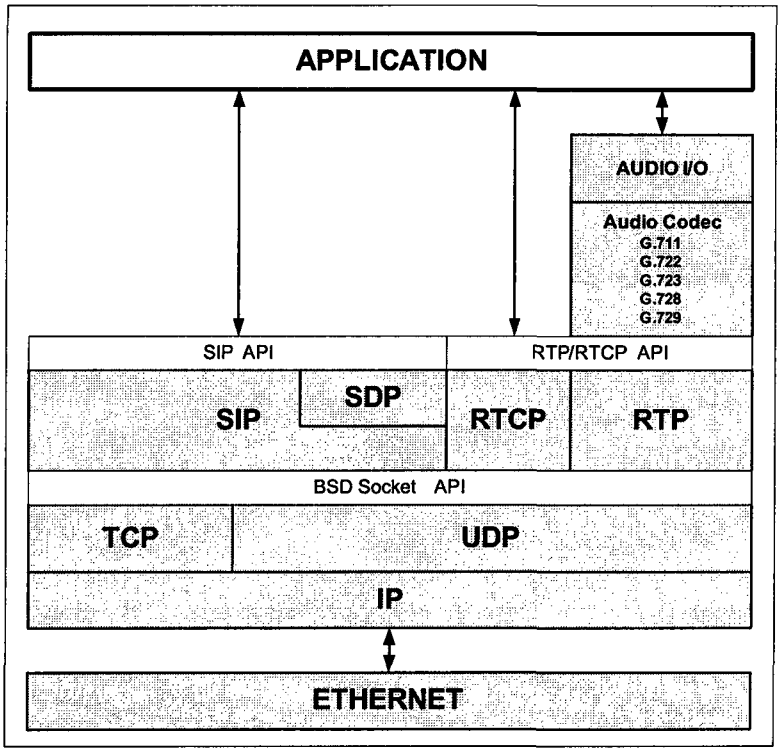

Figure 2.3: Pile de protocoles pour une application de téléphonie IP

2.8.2 La couche d'abstraction matérielle

Cette couche est aussi appelée HAL (*Hardware Abstraction Layer*) et sert d'interface entre une application et tout le matériel qui n'est pas géré par le système d'exploitation. Même les particularités du processeur peuvent être définies dans cette couche.

Par ailleurs, les systèmes embarqués sont caractérisés pour leur grande variété au niveau des caractéristiques matérielles, même pour des prestations fonctionnelles équivalentes. Différents processeurs et ensembles de puces (*chip set*) et différentes configurations de mémoire, par exemple, sont cachés derrière cette couche.

Dans une application de VoIP, la couche d'abstraction matérielle peut servir, par exemple, à isoler l'application des détails du sous-système de traitement audio (accès aux convertisseurs analogiques/numériques et numériques/analogiques), des ports d'entrée/sortie et des temporisateurs matériels (*hardware timers*).

2.9 Piles de protocoles pour une application de VoIP

Les sections précédentes ont permis de déterminer les besoins de base d'une application de téléphonie sur IP et les protocoles disponibles à cet effet. D'un côté, pour la parole numérisée, on peut se servir des protocoles RTP/RTCP (*RTP Contol Protocol*). D'un autre côté, pour la signalisation, les deux protocoles les plus utilisés sont le SIP de l'IETF, et le H.323 de l'ITU. Cependant, dans le présent projet, seule l'application de SIP est visée. La figure 2.3 illustre la structure résultante de la pile de protocoles. Il faut noter que la paire RTP/RTCP n'est supportée que par UDP, tandis que SIP peut utiliser UDP ou TCP pour le transport.

Dans la pile de protocoles, on a inclus SDP (*Session Description Protocol*), spécifié par la recommandation RFC 4566 de l'IETF (Handley et al., 2006). C'est un protocole simple qui sert à négocier les paramètres de la session, tels que le type de média (audio, vidéo, etc.) et le codec à utiliser (G.711a, G.729a, etc.). Il voyage embarqué dans SIP pendant l'établissement des sessions. Cette fonctionnalité n'est pas implantée dans la première version de ce projet de conception.

2.10 Architecture générale d'une application de terminal VoIP

L'analyse de différentes facettes de la téléphonie sur IP, faite tout au long de ce chapitre, a permis d'établir les grandes lignes d'une application de téléphonie sur IP. La figure 2.4 illustre son architecture générale. Le schéma met en évidence les blocs associés aux fonctions que l'on a identifiées comme essentielles pour de tels types d'applications. D'un côté, le couple de protocoles RTP/RTCP est à la base de la partie chargée de la transmission-réception de la parole. D'un autre côté, SIP gère les tâches de signalisation. Le bloc «Traitement de la parole» concentre les différentes fonctions de traitement de la parole, celles-ci incluent la compression - décompression, la suppression de silences, la génération du bruit de confort et l'annulation de l'écho.

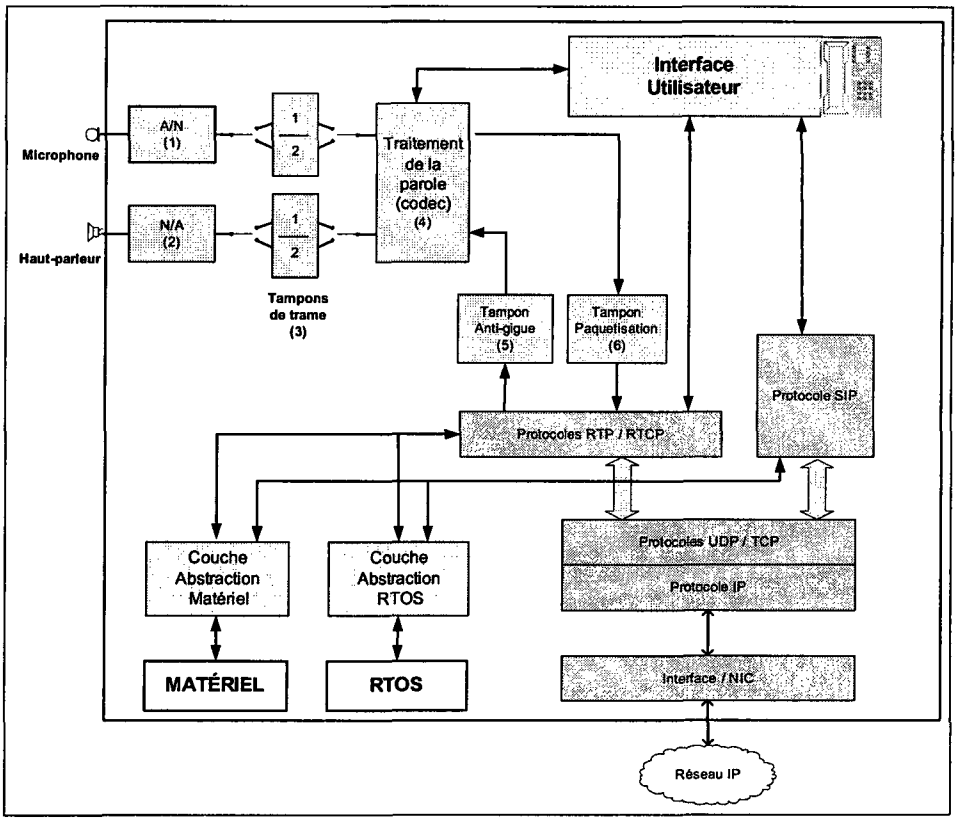

Figure 2.4: Architecture générale d'une application de terminal de VoIP

2.11 Conclusion

Ce chapitre a servi à identifier la problématique présentée pour l'utilisation des réseaux IP pour des applications de téléphonie. On a présenté, de manière très générale, les solutions proposées pour contourner les problèmes et on a introduit les protocoles disponibles pour arriver, à la fin du chapitre, à ébaucher l'architecture d'une application logicielle de VoIP. Dans les chapitres suivants, on fera une analyse plus en profondeur des protocoles SIP et RTP qui servira comme introduction à la description de leur mise en œuvre.

CHAPITRE 3
PROTOCOLE SIP

La problématique de base de la téléphonie sur IP peut être séparée, de façon générale, en deux parties. D'un côté, nous avons les aspects liés à la signalisation, soit l'ensemble des mécanismes permettant de localiser, d'établir, de gérer et de terminer une communication. D'un autre côté, nous devons faire face aux défis qui représentent la transmission de la parole numérisée sur un réseau IP. Le protocole SIP qui dérive d'HTTP, dont il reprend de nombreuses caractéristiques, est la réponse de l'IETF au premier de ces sujets. Son fonctionnement s'appuie sur des échanges de messages en mode texte. Simplicité, extensibilité et flexibilité sont des caractéristiques qui justifient la popularité d'un protocole qui rapidement est devenu un composant essentiel de la quasi-totalité des implémentations de VoIP.

Le but de ce chapitre est d'introduire les concepts fondamentaux du protocole SIP pour aider à la compréhension de sa mise en œuvre, décrite dans le chapitre 7. La première partie donne une vision générale du SIP, en décrivant ses fonctionnalités de base et les entités constitutives d'un réseau SIP. Ces points sont complétés par l'inclusion d'un exemple d'établissement d'une session. La deuxième partie présente l'analyse de la structure du protocole, incluant la description de la composition des messages, de sa grammaire et des éléments qui déterminent son comportement dynamique, comme les concepts de transaction et de dialogue. Finalement, il faut mettre en relief que certains aspects, notamment ceux liés aux machines à états finis qui décrivent les transactions et ceux liés aux dialogues, sont repris dans le chapitre 7 afin d'établir une liaison directe avec sa mise en œuvre.

3.1 Protocole SIP

SIP est un protocole de signalisation, de présence et de messagerie instantanée, situé dans la couche d'application du modèle OSI *(Open Systems Interconnection)*. Il permet d'établir, de modifier et de terminer des sessions multimédias, entre deux ou plusieurs participants, sur des réseaux IP. La première version de SIP, le RFC 2543, a été publiée par l'IETF en 1999 (Handley et al., 1999). Une deuxième version, apportant quelques modifications

et des corrections, a été publiée plus tard. Il s'agit du standard RFC 3261 (Rosenberg et al., 2002) qui a été suivi par plusieurs recommandations complémentaires telles que le RFC 3262 (Rosenberg and Schulzrinne, 2002a), le RFC 3263 (Rosenberg and Schulzrinne, 2002b), le RFC 3264 (Rosenberg and Schulzrinne, 2002c), le RFC 3265 (Roach, 2002) et le RFC 3853 (Peterson, 2004).

Il a été conçu en prenant certains éléments provenant de deux autres protocoles bien connus, HTTP (*HyperText Transfert Protocol*) et SMTP (*Simple Mail Transfert Protocol*). Du premier, SIP en a conservé le modèle client-serveur, l'utilisation des URIs *(Uniform Resource Indentifier)* et le schéma de codification textuelle. De SMTP, il a pris la structure de l'entête.

Sa simplicité, sa proximité avec HTTP, et sa focalisation sur les réseaux IP sont parmi les clés du succès initial de SIP, un protocole qui a rapidement gagné l'intérêt des entreprises et des concepteurs par rapport à son principal concurrent, le protocole H.323 de l'ITU.

En tant que protocole de signalisation, SIP supporte cinq aspects de l'établissement et de la terminaison de communications multimédias :

- **Localisation de l'utilisateur appelé**. SIP utilise des URIs comme identificateurs de portée globale. Ils permettent d'identifier de manière univoque les utilisateurs. La localisation consiste à établir la correspondance entre les URIs et les adresses IP des postes terminaux.
- **Disponibilité de l'utilisateur appelé**. Détermine si le poste appelé souhaite communiquer, et autorise l'appelant à le contacter.
- **Capacités de l'utilisateur**. Détermine le type de média et les paramètres à utiliser pendant la communication.
- **Établissement de la session**. Cet aspect regroupe les différentes étapes qui conduisent à l'établissement effectif de la communication.
- **Gestion de la session**. Elle comprend la finalisation ou le transfert d'une session, la modification des paramètres et l'invocation de services.

SIP ne définit pas un système de communications verticalement intégré. Il constitue plutôt un composant qui opère en collaboration avec d'autres protocoles de l'IETF afin de bâtir une architecture de communications multimédias intégrale. Quelques protocoles intervenants

sont RTP (Schulzrinne et al., 2003) et SDP (Handley et al., 2006). Pour le transport, SIP permet l'utilisation du protocole UDP autant que TCP.

3.2 Entités définies par SIP

Un réseau SIP comprend cinq types d'entités logiques : UA (User Agents), qui est sous-divisé, entre UAC (*User Agens Clients*) et UAS (*User Agent Servers*) ; PS (*Proxy Server*) ; LS (*Location Server*) ; RS (*Redirect Server*) et RG (*Registrar Server*). Un dispositif physique est capable d'assumer un seul ou plusieurs rôles en même temps ou en différents moments. En particulier, le rôle des UACs et des UASs se définit sur une base de transaction à transaction. Par exemple, un UA se comporte comme un UAS lorsqu'il reçoit une requête, mais il agit comme un UAC quand il l'émet.

- **UA** : C'est une entité logique terminale qui peut se comporter comme client ou comme serveur. Selon son rôle, un UA peut être classifié comme UAC ou UAS.
  UAC : C'est l'UA qui émet une requête SIP.
  UAS : C'est l'UA qui reçoit une requête et qui génère la réponse.
- **PS** : Entité intermédiaire qui aide à router des messages SIP vers sa destination. Il peut fonctionner comme UAC ou UAS dans le but d'émettre des requêtes ou d'y répondre en fonction d'autres UA. Un PS a la capacité d'interpréter et, si nécessaire, de modifier la requête originale.
- **LS** : Il fournit l'information de la possible position courante d'un UA. Ce service est utilisé par les PSs ou par les RSs.
- **RS** : C'est un UAS qui génère des réponses de redirection (avec de réponses codes qui se trouvent dans la plage 300 - 399) aux requêtes reçues d'un UAC. Le RS fait une association entre l'adresse de destination originale et une ou plusieurs adresses IP alternatives.
- **RG** : C'est un serveur qui offre un service de localisation et qui accepte des requêtes REGISTER.

3.3 Sip URI vs. URL

Les «Uniform Ressource Locators» (URLs) sont des noms utilisés pour représenter des emplacements ou des adresses IP liées aux ressources dans l'Internet. Ils ont été initiale-

ment spécifiés dans la recommandation RFC 1738 (Berners-Lee et al., 1994) de l'IETF, aujourd'hui obsolète. Le format général de sa syntaxe est :

<schéma> :<identificateur>

Le schéma représente le type de ressource utilisé tandis que l'identificateur, qui dépend du type de schème, détermine l'adresse IP de destination.

Un exemple d'URL est «http ://www.usherbrooke.ca». Le schéma indique qu'il s'agit d'un serveur http et l'identificateur www.usherbrooke.ca représente l'adresse IP du serveur qui héberge le site web de l'Université de Sherbrooke.

Par ailleurs, les «Uniform Ressource Identifiers» (URI) définissent de manière univoque un nom dans un espace des noms enregistrés avec une portée globale. Les URIs sont définis par le RFC 3986 (Berners-Lee et al., 2005), qui a la catégorie de standard. À différence des URLs, les URIs ne sont pas directement liés à des emplacements ou des adresses IP des objets nommés. La correspondance entre un URI et une adresse IP, qui peut être éphémère, est faite à l'aide des éléments complémentaires tels que bases de données ou serveurs spécialisés. C'est pour cette raison que le protocole SIP utilise des URIs étant donnée la mobilité associée aux entités SIP.

3.4 Exemple de l'opération de SIP

La figure 3.1 illustre l'établissement d'une session SIP entre deux agents utilisateur (UA). La topologie de l'exemple, qui inclut deux serveurs mandataires, est souvent nommée trapèze SIP. Chacun des utilisateurs est identifié par son URI (*Uniform Resource Identifier*). Les URIs sont l'équivalent des numéros de poste de la téléphonie traditionnelle, mais à différence de ceux-ci, les URIs ont une portée globale et sont associés à l'utilisateur plutôt que d'être rattachés à un terminal déterminé. La localisation, soit la liaison entre un URI et l'adresse IP qu'il prend à un moment donné, est faite à l'aide de serveurs spécialisés appelés **REGISTRAR**.

Dans l'exemple, François se sert de l'URI de Louis pour initier l'appel. Tout d'abord, l'UA appelant génère une requête INVITE qui montre l'intention d'établir une session. Étant

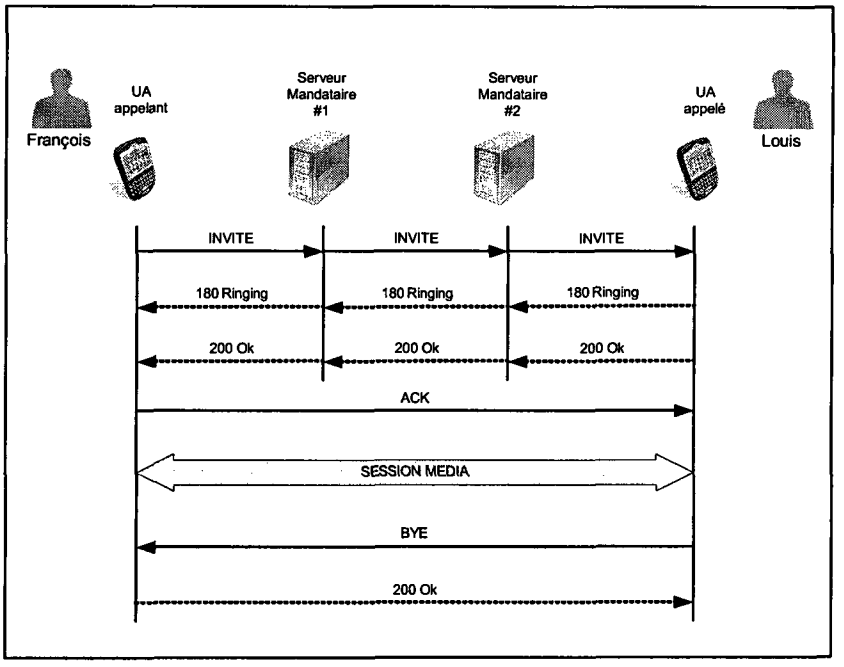

Figure 3.1: Exemple d'une session SIP

donné que François ne connaît pas la localisation actuelle de Louis, il envoie la requête vers son serveur proxy, qui l'analyse et détermine le domaine correspondant à l'URI de Louis. Puis, ce serveur renvoie la requête vers le serveur *proxy* servant ce domaine. Finalement, c'est ce dernier qui détermine l'adresse IP où se trouve actuellement Louis afin de lui acheminer la requête.

Chacune des entités SIP traversées par la requête, laisse une trace dans le message afin de permettre à la réponse de trouver le chemin de retour. Cette trace est matérialisée par l'ajout d'un en-tête **Via** par chaque serveur visité par le message.

Lorsque la requête arrive à l'UA de Louis, cela génère une première réponse automatique, «180 Ringing», pour indiquer que le message a été reçu et un avis, comme une sonnerie par exemple, est activé afin d'informer l'utilisateur appelé de l'appel. Quand Louis décide d'accepter l'appel, une deuxième réponse, «200 Ok», est envoyée. C'est une réponse finale que termine la transaction. Les deux réponses envoyées par l'UA de Louis remontent exactement le même parcours que la requête originale. Dans l'UA appelant, la réception d'une réponse finale positive déclenche l'envoi d'une requête de confirmation **ACK**. Contrairement aux

messages antérieurs, l'**ACK** est envoyé de bout en bout, en profitant de l'information de localisation additionnelle collectée pendant la transaction **INVITE**.

C'est à partir de ce moment qui commence l'échange d'information audio, qui va continuer jusqu'à ce qu'un des participants raccroche. Dans l'exemple, c'est Louis qui termine la communication, son UA génère une requête **BYE**. Lorsque la réponse est reçue, la transaction et la session sont, toutes les deux, finies.

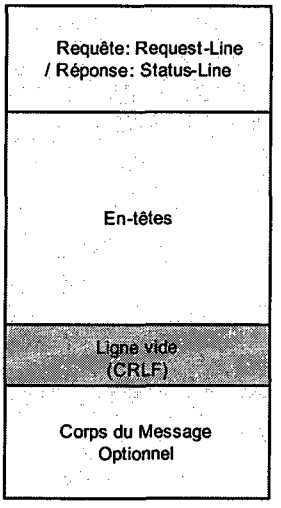

Figure 3.2: Structure des messages SIP

3.5 Structure du protocole SIP

SIP est un protocole structuré par couches, car son comportement est décrit par des stades de traitement faiblement couplés entre eux. Les éléments établis par SIP sont des éléments logiques plutôt que physiques et ils ne conditionnent pas la façon de mettre en œuvre le protocole.

La couche inférieure correspond à la syntaxe et au codage du protocole, qui est spécifié en utilisant la grammaire *Backus-Naur Form* augmentée (BNF). La grammaire du SIP est intégralement spécifiée dans la section 25 du RFC 3261 (Rosenberg et al., 2002).

La deuxième couche est la couche de transport. Elle définit comment les clients et les serveurs envoient et reçoivent des requêtes et des réponses sur le réseau.

La troisième est la couche de transaction. Elle constitue un élément central de SIP en établissant le fonctionnement des clients et des serveurs durant les interactions entre des entités SIP. La couche de transaction contrôle les retransmissions, la correspondance entre des requêtes et des réponses et les délais d'attente (*timeouts*). Le comportement de cette couche est modélisé par des machines à états finis.

3.6 Messages SIP

Un message SIP peut être une requête, qu'un client envoie vers un serveur, ou une réponse d'un serveur vers un client. En termes de sa grammaire, on peut spécifier un message SIP comme suit :

```
generic message = start-line
                  *message-header
                  CRLF
                  [ message-body ]
start-line      = Request-Line / Status-Line
```

La figure 3.2 illustre la structure d'un message. C'est à partir de la première ligne qu'on détermine son caractère de requête ou de réponse. L'ensemble d'en-têtes contient l'état de la session. La ligne vide est obligatoire et elle sert à démarquer la fin de la section d'en-têtes et il faut l'inclure indépendamment de la présence du corps du message, qui lui est optionnel.

3.6.1 Requêtes

Le format de la ligne de requête est le suivant :

```
Request-Line = Method SP Request-URI SP SIP-Version CRLF
```

Method : La recommandation RFC 3261 définit six méthodes : REGISTER, INVITE, ACK, CANCEL, BYE et OPTIONS. Des méthodes additionnelles sont définies dans des extensions à cette recommandation. Quelques exemples sont le RFC 3262 (Rosenberg and Schulzrinne, 2002a) pour la méthode PRACK, le RFC 3265 (Roach, 2002),

qui introduit les méthodes SUBSCRIBE et NOTIFY ou le RFC 3311 (Rosenberg, 2002), qui définit la méthode UPDATE.

Request-URI : Indique l'utilisateur ou le service destinataire de la requête.

SIP-Version : C'est la version du protocole utilisée. SIP le considère comme une chaîne de caractères qui, pour la version courante, doit être égale à «SIP/2.0».

3.6.2 Méthodes SIP

Les méthodes informent du caractère des requêtes et ils expriment l'intention de l'UAC de faire une action déterminée. Les six méthodes mentionnées dans le paragraphe précédent sont fondamentales pour le fonctionnement du protocole et doivent toujours être implémentées. La nature de chaque application déterminera si des méthodes additionnelles peuvent être nécessaires. Lorsqu'un UAS reçoit une requête avec une méthode qu'il ne supporte pas, il répond en indiquant que la méthode n'est pas implémentée.

À titre de référence, on inclut une très brève description des six méthodes définies dans le RFC 3261 :

REGISTER : Cette méthode est utilisée par un UA dans le but de notifier un réseau SIP de sa localisation courant, soit son «Contact URI» (adresse IP) afin que les requêtes INVITE qui lui sont destinées puissent l'atteindre.

INVITE : C'est la méthode la plus importante de celles définies par SIP et elle est utilisée par un UAC pour initier une session. Cette méthode peut aussi être utilisée lorsqu'une session est déjà établie afin de changer ses paramètres. Dans ce cas-là, on parle plutôt de re-INVITE.

ACK : Elle est utilisée pour accuser la réception des réponses finales pour la méthode INVITE. Quand les réponses finales sont négatives, l'ACK prend part à la même transaction que la méthode INVITE qui l'a générée et sa transmission est réalisée sur une base saut à saut (hop-by-hop). Par contre, si la réponse finale est positive (200 Ok), l'envoi de l'ACK, fait d'une extrémité à l'autre (end-to-end), est considéré comme une transaction indépendante.

CANCEL : La méthode CANCEL permet de terminer des transactions en cours et d'annuler une INVITE avant que la réponse finale ne soit générée.

BYE : Elle est utilisée pour terminer une session déjà établie.

OPTIONS : C'est une méthode qui est utilisée pour découvrir les capacités et la disponibilité d'un UA sans établir un dialogue avec lui.

Les noms de ces méthodes doivent toujours être écrits en majuscules.

3.6.3 Exemple de requête INVITE

```
INVITE sip:bob@biloxi.example.com SIP/2.0
Via: SIP/2.0/TCP client.atlanta.example.com:5060;branch=z9hG4bK74bf9
Max-Forwards: 70
From: Alice <sip:alice@atlanta.example.com>;tag=9fxced76sl
To: Bob <sip:bob@biloxi.example.com>
Call-ID: 3848276298220188511@atlanta.example.com
CSeq: 1 INVITE
Contact: <sip:alice@client.atlanta.example.com;transport=tcp>
Content-Type: application/sdp
Content-Length: 151

v=0
o=alice 2890844526 2890844526 IN IP4 client.atlanta.example.com
s=-
c=IN IP4 192.0.2.101
t=0 0
m=audio 49172 RTP/AVP 0
a=rtpmap:0 PCMU/8000
```

Figure 3.3: Exemple de requête INVITE (Johnston et al., 2003)

La figure 3.3 montre un exemple de requête INVITE extraite du RFC 3665, «*SIP Basic Call Flow Examples*» (Johnston et al., 2003).

La première ligne du message (*request line*) nous indique qu'il s'agit d'une requête **INVITE**. Puis on trouve l'ensemble d'en-têtes : *Via*, *Max-Forwards*, *From*, *To*, *Call-ID*, *CSeq*, *Contact*, *Content-Type* et *Content-Length*. L'en-tête *Via* indique que le protocole de transport est TCP. La valeur de *Content-Length*, d'inclusion obligatoire lorsque le transport est TCP, donne la longueur du corps du message et *Content-Type* dit que leur contenu est une offre SDP (Session Description Protocol) (Handley et al., 2006). Il faut aussi noter que la requête n'a qu'un seul paramètre *tag*, dans l'en-tête *From*. Celui de l'en-tête *To* devra être ajouté par le serveur. L'ensemble d'en-têtes est suivi d'une ligne vide et du corps du message, qui contient une offre SDP.

3.6.4 Réponses

La première ligne d'une réponse s'appelle **ligne d'état**. Sa structure est la suivante :

```
Status-Line = SIP-Version SP Status Code SP Reason-Phrase SP CRLF
```

Le ***status code*** est représenté par un numéro de trois chiffres, dont le premier définit la classe de la réponse. Dans la section 7.2, le RFC 3261 (Rosenberg et al., 2002) considère six classes de réponses :

- **1xx** : Réponse provisionnelle. La requête a été reçue et son traitement continue.
- **2xx** : Réponse finale positive.
- **3xx** : Redirection.
- **4xx** : Erreur du client. Possible erreur dans la requête.
- **5xx** : Erreur du serveur. La requête reçue est, apparemment, correcte, mais le serveur n'est pas capable de la traiter.
- **6xx** : Défaillance globale. La requête reçue ne peut être traitée par aucun serveur.

Reason Phrase est un bref texte inclus dans le seul but de faciliter la lecture humaine. La recommandation suggère des textes pour chaque *status code*. Pour faire le traitement des réponses, les implémentations ne doivent considérer que le code du *status code*, tout en ignorant les *reason phrases*.

3.6.5 Exemple de réponse à une requête INVITE

La figure 3.4 montre la réponse correspondante à la requête de l'exemple de la figure 3.3. Il a été pris, aussi, du RFC 3665 (Johnston et al., 2003).

On observe que la première ligne du message lui identifie comme une réponse. Celle-ci est finale et positive, car le code de la réponse se trouve dans la plage 200-299. On constate que le paramètre *received* a été ajouté dans l'en-tête *Via* afin d'indiquer l'adresse IP de la source du message. C'est important de noter que la réponse inclut le *tag* dans l'en-tête *To*. Étant donné que la réponse est positive, un dialogue sera établi. À partir de ce moment, le trio *Call-ID*, *From-tag* et *To-tag* vont l'identifier de manière univoque et ils doivent rester invariables et être inclus dans tous les messages échangés dans le contexte de ce dialogue.

Finalement, la réponse inclut aussi un corps du message contenant la réponse à l'offre SDP (Handley et al., 2006) reçue dans la requête.

```
SIP/2.0 200 OK
Via: SIP/2.0/TCP client.atlanta.example.com:5060;branch=z9hG4bK74bf9
  ;received=192.0.2.101
From: Alice <sip:alice@atlanta.example.com>;tag=9fxced76sl
To: Bob <sip:bob@biloxi.example.com>;tag=8321234356
Call-ID: 3848276298220188511@atlanta.example.com
CSeq: 1 INVITE
Contact: <sip:bob@client.biloxi.example.com;transport=tcp>
Content-Type: application/sdp
Content-Length: 147

v=0
o=bob 2890844527 2890844527 IN IP4 client.biloxi.example.com
s=-
c=IN IP4 192.0.2.201
t=0 0
m=audio 3456 RTP/AVP 0
a=rtpmap:0 PCMU/8000
```

Figure 3.4: Exemple de réponse 200 Ok (Johnston et al., 2003)

3.6.6 En-têtes SIP

Les champs des en-têtes SIP sont similaires à ceux d'HTTP en ce qui concerne aussi bien la syntaxe que la sémantique. La grammaire des en-têtes a le format suivant :

```
header = "header-name" HCOLON header-value *(COMMA header-value)
```

Plusieurs en-têtes possèdent un ou plusieurs paramètres séparés par des points-virgules. Ils sont décrits dans la grammaire de la manière suivante :

```
field-name : field-value  *( ;parameter-name=parameter-value)
```

Les champs des en-têtes peuvent comporter plusieurs lignes, à condition de précéder chaque ligne additionnelle avec, au moins, un espace (SP) ou une tabulation horizontale (HT).

L'ordre relatif des en-têtes avec des noms différents n'est pas significatif. Par contre, l'ordre des en-têtes avec le même nom est important et doit être respecté. Un cas notable est celui

des en-têtes ***Via***, qui sont utilisés par les entités SIP intermédiaires afin d'acheminer les réponses.

À la différence des noms des méthodes, les noms des en-têtes peuvent être écrits en majuscules ou en minuscules. Dans le but de réduire la taille des messages, plusieurs possèdent une représentation alternative abrégée qui consiste en une seule lettre.

Le RFC 3261 établit que tout message SIP valide doit contenir, au minimum, les en-têtes ***To***, ***From***, ***CSeq***, ***Call-ID***, ***Max-Forwards*** et ***Via***. D'autres en-têtes importants que se retrouvent souvent dans les messages SIP sont ***Contact***, ***Route***, ***Record-Route***, ***Require et Allows***.

3.7 Transactions

SIP est un protocole transactionnel, où toutes les interactions entre des composants sont constituées d'échanges de messages indépendants. On appelle ***transaction*** à un cycle complet comprenant une requête et toutes ses réponses, qui peuvent inclure aucune ou plusieurs réponses provisoires et une seule réponse finale.

La transaction qui correspond à la méthode INVITE est spéciale : elle inclut un ACK lorsque la réponse du serveur est négative (3xx - 6xx). Dans le cas de réponses positives 2xx (*success responses*), l' ACK qui suit la transaction INVITE constitue une requête indépendante.

Chaque transaction a un côté serveur et un côté client, ce dernier étant celui qui transmet la requête, tandis que le premier est celui qui la reçoit et qui envoie la réponse. Le rôle de client ou de serveur est redéfini à chaque transaction : un UA peut fonctionner comme client durant une transaction et devenir le serveur pour la suivante. Les serveurs mandataires jouent les deux rôles en même temps tel que montré dans la figure 3.5.

La RFC 3261 spécifie le comportement pendant les transactions à l'aide de machines à états finis (MEF). La recommandation définit un total de quatre MEFs, chacune d'elles décrivant une des quatre possibilités : INVITE-client, INVITE-serveur, NON-INVITE-client et NON-INVITE-serveur. Par ailleurs, la RFC 3261 définit quatre ensembles de temporisateurs, cor-

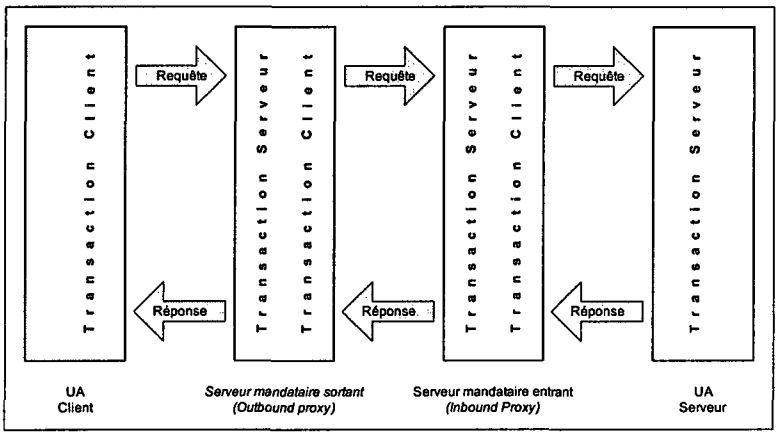

Figure 3.5: Relations transactionnelles client-serveur

respondant à chaque MEF. Une analyse détaillée des MEFs et du système de temporisation associé, est reportée au chapitre 7, où on décrit la mise en œuvre de SIP.

La RFC 3261 définit deux catégories de serveurs mandataires, ceux qui implémentent des MEFs (*stateful proxies*) et ceux qui ne le font pas (*stateless proxies*). Dans le premier cas, ils créent une transaction chaque fois qu'ils reçoivent une nouvelle requête. Ainsi, il est important que toutes les réponses à cette requête passent par lui dans le but de faire évoluer sa propre MEF. À cet effet, la trace du parcours du message par le réseau est enregistrée en empilant des en-têtes ***Via***. Ceux-ci sont copiés dans la réponse, et durant le chemin de retour, ces en-têtes sont enlevés par le même serveur qui les avait ajoutés. Ce fonctionnement permet de comprendre combien il est important de respecter l'ordre des en-têtes ***Via***.

Selon la RFC 3261, dans chaque entité SIP, les transactions sont identifiées par le paramètre «***Branch***» de l'en-tête ***Via*** la plus récente (le premier selon son ordre d'apparition). De cette manière, chaque entité utilise ce paramètre afin de trouver la structure de contrôle de la MEF qui gère la transaction.

3.8 Dialogues

SIP définit un dialogue comme une relation poste-à-poste (*peer-to-peer*) entre deux agents utilisateurs, qui dure un certain temps. Les dialogues donnent un cadre de référence pour l'échange de messages SIP, en regroupant des paramètres importants pour la génération

et la validation de requêtes et de réponses. Parmi ces paramètres, on trouve les numéros de séquences des commandes locales et distantes, les URIs de chaque UA intervenant, les paramètres d'identification du dialogue (*tags* et *Call IDs*), l'information de contact qui signale où envoyer les requêtes suivantes, les méthodes acceptées par l'UA distant et la route à suivre par les nouvelles requêtes. L'ensemble de cette information constitue «l'état du dialogue».

Dans chaque UA, un dialogue est identifié de manière univoque par trois éléments : la valeur ***Call-ID***, une étiquette locale (***local tag***) et une étiquette distante (***remote tag***). Ces trois éléments sont inclus dans tous les messages échangés dans le contexte d'un dialogue.

Dans le cadre de la RFC 3261, un dialogue est établi par une réponse positive (**200 Ok**), contenant une étiquette To «*To tag*», à une requête INVITE. C'est à partir de cet échange initial de messages que les UAs bâtissent l'état d'un dialogue. Cet état doit être maintenu jusqu'à la fin de la session comme conséquence de la réception d'une requête BYE. Au cours d'une session, la modification de l'état d'un dialogue n'est possible qu'à partir de la réception d'une nouvelle requête INVITE.

3.9 Conclusion

La problématique de la signalisation dans la téléphonie est complexe ce qui entraîne la complexité du protocole SIP. Pour cette raison, on a choisi d'introduire le protocole à partir d'un exemple. Puis, on a parcouru les facettes les plus importantes de ce protocole. Dans la première partie, on a analysé les composantes statiques, telles que la structure des messages et leur grammaire. La deuxième partie a permis d'introduire deux éléments incontournables pour le comportement dynamique du protocole : les transactions et les dialogues. De nombreux détails additionnels ont été reportés au chapitre 7, qui décrit la mise en œuvre de SIP.

CHAPITRE 4

LE PROTOCOLE RTP

Le transport en temps réel de flots de données sur des réseaux de commutation par paquets, tels qu'Internet, est un défi d'envergure. En effet, les réseaux IP n'ont été originalement conçus que pour la transmission de données indépendantes. Cependant, la popularisation des réseaux d'accès à haut débit, l'amélioration de la capacité et la robustesse de ces réseaux et, surtout, le développement de codecs très efficaces ont rendu possible la transmission des signaux en temps réel, comme la parole, sur des réseaux IP avec un niveau de qualité très acceptable. Le protocole RTP a été conçu spécifiquement à ce propos. Il est utilisé dans la majorité des applications actuelles de téléphonie sur Internet. Ce chapitre est une introduction aux protocoles RTP et RTCP. Il présente leurs caractéristiques principales, lesquelles seront à la base de la description de leur mise en œuvre, décrite dans le chapitre 8. La première partie du chapitre reprend la problématique de la transmission des flots en temps réel, déjà introduit dans le chapitre 2, afin d'établir les besoins requis par un protocole de transport en temps réel. C'est une introduction à la paire de protocoles RTP-RTCP, analysée ensuite dans la deuxième partie.

4.1 Le transport en temps réel des données

Les réseaux de commutation de paquets, tels qu'Internet, ont été conçus pour mieux servir les besoins en transmission des données. L'information à transmettre est divisée en blocs de taille appropriée et envoyée dans des paquets. Chaque paquet possède, dans son en-tête, toute l'information requise pour son identification et son acheminement ce qui permet au réseau de les considérer comme des entités individuelles. Chaque paquet est, alors, routé indépendamment. Au point de destination, ils sont mis en ordre puis validés et, finalement, dans le cas où il y a des erreurs ou il y a un paquet manquant, le récepteur peut demander sa retransmission. Cette solution est très appropriée pour un type de trafic qui est peu affecté par les délais de transmission et presque insensible à la gigue. Une autre caractéristique de ce type de transmission de donnée est que, en général, leur rythme n'est pas soutenu dans le temps et leur transmission se fait, plutôt, en rafales.

Par contre, ce schéma, bien adapté pour la transmission de données, n'est pas aussi approprié pour la transmission de flots d'information en temps réel. La principale raison en est que le temps disponible pour faire la codification, la transmission et le décodage de l'information, appelée délai de transit (*latency*), est très limité. Parmi les conséquences immédiates de cette limitation, on trouve que le fait de demander la retransmission d'un paquet perdu ou avec des erreurs n'a pas de sens, car la retransmission arrivera trop tard pour être utile. Il s'ensuit que l'utilisation du protocole TCP pour le transport de données en temps réel n'est pas recommandée, car les avantages qu'il donne sont contrebalancés par les contraintes de temps. UDP est donc, un protocole plus adapté à ces besoins.

Contrairement au trafic de données, les trafics en temps réel, tels que la transmission de la parole, sont plutôt de nature continue, soit soutenue, et de haut volume. Le livre de Colin Perkins (Perkins, 2006) présente des études très intéressantes sur le comportement des réseaux IP. Par exemple, il inclut, parmi d'autres, des informations sur la perte de paquets moyenne, les patrons de perte, la duplication et la corruption de paquets, et le délai de transit. Ces études montrent aussi les variations de ces paramètres pendant une période de temps donnée. Les conclusions tirées de l'analyse de ces données sont très utiles lorsqu'il est nécessaire de modéliser certaines situations auxquelles on doit faire face pendant la mise en œuvre des protocoles de transport en temps réel.

4.2 Les exigences de base pour un protocole de transport en temps réel

À partir des éléments mentionnés précédemment, on peut établir quels sont les besoins de base que doit satisfaire un protocole de transport de flots de données en temps réel :

- Il faut ajouter le moins de surcharge possible à la charge utile de chaque paquet : le volume de trafic est, généralement, assez élevé et le nombre des paquets transmis est considérable. Ainsi, afin de réduire le débit global, la surcharge associée à chaque paquet doit rester minimale.
- Étant donné que le délai du transit des paquets sur le réseau (*latency*) et sa variation, la gigue, sont des facteurs critiques, il faut incorporer des informations qui permettent de les mesurer de façon relative, afin de prendre les mesures nécessaires pour limiter leur impact sur la performance du système.

- Avec l'utilisation d'UDP comme protocole de transport, certaines conditions ne sont pas garanties, notamment l'arrivée des paquets et leur ordre relatif. Ainsi, il faut contrôler la séquence des paquets en utilisant un mécanisme de numérotation approprié. Cette précaution permet de détecter les paquets perdus et ceux qui arrivent en désordre.
- Afin de faciliter le traitement des données, il est convenable d'inclure la spécification du codec utilisé pour le traitement de l'information contenue dans la charge utile. Cette caractéristique permettra de changer le codec dynamiquement si nécessaire sans avoir besoin d'établir une négociation.

4.3 Les protocoles RTP et RTCP

RTP est un protocole de transport spécifiquement conçu pour faciliter la transmission poste-à-poste, en temps réel, des flots multimédia (audio ou vidéo) sur des réseaux IP tel que l'Internet. La première version de la spécification du protocole RTP est la RFC 1889 de l'IETF (Schulzrinne et al., 1996), publiée en 1996. La spécification actuelle correspond au standard RFC 3550 de juillet 2006 (Schulzrinne et al., 2003).

Un autre protocole, le RTCP, est utilisé conjointement à RTP. Il permet aux terminaux participants d'une session d'échanger des informations supplémentaires afin de permettre le démarrage des tâches de gestion. RTCP est aussi spécifié par la recommandation RFC 3550 (Schulzrinne et al., 2003). La paire de protocoles RTP-RTCP est située dans la couche application, et utilise UDP comme protocole de transport.

Les fonctionnements de RTP et de RTCP sont complémentaires : RTP ne transporte que les données des utilisateurs tandis que les paquets RTCP ne contiennent que des informations de supervision. La proximité entre ces deux protocoles est aussi reflétée par l'assignation de ports, qui sont toujours consécutifs, RTP étant toujours assigné à un port pair et RTCP au port impair immédiatement supérieur.

La RFC 3550 a été étendue par d'autres recommandations qui spécifient les profils pour différents types de flots. La RFC 3551 (*RTP Profile for Audio and Video Conferences*) (Schulzrinne and Casper, 2003), la RFC 3952 (*RTP Payload Format for internet Low Bit Rate Codec (iLBC) Speech*) (Duric and Andersen, 2004) et la RFC 4585 (*Extended RTP Profile for RTCP-based Feedback (RTP/AVPF)*) (Ott et al., 2006) sont parmi eux.

4.4 Structure du protocole RTP

La structure de base des paquets établie par RTP et montrée dans la figure 4.1, est relativement simple.

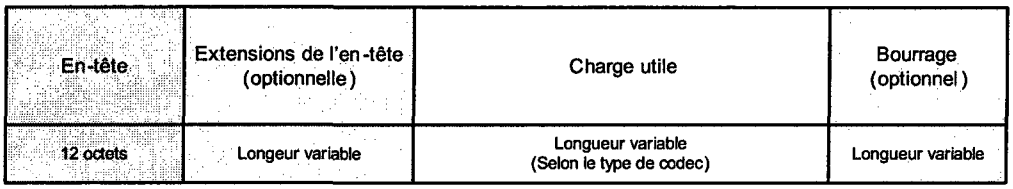

Figure 4.1: Structure d'un paquet RTP

En-tête : il a une longueur fixe de 12 octets. Il n'inclut que l'information essentielle requise par des applications en temps réel. Sa composition sera analysée en détail plus loin.

Extensions de l'en-tête : c'est un champ optionnel qui permet d'inclure des informations additionnelles requises par certains types de charge utile. L'identification des sources contributrices à un mélangeur, lorsqu'utilisée, doit être incluse dans ce champ.

Charge utile (*payload*) : ce champ transporte l'information audio ou vidéo numérisée et compressée. Sa nature dépendra du type de codec spécifié dans l'en-tête.

Bourrage (*padding*) : ensemble optionnel d'octets inclus à la fin du paquet. Le dernier octet du champ indique sa longueur en incluant lui-même. Ce champ, rarement utilisé, est plutôt employé lorsque l'information est encryptée.

4.5 En-tête d'un paquet RTP

L'analyse de l'en-tête d'un paquet RTP, comme illustré dans la figure 4.2, montre que ce protocole satisfait toutes les conditions estimées comme essentielles pour un protocole de transport en temps réel

La description de chaque champ est la suivante :

V : version du protocole RTCP employée (2 bits). La version actuelle et utilisée dans ce travail est la 2.

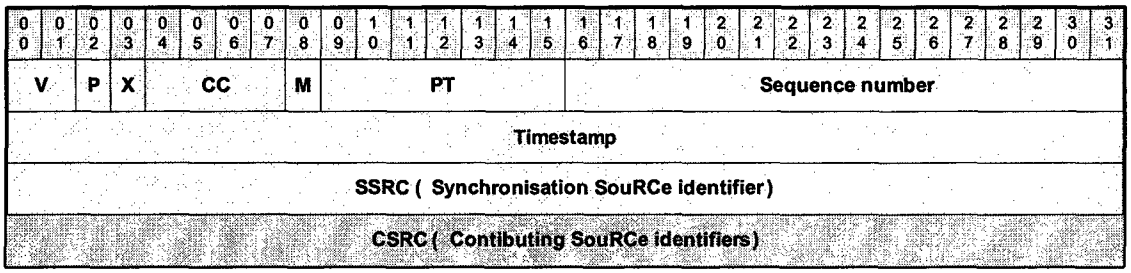

Figure 4.2: En-tête des paquets RTP (Schulzrinne et al., 2003)

P : (*padding flag*), ce bit indique la présence d'un champ de bourrage à la fin du paquet. La longueur du champ de bourrage est spécifiée par son dernier octet.

X : (*extension flag*), il indique l'inclusion d'une extension de l'en-tête.

CC : nombre de CSRC (*Contribution SouRCes*) qui suivent l'en-tête (4 bits). Ce champ n'est employé que par des mélangeurs (mixers) qui combinent différents flots RTP en un flot unique.

M : marqueur, son interprétation dépend du type d'application. Par exemple, dans le cas de transmission d'un signal vidéo, il indique le commencement d'une nouvelle trame.

PT : type de charge utile (*payload type*) (7 bits). Ce champ spécifie le type de charge utile, c'est-à-dire le codec employé.

Sequence number : nombre de 16 bits qui permet d'établir l'ordre des paquets, afin de les réordonner et de détecter ceux qui sont manquants.

Timestamp : champ de 32 bits qui reflète, de manière relative, l'instant d'échantillonnage du premier échantillon encodé de la charge utile. Il sert à calculer les délais de transit ainsi que la gigue.

SSRC : champ de 32 bits qui identifie de manière unique la source, sa valeur est choisie de façon aléatoire par l'application au moment de l'établissement de la session.

CSRC : champ de 32 bits qui permet d'identifier chaque contributeur à un flot RTP combiné. Il peut y avoir jusqu'à 15 instances de ce champ, le nombre de contributeurs étant spécifié par le champ CC.

Le dernier champ, CSRC, n'est ajouté que par des mélangeurs. En l'excluant, ce qu'il reste est la partie fixe de l'en-tête avec une longueur de 12 octets.

4.6 Le protocole RTCP

RTCP est un protocole complémentaire à RTP qui permet l'échange d'information de supervision et de contrôle. Parmi les informations transportées par RTCP nous trouvons le nombre de paquets transmis et reçus, le nombre de paquets manquants, le délai de transit et la gigue. La recommandation définit six différents types de paquets en incluant trois variantes de rapports :

SR : *Sender report*
RR : *Receiver report*
SDES : *Source description*
BYE : *Bye* (fin de la session RTP)
APP : *Application specific*
XR : *Extended report*

Ces types de paquets RTCP ne sont pas transportés individuellement, ils sont plutôt regroupés, et l'ensemble est transmis dans un seul paquet UDP. Ce regroupement est appelé «*compound RTCP packet*» et il est transmis de façon périodique. Les intervalles de transmission sont déterminés d'accord aux règles données dans le chapitre 6 du RFC 3550 (Schulzrinne et al., 2003). Selon ces règles, les rapports seront moins fréquents lorsque le nombre de participants à la session augmente afin de limiter la surcharge causée par l'activité de RTCP dans des valeurs acceptables. Une des raisons de cette mise à l'échelle du trafic RTCP est que, dans une conférence, le trafic avec de l'information audio reste limité, même si le nombre de participants s'accroît : ils ne peuvent pas parler tous en même temps. Par contre, dans le cas de RTCP le trafic peut augmenter de manière considérable, car chaque participant doit envoyer périodiquement des rapports RTCP à tous les autres.

La figure 4.3 montre le format général d'un paquet RTCP. Il a un en-tête commun à tous les types de paquets mentionnés auparavant, suivi d'une partie qui sera spécifique pour chacun d'eux. La description de chaque champ est la suivante :

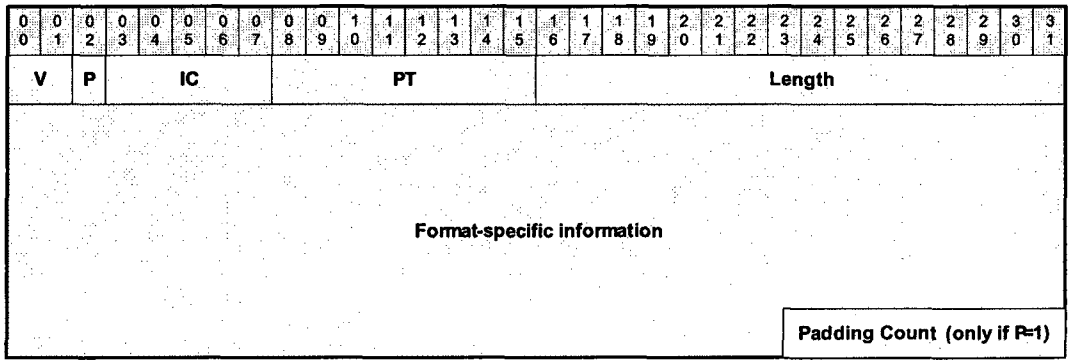

Figure 4.3: Format général d'un paquet RTCP (Schulzrinne et al., 2003)

V (*Version number*) : version du protocole RTCP employé (2 bits). La version actuelle est la 2.0.

P (*Padding flag*) :, ce bit indique la présence d'un champ de bourrage à la fin du paquet. La longueur du champ de bourrage est spécifiée par son dernier octet.

IC (*Item count*) : Le nom et le contenu de ce champ varient selon le type de paquet. En général, il indique le nombre d'items contenus dans le paquet.

PT (*Packet type*) : Identifie le type de paquet comme un des types mentionnés auparavant.

***Length* :** Taille du paquet, en excluant cet en-tête, exprimée en termes de mots de 32 bits.

L'étude de chaque type de paquet est au-delà de la portée de ce chapitre et peut être consultée directement dans la RFC 3550 (Schulzrinne et al., 2003) ou dans le livre de Collins (Perkins, 2006).

4.6.1 D'autres entités RTP

En plus des systèmes terminaux, la RFC 3550 (Schulzrinne et al., 2003) considère deux autres entités intermédiaires : les traducteurs (*translators*) et les mélangeurs (*mixers*). Les traducteurs transfèrent des paquets en laissant le champ SSRC (*Synchronisation Source Identifier*) intact. Par ailleurs, les mélangeurs combinent plusieurs flots RTP en un flot unique.

4.6.2 Les profils de RTP pour la transmission d'audio et de vidéo

Les particularités de l'utilisation de RTP avec différents codecs audio et vidéo sont données par la RFC 3551 (Schulzrinne and Casper, 2003). Les tables de l'annexe A donnent la liste des codecs supportés et le code de la charge utile correspondante (PT : *payload type*).

La spécification de la charge utile supporte des codifications orientées aux échantillons et d'autres orientées aux trames. Pour les premières, la longueur du champ de charge utile est arbitraire. Pour les deuxièmes, la longueur de ce champ doit être un multiple de la longueur de la trame correspondant à chaque codec. Le standard n'utilise pas de variable afin d'indiquer le nombre de trames contenues. Ainsi, cette détermination ne peut être faite que si le décodeur connaît ou peut déduire la longueur des trames contenues dans la charge utile. Évidemment, cette dernière ne peut contenir que des trames avec un même type de codification. Dans le cas de G.729 (ITU-T, 2007) par exemple, la RFC 3551 (Schulzrinne and Casper, 2003) établit que la longueur de la trame est de 10 octets et que, par défaut, les paquets contenant deux trames sont transmis à toutes les 20ms.

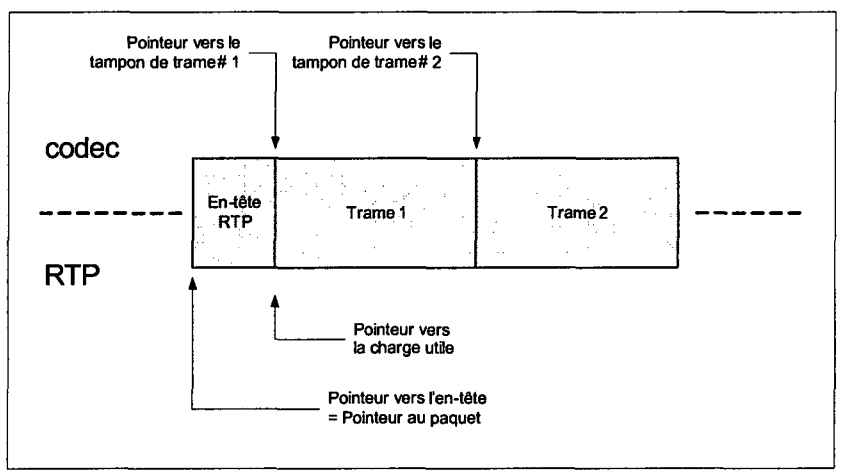

Figure 4.4: Paquétisation des trames avec RTP

La figure 4.4 montre la façon dont les trames sont organisées dans un paquet RTP pendant le processus de paquétisation, en considérant les points de vue de RTP et du codec. Ainsi, les trames contenues dans le paquet partagent un même en-tête RTP. Une première conséquence est que toutes les trames doivent être compressées avec le même codec. Par ailleurs, le *timestamp* correspond à la prise du premier échantillon de la première trame.

Le principal avantage de la paquétisation est une augmentation de l'efficacité dérivée d'une réduction de la surcharge relative. Cependant, la limitation du nombre de trames qui peuvent être regroupées dans un même paquet RTP est plutôt d'ordre pratique. En effet, la contribution au délai de transit faite par le délai de paquétisation grandira linéairement avec le nombre de trames. Un deuxième problème est que, dans le cas des paquets manquants, les trous de réception seront plus notoires et difficiles à masquer.

4.7 Conclusion

C'est à partir de l'analyse et de la compréhension du comportement des réseaux IP avec des flots de données temps réel que nous avons établi les besoins les plus importants pour un protocole de transport dans le but d'une telle utilisation. Puis, nous avons introduit le protocole RTP, en vérifiant s'il satisfait ces besoins.

Dans l'analyse, nous n'avons considéré que les éléments les plus représentatifs de la paire de protocoles RTP / RTCP, afin d'établir une base pour la meilleure compréhension de sa mise en œuvre, décrite dans le chapitre 8. Pour une étude plus en profondeur, le lecteur doit se remettre à la recommandation RFC 3550 (Schulzrinne et al., 2003) et au livre de Colin Perkins (Perkins, 2006).

CHAPITRE 5

LES SYSTÈMES EMBARQUÉS ET LES SYSTÈMES TEMPS RÉEL

Actuellement, les systèmes embarqués sont de plus en plus omniprésents dans tous les aspects de la technologie. On peut les trouver dans une vaste gamme d'appareils, en allant de simples montres jusqu'à des équipements militaires hautement sophistiqués. Parmi les exemples les plus notables se trouvent les téléphones cellulaires. En tant que points terminaux de complexes réseaux de communication, ils combinent dans une même unité, un ensemble de modules très représentatifs de l'état de l'art dans plusieurs domaines de la technique. Avec leur prolifération, la conception des systèmes embarqués est devenue une spécialité caractérisée par l'exigence de respecter des conditions de design très rigides.

Lors de l'analyse de ce projet, il est important de garder à l'esprit son orientation vers les systèmes embarqués. Ainsi, un jugement de la problématique de ce point de vue s'avère essentiel, dans le but de mieux comprendre les décisions prises et l'effort investi pendant la conception. L'objectif de ce chapitre est d'identifier les contraintes et les conditions particulières de la conception. Dans la première partie, on identifie les types de systèmes embarqués ciblés par ce projet afin d'établir, dans la deuxième partie, les points les plus importants à considérer lors de sa mise en œuvre.

5.1 Systèmes embarqués ciblés par ce projet

De nos jours, le concept de **système embarqué** englobe une très grande variété de systèmes, applicables à une également vaste diversité de situations. En conséquence, il est difficile de donner une définition générale. Ainsi, il est préférable de se focaliser sur le sous-ensemble des systèmes visés par ce projet. Dans ces termes, la définition suivante, prise du livre de Quing Li (Li, 2003), est tout à fait applicable :

«Un système embarqué est un système matériel-logiciel (un ordinateur), où le matériel (*hardware*) et le logiciel (*software*) sont fortement intégrés et conçus pour effectuer une

fonction spécifique. D'habitude, les systèmes embarqués sont complètement contenus ou encapsulés dans le système qu'ils contrôlent. Plusieurs systèmes embarqués peuvent coexister dans une même application.»

La considération de certaines conditions complémentaires permet de limiter la portée de cette définition sur les types de systèmes envisageables par les piles de protocoles à développer, lesquels doivent inclure les caractéristiques suivantes :

- Avoir de la connectivité aux réseaux IP.
- Être basés sur des microcontrôleurs, typiquement de 32 bits, avec une puissance de traitement suffisante pour supporter, en plus des applications qui utilisent les piles SIP et RTP, les exigences du traitement audio et d'une pile TCP/IP.
- Inclure un RTOS *(Real-Time Operating System)* multitâche.
- Pour les applications de téléphonie : inclure un sous-système de capture et de restitution audio.
- Pour les applications de contrôle : avoir une capacité d'entrée-sortie compatible avec l'application spécifique.

5.2 Systèmes temps réel

Selon une définition prise des notes de cours du Professeur Philippe Mabilleau (Mabilleau, 2001), de l'Université de Sherbrooke, «on dit qu'un système est temps réel, lorsqu'il interagit avec un environnement externe changeant avec le temps. Il effectue des actions à partir de stimulus externes, à une vitesse suffisamment rapide pour pouvoir exercer un certain contrôle sur cet environnement».

En fonction de la sévérité des contraintes temporelles, on classifie les systèmes embarqués temps réel dans deux catégories : les systèmes «*soft*» tolèrent des écarts dans le respect des règles temporelles avec une dégradation de la performance, mais sans conséquences graves. À l'inverse, pour les systèmes «*hard*», l'incapacité de rencontrer les contraintes temporelles peut entraîner des conséquences graves, voire la défaillance totale du système.

D'après cette classification, une pile SIP, destinée aux agents utilisateurs, appartient à la catégorie des systèmes *soft* temps réel. Par contre, une pile RTP peut être considérée comme plus proche des systèmes *hard* temps réel.

5.2.1 Systèmes d'exploitation temps réel

Les systèmes d'exploitation temps réel, ou RTOS, sont conçus pour opérer dans des environnements avec des contraintes temporelles, où la mémoire et la puissance de calcul sont souvent limitées. Ils doivent fournir leurs services dans des délais très limités et prédictibles. Il s'agit de systèmes multitâches, qui donnent, au minimum, des services de communication et de synchronisation des tâches, de temporisation et d'allocation dynamique de la mémoire.

La caractéristique la plus importante qui différencie les RTOS des systèmes d'exploitation généraux est leur comportement déterministe en fonction du temps. Autrement dit, les RTOS optimisent et rendent prédictible l'utilisation du temps. En même temps, ils réduisent au minimum la latence pendant les interruptions, en limitant l'utilisation des sections critiques, où les interruptions sont désactivées.

Par ailleurs, afin de réduire la taille finale du code compilé, il est possible en général, de configurer les RTOS en incorporant uniquement les services requis pour l'application supportée.

5.3 Conception de logiciels embarqués

Un concept qui découle directement de la définition d'un système embarqué est celui de la pénurie de ressources. En effet, étant donné que les systèmes embarqués sont conçus pour des applications spécifiques, ils sont équipés avec les ressources minimales compatibles avec leur finalité et avec l'environnement d'application. Les limitations de la puissance disponible, de la taille et du poids du produit final sont parmi les contraintes les plus courantes.

Un grand nombre de systèmes embarqués, une fois rendus à la phase de production, ne seront ni modifiés ni mis à jour pendant tout leur cycle de vie. Ainsi, l'incorporation d'éléments non indispensables, en plus d'être une source potentielle de problèmes, représente un gaspillage d'argent. Toutefois, il y a certaines exceptions. En effet, il peut arriver que, pour des raisons stratégiques telles qu'une réduction des coûts globale dans une ligne de produits, la plateforme matérielle choisie pour une application soit surdimensionnée et partagée entre différentes lignes de produits. Une autre situation, de plus en plus fréquente, est celle des systèmes embarqués réseautés susceptibles d'être mis à jour à distance. D'habitude,

ces systèmes possèdent une certaine marge dans l'attribution des ressources en prévision des modifications futures.

À partir de ce qu'on vient d'exposer, il s'ensuit que la conception des logiciels pour des systèmes embarqués doit toujours être faite dans l'hypothèse de ressources limitées.

5.3.1 Critères généraux d'optimisation de logiciels embarqués

Comme conséquence de l'importance grandissante des systèmes embarqués, une recherche active est menée dans l'objectif de développer des techniques d'optimisation pour la conception. Une partie importante de cet effort est mise sur le développement des compilateurs plus efficaces du point de vue de la taille du code. Un autre sujet qui attire l'attention est l'étude d'architectures logicielles permettant de réduire la consommation d'énergie.

Cependant, indépendamment du compilateur employé, l'utilisation de pratiques de codage appropriées est une condition fondamentale pour arriver à de bons résultats. Certaines de ces techniques ont un caractère général tandis que d'autres sont plus ou moins dépendantes de la plateforme matérielle. À titre de référence, le livre «ARM. System Developer's Guide» (Sloss et al., 2004) dédie deux chapitres à l'analyse de méthodes pour la programmation efficace en langages 'C' et assembleur pour la plateforme ARM.

Certains critères qui ont été utilisés en permanence pendant le projet pour améliorer la performance sont :

- Réduction au maximum des déplacements de données dans la mémoire.
- Utilisation d'un nombre limité de paramètres lors des appels de fonctions.
- Utilisation d'arithmétique entière autant que possible.
- Simplification des opérations arithmétiques : Pour les multiplications et pour les divisions, on utilise l'approximation consistant à remplacer un des facteurs à la puissance de deux la plus proche. Ceci permet d'utiliser d'opérations simples de décalage (*shifts*).
- Utilisation, de façon intensive, de tables pour l'accès aux données et pour la sélection de fonctions.

5.3.2 Coûts et limites de l'optimisation

L'application des critères d'optimisation mentionnés dans la section précédente implique la mise en jeu d'efforts, mesurables en temps et en argent, qui peuvent être considérables. Ainsi, il s'impose de pondérer le coût de cet effort en fonction du type de système visé afin d'établir les limites. Étant donné que le coût et le temps de développement sont de plus en plus critiques à l'heure de concevoir un système embarqué et dû à la grande diversité d'applications, le compromis entre l'optimisation et l'incrément des coûts dépendra de chaque situation. En général, on peut dire que la conception des librairies de base et de piles de protocoles, comme dans le cas de ce projet-ci, justifie de réaliser des efforts additionnels d'optimisation, car ils constituent la base sur laquelle s'appuieront les applications finales plus complexes.

Par contre, une situation complètement opposée se présente pour des applications destinées aux ordinateurs de propos général, par exemple, les PC, où la disponibilité de ressources est bien supérieure. Dans ces situations-ci, certaines optimisations, qui sont décisives lorsqu'il s'agit de systèmes embarqués, deviennent presque insignifiantes. Il s'ensuit que l'effort investi n'est pas justifié.

5.4 Gestion de la mémoire dynamique

Le problème de la gestion de la mémoire dynamique a une dimension très importante lors de la conception de systèmes embarqués. C'est pour cela qu'elle mérite une attention particulière.

Les problèmes plus importants qu'on peut identifier en relation au sujet sont la fragmentation du tas de mémoire dynamique (*heap*), le temps d'allocation des blocs de mémoire et la fuite de mémoire (*memory leak*). Les deux premiers sont dépendants des algorithmes utilisés et souvent, les RTOS incluent des fonctions qui utilisent des techniques spéciales pour l'allocation de mémoire. Le troisième problème est plutôt une conséquence d'une programmation défectueuse. Les raisons qui font que ces points soient si importants sont, en premier lieu que la mémoire RAM *(Random Access Memory)* disponible est toujours limitée et en deuxième lieu, que les systèmes embarqués peuvent fonctionner pendant de longues périodes de manière autonome.

Dans le chapitre 9, «Modules Complementaires», on reviendra sur ce sujet et sur la solution adoptée pour la résolution de la présente problématique.

5.5 Sip et RTP embarqués

La plupart des distributions de SIP et RTP disponibles aujourd'hui sont destinées aux applications de VoIP, et elles sont utilisées dans des systèmes embarqués : des téléphones mobiles, des passerelles, des serveurs mandataires, etc . Cependant, par différentes raisons, elles n'ont pas été conçues comme des applications embarquées sinon qu'il s'agit plutôt de logiciels destinés à être exécutés dans des environnements avec des systèmes d'exploitation génériques. De plus, il s'agit souvent de distributions qui doivent supporter plusieurs systèmes d'exploitation. Le fait d'utiliser un système d'exploitation générique comme plateforme de développement impose des limitations, parfois importantes, à telles implémentations. Par exemple, d'habitude les applications destinées à être utilisées avec des systèmes d'exploitation génériques sont conçues utilisant un nombre plutôt limité de fils d'exécution (*threads*). Par contre, lorsqu'un RTOS est utilisé, le nombre de fils d'exécution est significativement plus grand, ce qui permet d'optimiser au maximum l'utilisation les ressources et de tirer profit des caractéristiques «temps réels» du système d'exploitation.

Parmi les distributions avec source code libre (*open source*), il faut remarquer PJSIP (). Cette distribution, faite en langage 'C', a une empreinte petite (environ 130 Ko) et une bonne performance. PJSIP est très susceptible d'être embarqué dû à sa petite taille et il a été conçue originalement sur Linux. Néanmoins, les limitations mentionnées dans le paragraphe précédent lui sont applicables.

Étant RTP un protocole de transport de flot de données en temps réel, ses implémentations sont aussi affectées lorsqu'il est utilisé avec des systèmes d'exploitation génériques, tels que Linux ou Windows, qui ne sont pas des RTOS. Des études ont été faites afin de contourner ces limitations, tel que le décrit par l'article «*A Kernel-Level RTP for Efficient Support of Multimedia Service on Embedded Systems*» (Sun and Kim, 2005), où les auteurs proposent l'implémentation du RTP à niveau du noyau (*kernel*) avec le but d'augmenter leur performance.

Par rapport aux implémentations embarquées du protocole SIP, le projet HomeSIP (ENSEIRB, s.d.) se trouve parmi ceux qui attirent plus d'attention. Il est décrit dans l'article

«Mise en œuvre d'une pile SIP sur système embarqué pour le contrôle de capteurs» (Kadionik, 2006; Abid et al., 2006). Cet intéressant projet, né en 2006, continue aujourd'hui son évolution vers un contexte plus général M2M (*Machine To Machine*).

5.6 Conclusion

Ce bref chapitre sert à introduire les systèmes temps réel et les systèmes embarqués dans le contexte de ce projet afin d'identifier, de façon générale, les contraintes que ces deux caractéristiques imposent à la conception de la pile de protocoles.

DEUXIÈME PARTIE

MISE EN ŒUVRE

CHAPITRE 6

MISE EN ŒUVRE, INTRODUCTION

Ce chapitre, qu'introduit la mise en œuvre, décrit sommairement différents aspects de la planification du projet. On mentionne, premièrement, les objectifs généraux, la portée et les limitations du projet. À continuation, on explique les conditions imposées à la conception de la pile de protocoles en prenant en considération les exigences pour les protocoles SIP et RTP ainsi que pour les systèmes embarqués. Par la suite, cette introduction présente les outils de développement utilisés et d'autres implantations de SIP, disponibles sur Internet et qui ont servi de base à cette nouvelle conception.

6.1 Objectifs généraux et portée du projet

L'objectif de ce projet de maîtrise, tel que mentionné dans le chapitre d'introduction à ce mémoire, est de mettre en œuvre des piles de protocoles SIP et RTP sur système embarqué, visant des applications simples de téléphonie sur IP et des aplications de contrôle réseautés. Cette définition englobe beaucoup d'information en soi-même. En premier lieu, les recommandations qui définissent les protocoles SIP et RTP déterminent une partie importante des spécifications fonctionnelles et détaillées du projet. En deuxième lieu, le fait qu'il s'agit des piles embarquées ajoute un ensemble des besoins «implicites» propres de la nature de ce type de systèmes. Finalement, la sorte d'applications ciblées aide à réduire le champ de solutions possibles. Les spécifications techniques du projet sont organisées en quatre parties : les spécifications propres du protocole SIP, celles du protocole RTP, certaines exigences particulières pour le projet et finalement, les conditions additionnelles qui découlent de la conception d'un logiciel embarqué.

6.2 Conditions de design

6.2.1 Conditions de design pour le protocole SIP

Du côté du protocole SIP, l'objectif du projet est de concevoir une pile destinée aux agents utilisateur (UA). Elle doit permettre, au minimum, l'enregistrement d'un UA sur un serveur *REGISTRAR* et l'établissement des sessions SIP dans un réseau local. Intégré avec RTP, l'ensemble devra servir à la mise en œuvre d'une application simple de VoIP. Le fait de ne cibler que des UA a un impact sur les exigences en ce qui concerne la vitesse d'exécution, exigences qui peuvent être considérablement plus souples que lorsqu'il s'agit d'un serveur mandataire. Ainsi, pour le protocole SIP, les solutions qui amènent vers un code plus compact sont prépondérantes par rapport à celles qui priorisent la vitesse.

De plus, la solution doit inclure, au minimum, les six méthodes fondamentales définies dans le RFC 3261 (Rosenberg et al., 2002) ainsi que les en-têtes de base : **To**, **From**, **CSeq**, **MaxForwards**, **Contact**, **Call-ID**, **Via**, **Route** et **Record-Route**. L'incorporation des méthodes et des en-têtes additionnels doit être possible en utilisant des procédures simples.

6.2.2 Conditions de design pour le protocole RTP

Du côté du protocole RTP, l'objectif est d'implémenter une pile permettant l'échange d'information audio, en format G.711, entre deux unités. La conception doit prévoir l'inclusion d'autres codecs avec des formats orienté-échantillons ou orienté-trames. La section de réception doit considérer l'utilisation des tampons de paquétisation pour la transmission et des tampons anti-gigue pour la réception. L'implantation du protocole RTCP n'est pas envisagée dans cette première version du projet, mais il faut prévoir son incorporation future en élaborant les statistiques nécessaires à son fonctionnement.

Dans le cas de RTP, la vitesse d'exécution devient importante étant donnée la nature «*hard real time*» du protocole. En conséquence, pour coder, il faut considérer que les critères d'optimisation à utiliser seront ceux qui amènent vers l'exécution la plus rapide.

6.2.3 Conditions de design découlant de l'application dans des systèmes embarqués

Simplicité, portabilité, extensibilité, fiabilité et robustesse sont des caractéristiques toujours désirables dans la conception d'un système. Cependant dans le cas du design des logiciels embarqués, elles deviennent fondamentales. Voici une brève analyse de chacune d'elles du point de vue de ce type de systèmes :

Simplicité. C'est une condition préalable si on prétend atteindre certains niveaux de robustesse et de fiabilité. Habituellement, la simplification est un processus itératif qui demande plusieurs révisions de la problématique.

Portabilité. Souvent, les logiciels destinés aux systèmes embarqués sont fortement dépendants des systèmes d'exploitation et du matériel. La conception des applications embarquées portables demande l'effort additionnel d'identifier et d'isoler les points de dépendance du système d'exploitation et du matériel, tout en conservant la simplicité de l'ensemble. La couche d'abstraction appelée HAL, concentre les fonctions et les macros pour isoler les dépendances du matériel. En même temps, une autre couche d'abstraction logicielle permet de s'abstraire du système d'exploitation.

Extensibilité. SIP est un protocole facilement extensible, basé sur des messages de texte, et qui a été conçu pour évoluer. Du côté de ses implémentations, le concept d'extensibilité prend en compte l'ensemble des prévisions faites afin d'accompagner cette évolution. Ainsi, la conception d'une architecture modulaire, avec des interfaces nettes et bien définies, s'impose afin de prévoir et de délimiter l'impact des modifications futures sur la pile.

Fiabilité et Robustesse. En tant que qualités clés pour les systèmes embarqués, il faut considérer, dès le départ, l'utilisation de techniques de programmation défensive. Ces attributs sont prioritaires, car fréquemment, ces types de systèmes fonctionnent de façon autonome et, en cas de défaillances, les possibilités d'interagir avec eux sont très restreintes ou nulles.

Empreinte minimale. Dans le contexte général d'économie de ressources qui régit la conception des systèmes embarqués, ceci constitue un élément de jugement pour établir la qualité de la conception par rapport à la complexité de l'application.

6.2.4 Conditions particuliers

- La conception doit s'appuyer sur le système d'exploitation MicroC/OS-II avec la pile MicroC/TCP-IP de Micrium. L'inclusion d'une couche d'abstraction logicielle permettra de minimiser le niveau de dépendance du système d'exploitation aux effets de sa portabilité
- Afin de simplifier la lisibilité du code, il doit être fait en langage 'C', en suivant les standards de codage définis par Micrium Inc.
- Dans le but d'assurer une utilisation efficace de la mémoire dynamique, une autre condition imposée est que la fonction **malloc()** ne doit pas être utilisée à cause des problèmes de fragmentation qu'elle occasionne. À sa place, il faut prévoir des fonctions pour l'allocation des blocs de mémoires. Cette problématique est expliquée plus en profondeur dans le chapitre 9.
- Dans le cas d'une pile SIP, il faut constamment allouer et prélever de nombreuses structures et chaînes de caractères, qui sont souvent de petite taille. Ceci donne comme conséquence une utilisation physique de la mémoire assez inefficace. Pour une utilisation optimale de la mémoire, l'ensemble doit inclure, en plus des fonctions d'allocation de blocs de mémoire, des fonctions spécialisées pour l'allocation de chaînes de caractères. Une explication détaillée de ce problème et des solutions implantées se trouve aussi, dans le chapitre 9.
- Avec le but de simplifier le débogage et la mise en marche des piles, la solution proposée doit inclure un système simple de traces. Il devra être capable d'identifier les situations suivantes : «*Fatal Error*», «*Error*», «*Warning*», «*Info*», «*Debug*» et «*Trace*».
- D'autres points importants à considérer sont : n'utiliser que de l'arithmétique entière et, autant que possible, calculer les multiplications et les divisions par approximations qui utilisent des puissances de 2. Cette estimation rend possible l'implémentation avec des opérations de décalage (*shift*), avec un incrément notable de la performance.

6.3 Outils de développement

6.3.1 Logiciels

La solution du projet a été réalisée à l'aide de la suite d'outils de GNU pour ARM, avec le compilateur gcc V3.3.1. Éventuellement, des compilateurs Visual C++ 6.0 et 8.0 de Microsoft ont été utilisés afin de coder et de déboguer certains morceaux de code, notamment, le *parser*.

Le contrôle de versions a été fait avec **SVN**. **Doxygen** a été utilisé pour générer la documentation à partir du code source.

6.3.2 Partie Matériel

La plateforme matérielle utilisée est un système de développement, basée sur des processeurs RISC de technologie ARM, et constituée par les éléments suivants :

- CerfPDA d'Intrinsyc, avec un microprocesseur strongARM SA1110 @ 192 Mhz, équipé de 32 Mo de mémoire Flash, 64 Mo de SDRAM, écran couleur LCD $\frac{1}{2}$ VGA de 3.8" avec écran tactile, clavier alphanumérique, interfaces 10Base-T, RS232 et capacité de traitement audio.
- Carte Zoom de LogicPD, avec un microprocesseur ARM7TDMI LH79520 @ 77 MHz. La carte est équipée de 32 Mo de DRAM, carte Compact Flash de 32 Mo, avec des interfaces 10/100 Base-T et RS232. Cette carte inclut un convertisseur numérique-analogique permettant la restitution audio, mais la capture audio n'est pas supportée.
- JTAG Raven de Mcraigos Systems.

6.3.3 Documentation

Pour l'édition de la documentation, on s'est servi de la suite Office de Microsoft, et en particulier, tous les diagrammes ont été faits avec Visio. Finalement, ce mémoire a été rédigé sur **LaTeX**, en utilisant **MiKTeX 2.4**. L'éditeur choisi a été *TEXnicCenter*.

6.4 Implantations de SIP disponibles sur Internet

Nombreuses implantations de SIP, des libraires et des piles sont actuellement disponibles sur Internet sous la modalité de code source libre (opens source), certaines d'elles sont très complètes et élaborées. Dans la liste suivante, on mentionne quelques unes qui ont été prises comme référence pendant le déroulement de ce projet. En particulier, le projet PJSIP et l'implantation de SIP d'Open Solaris ont été les plus consultés.

- VOCAL. (oo C++), URL=http ://www.vovida.org
- SipX. (oo C++), URL=http ://www.sipfoundry.org/sipx
- ReSIProcate (oo C++), URL=http ://www.resiprocate.org
- oSIP. (ANSI-C), URL=http ://www.gnu.org/software/osip
- Sofia-SIP (ANSI-C), URL=http ://sofia-sip.sourceforge.net
- PjSIP (ANSI-C), URL=http ://www.pjsip.org
- Open Solaris SIP Stack (ANSI-C), URL=http ://www.opensolaris.org

6.5 Conclusion

À titre d'introduction à la mise en œuvre des protocoles SIP et RTP, ce chapitre commence avec un résumé des objectifs et des limitations du projet. Il continue avec une énumération des conditions de design identifiées pour chacun des protocoles implantés et suivi par une description de l'environnement de développement.

CHAPITRE 7

MISE EN ŒUVRE DU PROTOCOLE SIP

Dans le chapitre 3, on a déjà mentionné que SIP est un protocole très extensible et flexible, qui donne une vaste marge de manœuvre aux concepteurs. L'effet combiné de ces facteurs explique l'étendue du spectre de solutions déjà disponible, allant de piles assez simples à des systèmes très complets et complexes. Le champ d'application, étant si vaste et attirant, justifie une telle diversité et la majorité des implémentations trouvent leur place.

Le type d'applications visées et les plateformes matérielles considérées déterminent le caractère de chaque implémentation. On pourrait ainsi regrouper les piles dans deux grandes catégories : celles du domaine général, et celles plus ou moins dédiées, plus ciblées à certaines applications particulières. Cette classification a une étroite relation avec les critères de design et le niveau d'optimisation, car il y a souvent une opposition entre généralisation et optimisation.

La pile conçue dans le cadre de ce projet vise des applications dans les systèmes d'extrémité (*end-point systems*) sous le rôle d'agent utilisateur (UA). Cette condition exclut spécifiquement les mandataires et autres entités intermédiaires, pour lesquelles le nombre de transactions exécutées par unité de temps est un des facteurs de performance. Étant donné que le nombre moyen de transactions attendues par un UA est très inférieur à celui d'un mandataire, la vitesse d'exécution n'est pas critique. En conséquence, les efforts ont été orientés vers l'amélioration d'autres caractéristiques, comme l'optimisation de la taille du code et de l'utilisation de la mémoire.

Par ailleurs, cette pile vise des applications dans des systèmes d'extrémité (*end-point systems*) sous le rôle d'agent utilisateur (UA). Cette condition exclut spécifiquement les mandataires et autres entités intermédiaires, pour lesquelles le nombre de transactions exécutées par unité de temps est un des facteurs de performance. Étant donné que le nombre moyen de transactions attendues par un UA est très inférieur à celui d'un mandataire, la vitesse d'exécution n'est pas critique. Dans le cadre de ce projet, les efforts ont été plutôt orientés vers l'amélioration d'autres caractéristiques, comme l'optimisation de la taille du code et de l'utilisation de la mémoire.

Ce chapitre décrit avec détail la conception de la pile SIP. La première partie donne une idée générale de l'architecture et de la dynamique de son opération. La deuxième rentre dans l'analyse en profondeur de chaque module. La troisième partie décrit la gestion d'événements qui permettent la communication entre modules, un élément qui a un rôle important dans l'implémentation. Une attention spéciale a été mise sur les justifications des décisions prises pendant la conception dans le but d'aider à mieux comprendre la philosophie de cette implémentation. Ces décisions pourront, bien entendu, être réévaluées à l'heure de faire une réingénierie du système.

7.1 Architecture générale

7.1.1 Description

La figure 7.1 présente un schéma de l'architecture de la pile qui montre les différents modules et leurs interrelations. Avec le but de rendre le schéma plus facile à interpréter, quelques liaisons secondaires ont été omises.

Les différents modules ont été regroupés, en utilisant un critère fonctionnel, dans quatre couches :

- **Couche de transport** : elle regroupe les fonctions de transmission et de réception de messages SIP sur le réseau.
- **Messages** :C'est le module qui définit la structure interne des messages et toutes les fonctions nécessaires pour leur traitement.
- **Couche de transactions** : Gère les transactions SIP.
- **Couche UA** : Inclut le TU (*Transaction User*), la définition de l'agent utilisateur et la gestion de dialogues SIP.

Voici une brève description des principaux modules :

- **Transport**. Module chargé de la transmission-réception de messages sur UDP/TCP. Dans cette implémentation, la couche de transport fait une totale abstraction du contenu des messages qui sont traités comme de simples blocs de texte.
- **Messages**. C'est le module qui définit la structure interne et toutes les fonctions de traitement des messages.

- **Parser**. Il fait la conversion des messages en format texte vers leur représentation interne. Il est appelé par le gestionnaire des messages immédiatement après l'arrivée de ceux-ci.
- **Impression de messages**. Ce module fait le travail complémentaire au parser, c'est à dire qu'il réalise la conversion de messages en format interne vers le format externe ou textuel.
- **Transactions**. Cet important module met en œuvre les machines à états finis définis par le protocole SIP. Ses fonctions principales sont : gérer la correspondance entre les requêtes et les réponses, gérer les retransmissions lorsqu'on utilise un transport non fiable tel que UDP et contrôler la temporisation des transactions (*timeout control*).
- **Gestionnaire de réception**. Au cœur de l'application, ce module est le point de destination de tous les messages SIP qui atteignent l'UA. Il est chargé de gérer le traitement des messages reçus et de les acheminer vers leur destination à l'intérieur de la pile. Les messages SIP sont envoyés vers d'autres modules, notamment «**Dialogues**» et «**Transactions**», en utilisant des messages d'événements.
- **TU (*Transaction User*)**. Ce module, qui prend le nom de la couche à laquelle il appartient, gère l'opération de la couche de transactions, la création de dialogues et la création des sessions.
- **Dialogues**. Module chargé de gérer les dialogues SIP. Appartenant à la couche *Transaction User*, il a été conçu comme un module indépendant.
- **SDP**. Module gestionnaire du protocole SDP. Il ne fait pas partie de la pile, mais il accompagne presque toujours SIP en permettant la négociation des paramètres d'une session. Ce module n'est pas implémenté dans le présent projet et il devra être incorporé dans une prochaine révision.
- **UA**. Couche intermédiaire qui contient le profil de l'utilisateur local et l'ensemble de fonctions de rappel (*callbacks*) enregistrées par l'application et qui font partie de l'API.

7.1.2 Séquence d'initialisation de la pile

La procédure d'initialisation de la pile est plutôt simple. La première étape consiste à initialiser les variables et structures en appelant les fonctions d'initialisation de l'API.

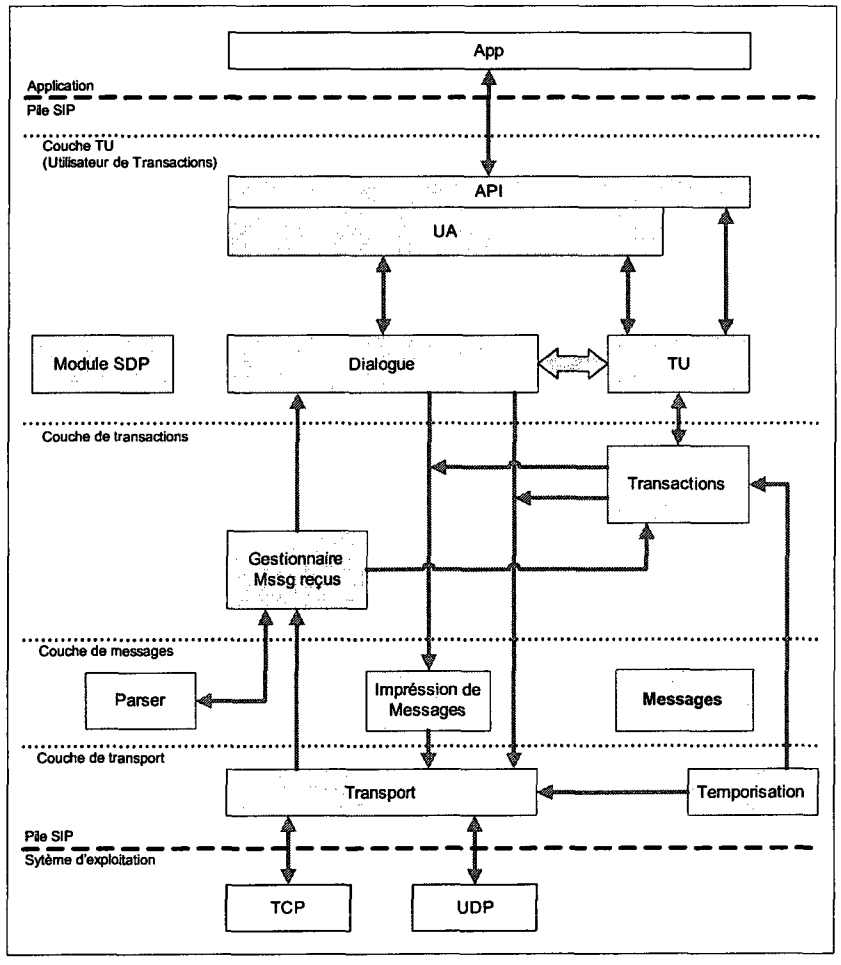

Figure 7.1: Architecture de la mise en œuvre du protocole SIP

Pendant la deuxième étape, l'application doit enregistrer les fonctions de rappel (*callbacks*) de l'API : **onIncommingCall()**, **onCallingTone()**, **onCallBusy()**, **onCallRejctd()** et **onCallError()**.

Finalement, la troisième étape consiste à créer un UA et à enregistrer les paramètres qui définissent le profil de l'utilisateur tel que son URI et le nom à afficher (*display name*), information de contact, etc. La création d'un UA déclenche immédiatement la procédure d'enregistrement de l'utilisateur dans un serveur REGISTRAR. La pile est maintenant prête à opérer.

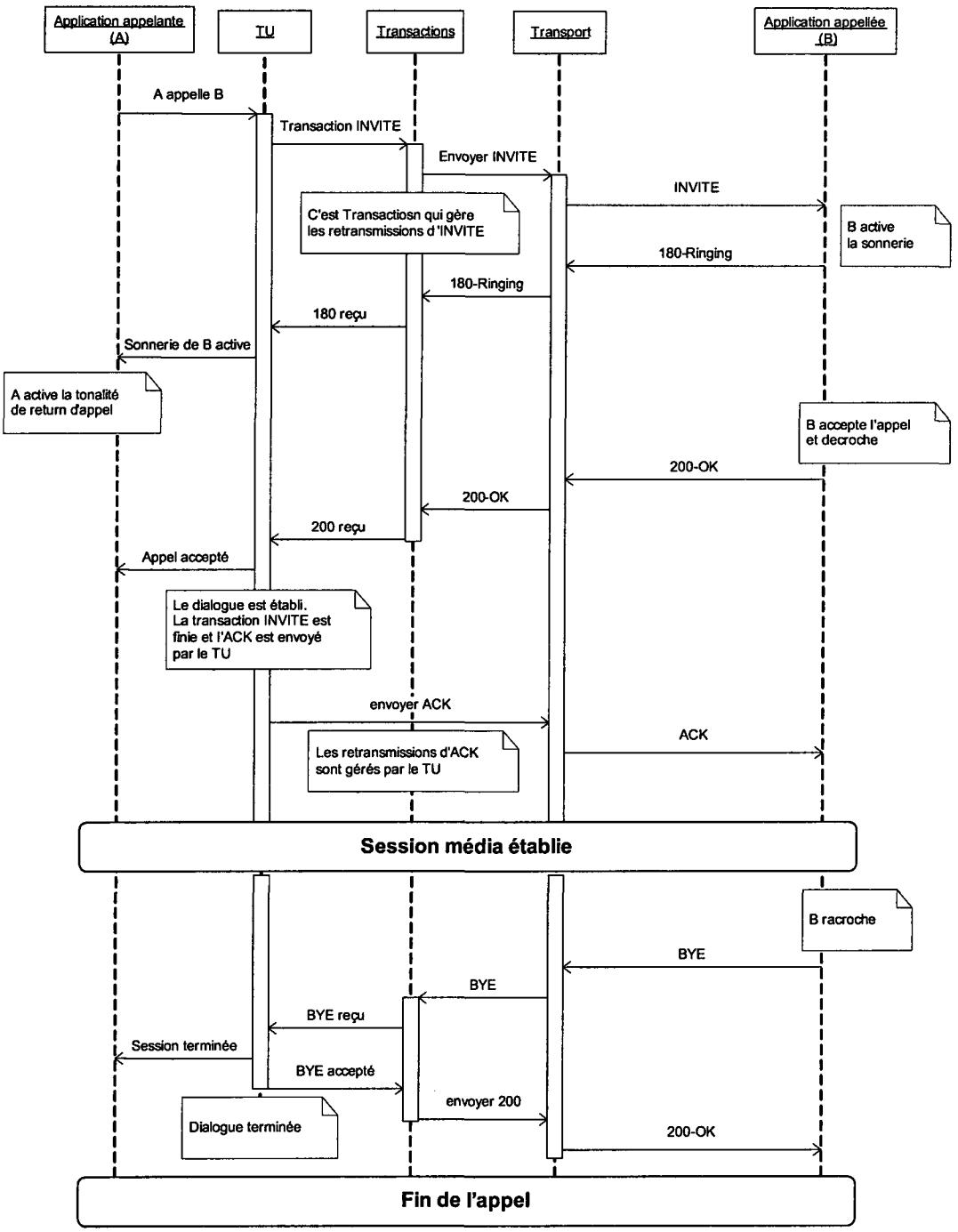

Figure 7.2: Diagramme de séquence de l'établissement d'une session SIP

7.1.3 Établissement d'une session

Alors que la description de l'architecture donne un portrait plutôt statique de la pile, la description de l'établissement d'une session aide à mieux comprendre sa dynamique et illustre son comportement avec la méthode la plus importante : INVITE.

Il faut se rappeler que, dans SIP, le rôle de client ou de serveur est défini transaction par transaction et que même si un UA est le client au moment de l'établissement de la session, il peut très bien devenir le serveur à la fin de cette session.

7.1.4 Comportement d'un client SIP

Pour établir une nouvelle session SIP, l'application doit appeler la fonction erUA_makeCall(), qui prend comme arguments l'URI appelé et l'objet de l'appel. Cette fonction déclenche dans le TU une série d'activités. En premier lieu, elle va créer une structure qui contiendra l'état de l'appel. Cette structure-ci est, dans cette implantation, une extension de la structure de l'état du dialogue défini dans le RFC 3261 (Rosenberg et al., 2002). La deuxième étape est la création de la requête INVITE initiale avec la fonction erGenMssg_INV(). La troisième étape consiste à créer une transaction avec erTrnxCreate(), pour gérer la transmission de la requête INVITE. La dernière activité de erUA_makeCall() est l'envoi d'un message-événement vers le module «*Transactions*» afin de déclencher son opération. À partir de ce moment, c'est le module «*Transactions*» qui prend en charge l'opération. Il transmet la requête et attend la réponse. Si le transport est non fiable (UDP), il gère aussi les retransmissions. Dans le cas d'une réponse négative, c'est la couche de transactions celle qui gère la transmission de l'ACK (*Acknowledge*). Un message-événement, envoyé vers la TU, indique le refus de l'appel. La couche TU, elle en informe l'application. Si par contre, l'INVITE est acceptée, la transaction est conclue et l'envoi de l'ACK est géré par le module **Dialog**. La session est établie et les UAs peuvent commencer à échanger l'information audio.

La figure 7.2 montre le diagramme de séquence de l'établissement d'une session SIP du point de vue d'une UAC. Le diagramme met en évidence le rôle de chaque couche pendant l'appel et les événements plus importants tels que la création et la destruction des transactions et du dialogue.

7.1.5 Comportement d'un serveur SIP

Lorsqu'un message SIP arrive, la couche de transport signale l'événement aux couches supérieures et met le message en file d'attente. Quand le module qui gère la réception est prêt, il demande le message et appelle immédiatement le parser pour faire sa conversion vers leur représentation interne. Puis, le message est validé et passé au TU afin de déterminer s'il est dans le contexte d'un dialogue, et si c'est le cas, s'il appartient également à une transaction. Une fois l'appartenance déterminée, le message est acheminé vers sa destination interne. Les messages qui sont hors du contexte d'un dialogue sont aussi envoyés vers le module TU, mais, dans ce cas, seules les requêtes avec les méthodes INVITE ou OPTIONS sont considérées. Dans le cas particulier des requêtes INVITE, le TU réagit en créant un nouveau dialogue et une transaction INVITE-SERVEUR. Puis, il envoie un message-événement vers la couche d'application afin d'avertir l'utilisateur à propos de l'appel entrant. D'habitude, l'application génère aussi un signal sonore (*ring*) ou graphique afin d'avertir à l'utilisateur. En même temps, le TU génère automatiquement une réponse provisoire qui est passée aux transactions pour sa transmission.

La suite dépend de la décision de l'utilisateur. Si l'appel est rejeté, le TU génère une réponse finale négative et la passe à la couche de transactions. Si par contre, l'utilisateur accepte l'appel, le TU génère une réponse finale positive (200 Ok). Dans les deux cas, la couche de transactions s'occupe de la transmission de la réponse, mais seulement dans la première situation, elle va aussi gérer la réception de l'ACK. Pourtant, dans le cas de réponse positive, c'est le TU même qui va s'en occuper. Une fois que ce dernier arrive, la session est établie et le TU informe le protocole RTP qu'il est prêt pour commencer avec la session audio.

7.2 Représentation interne des messages

Les messages SIP encodés en format texte ne sont pas appropriés pour être traités par un système logiciel. Pour cette raison, à l'interne, ils ont une représentation basée sur des structures de données, plus adaptée, alors, à la manipulation postérieure. Le parser fait la conversion du format texte vers le format interne tandis que le module d'impression fait le travail inverse.

```
typedef struct {
   erINT16U          users;          /* users count                                  */
   erINT16U          mssgClss;       /* incomming message-class:                     */
                                     /*                0xx  Request                  */
                                     /*                1xx  Provisional response     */
                                     /*                2xx  Response Success         */
                                     /*                3xx  Response Redirection     */
                                     /*                4xx  Response Client error    */
                                     /*                5xx  Response Server failure  */
                                     /*                6xx  Response Global failure  */
   union {                           /* message type:                                */
       sipMethdId_e  rqustMthd;      /*      request  Id (method)                    */
       sipRespCod_e  respId;         /*      response Id                             */
       }mTyp;
   erINT16U          dialogID;       /* dialog ID                                    */
   sipTrnsProt_e     trnspProto;     /* transport protocole: UDP, TCP, TLS, SCTP     */
   erSipUri_t       *rmoteTargetUri; /* remote target URI to be used in request line */
   sipUArole_e       clientSRVR;     /* may be UA_CLIENT or UA_SERVER                */
   union {
       erSipUri_t    *rqustURI;      /*   request URI                                */
       erCHAR        *reasonPhrase;  /*   reason phrase                              */
       }mInf;
   erCHAR            *fromTag;       /*  from tag                                    */
   erCHAR            *toTag;         /*  to tag                                      */
   erCHAR            *callId;        /*  Call-Id                                     */
   erINT32U           cSeqNum;       /*  sequence number                             */
   sipMethdId_e       cSeqMth;       /*  command (sequence number)                   */
   erINT32U           keyVal;        /*  dialog key value                            */
   erINT16U           mssgError;     /* non critical error codes                     */
   erINT16U           rfc3261Branch; /* flag: branch parameter is RFC3261 compliant  */
   erCHAR            *topViaBranch;  /* top Via branch (for transaction-ID)          */
   erCHAR            *topViaRcvdIp;  /* top Via received parameter                   */
   erINT16U           topViaRcvdPort; /* top Via received port                       */
   erCHAR            *targetAdd;     /* first hop target address                     */
   erINT16U           targetPort;    /* first hop target port                        */
   erINT16U           bodyLen;       /* body length                                  */
   erCHAR            *body;          /* message body pointer                         */
   erHdrLst_t        *nxtHdr;        /* header list                                  */
   } erMessage_t;
```

Figure 7.3: Structure utilisée pour la représentation interne des messages SIP

Cette section présente différents aspects en relation avec les structures de messages : leur définition, leur création, leur destruction, et une description du parser. La section se termine avec la présentation de la procédure à suivre afin d'ajouter de nouveaux en-têtes SIP.

7.2.1 Description des structures de messages

La figure 7.3 montre la déclaration de la structure qui décrit les messages SIP. Elle permet de représenter tous les types de messages, requêtes ou réponses, sous le rôle de client ou de serveur.

Dans un premier niveau, la structure regroupe les éléments principaux du message afin de rendre son traitement plus simple . Ainsi, plusieurs valeurs et paramètres appartenant aux en-têtes, qui ne sont accessibles que d'une manière détournée, sont rendus disponibles directement avec des pointeurs. Tel est le cas, par exemple, des valeurs de *«toTag»*, *«fromTag»* ou *«callId»*.

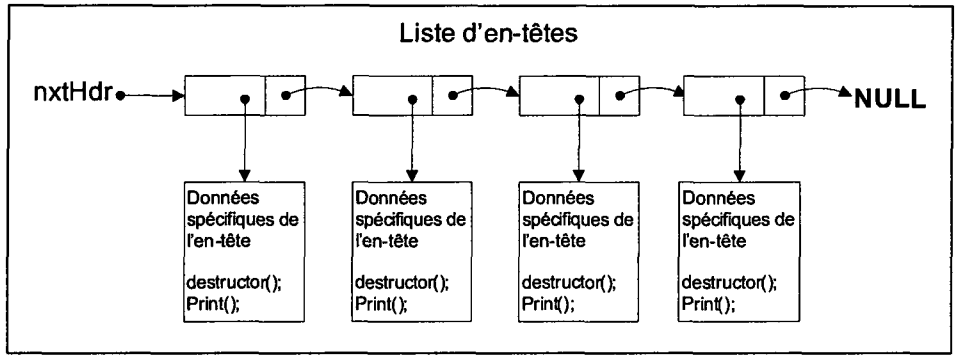

Figure 7.4: Liste chaînée des en-têtes SIP

Les en-têtes SIP sont organisés dans une liste chaînée, comme illustré dans la figure 7.4. Une particularité de cette liste est qu'elle est constituée par des maillons non homogènes. Ainsi, chaque maillon est formé par un agrégat de deux structures. La première possède un format invariant et contient les liens pour la liste et les accès vers la deuxième structure. Celle-ci a un format variable et comprend l'ensemble des données correspondant à chaque type d'en-tête.

```
struct erHdrLst{
  struct erHdrLst  *nxtHdr;              /* next header struct pointer             */
  erINT16U          nSeq;                /* sequential number (only for debug)     */
  erINT16U          headerId;            /* header identification                  */
  erINT16U          errCod;              /* errCod generated by the parser         */
  erHdrParmLst_t   *nxtParm;             /* ptr to the non-specific parameter list */
  void             *hd;                  /* ptr to header specific data            */
  erINT16U         *sUsrs;               /* ptr to the reference counter           */
  void       (*dstructor)(void* thisP);  /* ptr to the destructor function         */
  erINT16U   (*printHeader)(void* thisP, /* ptr to the header print function       */
                            erCHAR *buf,
                            erINT16S *len);
  };

typedef struct erHdrLst erHdrLst_t;
```

Figure 7.5: Maillon de la liste chaînée d'en-têtes SIP

Certains éléments de la structure des maillons sont plus particulièrement intéressants. Le pointeur **hd**, voir la figure 7.5, est le lien interne entre les deux structures de l'agrégat. Les pointeurs vers les fonctions **dstructor()** et **printHeader()** permettent d'adresser les bonnes fonctions de destruction et d'impression selon le type d'en-tête en implémentant une forme de polymorphisme. Cette modalité d'accès simplifie beaucoup la destruction et l'impression de messages, car elle rend possible l'utilisation des itérateurs. Finalement, **sUsrs** est un pointeur vers la variable **users**, qui indique le nombre de références, donc d'utilisations, de la structure (ces structures peuvent être partagées par plus d'une liste). À titre d'exemple, la figure 7.6 montre la partie variable d'un maillon correspondant à l'en-tête «To».

```
typedef struct{
    erINT16U       users;              /* struct user count                */
    erCHAR        *tDsplayName;        /* "To" display name                */
    erSipUri_t    *toUri;              /* "To" URI                         */
    erCHAR        *toTag;              /* "To" tag                         */
    }erSipHdrTo_t;
```

Figure 7.6: Structure pour l'en-tête «To»

Dans le cadre de ce projet, seules les en-têtes SIP les plus importants ont été implémentés : **To**, **From**, **Via**, **Call-ID**, **CSeq**, **Contact**, **Content-Type**, **Content-Lenght**, **Max Forwards**, **Route**, **Record-Route**, **Supported**, **Require** et **Subject**.

La définition d'un type d'en-tête générique permet de considérer les cas non supportés. L'en-tête générique emmagasine le contenu de la ligne sans faire aucun type d'interprétation.

7.2.2 Création et destruction de messages

La procédure pour la création d'un message SIP consiste à créer la structure de base montrée dans la figure 7.3 et d'annexer à la liste, un par un, les en-têtes requis. Les fonctions **newMessage()** et **newHeaderFrom()** sont deux exemples des fonctions disponibles pour la création de tous ces éléments. Elles retournent des pointeurs vers des structures vides, qui doivent être remplies par la fonction appelante.

Par ailleurs, lorsqu'il s'agit de la création de requêtes, il y a des fonctions qui automatisent cette procédure. Dans ces cas, une partie importante des valeurs des paramètres des en-têtes est prise de la structure utilisée pour définir l'état des dialogues. Ainsi, l'existence de cette structure est une condition préalable à l'utilisation des fonctions de création de requêtes.

Même dans certaines situations où le RFC 3261 (Rosenberg et al., 2002) dit qu'il ne doit pas s'établir un dialogue, ces structures doivent être créées, ayant un caractère éphémère.

La génération des réponses est aussi automatisée. La procédure est semblable au cas des requêtes sauf que plusieurs en-têtes de la réponse sont «copiés» à partir de la requête reçue. La fonction **erMssgHdrCopy()** fait ce travail. Il faut mettre en évidence que les structures ne sont pas copiées physiquement, elles sont plutôt partagées pour les deux messages, requête et réponse, qui ont des références vers elles.

Du point de vue de sa destruction, un message peut être considéré comme une collection de chaînes de caractères et de structures. En conséquence, pour le détruire, une fonction doit parcourir et supprimer systématiquement chaque chaîne et chaque structure de cette collection. Mais, avant de procéder, le destructeur doit vérifier s'il y a encore des références en examinant la valeur de la variable **users**. Cette variable est incluse dans chaque structure susceptible d'être partagée.

```
void hdrDstructor(erHdrLst_t* thisP)
{
    if( thisP){
        if(thisP->nxtParm   != NULL)
            erParmListDstructor(thisP->nxtParm);  /* call parameter-list destructor */
        if( thisP->hd)
            (thisP->dstructor)(thisP->hd);        /* destr hdr-specific data struct */
        if( thisP->nxtHdr );
            hdrDstructor( thisP->nxtHdr );        /* call next header destructor    */
        erMemFree((void*)thisP);                  /* free allocated memory         */
    }
}

void hdrDstructorCSeq(void* thisP)
{
    if (! --((erSipHdrCSeq_t*)thisP)->users )     /* if no more struct users        */
        erMemFree((void*)thisP);                  /*   free memory block            */
}
```

Figure 7.7: Fonction destructrice de la structure de l'en-tête CSeq

Par exemple, la séquence de destruction d'un message inclut la destruction de la liste d'en-têtes. Cette tâche est accomplie par la fonction récursive **hdrDstructor()**, montrée dans la partie supérieure de la figure 7.7. Cette fonction appelle, en même temps, au destructeur de la structure complémentaire dans l'agrégat en utilisant le pointeur vers la fonction **(thisP->dstructor)(thisP->hd)**. Finalement, la partie inférieure de la figure 7.7 illustre la

fonction destructrice pour la structure de données spécifique de l'en-tête Cseq. Il faut noter qu'avant de procéder, la variable **users** est décrémentée et vérifiée.

7.2.3 Parser

Comme on a déjà mentionné, le module Parser fait l'analyse syntaxique des messages en format texte pour les transformer en leur représentation interne. La grammaire du SIP est décrite, dans la section 25 du RFC 3261 (Rosenberg et al., 2002), en utilisant **ABNF** (*Augmented Backus-Naus Form*) défini par le RFC 2234 (Crocker and Overell, 2008). Cette grammaire contextuelle (*context sensitive*), qui présente quelques ambiguïtés, n'est pas résolue efficacement par les générateurs de parsers. En conséquence, les analyseurs syntaxiques pour SIP sont fréquemment écrits à la main. Cette dernière option est l'approche suivie dans ce projet en utilisant un parseur descendant (*top-down*) complémenté avec un analyseur lexicographique (*lexer*), aussi écrit à la main, chargé de partitionner le message en tokens. La figure 7.8 montre la définition formelle d'un message SIP.

```
SIP-message     =  Request / Response
Request         =  Request-Line
                   *( message-header )
                   CRLF
                   [ message-body ]
Response        =  Status-Line
                   *( message-header )
                   CRLF
                   [ message-body ]
```

Figure 7.8: Définition formelle d'un message SIP

En conséquence, le parseur doit, en premier lieu, identifier le message comme une requête ou une réponse, et ensuite faire l'analyse de la ligne de requête ou de la ligne d'état, selon le cas. Puis, le message rentre dans une boucle pour analyser les en-têtes jusqu'à trouver une ligne vide, qui signale la fin du bloc d'en-têtes. On trouve ensuite le «*message-body*» optionnel, qui à ce niveau, n'est considéré que comme un simple bloc de données. Étant donné que cette pile est destinée aux UA, le parser fait le traitement au complet du message. Par contre, lorsqu'il s'agit de mandataires, le parser fonctionne souvent sur demande, en analysant seulement les en-têtes requis. Dans ce cas-ci, on parle de «*lazy parsers*».

Le parser inclut une table d'en-têtes, où tous les en-têtes reconnus par la pile doivent être enregistrés pendant l'initialisation. La figure 7.9 montre la structure d'une entrée de

cette table qui définit : une variable de dispersion (**hvalue**), un pointeur vers l'analyseur syntaxique spécifique (**hfntion**), et le nom et la longueur de l'en-tête enregistré (**hdrName** et **hdrNameLen**). Les en-têtes avec une notation abrégée requièrent deux enregistrements, pointant tous les deux vers le même parseur. Cette solution permet de simplifier l'accès à la table et de rendre son traitement global plus efficace et systématique. Par ailleurs, les en-têtes qui ne sont pas reconnus sont traités, de manière transparente, comme des «en-têtes génériques». Finalement, il faut noter que l'analyseur syntaxique est facilement extensible et que l'ajout de nouveaux en-têtes devient très simple, car il faut tout simplement définir le parseur spécifique et l'enregistrer pendant l'initialisation.

```
typedef struct {
    erINT32U        hValue;                         /* hash value                  */
    erHdrLst_t*     (*hfntion)(erINT16U* error);    /* header handler pointer      */
    erINT16U        hdrNameLen;                     /* header name length          */
    const erCHAR*   hdrName;                        /* header name                 */
} erHdrTabReg_t;
```

Figure 7.9: Structure pour l'enregistrement des en-têtes

Une fois la table d'en-têtes introduite, on peut décrire comment le parser fait le traitement des lignes d'en-têtes. Ce processus est fait en deux étapes. Dans la première, le parser identifie le nom de l'en-tête et le cherche dans une table. S'il le trouve, il passe immédiatement à la deuxième étape, consistant à compléter l'analyse du reste de la ligne. Pour le faire, le parser appelle la fonction spécialisée enregistrée dans la table. Le cas échéant, le parser considère l'utilisation d'un en-tête générique et emmagasine toute la ligne sans continuer l'analyse.

```
typedef struct {
    erINT16U        type;               /* token identifier                */
    erCHAR*         lexp;               /* lexeme pointer                  */
    erINT16U        lexlen;             /* lexeme length                   */
    erINT32U        value;              /* number Value                    */
} erToken_t;
```

Figure 7.10: Structure pour la déclaration des tokens

Il faut souligner que toute l'opération du parser est faite sur le tampon de données original, sans modifier le texte et sans faire de copies des données. En particulier, l'emmagasinage dynamique des chaînes de caractères est fait avec les fonctions spécialisées **erLibStrSto()** et

erLibnStrSto(). Elles ont été conçues pour gérer cette tâche d'une manière très efficace tout en évitant les effets de la fragmentation de la mémoire.

L'analyseur lexicographique, ou lexer, a la mission d'identifier les tokens qui seront passés au parser. Il inclut aussi des fonctions de corrélation de patrons qui sont appelées par le parseur au moment de faire des prédictions.

À la base de l'opération du lexer on trouve une fonction qui permet de classer les caractères selon leur appartenance à un des sous-alphabets définis par SIP. Cette importante fonction s'appuie sur l'utilisation d'une table de classification qui considère les sous-groupes de caractères montrés dans le tableau 7.1.

Digit
Hexadécimal
Alphabétique
Alphabétique, majuscule
Alphabétique, minuscule
Alphanumérique
Espaces séparateurs
Caractères réservés pour SIP
Marques SIP
Tokens SIP
Séparateurs SIP
Séparateurs nus SIP
Mots SIP
Caractères valides entre des guillemets

TABLEAU 7.1: Sous-alphabets utilisés pendant le parsing des messages SIP

Les sept premiers groupes du tableau 7.1 sont plutôt de nature générale, tandis que les derniers sont spécifiques à SIP. Ils sont définis dans la section 8.1 du RFC 3261 (Rosenberg et al., 2002). Il faut noter que bien que certains de ces ensembles de caractères ne soient pas définis explicitement par SIP, ils aident à définir d'autres ensembles par agrégation. C'est le cas, par exemple, du groupe «séparateurs nus», qui permet de définir l'ensemble «mots (*words*) SIP» comme l'agrégation de «tokens SIP» avec «séparateurs nus SIP».

Les tokens sont retournés vers le parser en utilisant la structure montrée dans la figure 7.10. Elle inclut, en plus d'un pointeur vers le lexème, la longueur de ce dernier. Ceci est nécessaire, parce que, afin d'éviter des mouvements de données, le lexème doit rester tout le temps dans le tampon qui contient le texte original. Ainsi, pour le comparer avec un patron déterminé ou pour le stoker, il faut connaître sa longueur. Finalement, la variable **value** sert à retenir la valeur du token, lorsqu'il est applicable.

7.2.4 Ajout des nouveaux en-têtes SIP

Une des caractéristiques remarquables de SIP est sa capacité d'évoluer et d'incorporer de nouvelles capacités d'une manière simple. Il est raisonnable d'attendre que ses implémentations accompagnent cette philosophie. Pour cette raison, l'extensibilité de la pile est une des conditions les plus importantes de sa conception. En conséquence, un effort particulier a été fait afin de simplifier l'ajout des nouveaux en-têtes. Les étapes pour le faire sont :

1. Ajouter l'identification du nouvel en-tête à l'énumération **sipHdrId_e**.
2. Définir une structure de données afin de garder la valeur des paramètres de l'en-tête.
3. Définir les fonctions pour la création, la destruction, l'impression et la génération de la structure de l'en-tête à partir de l'état du dialogue.
4. Définir la fonction spécifique pour faire l'analyse syntaxique (parsing) de l'en-tête.
5. Dans la fonction **hdrHndlerTabSetup()**, enregistrer le parser de l'en-tête pendant l'initialisation.

De nombreuses fonctions, implémentées dans la pile, peuvent être prises comme des modèles pour aider à la réalisation de cette procédure.

7.3 Production et impression de messages SIP

Une première classification faite sur les messages SIP les catégorise entre des **requêtes** et des **réponses**. Les mécanismes de génération sont différents pour chaque catégorie et ils seront analysés indépendamment plus loin dans cette section.

Par ailleurs, dans le contexte de cette implémentation, on appelle «impression de messages» la conversion d'un message vers le format textuel. Par opposition à la production de messages, cette opération ne fait pas de distinction entre des requêtes et des réponses, et la tâche est accomplie pour la même fonction dans les deux cas.

7.3.1 Requêtes

Les requêtes sont le moyen utilisé par le protocole pour initier les échanges d'informations entre deux UAs. Elles sont générées lorsqu'un UA agit comme client, et sont associées à un comportement proactif. La première ligne d'une requête SIP, la «*Request line*», exprime le but du message et elle est suivie d'une série d'en-têtes contenant l'information relative à l'état de l'échange de messages et des UAs participants.

Dans le chapitre 3, on a mentionné que, dans le contexte du SIP, la relation temporaire entre deux UAs s'appelle **dialogue**. L'état de cette relation, appelée «**état du dialogue**», est stocké dans une structure qui est décrite avec détail un peu plus loin, dans la section «Dialogues» de ce chapitre. Dans cette implantation, la systématisation des procédures pour la production de requêtes utilise la structure «état du dialogue» comme point de parti du processus de création de requêtes. Ainsi, l'existence de cette structure est un préalable à la construction d'une requête. Cependant, selon la recommandation RFC 3261 (Rosenberg et al., 2002), il y a certains échanges des messages SIP qui n'établissent pas des dialogues. Afin de contourner ce problème, le concept de dialogue défini pour la recommandation a été étendu avec la création des «dialogues éphémères». La principale différence de ces dernières avec les «dialogues SIP» est qu'ils sont terminés aussitôt que la transaction qui a demandé leur existence est finie. Étant donné que l'implémentation des dialogues est faite en utilisant des machines à états finis, il est relativement simple d'ajouter ce comportement avec la création d'un état initial particulier pour ce type de transaction.

À partir des éléments exposés, il est possible de rédiger la séquence générale à suivre pendant la production de requêtes :

1. Détermination de la méthode.
   Les requêtes sont générées sur demande, dont l'origine peut être externe, c'est à dire l'application, ou interne, c'est-à-dire la pile même. La détermination de la méthode provient de l'interprétation de l'intention de la demande. Par exemple, l'utilisateur du système veut établir une session (INVITE) ou bien finir une session déjà établie (BYE). Un autre exemple, mais d'origine interne, est le cas d'un enregistrement envoyé vers un serveur REGISTRAR, lors de l'initialisation de la pile. Finalement, l'utilisation ou non d'une méthode, à un moment donné, implique des considérations contextuelles.

2. S'il n'y a pas de structure d'état de dialogue, il faut la créer et la remplir avec les données pertinentes.
3. Incorporer les en-têtes qui constituent l'information d'état du message : To, From, Call-ID, CSeq, Via, Max-Forwards, Route.
4. Incorporer des en-têtes additionnels, dépendants du contexte.
5. Ajouter le corps du message (*message-body*) au besoin.

Les fonctions disponibles pour la création de requêtes sont presentées à la figure 7.11. La fonction **erGenMssg_RQST()** permet de créer des requêtes génériques, par contre les fonctions **erGenMssg_INV()** et **erGenMssg_REG()** génèrent des requêtes INVITE et REGISTER respectivement.

```
erMessage_t* erGenMssg_RQST( erSipDialogStat_t  *dlgP,
                             sipMethdId_e        mthod,
                             erINT16U            *errorP)
erMessage_t* erGenMssg_INV(  erSipDialogStat_t  *dlgP,
                             erCHAR              *subjet,
                             erINT16U            *errorP)
erMessage_t* erGenMssg_REG(  erSipDialogStat_t  *dlgP,
                             erCHAR              *subjet,
                             erINT16U            *errorP)
```

Figure 7.11: Fonctions de création de requêtes

7.3.2 Réponses

Les réponses sont générées pour le TU pour indiquer l'acceptation ou le refus d'une proposition faite sous forme de requête. C'est un comportement réactif. Ainsi, la réponse est générée en prenant la requête originale comme message de référence. Dans la figure 7.12, on montre la fonction génératrice de réponses. À partir de l'analyse des arguments, il est possible de déduire les éléments clés : le code de la réponse, le message original (la requête) et l'état du dialogue.

Plusieurs en-têtes de la réponse sont copiés directement à partir du message original. Cette copie est effectuée pour la fonction **erMssgHdrCopy()**, qui cherche dans le message original toutes les instances de l'en-tête spécifié et les ajoute à la réponse en respectant l'ordre d'occurrence. Il faut mettre en évidence que ce mécanisme de copie n'implique pas de mouvements de données dans la mémoire, car l'opération est effectuée en utilisant des pointeurs vers les structures impliquées. La variable **users** de ces structures, qui indique le

nombre de références, est incrémentée pour indiquer que leur utilisation est partagée entre le message original et la réponse.

Par ailleurs, certains en-têtes doivent être générés (ou régénérés) localement, parce que l'en tête original a été modifié pendant le parcours du message par le réseau. Un exemple typique est **Max-Forwards**. Une autre situation arrive lorsqu'ils dépendent des valeurs locales, tel que **Contact**. Il y a des fonctions génératrices, spécialisées pour chaque type d'en-tête, qui reçoivent comme arguments, le message de destination et l'état du dialogue pertinent.

```
erMessage_t* erGenRsponseMssg( sipRespCod_e          rspnsCode,
                               erMessage_t          *rxMssgP,
                               erSipDialogStat_t    *dlgP,
                               erINT16U             *errorP)
{
    erMessage_t          *txMssg     = NULL;


    if( !dlgP || !rxMssgP || !errorP ){              /* argument validation         */
        *errorP = EC_NULLPOINTER;
        return NULL;
    }
    if ( dlgP && (txMssg = newMessage()) ){          /* creates new sip mssg struct */
        txMssg->clientSRVR     = UA_SERVER;          /* UA is server                */
        txMssg->mTyp.respId    = rspnsCode;          /* response type               */
        txMssg->rfc3261Branch = rxMssgP->rfc3261Branch;
        txMssg->dialogID       = dlgP->dialogID;     /* dialog  Id                  */
        txMssg->trnspProto     = dlgP->trnsprtProto; /* set transport protocol      */
                                                     /* cpy frst hop target address */
        txMssg->targetAdd      = (erCHAR*)erLibStrSto( rxMssgP->topViaRcvdIp );
        txMssg->targetPort     = rxMssgP->topViaRcvdPort; /* cpy frst hop target port*/

        erMssgHdrCopy( txMssg, rxMssgP, SHDR_ID_VIA );  /* copy Via headers         */
        erMssgHdrCopy( txMssg, rxMssgP, SHDR_ID_TO );   /* copy "To" header         */
        erMssgHdrSetRespToTag( txMssg, dlgP );          /* set "To" tag = local tag* /
        erMssgHdrCopy( txMssg, rxMssgP, SHDR_ID_FROM ); /* copy "From" header       */

        erMssgHdrCopy(txMssg, rxMssgP, SHDR_ID_CALLID); /* copy "CallId" header     */
        erGenMssgAddHdr_Contact(txMssg, dlgP);          /*   add header Contact     */
        erGenMssgAddHdr_MaxForwards(txMssg, dlgP);      /*   add header MaxForwards */
        erMssgHdrCopy(txMssg, rxMssgP, SHDR_ID_CSEQ);   /* copy CSeq header         */
        erMssgHdrCopy(txMssg, rxMssgP, SHDR_ID_GENERIC);/* copy Generic headers     */
        erGenMssgAddHdr_Route(txMssg, dlgP);            /*   add header Route       */

    }
    return txMssg;
}
```

Figure 7.12: Génération de messages réponse

7.3.3 Impression de messages

On appelle «impression» la procédure qui s'occupe de la génération d'une copie du message SIP en format texte à partir de sa représentation interne. Cette copie est passée à la couche de transport aux fins de sa transmission. Dans certaines circonstances, cette fonctionnalité est utilisée pour générer l'information qui permettra la mise au point du système.

En se rappelant qu'un message SIP consiste en une ligne de requête ou d'état et en une succession de définitions d'en-têtes SIP, la procédure d'impression devient très simple. La première étape consiste à générer la ligne d'en-tête, qui peut être une requête ou une ligne d'état, à partir de l'information disponible directement dans l'instance de la structure erMessage_t du message. Dans une deuxième étape, un itérateur permet de parcourir la liste chaînée des structures d'en-têtes en appelant les pointeurs vers les fonctions (*printHeader)(), qui donnent accès aux routines spécialisées d'impression pour chaque type d'en-tête. La troisième étape consiste à délimiter la fin de cette section du message avec la génération d'une ligne vide. Éventuellement, s'il y a lieu, on ajoute le corps du message (*message-body*).

Comme on a indiqué précédemment, un critère adopté dans cette implémentation spécifie que la couche de transport fait une totale abstraction du contenu des messages. Elle n'effectue ni interprétation ni modification. Cette condition oblige à accomplir certaines opérations juste avant l'impression du message. Le cas le plus remarquable est celui de l'en-tête **Via**, où le champ «envoyé par» doit être complété avec l'information de l'adresse IP et du port utilisés pour la transmission. Ainsi, son contenu est rempli avec l'information demandée à la couche de transport pendant l'étape de préparation à l'impression. Un autre cas notable se présente lorsque le transport spécifié est TCP. Cette situation est décrite un peu plus loin.

La longueur maximale d'un message SIP dépend, en principe, du type de transport. Lorsqu'il s'agit d'UDP, tout le message doit être contenu dans un seul datagramme afin d'éviter des fragmentations. La recommandation RFC 3261 (Rosenberg et al., 2002) établit cette limite à 200 octets en dessous du MTU (*Maximum Transmission Unit*) du chemin (*path MTU*), si cette valeur est connue, ou bien 1300 octets si elle ne l'est pas. La raison pour laquelle nous prenons 200 octets de marge est justifiée par le fait qu'un message peut grandir pendant son parcours à travers du réseau à cause de l'addition des en-têtes **Via**, ou bien dû à l'inclusion d'un en-tête **Record-Route**. Si la longueur du message est plus grande que

```
erINT16U hdrPrintCSeq( void     *thisP,
                       erCHAR   *buffer,
                       erINT32S *len)
{
    erINT32S            hdrLen;
    erCHAR              *CSeqMethode;
    erSipHdrCSeq_t*     CSeqP;
    erCHAR              locBuf[30];


    if( !thisP || !buffer )
        return EC_NULLPOINTER;


    CSeqP = (erSipHdrCSeq_t*)thisP;          /* short cut                          */

    CSeqMethode = (erCHAR*)mthIdGetName(CSeqP->cSeqMth);
    sprintf(locBuf,"%u",CSeqP->cSeqNum);

    hdrLen =  strlen(msg_HDR_CSEQ)           /* check if buffer is big enough      */
            + strlen(locBuf)
            + strlen(CSeqMethode)
            + 5;
    if( hdrLen < *len){
        sprintf( buffer, "%s: %s %s\r\n",    /* print header                       */
                         msg_HDR_CSEQ,
                         locBuf,
                         CSeqMethode);
        *len -= hdrLen;                      /* update "available length"          */
    }
    else
        return EC_M_NOTENOUGHPLACE;          /* not enough place in the buffer     */
    return EC_ERRNONE;
}
```

Figure 7.13: Fonction d'impression d'un en-tête Sip

le maximum admis, il faudra utiliser TCP au lieu d'UDP pour le transport. Par ailleurs, quand le transport spécifié est TCP, le message peut être fragmenté. Dans ce cas, et dans le but de permettre de le délimiter et de vérifier son intégrité, le RFC 3261 (Rosenberg et al., 2002) exige qu'un en-tête **Content-Length** soit ajouté indépendamment de la présence ou non du *message-body* dans le message. Si *message-body* n'est pas présent, la valeur du paramètre de **Content-Length** sera zéro. La vérification de la longueur du message et l'ajout éventuel de l'en-tête mentionné sont faits juste avant le processus d'impression. Par exemple, la figure 7.13 montre la fonction d'impression pour l'en-tête **Cseq**.

Dans certains cas, il est difficile de prévoir la place que prendra l'impression d'un en-tête. Ainsi, l'utilisation des tampons temporaires locaux permet d'éviter l'écrasement de données et une fois que la longueur de la chaîne est vérifiée, son contenu est copié dans le tampon de destination.

7.4 Réception de messages

Dans les systèmes de communication, la partie de réception est normalement plus complexe que celle de transmission. En grandes lignes, la raison de cette différence est que la réception doit gérer des contingences externes au système, qui se produisent, par exemple, dans le réseau ou sur un terminal distant. Lorsque plus de contingences de réception sont gérées, plus stable et robuste sera le système. Par contre, l'opération du transmetteur est plus prévisible, car le nombre de facteurs incidents qui sont hors du système est très inférieur.

Cette section décrit une partie importante de la pile, le *front-end* de réception, qui inclut l'opération du segment de réception de la couche de transport et du gestionnaire de réception.

```
struct sipRxMssg{
   erINT16U            rxMssgId;      /* message Id                                    */
   erINT16U            stat;          /* message stat: 0 = unused; 1 = in use/used     */
   erCHAR             *buf;           /* message buffer                                */
   erINT16U            bufLen;        /* message length                                */
   sipTrnsProt_e       trnspProto;    /* transnport type: UDP, TCP, TLS, SCTP          */
   erINT16U            srce_port;     /* sip source port                               */
   erCHAR              srce_ipadd[16];/* source Ip Address (dotted decimal notation)   */
   struct sipRxMssg   *next;          /* pointer to the next element in the list       */
    };
typedef struct sipRxMssg    sipRxMssg_t;
```

Figure 7.14: Structure de la file d'attente de réception de messages

7.4.1 Partie réception de la couche de transport

Une fois que le message est entré dans le système, il est mis en file d'attente jusqu'à son traitement par le gestionnaire de réception. La file d'attente est implémentée comme une liste chaînée, la figure 7.14 montre la structure des maillons. Ceux-ci incluent des informations complémentaires telles que l'adresse IP, le port de la source et le protocole de transport. Ces données permettront, dans une phase ultérieure, de compléter le **top Via header** avec le paramètre **received**.

Par ailleurs, la variable **stat** joue le rôle de fanion afin d'indiquer si le message a déjà été lu par le gestionnaire de réception au non. Ceci est nécessaire, car les messages restent tout le temps dans la file d'attente, tout au long de leur traitement, jusqu'à ce que le gestionnaire les détruise. Encore, afin d'éviter de déplacements des données, tous les tampons alloués

aux messages reçus restent physiquement à la même place. Le corps du message est stocké dans un tampon de texte pointé par **buf**, sa longueur étant indiquée par la variable **buflen**.

Pour informer les couches supérieures de l'arrivée d'un message, la couche de transport appelle une fonction de rappel enregistrée pendant l'initialisation par le gestionnaire de réception. Cette fonction ne fait que signaler un sémaphore afin de déclencher l'opération du gestionnaire.

Les prototypes des méthodes qui permettent au gestionnaire d'acquérir et de détruire les messages qui restent dans la file d'attente sont montrés dans la figure 7.15.

```
sipRxMssg_t *erSipRxMssgGet    ( erINT16U *errorP );
erINT16U     erSipRxMssgFree ( erINT16U   mssgId );
```

Figure 7.15: Prototypes des méthodes d'acquisition et de destruction des messages reçus

7.5 Gestionnaire de réception de messages

Le gestionnaire de réception joue un rôle important dans la pile, car c'est lui-même qui fait le traitement initial des messages et les achemine vers leur destination interne finale. La séquence de tâches inclut la conversion du message vers le format interne, sa validation, le test pour déterminer l'appartenance du message à un dialogue ou à une transaction, et finalement l'acheminement.

La première étape du traitement du message, accomplie par le module parser, consiste à changer sa représentation vers le format interne. Ce dernier et le fonctionnement du parser ont déjà été décrits dans ce chapitre. Si le parser n'arrive pas à compléter sa tâche, il génère une erreur et le gestionnaire abandonne le message.

La deuxième étape consiste à valider la structure du message, comme indiqué dans la section 8 du RFC 3261 (Rosenberg et al., 2002). Cette tâche consiste à vérifier si l'URI de destination du message correspond à l'URI local et si les en-têtes définis comme essentiels sont présents.

La troisième étape consiste à déterminer si le message reçu est à l'intérieur ou à l'extérieur du contexte d'un dialogue. Dans le cas affirmatif, il faut déterminer s'il est aussi contrôlé par une transaction. Avec cette information, le gestionnaire est capable de déterminer la

destination finale du message et de procéder à l'acheminement. Cette procédure de prise de décision est illustrée dans l'ordinogramme de la figure 7.16.

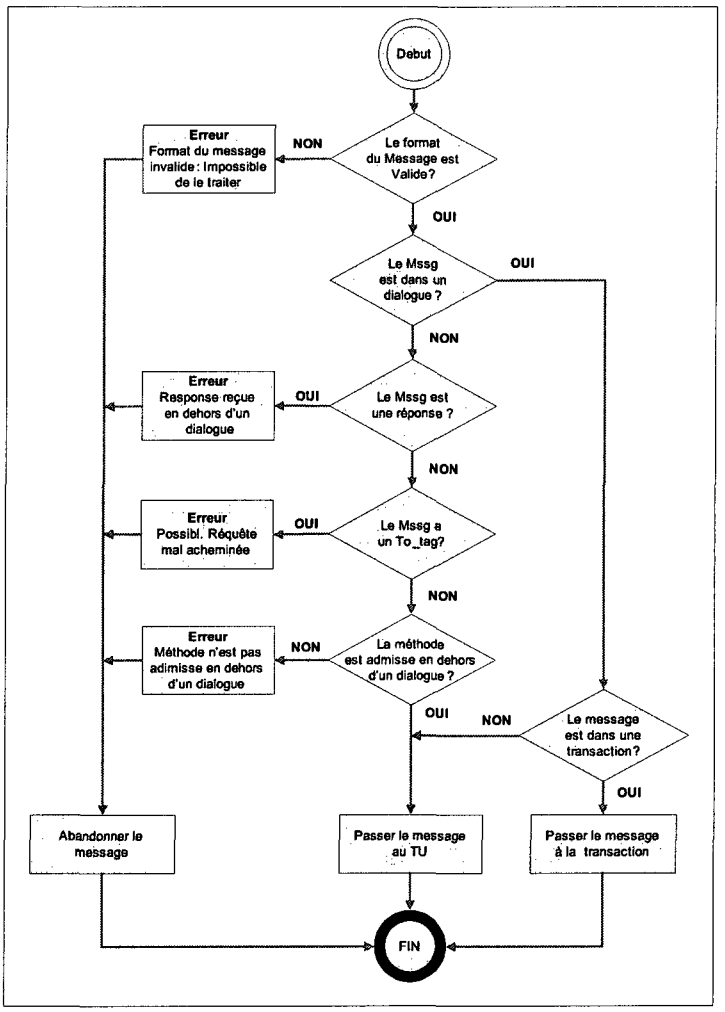

Figure 7.16: Traitement des messages reçus

7.6 Transactions SIP

Étant donné que SIP est un protocole transactionnel, l'importance de l'implémentation des modules chargés de gérer les transactions est évidente. Il s'agit, d'ailleurs, d'une des parties qui possèdent le plus haut niveau de difficulté. Alors, sa description sera présentée de manière soignée et extensive.

Le RFC 3261 (Rosenberg et al., 2002) définit une transaction comme l'ensemble d'une requête et de toutes les réponses associées, soit une réponse finale et zéro ou plusieurs réponses provisoires. Dans ce contexte, le comportement du protocole est spécifié à l'aide de la définition des machines à états finis (MEF). Celles-ci sont analysées dans la première partie de cette section pour passer ensuite à la description de son implémentation dans la deuxième partie.

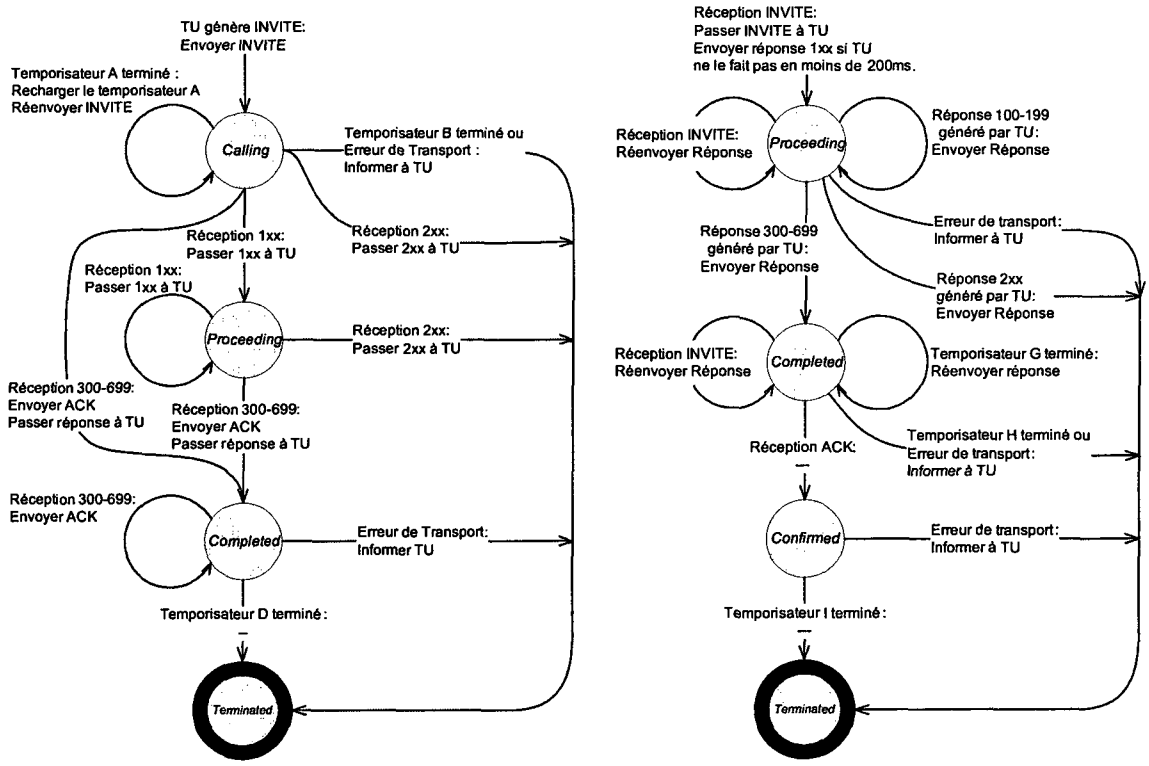

Figure 7.17: Machines à états finis. INVITE/client, INVITE/serveur

7.6.1 Machines à états finis définies par le RFC 3261

Chaque transaction possède un côté serveur et un côté client. Ce dernier est celui qui transmet la requête, tandis que le serveur est l'entité qui la reçoit et qui s'occupe de générer et de transmettre la réponse. En plus de considérer les côtés client et serveur, la recommandation RFC 3261 (Rosenberg et al., 2002) considère un comportement particulier pour les transactions reliées à la méthode INVITE. Ainsi, on parle des transactions «NON-INVITE» dans tous les autres cas.

La combinaison de ces possibilités donne lieu à un total de quatre machines à états finis : INVITE/client, INVITE/serveur, NON-INVITE/client et NON-INVITE/serveur. Dans la figure 7.17 on montre les machines à états finis correspondantes aux requêtes INVITE, et dans la figure 7.18, celles pour les requêtes NON-INVITE.

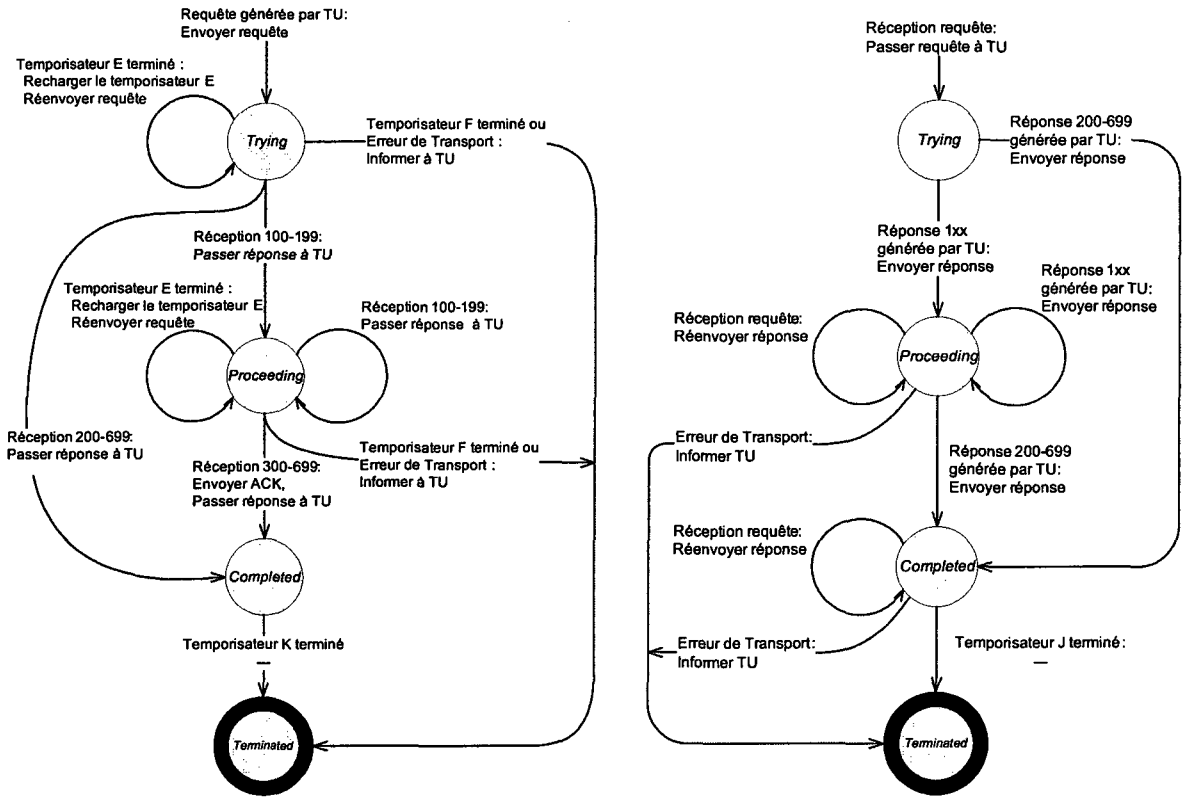

Figure 7.18: Machines à états finis. NON-INVITE/client, NON-INVITE/serveur

La méthode INVITE est la plus importante de celles définies pour SIP, car elle sert à établir une session. En premier lieu, cette méthode prend en considération le fait que les délais, avant que l'UA serveur ne génère une réponse finale, peuvent être considérables. De plus, comme INVITE sert à établir et à négocier les paramètres à utiliser par RTP, c'est l'unique méthode SIP qui utilise un *3-way handshake*. Ainsi, un message ACK est toujours associé à l'INVITE. De l'analyse des définitions des MEF, on peut constater que la transaction n'inclut qu'un ACK lorsque la réponse du serveur est différente de 2xx, autrement dit, lorsque le serveur donne une réponse finale négative. Par contre, lorsque le serveur donne

une réponse finale positive (200 Ok), la transmission de l'ACK est gérée pour une nouvelle transaction.

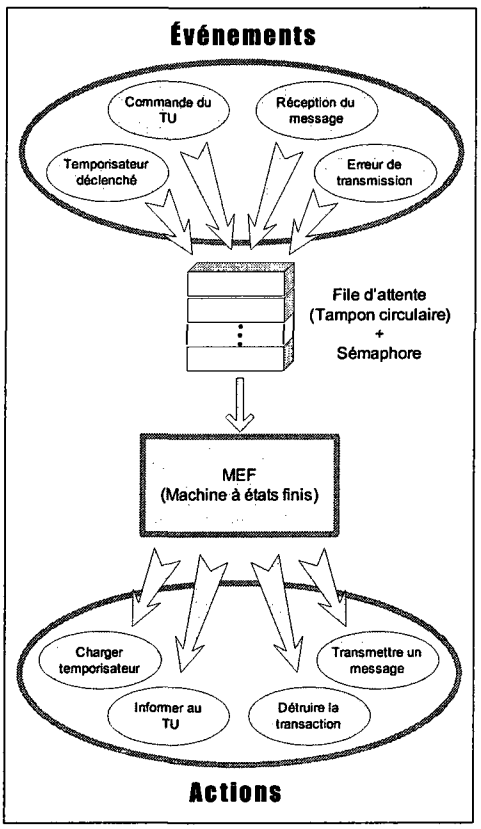

Figure 7.19: Modèle de fonctionnement des MEFs définies pour SIP

7.7 Modèle de fonctionnement

La figure 7.19 montre la modélisation faite pour les MEF. On peut voir qu'il y a quatre sources principales d'événements :

- La couche TU est la principale source d'événements pour les MEFs. TU génère presque toutes les requêtes et les réponses du système et c'est elle-même qui crée les transactions. La couche TU est aussi le principal destinataire des événements provenant de la machine.
- Le système de temporisation, qui signale le déclenchement des temporisateurs.

- Le gestionnaire de réception, qui informe l'arrivée d'une requête ou d'une réponse destinée aux transactions.
- La couche de transport, qui signale des problèmes liés à la transmission de messages, des requêtes ou des réponses.

Les MEFs associées aux transactions SIP sont exécutées dans une tâche indépendante. Le fonctionnement de cette tâche est synchronisé en utilisant un modèle producteur - consommateur matérialisé avec une queue de messages événements.

Les événements sont modélisés avec une structure qui inclut le type d'événement, l'identification interne de la transaction et un pointeur vers un type de donnée arbitraire. Ceci est typiquement utilisé pour attacher des messages SIP aux événements. Le gestionnaire d'événements inclut un tampon circulaire qui permet d'ordonner et de gérer les événements asynchrones provenant de différentes sources. Il inclut aussi des sémaphores qui servent à synchroniser le fonctionnement de la tâche qui exécute les MEF.

Pour chaque transaction, il y a une structure de contrôle qui définit son état courant. Une référence plus détaillée à cette structure est faite plus loin. Pour le moment, il est suffisant de mentionner qu'elle inclut deux éléments fondamentaux pour son fonctionnement : une variable, qui spécifie le type de transaction (une des quatre définies pour SIP), utilisée pour sélectionner la MEF, et une autre qui indique l'état courant de la MEF.

		États ⇒			
		état_0	**état_1**	**état_2**	**état_3**
Événements ⇓	**évnmt_1**	(*fn_1)() état_1	(*fn_6)() état_2	(*fn_4)() état_3	(*fn_8)() état_4
	évnmt_2	(*fn_1)() état_1	(*fn_6)() état_2	(*fn_4)() état_3	(*fn_8)() état_4
	évnmt_3	(*fn_1)() état_1	(*fn_6)() état_2	(*fn_4)() état_3	(*fn_8)() état_4
	évnmt_4	(*fn_1)() état_1	(*fn_6)() état_2	(*fn_4)() état_3	(*fn_8)() état_4

TABLEAU 7.2: Exemple de table de transitions

7.7.1 Détails de la mise en œuvre des MEF

Le comportement des MEFs est spécifié en utilisant des tables de transitions. Celles-ci permettent, à partir de l'état actuel et de l'événement, de déterminer la fonction chargée de gérer l'action pertinente et l'état suivant. L'utilisation des tables de transition est justifiée par le fait qu'elles permettent d'arriver à des solutions plus compactes et plus efficaces qu'avec les méthodes alternatives couramment utilisées avec des langages procéduraux, telles que des structures «*switch - case*» imbriquées. Par ailleurs, la complexité de leur mise à jour, un problème qui est normalement attribué à ce type d'implantation, a peu d'incidence dans ce cas. En effet, les MEF sont spécifiées dans la recommandation qui définit le protocole et les modifications qui pourraient changer son comportement sont peu probables.

Il y a quatre tables de contrôle en correspondance avec les quatre MEF définies pour SIP. Le tableau 7.2 montre une disposition typique d'une table de transition. Pour chaque état de la machine (colonnes), les différents événements (lignes) permettent d'adresser une cellule de la table en particulier. Chaque cellule contient deux éléments : le premier est un pointeur vers la fonction d'état qui accomplit les actions correspondant à la combinaison **état-événement** associée. Le deuxième élément est le prochain état vers lequel la machine doit évoluer.

7.7.2 Structure de contrôle pour les transactions

Une transaction possède, en plus de l'état de la MEF et de la définition de son type, plusieurs variables d'état. Elles sont toutes regroupées dans une structure, montrée dans la figure 7.20. Les plus importantes sont :

trsxStat : état courant de la MEF
tType : type de MEF : client/serveur ; INVITE/NON-INVITE
trnsportProto : protocole de transport : UDP / TCP
transportID : ID du transport
branch : «*top Via branch*», c'est l'ID univoque de la transaction associé au message.

Il faut remarquer que les temporisateurs associés aux machines à états finies ont été embarqués dans les structures de contrôle, et par conséquence, ils sont dédiés. Comme on le verra ultérieurement, cette approche présente plusieurs avantages.

```
struct erSipTransactn{
    erINT16U                trnxID;          /* transaction ID                        */
    erINT32U                trnxKey;         /* transaction Key (hash value)          */
    erINT16U                dialogID;        /* dialog ID (0: no dialog)              */
    trnxType_e              tType;           /* transaction type: INVC, INVS,         */
                                             /*           NON-INVC, NON-INVS          */
    sipMethdId_e            methdId;         /*   method identification               */
    sipTrnsProt_e           trnsportProto;   /* transport protocol: UDP / TCP / TLS */
    erINT16U                transportID;     /* transport ID                          */
    erCHAR                 *branch;          /*   Via branch                          */
    erINT16U                trsxStat;        /*   trsx state                          */
    erINT16U                txCntr;          /*   retransmission counter              */
    erMessage_t            *tmssgP;          /* sip message to transmit               */
    erMessage_t            *rmssgP;          /* received sip message                  */
    erINT16U                lstMssId;        /*   last tx message ID                  */
                                             /* transact tmrs: [0..655]sec            */
                                             /*       resolut: 10ms                   */
    erINT16U                tmr_1;           /*   timer 1, used for: reTx timer       */
    erINT16U                tmr_2;           /*   timer 2, used for: time out timer   */
    erINT16U                tmr_3;           /*   timer 3, used for: hold on timer    */
    erINT16U                tmr_4;           /*   timer 4, master time out (not a     */
                                             /*            RFC3261 timer)             */
    struct erSipTransactn  *nxtTrnx;         /* next transaction                      */
    };

typedef struct erSipTransactn erSipTransactn_t;
```

Figure 7.20: Structure de contrôle d'une transaction

Étant donné qu'un agent utilisateur (UA) peut maintenir actives plusieurs transactions en même temps, l'ensemble des structures de contrôle associées est stocké dans une liste simplement chaînée. Cette structure des données est bien adaptée à la dynamique des transitions SIP. Les nouvelles transactions, qui sont présumées plus actives, sont ajoutées dans la tête de la liste.

7.7.3 Temporisateurs pour les machines à états finis

Comme on peut l'observer dans le tableau 7.3, il existe plusieurs temporisateurs associés au fonctionnement des MEF définis par la recommandation RFC 3261 (Rosenberg et al., 2002). Les lignes de la table regroupent les temporisateurs selon leur fonction, tandis que les colonnes le font selon le rôle de la transaction qui les utilise. Évidemment, ces rôles sont disjoints et en conséquence, les temporisateurs de chaque ligne peuvent être regroupés et implémentés en utilisant un unique temporisateur réel. Ceci est l'approche finalement adoptée.

Les temporisateurs A, E et G sont employés pour déclencher des retransmissions lorsqu'on utilise des transports non sécurisés, tel que UDP. Par ailleurs, B, F et H contrôlent les délais d'attente (*time-outs*), afin de forcer l'extinction de la transaction face à des situations d'exception. Finalement, les temporisateurs D, I, J et K accomplissent une fonction de rétention (*hold-on*) avant la destruction finale d'une transaction afin de maîtriser l'arrivée des messages tardifs.

Rôle / Fonctions	INVITE		NON-INVITE	
	Client	Serveur	Client	Serveur
Retransmission	A	G	E	-
Time out	B	H	F	-
Hold on	D	I	K	J

TABLEAU 7.3: Temporisateurs définis pour SIP

Une analyse de la solution montre que les structures de contrôle des transactions incluent un total de quatre temporisateurs. Les trois premiers sont reliés au comportement défini dans la recommandation RFC 3261 (Rosenberg et al., 2002) et chacun d'eux est lié à une des fonctions indiquées dans le tableau 7.3.

Le fait que les temporisateurs soient embarqués dans les structures de contrôle permet de les gérer plus facilement. En premier lieu, ils sont créés et détruits avec la structure associée. En deuxième lieu, comme ces structures sont organisées comme une liste chaînée, l'actualisation des temporisateurs est bien simple. Elle consiste à parcourir la liste périodiquement avec un itérateur à une période égale à la résolution choisie (10 ms).

La recommandation RFC 3261 (Rosenberg et al., 2002) inclut une récapitulation des différents temporisateurs ainsi que leurs valeurs de charge par défaut : T1, T2, et T4. Ces derniers sont montrés dans le tableau 7.4. Il faut noter que, dans certains cas, l'initialisation des temporisateurs dépendra du type de transport employé, en faisant une distinction entre les protocoles de transport fiables, tel que TCP, et les non fiables comme UDP.

T1	T2	T4
500ms	4sec	5sec

TABLEAU 7.4: Valeurs par défaut de T1, T2 et T4

7.8 Dialogues

Selon le RFC 3261 (Rosenberg et al., 2002), un dialogue SIP est une relation pair-à-pair (*peer-to-peer*) entre deux entités SIP qui dure un certain temps. Un dialogue contient un certain nombre d'informations d'état qui sont nécessaires pour des échanges de messages ultérieurs entre les UA. L'ensemble de ces informations constitue «l'état du dialogue». Dans cette implantation, «l'état du dialogue» est stocké dans la structure qui se trouve à la figure 7.21. Ces structures, qui sont organisées dans une liste simplement chaînée, incluent plusieurs éléments qui permettent d'identifier le dialogue et de le corréler avec un message SIP. Ainsi, la variable **dialogID** donne un identificateur local unique bâti à partir des chaînes de caractères pointés par **call_id**, **localTag** et **remoteTag**. Tel que mentionné dans le chapitre 3, ces chaînes-ci sont utilisées par SIP afin d'identifier de manière univoque un dialogue.

Les variables de hachage **key** , et **earlyKey** , sont calculées à partir des chaînes de caractères nommées et elles permettent de faire des comparaisons simples à l'heure d'établir la correspondance entre un dialogue et un message donné. Les algorithmes utilisés pour cette tâche sont plutôt simples et linéaires, avec la présomption que le nombre de dialogues simultanés à gérer sera souvent limité. Une précaution simple est de placer les nouveaux dialogues, présumés d'avoir plus d'activité, à la tête de la liste.

L'évolution des dialogues est contrôlée avec une machine à états finis. La figure 7.22 présente une version simplifiée d'une telle machine où certains états et événements ont été omis afin de simplifier le dessin. Différemment des machines à états finis utilisées pour contrôler les transactions, celle-ci a été implantée en utilisant une table de fonctions d'état tandis que les événements sont gérés avec des *switch-case*. La machine s'exécute dans une tâche indépendante synchronisée avec l'arrivée des événements. Ceux-ci proviennent principalement du module de transactions, de la couche d'application et du gestionnaire de réception de messages.

D'autres actions de la MEF de dialogues ont comme destination d'autres modules à l'interne de la pile. Dans ces cas, le mécanisme de communication préféré est d'envoyer des messages-événements. Il faut noter que la majorité des messages SIP générés dans la pile proviennent de la couche de dialogue. En général, ces messages ont comme destination la couche de transport ou les transactions.

```
struct erSipDialogStat{
                                          /* dialog id info                          */
    erINT16U                dialogID;     /*   dialog ID                             */
    erCHAR                 *call_id;      /*   call ID                               */
    erCHAR                 *localTag;     /*   To tag                                */
    erCHAR                 *remoteTag;    /*   From tag                              */
    erINT32U                key;          /*   dialog key                            */
    erINT32U                earlyKey;     /*   dialog early key (whitout remoteTag)*/
    erINT16U                stat;         /* dialog stat                             */
    erINT16U                clientSrvFlg; /* Flag: client = 0; server = 1;           */
    erBOOLEAN               secure;       /* secure flag                             */
    erSipPartyInfo_t       *local;        /* local parameters                        */
    erINT16U                lcalsqn;      /*   local sequence number                 */
    erSipPartyInfo_t       *remote;       /* remote parameters                       */
    erINT16U                rmotsqn;      /*   remote sequence number                */
    erSipUri_t             *remoteTarget; /*   remote target URI (to request URI)  */
    erMessage_t            *lastRxMssgP;  /* received message                        */
                                          /* general parameters                      */
    erINT16U                maxForwards;  /*    max Forwards                         */
    sipTrnsProt_e           trnsprtProto; /*    transport protocol                   */
    erINT16U                trnsprtId;    /*    transport Id                         */
    erINT16U                curTransact;  /*    current transaction                  */
    erINT16U                cancelingFlg; /*    canceling flag                       */
    erINT16U                trnsactId;    /*    transaction Id                       */
    sipMethdId_e            method;       /*    current method                       */
    void                   *mssgBody;     /*    message body                         */
    erCHAR                  mssgBodyLen;  /*    message body length                  */
    erSipRouteUri_t        *routeSet;     /* route set                               */
    erINT16U                routeSetFlg;  /* route set flg: if set: skip the first */
                                          /*   route uri                             */
                                          /*     and append the remote target        */
    struct erSipDialogStat *next;         /* next dialog in the list                 */
    };

typedef struct erSipDialogStat erSipDialogStat_t;
```

Figure 7.21: Structure de l'état du dialogue

Un dialogue peut être créé en réponse à un événement interne, tel qu'une demande de la couche d'application pour établir une session, ou externe, tel que l'arrivée d'une requête INVITE. Pour chaque cas, on a prévu une fonction de création de dialogues spécifique. La fonction **sipDialogCreateUAC()** , permet de créer un dialogue comme client tandis que **sipDialogCreateUAS()** permet de le faire comme serveur. Dans les deux cas, la structure établie est essentiellement la même. Les différences entre les deux fonctions se trouvent dans la procédure. Par ailleurs, la destruction de dialogues est réalisée par **erDialogDestructor()** .

Dans la figure 7.22, on observe que les états initiaux de la MEF sont différents selon que l'UA initie le dialogue comme serveur ou comme client. Cependant, l'état «*Active*» est le même dans les deux cas et une fois que le dialogue est dans cet état, la session peut finir par une décision locale, lorsque l'utilisateur raccroche, ou distante, lorsqu'une méthode BYE

est reçue. Dans le premier cas, l'UA joue en rôle de client, alors que dans le deuxième, celui de serveur.

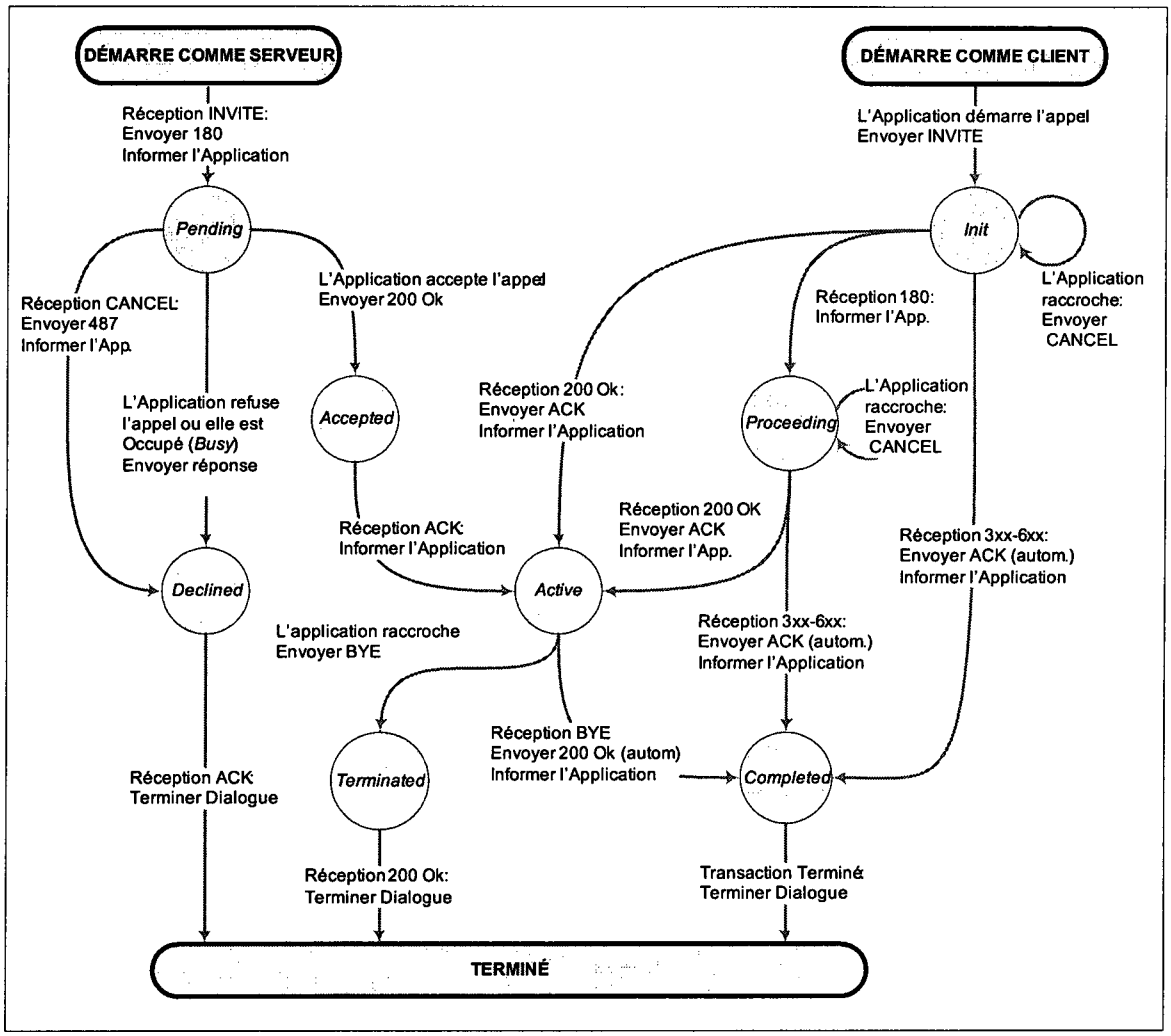

Figure 7.22: Machine à états finis utilisé pour le contrôle des dialogues

7.9 Couche de transport

La couche de transport est chargée de gérer la transmission et la réception de messages SIP sur le réseau. À la différence de la plupart des implantations de SIP, la fonctionnalité de cette couche se limite exclusivement aux tâches qui sont strictement en relation à la transmission et à la réception de messages, avec une totale abstraction de leur contenu. Cette couche

définit des objets de transport (*transports*) qui contiennent l'état des communications vers une destination donnée. Ceci permet que, une fois qu'un *transport* est créé, les couches supérieures ne doivent qu'inclure une référence à lui, son numéro d'ID, dans leurs appels aux méthodes de transport et la couche de transmission sera capable de gérer tous les détails de la transmission. Les *transports* sont organisés dans une liste simplement chaînée, la figure 7.23 montre la définition de la structure qu'ils utilisent.

Deux types de transport ont été définis : les éphémères et les permanents. Les transports éphémères se détruisent automatiquement après un certain temps d'inactivité et ils sont utilisés pour le trafic général. Les transports permanents, par contre, restent actifs pendant tout le fonctionnement de la pile. C'est le cas des transports ouverts par défaut pendant l'initialisation : une écoute (*listener*) TCP, et un port UDP, tous deux sur le port 5060.

```
struct erSipTrnsport{
    erINT16U              trnspID;          /* transport Id                          */
    erINT16U              users;            /* users reference counter               */
    struct erSipTrnsport  *parent;          /* parent struct (used with UDP)         */
    erINT32U              keyVal;           /* transport key                         */
    sipTrnsProt_e         trProto;          /* protocol = UDP / TCP / TLS e SCTP     */
    erINT16U              socket;           /* socket                                */
    sipTrnsportStat_e     stat;             /* stat = WAITING/BINDED/CONNECTED/ERROR*/
    erINT16U              clientSrvFlg;     /* Flag: client = 0; server = 1          */
    erCHAR                remoteIpAdd[16]; /* remote IPadd (dotted decimal notatat)*/
    erINT16U              remotePort;       /* remote port                           */
    struct sockaddr_in    remoteSockAdd;    /* remote socket address                 */
    erINT16U              localPort;        /* local port                            */
    struct sockaddr_in    localSockAddr;    /* local socket address                  */
    erINT16U              PMTU;             /* path MTU                              */
    erINT16U              timeout;          /* time to close the Transport           */
    struct erSipTrnsport  *next;            /* list pointer                          */
    };

typedef struct erSipTrnsport    erSipTrnsport_t;
```

Figure 7.23: Structure de transports

Pendant la procédure de transmission, avant de passer un message à la couche de transport pour sa transmission, l'émetteur (typiquement la couche de transactions ou la couche de dialogues) doit acquérir un *transport* vers l'adresse et le port de destination spécifié. Cela est fait à l'aide de la fonction **erSipGetTrnsportID()** . D'autres paramètres sont le protocole de transport (UDP ou TCP) et le rôle de l'UA (client ou serveur). Si nécessaire, cette fonction résout l'adresse IP avec le client DNS. Puis, elle cherche dans une liste s'il y a déjà un transport qui satisfait les conditions. Le cas échéant, un nouveau transport est créé.

La transmission est faite d'une façon synchronique. Les messages à transmettre sont arrangés dans une file d'attente. La liste est parcourue cycliquement afin de vérifier si le *transport* spécifié dans chaque message est prêt (état = *READY*). Dans ce cas-ci, elle enlève le message de la liste et le transmet.

Du côté de la réception, une méthode fait cycliquement le *polling* des *sockets* liés aux *transports* afin de vérifier s'il y a des messages qui viennent d'arriver. Les messages reçus sont mis en file d'attente et leur arrivée est signalée aux couches supérieures avec une fonction de rappel **cback_onRcvedMssg()**. Cette fonction signale un sémaphore qui réveille la tâche du gestionnaire de réception.

Il faut mettre en évidence que le RFC 3261 (Rosenberg et al., 2002) établit que toutes les implémentations de SIP doivent supporter, au minimum, les protocoles de transport UDP et TCP. Dans le cadre de ce projet, seule l'utilisation d'UDP a été prévue, cependant la couche de transport a été conçue de manière à ce que l'incorporation de TCP soit presque immédiate.

Comme on a déjà mentionné dans la description de l'architecture (section II, architecture générale), on a adopté comme critère de conception que la couche de transport doit faire abstraction du contenu des messages. Autrement dit, les messages sont traités comme des blocs de texte sans les modifier et sans les interpréter. Ce critère oblige à ajouter des méthodes d'interface qui fournissent des informations additionnelles aux couches supérieures.

7.10 Système de temporisation

L'implantation de SIP demande de nombreux temporisateurs. Dans ce chapitre, on a déjà signalé ceux correspondants aux transactions. Cette section présente, d'une façon plus générale, l'approche adoptée pour résoudre les différents aspects liés à la temporisation du protocole.

Durant l'élaboration d'une solution pour cette problématique, plusieurs aspects ont été considérés. Parmi les plus importants, on souligne les suivants :

- **Résolution**. En général, les intervalles de temps qui sont mis en jeu dans le protocole SIP sont plutôt longs. Par exemple, pour les MEF, l'intervalle de temps le plus petit spécifié est de 500 ms. En conséquence, une résolution de 10 ms a été considérée comme

```
void tmrCtrlTsk (void *pData)
{
    erINT16U    ii;
    tmrReg_t    *pp;

    pData = pData;
    while (TRUE) {                                  /* Task body                          */
        erSemaphPend(sipTmrrSem,0);                 /* wait the sync semaphore            */
        pp = &tmrV[0];                              /* pp point to the top of the tab     */
        for(ii = 0; ii < MAX_TMR_REG; ii++){        /* for every entry in the tab         */
            if( pp->tFn )                           /*   if callback defined              */
                if( --(pp->tcnt) <= 0  ){           /*      update (dec) the prescaler    */
                    pp->tcnt = pp->tval;            /*      if presc == 0: reload it      */
                    (pp->tFn)();                    /*           call callback fucntion   */
                }
            pp++;                                   /* next tab entry                     */
        }
       }
}
```

Figure 7.24: Tâche utilisée pour générer la temporisation

adéquate. Avec cette valeur, il est possible de combiner une granularité temporelle suffisamment petite avec une surcharge du système tout à fait acceptable.

- **Regroupement fonctionnel**. Une analyse des différents temporisateurs utilisés par SIP montre qu'ils peuvent être regroupés, selon un critère fonctionnel, dans un nombre réduit d'ensembles. Ceci permet de les traiter de manière uniforme et systématique, avec des gains en efficacité en termes de taille du code et du temps d'exécution.
- **Réduction de redondance**. Plusieurs temporisateurs, qui sont équivalents en termes de fonctionnalité, ne coexistent pas dans le temps. En conséquence, ils peuvent être regroupés en ensembles liés à un unique temporisateur réel. Cette solution est combinée avec l'emploi de macros afin de maintenir la nomenclature originale de la recommandation et de conserver la lisibilité du code.

À l'égard de ce que l'on vient d'exposer, la solution finalement adoptée est orientée vers le traitement en lots des temporisateurs avec des caractéristiques communes par des fonctions locales.

Au cœur du sous-système de temporisation on trouve la fonction montrée dans la figure 7.24. Elle est exécutée à chaque 10 millisecondes dans une tâche indépendante. Un sémaphore sert à synchroniser son fonctionnement avec une référence temporelle absolue prise auprès du système d'exploitation. Le sémaphore est du type compteur, car il permet de rattraper

```
typedef struct {
    void            (*tFn)(void);                   /* timer-adjustment fucntion   */
    erINT16S        tcnt;                           /* prescaler counter           */
    erINT16S        tval;                           /* prescaler reload value      */
} tmrReg_t;

erINT16U  tmrRegFn (void        (*fn)(void),
                    erINT16U    tVal)
{
    erINT16U    ii;
    erINT16U    error;
    tmrReg_t   *pp;
    OS_CPU_SR   cpu_sr;

    OS_ENTER_CRITICAL();
    error = EC_TMR_TABFULL;
    pp = &tmrV[0];
    for( ii = 0; ii < MAX_TMR_REG; ii++){           /* search for an empty register */
        if( !pp->tFn ){
            pp->tcnt = tVal;                        /* set register                 */
            pp->tval = tVal;
            pp->tFn  = fn;                          /* fn must be the last item     */
            error = EC_ERRNONE;  /* set error to NONE    */
            sipTmrEna = 1;
            break;
        }
        pp++;
    }
    OS_EXIT_CRITICAL();
    return error;
}
```

Figure 7.25: Enregistrement des fonctions locales pour l'ajustement des temporisateurs

le temps du système dans des conditions éventuelles de surcharge. La priorité de cette tâche est basse, en cohérence avec les considérations faites sur la résolution temporelle. Le sémaphore est signalé par une fonction de rappel (*callback*) enregistrée dans un crochet (*hook*) de la routine d'interruption du temporisateur du système.

Pendant l'initialisation du système, chacun des modules utilisateur du sous-système de temporisation doit enregistrer sa propre fonction locale d'actualisation des temporisateurs. La figure 7.25 montre la fonction d'enregistrement et la structure de données utilisée. L'analyse de cette dernière montre qu'elle ne contient que trois éléments : le pointeur vers la fonction locale d'actualisation des temporisateurs, un compteur «*prescaler*» et sa valeur de recharge. Ces structures sont stockées dans une table dont sa taille est fixée au moment de la compilation. Chaque fois que la tâche de temporisation est réveillée, elle parcourt la

table afin d'actualiser les compteurs *prescalers*. Lorsqu'ils deviennent égaux à zéro, ils sont remis en marche avec leur valeur de recharge et la fonction enregistrée est appelée.

Les fonctions locales de temporisation sont chargées de faire l'actualisation des temporisateurs et de déclencher les événements lorsque les temporisateurs expirent. Par exemple, la figure 7.26 montre la fonction qui actualise les temporisateurs des transactions. Elle utilise un itérateur pour parcourir la liste de transactions actives et, pour chacune d'elles, actualise les temporisateurs associés. Lorsqu'un temporisateur expire, un message-événement est généré. Un sémaphore *mutex* est utilisé afin de régler le problème d'accès concourant aux temporisateurs.

```
void erTrnxTmrAdj(void)
{
    erSipTransactn_t    *nxtFSM;

                                                /* execute every 10 ms          */
    erMutexPend(sipFSMmutex, 0);                /* adquire the mutex            */
    nxtFSM = trnxList;
    while( nxtFSM){                             /* adjust timers for every FSM  */
        if(nxtFSM->tmr_1)                       /*   timer 1                    */
            if( !--nxtFSM->tmr_1 )
                erTrnsNewEvnt(erTEvFIRED_TMR_1, nxtFSM->trnxID, NULL);
        if(nxtFSM->tmr_2)                       /*   timer 2                    */
            if( !--nxtFSM->tmr_2 )
                erTrnsNewEvnt(erTEvFIRED_TMR_2, nxtFSM->trnxID, NULL);
        if(nxtFSM->tmr_3)                       /*   timer 3                    */
            if( !--nxtFSM->tmr_3 )
                erTrnsNewEvnt(erTEvFIRED_TMR_3, nxtFSM->trnxID, NULL);
        if(nxtFSM->tmr_4)                       /*   timer 4                    */
            if( !--nxtFSM->tmr_4 )
                erTrnsNewEvnt(erTEvFIRED_TMR_4, nxtFSM->trnxID, NULL);
        nxtFSM = nxtFSM->nxtTrnx;
    }
    erMutexPost(sipFSMmutex);                   /* release the mutex            */
}
```

Figure 7.26: Fonction d'actualisation des temporisateurs des machines à états finis

Cette solution présente des avantages face à l'alternative d'utiliser des objets temporisateurs de caractère général. En premier lieu, le code final est plus compact, car on profite des caractéristiques communes des groupes de temporisateurs. En deuxième lieu, les schémas d'exclusion mutuelle deviennent plus simples et économiques en termes de ressources du système d'exploitation. En troisième lieu, les temporisateurs peuvent être embarqués dans les structures qui les utilisent et, en plus d'être créés et détruits automatiquement avec elles, ils deviennent susceptibles d'être traités en lots (*«batch processing»*).

7.11 Gestionnaire d'événements

L'implantation de SIP conçue dans ce projet requiert plusieurs tâches. Pour son fonctionnement, ces tâches doivent se synchroniser et échanger des informations. Cette problématique a été résolue à l'aide des files d'attente de messages-événements (*event message queues*).

Cette approche rend possible la communication asynchrone entre des tâches et synchronise leur fonctionnement avec l'occurrence des événements pertinents.

Le gestionnaire de messages événements utilisé dans la pile SIP a été conçu spécifiquement pour cette application. Les fondements de cette décision sont mentionnés un peu plus loin dans cette section. Sa conception a été faite en deux niveaux. Le niveau le plus bas correspond aux fonctions génériques pour la gestion de la file d'attente alors que le niveau le plus haut est formé d'une «façade» embarquée dans chaque module utilisateur, qui personnalise les types d'événements et permet une meilleure gestion des erreurs. La figure 7.27 montre les structures utilisées pour la définition d'événements et par le contrôle de chaque file d'attente.

Il faut dire qu'une solution alternative aurait été d'utiliser le service de «*Message Queue*», fourni par MicroC/OS-II. Cependant, et selon le point de vue de l'auteur, l'approche prise présente des avantages, car elle permet de profiter de certaines caractéristiques particulières de l'application. Par contre, une solution générale, telle que celle fournie par MicroC/OS-II, est conçue pour être appliquée dans le cas plus général tout en rendant le temps d'exécution prédictible. Parmi les facteurs considérés qui justifient cette décision, on peut mentionner les suivants :

- La taille des messages est fixe et connue. Ainsi, la file d'attente peut être implémentée avec une table de structures d'événements plutôt qu'avec une table de pointeurs, en économisant un niveau d'indirection.
- Le style de conception de cette pile SIP priorise la réduction de la taille de l'application face à la vitesse d'exécution. Ce fait, qui s'ajoute à la connaissance a priori de certaines caractéristiques de l'application, permet de simplifier la conception du gestionnaire. Le coût, par contre, est une certaine perte de généralité.
- Le travail nécessaire pour la conception du gestionnaire est comparable à celui requis pour la conception de la couche d'abstraction et les adaptations requises par «*Message Queue*».

- L'utilisation d'une solution embarquée dans la pile améliore sa portabilité.

```
typedef struct{
    erINT16U            type;                       /* event type                  */
    erINT16U            evntParam1;                 /* event parameter #1          */
    erINT16U            evntParam2;                 /* event parameter #2          */
    void               *dataP;                      /* event general data pointer  */
    }erSipEvnt_t;


/**
 * @struct  erSipEvntBuff_t
 * @brief   Control struct of the Sip event buffer
 *
 */
typedef struct{
    erINT16U            headNdx;                    /* head index                  */
    erINT16U            tailNdx;                    /* tail index                  */
    erINT16U            bufSize;                    /* buffer size                 */
    erSipEvnt_t       **buf;                        /* event buffer pointer        */
    erHANDLE            semaph;                     /* buffer semaphore            */
    }erSipEvCtrl_t;
```

Figure 7.27: Définition des structures d'événements et de contrôle de la file d'attente

7.12 Conclusion

La conception de cette pile SIP a demandé un travail considérable. Elle compte plus de 10000 lignes de code source et de nombreuses heures investies sur des tâches complémentaires toujours présentes lors de la conception des systèmes embarqués. Mais cet effort a été bien récompensé par les résultats obtenus. La taille du code compilé reste en dessous de 75 K-octets pour l'architecture ARM. Une valeur qui, même en considérant les parties qui restent à implémenter, reste plus qu'intéressante pour ce type d'implantation sur système embarqué.

Parmi les sujets qui restent à compléter, le support de TCP dans la couche de transport et la gestion du protocole SDP sont les plus prioritaires et ils devront être incorporés parmi les premières itérations.

En résumé, il reste encore beaucoup de travail à faire. Cependant, la pile SIP conçue dans cette partie du projet se présente comme une option très intéressante et avec beaucoup de potentiel à l'heure d'utiliser SIP sur des systèmes embarqués.

CHAPITRE 8

MISE EN ŒUVRE DE RTP

La mise en œuvre des applications temps réel présente toujours des défis particuliers où l'efficacité et la vitesse d'exécution jouent un rôle central. De plus, dans le cas des applications audio, les défauts de conception et le manque de qualité sont rapidement perçus par les utilisateurs. Ainsi, une conception soignée s'impose afin d'arriver à des résultats qui soient au dessus du seuil d'acceptabilité. Lorsqu'il s'agit de la conception d'une librairie de base, le contexte est plus restreint encore, en ajoutant de fortes conditions sur la versatilité, l'extensibilité et la robustesse.

Ce chapitre décrit l'implémentation du protocole RTP fait dans le cadre du projet, en expliquant avec détail, tous les points qui ont mérité plus d'attention, tel que l'implémentation du tampon anti-gigue, par exemple. Comme d'habitude lorsqu'on travaille avec des systèmes embarqués, une bonne quantité de travail additionnel a été nécessaire afin de bâtir des modules complémentaires requis par le protocole RTP, tels quels les pilotes de bas niveau pour la capture et la restitution audio.

Dans la première partie de ce chapitre, on fait une analyse plutôt générale de la mise en œuvre du RTP. La deuxième partie revient sur les objectifs et la portée imposés au projet. La troisième et la quatrième parties donnent une description détaillée de l'émetteur et du récepteur. Finalement, dans la conclusion, sont mentionnés les résultats observés, en faisant un bilan de cette partie du projet.

8.1 Analyse de la mise en œuvre du protocole RTP

La figure 8.1 montre un possible schéma-bloc illustrant les processus de transmission et de réception d'audio temps réel avec le protocole RTP. En restant dans la généralité, un codec G.729 fut choisi pour exemple. La partie supérieure représente l'émetteur, celle inférieure, le récepteur. Le diagramme est divisé en trois zones, où chacune d'elles correspond à un ensemble de fonctions qui, dans le cas de systèmes d'exploitation multitâche, sont typiquement exécutées dans des tâches ou fils d'exécution (*threads*) séparés et synchronisés

avec des événements différents. Les sections suivantes décrivent les caractéristiques les plus importantes de chaque bloc.

8.1.1 Comportement d'un émetteur RTP

La fonction de l'émetteur, exprimée en termes très généraux, est de capturer l'information audio, de l'encoder dans un format plus compressé, de générer les paquets RTP et de les transmettre. Éventuellement, l'émetteur peut modifier son comportement à la demande du destinataire. Par exemple, il peut changer le codec ou prendre des mesures en cas de congestion.

Le signal audio provenant du microphone, ou d'une autre source externe, est échantillonné et capturé dans un des tampons de capture. Ceux-ci regroupent les échantillons dans des blocs, appelés trames, afin de simplifier le traitement postérieur. La taille des trames est habituellement spécifiée par sa durée équivalente exprimée en millisecondes, car cela permet d'établir une liaison plus directe avec les retards du traitement et du stockage dans les tampons. Typiquement, les échantillons sont encodés en utilisant une échelle linéaire de 16 bits et prises au rythme de 8000 échantillons par seconde.

Les trames disponibles à la boucle de capture sont traitées par les algorithmes de détection de silences et ensuite par le codec, qui génère des trames compressées. Pour une implémentation donnée, il y a une variété de codecs disponibles avec différents niveaux de compression et de complexité. Celui qui sera utilisé dans une session donnée est déterminé dans la phase de négociation avec l'autre terminal. Mais il peut changer dynamiquement. Pour chaque trame, le système choisira le codec plus approprié pour le type de signal à encoder parmi ceux qui sont disponibles aux extrêmes émetteur et récepteur. Dans la pratique, ce choix ne se produit qu'entre deux types de codecs, un plus adapté à la compression de la parole et l'autre pour d'autres types de signaux. Par exemple, pendant que le signal à compresser est la parole, le système peut utiliser un codec G.729, mais si dans un moment donnée, une signalisation DTMF est envoyée, il faudra changer vers un codec à G.726, car G.729 est incapable de gérer ces types de signaux adéquatement. Les facteurs de compression sont souvent très importants. Par exemple, la taille d'une trame de 10 ms avec des échantillons de 16 bits est de 160 octets, mais d'après la codification G.729 elle ne prend que 10 octets.

L'étape suivante est d'encapsuler les trames compressées dans des paquets RTP. Dans ce but, le module «Codeur RTP» génère l'en-tête des paquets. Dans certaines situations, un

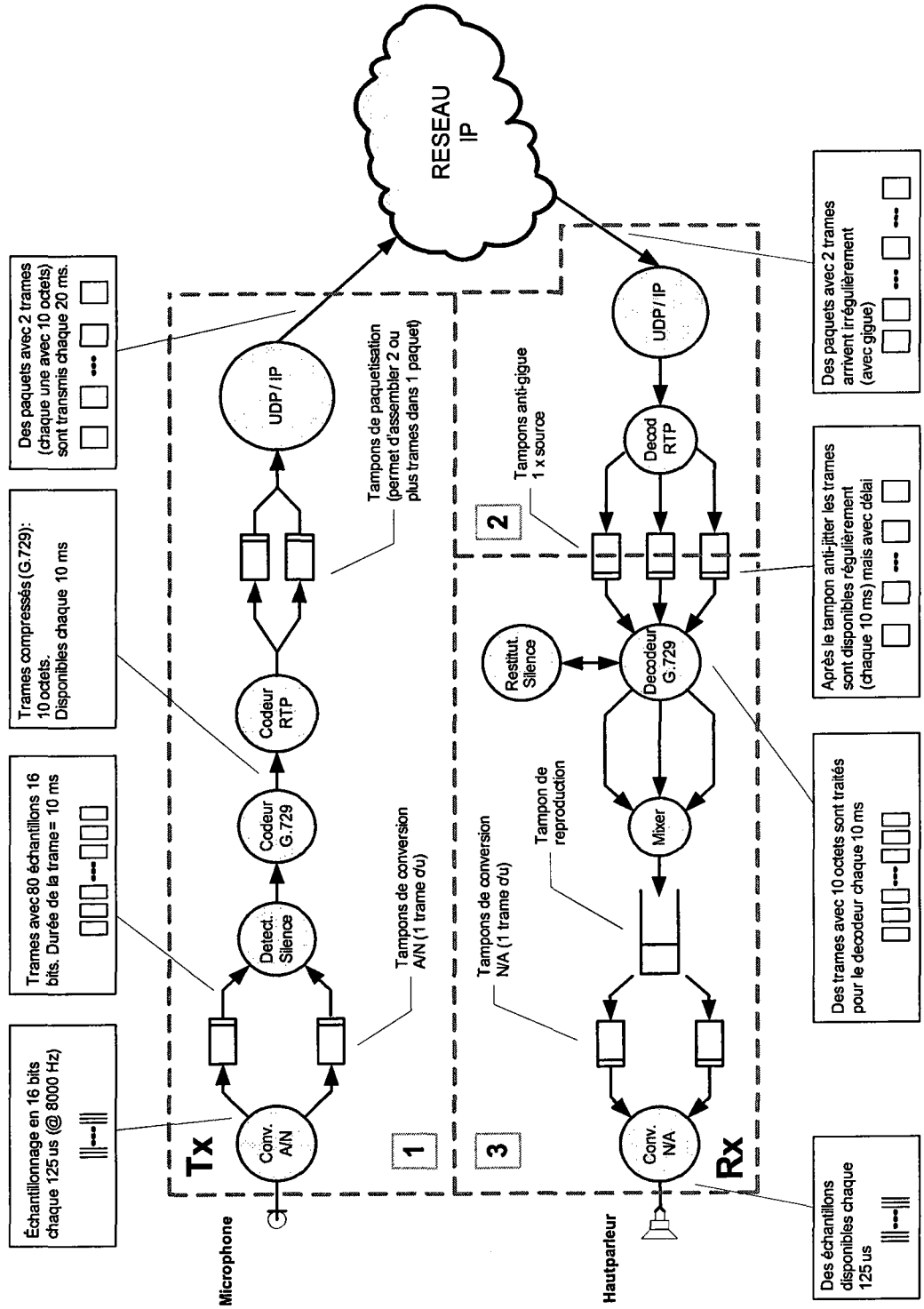

Figure 8.1: Processus de transmission et de réception de paquets RTP

paquet peut inclure plus d'une trame. Par exemple, pour le codec G.729, chaque paquet inclut deux trames. La tâche d'amasser les trames, en attendant que le nombre spécifié soit atteint, est accomplie par les tampons de paquetisation.

Finalement, le module de transport est chargé d'envoyer les paquets vers le(s) destinataire(s) en utilisant le protocole UDP. L'émetteur est aussi responsable de générer périodiquement des rapports d'information d'état et de synchronisation.

8.1.2 Comportement d'un récepteur RTP

Le fonctionnement du récepteur est décrit en termes de fonctions qui réalisent des opérations complémentaires à celles de l'émetteur, mais en incorporant des mécanismes pour compenser les effets dérivés du transit des paquets par le réseau (pertes, délais, duplications, etc.).

La première étape du processus de réception est celle de collecter les paquets RTP. Avant que les paquets provenant d'une source déterminée soient acceptés par la pile, la source doit être validée en suivant une procédure spécifique définie dans le RFC 3550 (Schulzrinne et al., 2003). Des structures de contrôle et des tampons anti-gigue sont créés dynamiquement pour chaque source RTP qui a été validée.

Dans une deuxième étape, les paquets acceptés sont décortiqués afin d'analyser l'information de séquence, de temporisation, ainsi que l'origine du paquet. En même temps, l'information statistique qui sera reportée à l'émetteur est générée.

La troisième étape consiste à envoyer les paquets vers le tampon anti-gigue correspondant. Normalement, ces tampons-ci contiennent un nombre minimum de paquets qui leur permettent de compenser les variations dans leur rythme d'arrivée. Dans certains cas, ils ont un comportement adaptatif, en contrôlant le délai qu'ils introduisent en fonction de la gigue estimée. C'est la stratégie utilisée dans le présent projet.

À continuation, le décodeur prend les trames des tampons anti-gigue à un rythme régulier pour les décompresser. Lorsqu'il y a plus d'un flot simultané, le mélangeur permet de combiner l'information audio provenant des différentes sources. Le tampon de reproduction sert à adapter les flots audio. Son inclusion ou non dépendra de chaque implantation.

Finalement, la boucle de restitution est chargée de faire la conversion numérique-analogique dans le but de récupérer le signal audio. Similairement à la boucle de capture, elle travaille en alternant l'utilisation de deux tampons.

Il est évident que le processus de réception est plus complexe que celui de transmission. L'origine de cette complexité se trouve dans le fait que c'est le récepteur qui doit traiter les erreurs et compenser les incertitudes ajoutées après le transit des paquets sur le réseau.

8.1.3 Temporisation et synchronisation des tâches dans RTP

À la différence de ce qui arrive avec le protocole SIP, dans RTP la temporisation joue un rôle central. La durée de la trame détermine l'intervalle de temps dans lequel toutes les tâches accomplies par l'émetteur ainsi que par le récepteur doivent être achevées. Dans la figure 8.1, les blocs sont regroupés en trois zones. Chacune d'elles correspondant à une source de synchronisation différente.

La zone 1, associée à l'émetteur, est synchronisée avec la génération des trames par la boucle de capture. Chaque fois que cette dernière rend disponible une nouvelle trame, il faut la traiter et la transmettre le plus rapidement possible afin d'éviter des pertes de temps inutiles qui contribueront à la latence. Comme référence, on peut prendre des valeurs typiques de 10 à 20 millisecondes pour l'intervalle qui s'écoule entre la génération de deux trames, dépendamment du codec choisi. La zone 2 regroupe les fonctions liées à la réception, la validation et l'acheminement interne des paquets RTP. Son fonctionnement est synchronisé avec l'arrivée des paquets. Les tampons de réception anti-gigue établissent la liaison entre le fonctionnement des zones 2 et 3.

Par ailleurs, le récepteur doit fournir des trames au convertisseur numérique-analogique de façon régulière. Cette condition détermine une autre zone qui doit opérer à la demande du processus de conversion et être synchronisé avec lui. C'est la zone 3. Dans la pratique, lorsqu'on travaille avec des systèmes d'exploitation multitâches, les fonctions comprises dans chaque zone sont accomplies par des tâches séparées. C'est l'approche choisie pour ce projet. L'accès concurrent des fonctions des zones 2 et 3 aux tampons anti-gigue demande l'emploi de sémaphores du type mutex.

8.2 Architecture

Dans la figure 8.2 on retrouve un schéma-bloc qui décrit l'architecture de la pile RTP. Les objectifs de la plupart des blocs ont été expliqués pendant la description des processus de

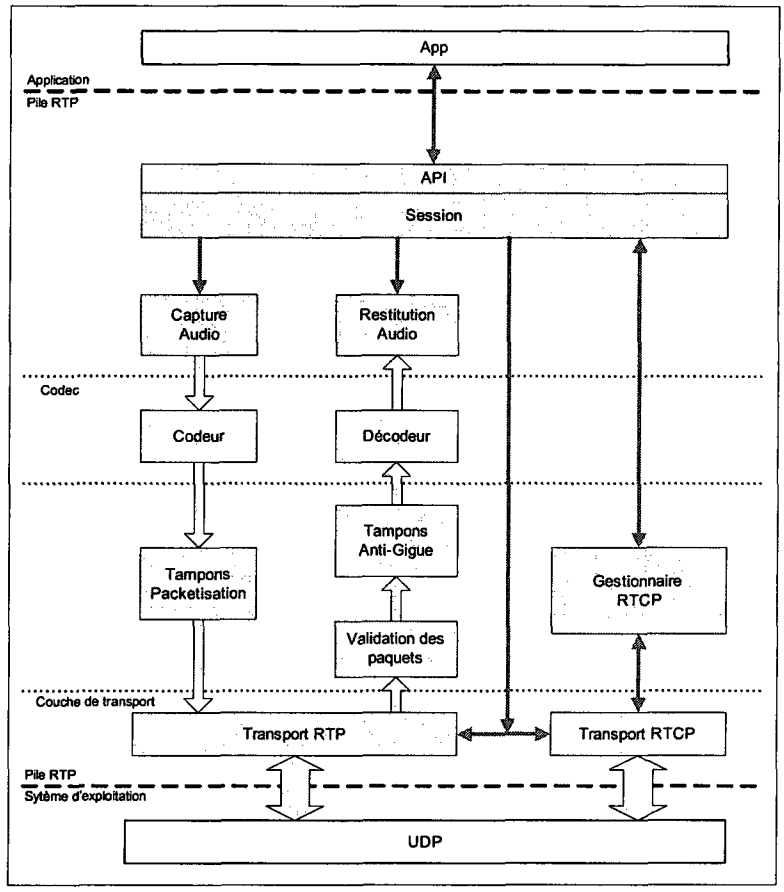

Figure 8.2: Architecture de la pile RTP

réception et de transmission. La figure met en évidence la couche de Session, qui implémente l'API de la pile.

8.3 Transmission

8.3.1 Boucle de capture audio

Le module de capture audio est responsable de prendre la sortie d'une source externe, par exemple un microphone, la numériser avec un format de 16 bits et la rendre disponible au codeur afin de la compresser.

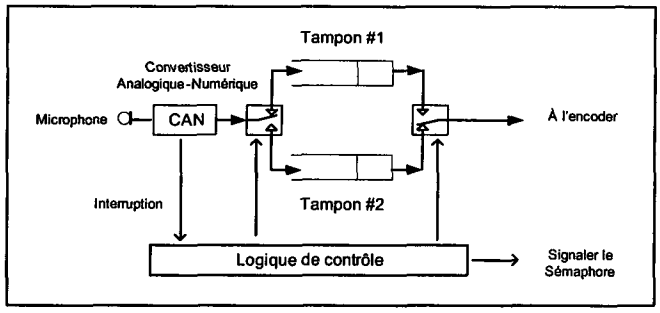

Figure 8.3: Capture audio

À la figure 8.3, on observe la structure utilisée. L'entrée audio est numérisée en 16 bits par le convertisseur. Deux tampons sont utilisés. C'est-à-dire que pendant que les échantillons sont insérés dans un des deux tampons par un dispositif d'accès direct à la mémoire, le DMA (*Direct Memory Acces*), le codeur traite les données dans l'autre. Le premier est appelé tampon actif et lorsqu'il devient plein, le DMA génère une interruption qui permet à la logique de contrôle de commuter le tampon actif et de signaler un sémaphore afin de déclencher le fonctionnement de l'émetteur.

L'implémentation de ce module est très dépendante de la couche matérielle sous-jacente. La définition d'un ensemble simple de fonctions d'interface (API) a permis d'isoler cette couche et ses pilotes de bas niveau. Les fonctions de capture comprises dans cet API sont montrées dans la figure 8.4.

```
void    inChnnlInit      (frSampl_e samplRate,
                          INT16U  bufSize,
                          void   (*onBufDone)(void));
void    inCnnlClrBuf     (void);
INT16S  inChnnlStart     (void);
INT16S  inChnnlStop      (void);
void    inChnnlReset     (void);
void   *inChnnlNxtBuf    (void);
void   *inChnnlBufInUse  (void);
```

Figure 8.4: API du pilote de capture audio

Toutes les fonctions en relation avec la capture de l'audio ont le préfixe «inChnnl». Il a été considéré comme acquis qu'il est possible d'échantillonner une signal audio avec une résolution de 16 bits, aux vitesses demandées en utilisant le DMA avec des interruptions.

La fonction **inChnnlInit()** permet d'initialiser le système de capture. Elle prend comme arguments la vitesse d'échantillonnage, la taille des tampons et un pointeur vers une fonction de rappel (*callback*), qui sera appelée chaque fois que le remplissage de chaque tampon de capture est complété. Afin de simplifier, il a été supposé que la conversion est toujours faite avec 16 bits, ainsi la taille des échantillons peut être exclue de la liste d'arguments. La taille des tampons de capture est définie à partir des besoins particuliers de chaque codec et elle est exprimée en échantillons. Le pointeur **onBufDone** permet d'appeler une fonction de rappel afin de synchroniser le fonctionnement du transmetteur RTP. Chaque fois qu'une nouvelle trame est prête, cette fonction est appelée pour signaler un sémaphore afin de déclencher l'exécution de la tâche de transmission, qui englobe le traitement de trames et l'envoi des paquets RTP.

La fonction **inChnnlReset()** réinitialise le pilote de capture audio à ses valeurs par défaut et **inCnnlClrBuf()** remplit les tampons de capture avec des zéros. Les fonctions **inChnnlStart()** et **inChnnlStop()** permettent de démarrer et d'arrêter le processus de capture audio.

Finalement, les fonctions **inChnnlNxtBuf()** et **inChnnlBuflnUse()** informent quel est le tampon de capture qui est disponible pour traitement et lequel est en train d'être rempli par le convertisseur. Les pilotes de bas niveau pour les deux plateformes matérielles, la carte Zoom et la carte CerfPDA, utilisées ont été développées en accord avec ce modèle.

La plupart des éléments de temporisation utilisés pour RTP sont relatifs à la période d'échantillonnage. Ceci a amené l'idée de générer la temporisation requise directement à partir du sous-système de capture audio.

```
void    iniSampleTime(INT32U iniVal);
INT32U  getSampleTime(void);
INT32U  getCurrentSTime(void);
```

Figure 8.5: Fonctions de temporisation

les prototypes des fonctions de temporisation sont montrés dans la figure 8.5. La fonction **iniSampleTime()** permet d'initialiser le compteur du système de temporisation avec une valeur aléatoire, comme demandé pour RTP. La valeur retournée par la fonction **getSampleTime()** indique le moment correspondant à la prise du premier échantillon de la trame. Finalement, la fonction **getCurrentSTime()** retourne le temps courant.

8.3.2 Compression audio et génération des paquets RTP

Lorsqu'une trame devient disponible, elle est traitée par le codeur, qui est le chargé de la compresser. La sortie du codec reste dans le tampon **outBuf** afin d'ajouter l'en-tête RTP. Comme cette implémentation n'est destinée qu'à être employée sur des agents utilisateur (UA), l'en-tête n'inclut pas des «*contribution sources*» et sa taille sera fixe et égale à 12 octets. Ceci permet de prévoir sa place au moment d'allouer le tampon en évitant des mouvements de données.

Le codeur RTP est appelé une seule fois par paquet : après la génération de la première trame incluse. Il ajoute la version, le type de codeur, l'identification de la source (*SSRC*), le numéro de séquence et le *timestamp*.

8.3.3 Paquetisation et Transmission

Le RFC 3551 (Schulzrinne and Casper, 2003) recommande que les codecs audio basés sur des trames, doivent être capables d'encoder et de décoder plusieurs trames consécutives dans un même paquet. Dans le projet, cette caractéristique a été implémentée en incorporant une variable qui détermine le nombre de trames à regrouper dans un même paquet. La valeur par défaut est définie dans la structure de définition des codecs, mais il est possible de la modifier dynamiquement si nécessaire. Évidemment, pour les codecs basés sur des échantillons, tels que G.711, cette variable doit rester toujours à 1. Une fois que le nombre de trames par paquet spécifié est atteint, le paquet est passé à la couche de transmission afin de l'envoyer vers la destination en utilisant le protocole UDP / IP.

8.4 Réception

La section de réception est divisée en deux grandes parties qui, dans le contexte d'un système d'exploitation multitâche, sont exécutées dans des tâches séparées. D'un côté, on a l'ensemble des modules chargés de la capture et de la validation des paquets RTP. À ce point, tout le traitement est fait au niveau de paquets, en faisant abstraction du type de codec. D'un autre côté, on trouve les modules chargés du décodage et de la restitution audio, où le travail est fait plutôt au niveau des trames. Les liaisons entre ces deux parties sont établies par les tampons anti-gigue, qui jouent un rôle très important dans cette implémentation.

En principe, les tampons anti-gigue doivent accomplir deux fonctions principales. La première est celle d'éliminer ou de réduire l'effet de la gigue, c'est-à-dire d'adapter le traitement fait selon le rythme irrégulier d'arrivée des paquets au fonctionnement synchrone imposé par la conversion numérique à analogique. Afin d'optimiser le point de travail, les tampons ont un comportement adaptatif. La deuxième fonction est d'adapter le mode de travail par paquets, associé aux fonctions de réception, au traitement par trames de la partie de décodage et restitution audio.

Les sections suivantes sont destinées à présenter, de manière détaillée, une description de chacun des modules de réception.

8.4.1 Réception et traitement des paquets RTP

Comme on l'a déjà mentionné dans l'analyse de l'architecture générale, la réception et le traitement des paquets RTP sont faits dans une tache spécifique dont l'exécution est synchronisée avec l'arrivée des paquets RTP. La séquence d'opérations accomplies comprend les étapes suivantes :

1. Capture des paquets RTP
2. Validation des paquets
3. Validation de la source
4. Contrôle de la séquence
5. Estimation de la gigue
6. Génération d'information pour le protocole RTCP
7. Insertion des paquets dans les tampons anti-gigue
8. Contrôle du fonctionnement adaptatif des tampons anti-gigue

Il s'ensuit une brève description de chaque chaque étape.

8.4.2 Capture des paquets RTP

Comme il a déjà été remarqué, cette implémentation ne supporte que UDP pour le transport des paquets RTP. Ainsi, la réception est faite en appelant la fonction recvfrom() en mode bloquant avec temporisation (*timeout*). De cette manière, l'exécution de la tâche reste synchronisée avec l'arrivée des paquets.

La route suivie par les paquets peut traverser différentes entités RTP, telles que mélangeurs, traducteurs et passerelles. Ceux-ci vont masquer les adresses IP d'origine, qui, en conséquence, n'apportent pas une indication fiable de l'identité de la source. Pour cette raison, l'adresse IP d'origine n'est pas retenue et seuls le contenu du paquet RTP et sa taille sont passés aux modules de validation.

8.4.3 Validation des paquets et de la source

À l'arrivée d'un paquet, la première étape consiste à le décoder tout en faisant quelques vérifications simples afin d'écarter ceux non conformes avec RTP. Les différents composants de l'en-tête RTP sont copiés dans une structure montrée dans la figure 8.6 Afin de calculer la gigue, le moment de l'arrivée est enregistré dans la variable **arrival**. Aux fins de RTP, le temps est mesuré de façon relative en utilisant la période d'échantillonnage comme unité.

```
typedef struct{
    erINT16S            pktLen;             /* packet lenght                    */
    void               *pBuf;               /* buffer for rtp packet reception */
    void               *payload;            /* pointer to the rtp payload      */
    erINT8U             mFlg;               /* marker flag                      */
    erINT16U            payloadLen;         /* payload lenght                   */
    erINT16U            pyldType;           /* payload type                     */
    erINT16U            seqNum;             /* sequence number (16 bits)        */
    erINT32U            ts;                 /* time stamp                       */
    erINT32U            arrival;            /* relative arrival time            */
    erINT32U            ssrc;               /* synch. source                    */
    } rcvedRtpPck_t;
```

Figure 8.6: Structure contenant des paquets arrivés

La deuxième étape correspond à la validation de la source. Chaque fois qu'un paquet provenant d'une nouvelle source est reçu, le système de réception crée une structure et l'ajoute à une liste de sources. Avant d'acheminer ces paquets vers d'autres modules, il faut vérifier leur origine en validant sa source. La procédure de validation consiste à atteindre la réception d'une succession de paquets, provenant de la source à confirmer, avec des numéros de séquence consécutifs. La longueur de la série est prédéterminée et elle est un compromis entre la fiabilité de la validation et le délai introduit. Ce mode de fonctionnement du système de réception est appelé *probation*. Les paquets reçus dans ce mode sont ignorés et ils ne sont pas passés au décodeur. Un mécanisme de *timeout* permet d'éliminer les sources qui n'arrivent pas à êtres validées. L'implémentation de cette partie suit le code exemple qui se trouve dans l'appendice 'A' du RFC 3550 (Schulzrinne et al., 2003).

8.4.4 Contrôle de la séquence

Le numéro de séquence qui voyage avec un paquet RTP n'a que 16 bits. Ainsi, dans l'hypothèse que les paquets sont générés à un rythme de 50 paquets par seconde, il va boucler chaque 22 minutes approximativement. Afin de simplifier les opérations internes, le numéro de séquence est étendu à 32 bits.

Étant donné que les paquets peuvent arriver de manière désordonnée, la procédure d'extension du numéro de séquence n'est pas évidente. Un premier niveau de filtrage permet d'écarter les paquets qui ne sont pas «en séquence». À ce propos, deux «fenêtres d'acceptation» sont définies : **MAX_DISORDER** qui s'étend vers l'arrière du paquet reçu avec le numéro de séquence plus haut et **MAX_DROPOUT** qui s'étend vers l'avant. Des valeurs de 100 et de 3000 respectivement sont suggérées par le RFC 3550 (Schulzrinne et al., 2003). Elles représentent une tolérance de 2 secondes pour les paquets hors ordre et de 1 minute pour les interruptions du flux. Par ailleurs, l'arrivée de 2 paquets hors séquence consécutifs est considérée comme le début d'une nouvelle séquence, en prenant l'hypothèse que la source a été réinitialisée.

8.4.5 Estimation de la gigue

L'estimation de la gigue à deux utilités : en premier lieu, la gigue est reportée à l'émetteur dans les rapports de réception de RTCP. En deuxième lieu, elle est considérée pour établir le point de travail des tampons anti-gigue.

L'algorithme pour l'estimation suit la procédure générale suggérée dans l'appendice A du RFC 3550 (Schulzrinne et al., 2003). Le temps de transit relatif des paquets est calculé comme la différence entre le temps d'arrivée et le *timestamp* des paquets. Comme l'horloge locale n'est pas synchronisée avec celle de la source, le temps de transit ne peut être calculé que d'une manière relative, en incluant un certain décalage. Cela n'est pas cependant un problème parce que la gigue se calcule à partir de la différence des temps de transit des paquets consécutifs, donc ce décalage s'annule. Ainsi, en appelant *delta* cette différence, la gigue se calcule à partir de :

$$transit = arrivalTime - timestamp \tag{8.1}$$

$$delta = transit_n - transit_{n-1} \tag{8.2}$$

$$jitter_n = \frac{15}{16} jitter_{n-1} + \frac{1}{16} delta \tag{8.3}$$

L'équation 8.3 correspond à une approximation de la moyenne mobile calculée à partir des dernières 16 valeurs de *delta*.

L'exactitude de l'estimation de la gigue est un facteur important. Plusieurs algorithmes ont été proposés afin de l'améliorer. En particulier, le calcul de la gigue de phase, soit la différence entre le temps d'arrivée d'un paquet et le temps auquel il était attendu, donne des résultats plus précis. Un autre point qui mérite d'être considéré est la détection et le traitement des valeurs de pointe (*spikes*) car elles peuvent déplacer la moyenne significativement et modifier le point de travail du récepteur. Ces améliorations sont hors de la portée du présent projet et elles pourraient être incorporées dans des versions futures.

8.4.6 Tampons anti-gigue

Led tampond anti-gigue sont des éléments de grande importance dans l'architecture du récepteur. Leur principale fonction est d'enlever la gigue tout en introduisant des retards les plus petits possibles pour un certain niveau de perte des paquets. Ils établissent la liaison entre la section de capture et de traitement des paquets RTP et celle de décodage et restitution audio. Les tampons anti-gigue sont créés en correspondance avec les sources actives, autrement dit, il y a un tampon pour chaque source des paquets RTP.

En plus de leur fonction primaire, qui est d'offrir un flux régulier de trames au décodeur, ces tampons permettent d'adapter le mode de traitement par paquets utilisés dans la partie de réception, au mode de traitement par trames utilisé dans la section de décodage et de restitution audio.

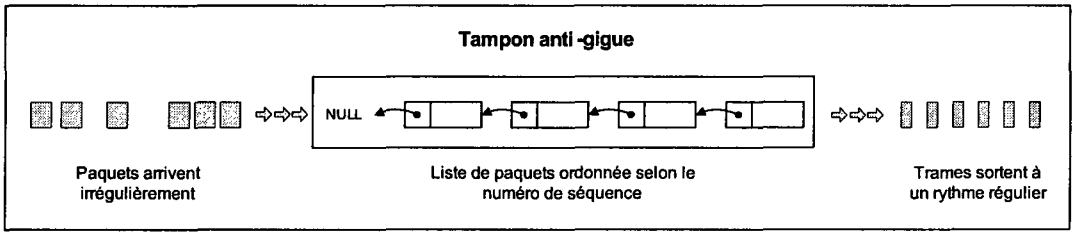

Figure 8.7: Fonctionnement du tampon anti-gigue

Le diagramme de la figure 8.7 illustre le fonctionnement du tampon anti-gigue. Des paquets qui arrivent de manière irrégulière et, peut-être, désordonnée sont ajoutés à une liste chaînée, où ils sont ordonnés selon leur numéro de séquence. La tête de la liste correspond

au paquet le plus ancien et qui est en train d'être décodé. La longueur maximale de la liste est définie par une variable qui permet d'établir une «fenêtre d'acceptation» des paquets. Ceux qui arrivent trop tard ou trop tôt sont écartés.

Le tampon anti-gigue introduit un retard équivalent au temps de restitution audio des paquets qui y sont contenus. Avec le but de minimiser ce retard, le contrôleur du tampon a un comportement adaptatif, en modifiant le point de travail en fonction de la gigue estimée.

8.4.7 Structures du tampon anti-gigue

L'ensemble des variables qui déterminent l'état du tampon est regroupé dans la structure montrée à la figure 8.8.

```
typedef struct{
    erINT16S          curlevel;        /* current level [tSamples]                  */
    erINT16S          thresholdlvl;    /* current threshold level [tSamples]        */
    erINT32S          averagelvl;      /* average level (x 32) [tSamples]           */
    erINT16S          minlevel;        /* minimun threshold level [tSamples]        */
    erINT16S          maxlevel;        /* maximun threshold level [tSamples]        */
    erINT16S          adjlevel;        /* current level compensation term           */
    erINT16U          jitter;          /* estimate jitter [tSamples]                */
    erINT16U          mode;            /* jitter buffer stat: [RUNNING, PRBUFRING]  */
    erINT16U          maxPBufnum;      /* maximun number of packet buffers          */
    erINT16U          bufCount;        /* current number of packet buffers          */
    erINT32U          lastXSeqNum;     /* last sequence number passed to codec      */
    erINT32U          totalRxPcks;     /* total number of frames processed          */
    erINT32U          dropRxPcks;      /* dropped packet count                      */
    erINT32U          missingPckt;     /* missing packet count                      */
    erINT32U          updateCount;     /* parameter update counter                  */
    erHANDLE          mutex;           /* jitter buffer mutex                       */
    jbBufList_t      *jblist;          /* packet buffer list                        */
    } erJBuf_t;
```

Figure 8.8: Structure du tampon antigigue

Voici une définition des principaux éléments de la structure :

- curlevel : temps équivalent des paquets stockés dans le tampon en prenant comme unité la période d'échantillonnage.
- thresholdlvl : niveau de seuil utilisé pour passer en mode de fonctionnement normal. Il est calculé dynamiquement en fonction de la gigue et du niveau moyen.
- averagelvl : moyenne mobile de curlevel sur ses 32 dernières valeurs.
- minlevel, maxlevel : valeur minimale et maximale admissible pour thresholdlvl.
- updateCount : fréquence d'actualisation des paramètres du tampon.
- jblist : liste ordonnée de paquets RTP.

Dans le tampon, les paquets RTP restent dans une file d'attente. Ils sont organisés dans une liste chaînée ordonnée grâce à leur numéro de séquence. La figure 8.9 montre la structure utilisée comme maillon de la liste. Ici, certaines variables méritent une mention spéciale :

```
struct jbBufList {
    struct jbBufList *next;          /* list pointer                    */
    erINT16U          payload;       /* packet payload type             */
    erINT16U          fsize;         /* packet payload size             */
    erINT32U          xseqnum;       /* extended sequence number        */
    erINT32U          ts;            /* timestamp                       */
    erINT32U          arrivalT;      /* arrival time                    */
    erINT16U          frameOffset;   /* frame start offset              */
    void             *dBuf;          /* packet payload pointer          */
    };
typedef struct jbBufList jbBufList_t;
```

Figure 8.9: Structure du maillon du tampon anti-gigue

- payload : type de charge utile en accord avec RFC 3551 (Schulzrinne and Casper, 2003). Elle permet de sélectionner le bon décodeur.
- arrivalT : temps d'arrivée du paquet mesuré de manière relative par une horloge locale. Elle sert à calculer durant combien de temps le dernier paquet inséré est resté dans le tampon avant que le décodeur demande la prochaine trame.
- frameOffset : début de la prochaine trame dans le tampon. Elle permet de séparer la charge utile dans les trames in situ, en évitant des mouvements de données.

8.4.8 Description de l'opération

Le comportement du tampon change selon son état. Deux modes de fonctionnement sont définis : le mode d'emmagasinage et le mode normal. Le premier est sélectionné lorsque le tampon est initialisé ou bien quand il devient vide.

Pendant le mode d'emmagasinage, le tampon reçoit et accumule des trames RTP mais il refuse toutes les demandes d'extraction. Ce mode persiste jusqu'à ce que le délai équivalent aux trames stockées dépasse un seuil préétabli donné par **thresholdlvl**. De cette façon, lorsque le tampon passe en mode d'opération normale, il contient déjà un certain nombre de paquets (et de trames) accumulés.

Une fois dans le mode normal, le tampon accepte les demandes d'extraction de trames qui sont faites, à un rythme régulier, par le décodeur. Le temps équivalent aux trames qui restent dans le tampon détermine le délai qui sert à compenser la gigue. Des éléments de

contrôle permettent de maintenir ce délai au minimum tout en conservant une sortie audio de bonne qualité.

Lorsqu'un nouveau paquet est disponible pour être inséré dans le tampon, un nouveau maillon contenant l'information du paquet est créé. Puis, le maillon est inséré dans la liste. À cet effet, il faut déterminer si le paquet est dans la fenêtre d'acceptation du tampon ou si, par contre, il est arrivé trop tard ou trop tôt. Cette fenêtre est définie à partir du numéro de séquence du paquet qui se trouve dans la tête de la liste et de la longueur maximale de la liste. Ainsi, la condition d'acceptation peut être exprimée comme :

$$NumSeqTête < NumSeq < (NumSeqTête + LongueurMaximaleListe) \tag{8.4}$$

La justification est simple : la tête de la liste correspond au paquet qui est en train d'être décodé. Si le numéro de séquence du paquet arrivant est inférieur à celui de la tête, alors le moment pour le restituer est déjà passé, le paquet a été compté comme disparu et substitué. En conséquence, il ne sera plus utile. La deuxième condition découle simplement de considérer le maximum du nombre de paquets qui peuvent rester dans la liste à un moment donné.

La condition d'acceptation peut être simplifiée un peu plus. Si on considère que les numéros de séquence sont définis comme des entiers sans signe, elle peut s'exprimer comme :

$$NumSeq - NumSeqTête < LongueurMaximaleListe \tag{8.5}$$

Si la condition d'acceptation est satisfaite, le paquet est alors ajouté à la liste en faisant attention à ce qu'il ne s'agisse pas d'une répétition. L'algorithme d'insertion est plutôt simple en partant du fait que, dans une application normale, le nombre de maillons de la liste sera toujours petit (inférieur à 5 ou 6 paquets).

8.4.9 Calcul du niveau instantané du tampon anti-gigue

La finalité du tampon est d'éliminer l'effet de la gigue afin d'obtenir une sortie audio de qualité acceptable. Ainsi, il accumule un certain nombre de trames ce qui lui donne un certain temps de garde ou de protection. Afin d'établir correctement le point de fonctionnement du tampon, il est important de mesurer ce niveau.

Une première approximation serait simplement de prendre le nombre de maillons dans la liste comme une valeur représentative du niveau du tampon étant donné que chacun contient un paquet. Le problème que présente cette méthode est que la conception du récepteur ne peut pas se baser sur l'hypothèse que la durée du paquet est de 20 ms, même si celle-ci est la valeur la plus fréquente. Ainsi, la détermination du temps équivalente devra se faire en considérant le type de codec et le nombre des trames pour chaque paquet.

La solution choisie est de calculer le niveau du tampon en utilisant les *timestamps*. Elle permet d'arriver à des résultats plus précis tout en faisant abstraction du type de codec et avec un niveau de complexité raisonnablement bas.

Pour deux paquets consécutifs, le temps équivalent du premier est égal à la différence entre ses *timestamps*. Ainsi, si dans la file d'attente il y a n paquets consécutifs, le temps équivalent des $n-1$ premières peut se calculer comme la différence entre les *timestamps* du dernier et du premier paquet. Le temps équivalent du dernier paquet doit être estimé. Pour le faire, une possibilité est d'assumer que la durée du dernier paquet est égale à celle du précédent. Cette hypothèse est tout à fait raisonnable à partir du fait que l'émetteur ne change pas du type de codec fréquemment et, par conséquant, il y aura de longues séquences des paquets du même type. Même, en cas de changement du type de codec ou de la taille des paquets, l'erreur sera corrigée à l'arrivée du paquet suivant. Une deuxième possibilité est d'approximer le temps équivalent du dernier paquet par une valeur constante, par exemple de 20 ms. Cette méthode est un peu moins précise que la précédente, mais les résultats sont très acceptables et sa complexité est mineure. La justification est basée sur des considérations d'ordre pratique : la durée équivalente des paquets reste toujours limitée. Elle ne peut pas être trop petite, au risque de générer des congestions, ni trop grande, parce que cela provoquerait une augmentation de la granularité de la latence et les trous provoqués par des paquets perdus deviendraient plus évidents à l'écoute.

Pour le calcul du niveau du tampon, il y a certains facteurs de correction qu'il faut considérer. En premier lieu, dans le cas de codifications basées sur des trames, les paquets seront traités partiellement, une trame à la fois jusqu'à les épuiser. En conséquence, il faut décrémenter du niveau du tampon la partie déjà traitée du premier paquet. Cette correction a été estimée par la formule (8.6).

$$jbufferLevel- = \sum_{n} \frac{DefaultPacketTime}{2^n} \tag{8.6}$$

Où n est le nombre de trames du paquet déjà traitées. Une fois de plus, l'emploi d'un temps par défaut constant permet de faire abstraction du type de codeur. La division par puissances de 2, malgré l'introduction d'une erreur, simplifie les mathématiques en permettant l'utilisation des opérations de décalage (*shift*). Il est facile de démontrer que, en prenant des valeurs réalistes pour la durée des paquets et pour le nombre de trames contenues, cette approximation donne des résultats très acceptables.

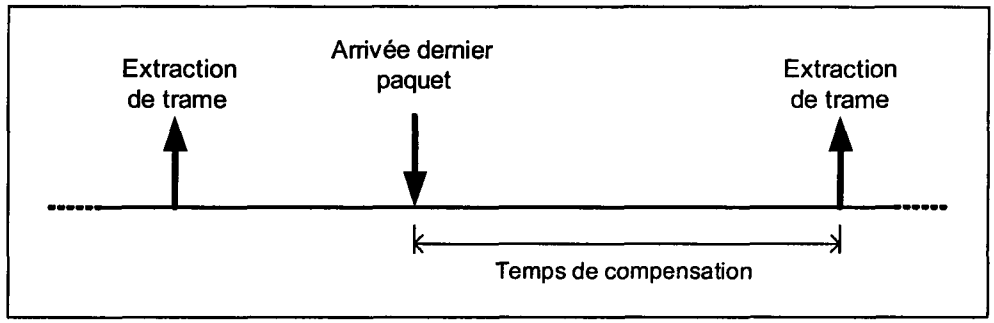

Figure 8.10: Compensation du temps d'arrivée du paquet

Un dernier facteur de correction considère le temps écoulé entre l'arrivée du dernier paquet et le moment de l'extraction de la trame suivante tel que s'illustre dans la figure 8.10. Cette correction permet de réduire la granularité dans le calcul du niveau du tampon.

La formule de correction est la suivante :

$$jbufferLevel+ = tempsExtractTrame - tempsArriveeDernierPqt \tag{8.7}$$

Le moment d'extraction de la trame suivante et le moment d'arrivée du dernier paquet sont, tous les deux, mesurés en unités de temps d'échantillon. En conséquence, aucune conversion d'échelle n'est nécessaire.

8.4.10 Niveau moyen du tampon

Afin de réduire l'effet des certains phénomènes, comme des points et des creux dans la réception des paquets, il faut calculer la moyenne mobile du niveau du tampon sur les dernières 32 valeurs en utilisant l'équation 8.8.

$$jbAvrgLevel_n = jbAvrgLevel_{n-1} + \frac{jbufferLevel_n}{N} - \frac{jbAvrgLevel_{n-1}}{N} \quad ; N = 32 \quad (8.8)$$

Selon des observations faites par Colin Perkins (Perkins, 2006), dans de nombreux cas, la gigue peut être considérée comme un phénomène aléatoire. L'intervalle entre l'arrivée des paquets possède une distribution qui se rapproche de la normale. Dans une première approximation, la gigue, estimée selon la procédure décrite plus haut, peut alors être vue comme la déviation standard. Ainsi, le taux de paquets écartés parce qu'ils sont arrivés trop tard sera inférieur à 2,5% ou 0,5% pour des temps de protection de 2 ou 3 fois supérieures à la valeur de la gigue, respectivement.

Pour ce projet, le point de fonctionnement de référence est donné pour la variable **threshodllvl** et il est calculé avec l'équation 8.9.

$$thresholdlvl = \frac{(jitter * JBLFACTOR)}{4} \quad (8.9)$$

Le facteur **JBLFACTOR** est un paramètre défini à la compilation et sa valeur est un entier qui vaut typiquement entre 8 à 12. La division par 4 est implémentée avec un décalage à gauche. Ainsi, **threshodllvl** aura une valeur entre deux et trois fois la valeur de la gigue. Les variables **minlevel** et **maxlevel** contiennent les valeurs minimale et maximale de l'échelle des valeurs admissibles pour **thresholdlvl**.

Périodiquement, le niveau moyen du tampon est comparé avec **thresholdlvl**. S'il est trop élevé, quelques trames seront écartées graduellement afin de réduire le délai. Si par contre, il est plus petit, la correction sera déclenchée à partir de la détection de l'incrément du taux moyen de paquets écartés pour être arrivés en retard. Une troisième condition qui peut déclencher une correction est la détection d'une série de paquets consécutifs qui sont tous arrivés en retard. La fréquence de l'actualisation des paramètres du tampon est aussi définie durant la compilation par **JB_UPDATE_FRQNCY**.

8.4.11 Extraction des trames

Les routines de décodage travaillent au niveau des trames. En conséquence, il faut identifier et séparer les trames parmi la charge utile des paquets. Le protocole RTP ne prévoit pas des éléments pour connaître le nombre de trames contenues et leur position dans la charge

utile. Selon le RFC 3550 (Schulzrinne et al., 2003), le récepteur doit être capable de déduire cette information à partir de sa connaissance du type de codec. En conséquence, dans cette implémentation, cette tâche est déléguée à la fonction qui fait le décodage et les tampons anti-gigue incluent des éléments qui facilitent cette opération. Le principal avantage de cette disposition est qu'elle permet de concentrer tous les traitements qui sont spécifiques aux codecs, à un seul point dans la réception. Un deuxième avantage est une réduction des opérations de mouvement interne des données.

La variable **frameOffset** indique la position de la prochaine trame à être traitée pour le décodeur par rapport au début de la charge utile du paquet. Sa valeur est mise à jour à partir de l'information retournée pour la fonction de décodage :

```
erINT16U (*decoder)(void* inBuf, void* outBuf, erINT16U *size);
```

Avant l'appel à la fonction **decoder()**, la variable pointée par size contient le nombre d'octets qui restent dans le paquet. Au retour, elle contiendra le nombre d'octets effectivement traités par la fonction de décodage.

8.4.12 Considérations sur la gestion de la mémoire

Concernant à la gestion de la mémoire de données, la stratégie suivie est de la gérer localement : toutes les allocations et libérations des blocs de mémoire sont faites à l'interne du module tampon anti-gigue. Le principal avantage est une simplification importante des tâches de création et destruction des maillons de la liste chaînée. Le coût à payer est un mouvement des données additionnel (*memcopy*) au moment de l'insertion des paquets dans le tampon.

8.4.13 Sélection du codec

Une caractéristique d'un flot RTP est que l'émetteur peut changer la spécification du type de codec «à la volée» (*on the fly*) en choisissant lequel s'adapte mieux au type de signal transmis. Le récepteur doit sélectionner le décodeur approprié à partir de l'information qui se trouve dans l'en-tête du paquet. Cette opération est reléguée au moment de l'extraction de trames du tampon anti-gigue. Chaque fois que la première trame d'un paquet est demandée par le module de décodage, le système compare le schéma de codage spécifié

dans l'en-tête du paquet avec celui du codec courant. S'ils ne se correspondent pas, le récepteur sélectionne le codec approprié. De cette manière, le codec courant, celui spécifié dans le paquet précédent, est pris comme la première option, en économisant du temps de traitement.

8.4.14 Compensation des paquets manquants

Lorsqu'un paquet RTP contenant de l'information audio est perdu, le récepteur doit générer un remplacement afin de conserver la temporisation du flot. La solution adoptée est de répéter la dernière trame valide. La complexité de cette méthode n'est qu'un peu plus élevée que celle de la méthode de remplacement par des trames de silence, mais avec de meilleurs résultats. Voir la référence (Perkins, 2006).

À cet effet, une fois que la trame est décodée, elle est copiée dans un tampon en prévision d'une possible substitution à la trame suivante. En exploitant des particularités de l'implémentation, il est possible de faire certaines économies sur le traitement. En premier lieu, dans les cas où plusieurs trames sont encodées dans le même paquet, il ne faut sauvegarder que la dernière trame de l'ensemble. En deuxième lieu, le tampon anti-gigue permet de connaître à l'avance si le paquet suivant est déjà disponible. Dans ce cas-ci, la copie n'est pas nécessaire.

8.4.15 Mélangeur

Lorsqu'il faut jouer l'audio provenant de plusieurs sources (RTP ou locales) on doit les mélanger en faisant l'addition saturée des échantillons. Ce module a été considéré hors de la portée du projet et il pourrait être implémenté dans des révisions futures.

8.4.16 Boucle de restitution audio

La restitution audio constitue le dernier maillon de la chaîne. Son fonctionnement est complémentaire à celui de la capture audio. Étant exécutée dans une tâche séparée, elle utilise aussi deux tampons. Pendant qu'un d'eux est rempli pour le décodeur, l'autre est vidé pour le DMA associé au convertisseur numérique à analogique.

D'habitude, les boucles de capture et de restitution audio ont chacun sa propre source de synchronisation. Cependant, dans ce projet, on a choisi de les unifier. Cela est rendu

possible par le fait que toutes les deux opèrent au même rythme avec un certain décalage du temps. La synchronisation de la capture et de la reproduction équivaut à fixer la valeur de ce décalage. L'avantage de cette stratégie est une réduction du temps d'exécution pendant les interruptions générées pour les DMA. Par contre, elle introduit un délai moyen équivalent à la moitié de la durée d'une trame. De plus, on renonce à l'utilisation des mécanismes de contrôle fin du point de restitution. Cependant, cette caractéristique serait facilement incorporable dans une révision future.

8.5 Codeurs audio

Le nombre et le type de codeurs audio inclus varient selon l'application. La définition d'une interface commune permet de systématiser l'appel aux fonctions liées au codec. Chaque codeur doit être codé séparément et toutes les caractéristiques pertinentes pour le système sont regroupées dans une structure. La déclaration de cette structure est montrée à la Figure 8.11.

```
typedef struct{
  erCodecID_e      codecID;                /* codec ID                              */
  erCHAR           codecName[10];          /* codec name                            */
  erCodecEncodng_e encodngType;            /* encoding type: sample/frame based     */
  erINT16U         rtpPayLoadType;         /* payload type acording to RFC 3551     */
  erINT16U         bitSample;              /* number of bits/sample                 */
  erINT16U         sampleRate;             /* sample rate                           */
  erINT16U         aframesize;             /* audio (uncoded) frame size [sampl]    */
  erINT16U         nframesize;             /* network (coded) frame size [bytes]    */
  erINT16U         maxRxframesize;         /* max possible rx frame size [bytes]    */
  erINT16U         frame4pckt;             /* frames per packet                     */
  erINT16U         (*decoder)( void* inBuf, /* pointer to the decoder function      */
                               void* outBuf,
                               erINT16U* size);
  erINT16U         (*encoder)( void* inBuf, /* pointer to the encoder function      */
                               void* outBuf,
                               erINT16U size);
  erINT16U         (*init)(void *pData);   /* pointer to the init function          */
  erINT16U         (*onSilence)(erINT16S*, /* pointer to the "on silence" fnct      */
                                erINT16U);
    } erCodecSpec_t;
```

Figure 8.11: Structure de spécification d'un codec

Les pointeurs **decoder**, **encoder**, **init** et **onSilence** donnent l'accès aux fonctions de l'interface. Chaque codeur doit inclure la définition de cette structure avec ses valeurs par défaut. Pendant la procédure d'initialisation, les structures seront insérées dans une liste gérée par

le gestionnaire de codeurs. Par exemple, la figure 8.12 montre le code pour enregistrer le G.711A.

8.5.1 Addition d'autres types de codecs

Seuls les codecs G.711 sont disponibles dans l'implémentation présente, cependant l'addition d'autres codecs est un processus simple.

La première étape consiste à définir le codec dans un ou plusieurs fichiers indépendants. Il doit inclure les fonctions **decoder()**, **encoder()**, **init()** et **onSilence()**, qui constituent son interface publique.

La deuxième étape consiste à définir une structure **erCodecSpec_t**, qui englobe les caractéristiques du codec afin de permettre au logiciel d'adapter les différents paramètres requis pour son exécution.

Finalement, il faut créér la fonction pour enregistrer le codec pendant la procédure d'initialisation.

```
erINT16U erG711A_alloc (void)
{
    erCodecList_t   *codecP;

                                                /* alloc the codec PCM A          */
    if((codecP=erNewCodec()) == NULL)           /* create a new codecSpec struct  */
        return RPT_ERR_ALLOCCODEC;

    codecP->codecID     = COID_PCMA;            /* set codec ID                   */
    codecP->codecNameP = codecG711A.codecName; /*      codec name                */
    codecP->next        = NULL;                 /*      link pointer = NULL       */
    codecP->codecDef    = &codecG711A;          /*      pointer to coded def.     */
    erCodecAdd2List(codecP);                    /* add the codec to the list      */

    return  RTP_ERRNONE;                        /* done, return RTP_ERR_NOERROR   */
}
```

Figure 8.12: Fonction pour l'enregistrement d'un codec

8.6 Protocole RTCP

La transmission, la réception et le traitement des paquets RTCP n'ont pas été implantés. Cependant, la majorité de l'information nécessaire pour les rapports est déjà disponible.

8.7 Conclusion

Une première série de tests, réalisés sur les deux plateformes disponibles (architectures ARM7 et StrongARM), ont donné des résultats très satisfaisants. En particulier, le comportement du tampon anti-gigue a été vérifié en provoquant des conditions qui ont permis d'analyser l'évolution dans le temps du niveau du tampon. Ces premiers résultats sont très encourageants et ils sont une source de motivation pour continuer le développement dans l'avenir. Cependant, une série de tests exhaustifs sera nécessaire afin de connaître les limites réelles de la présente implémentation.

En comptant un peu plus de 2000 lignes de code, les attentes en relation à l'empreinte ont aussi été atteintes. La taille du code compilé reste plutôt petite : environ 14 ko plus 1,5 ko additionnels utilisés pour le codec G.711. Plusieurs sujets restent en attente afin d'être développés dans des révisions futures. Parmi eux, les plus importants sont : compléter le protocole RTCP, incorporer le support pour la détection de silences et le melangeur (*mixer*). Du côté des codeurs, il faudra incorporer des types de codecs plus efficaces, tels que le G.729A par exemple. En bref, on peut dire que tous les objectifs par rapport au protocole RTP ont été atteints, et que certains objectifs ont été dépassés.

CHAPITRE 9

MODULES COMPLEMENTAIRES

Ce bref chapitre est destiné à présenter les modules complémentaires plus importants qui ont été conçus pendant le développement du projet et qui sont en étroite relation avec le fonctionnement des piles SIP et RTP. Les deux premières sections présentent le système d'allocation de mémoire, qui inclut deux modules : un module spécialisé en l'allocation de chaînes de caractères et un module d'allocation de blocs de mémoire générale. Finalement, le système de traces est présenté dans la troisième section.

9.1 Gestion dynamique de la mémoire

Un problème particulièrement important des systèmes embarqués est la gestion dynamique de la mémoire. La bibliothèque standard de C fournit les fonctions **malloc()** et **free()**. Malheureusement, l'utilisation continue de ces fonctions, dans certaines conditions, amène à un problème appelé «fragmentation de la mémoire» qui finit par bloquer le fonctionnement du système, voir (Li, 2003) et (Johnstone and Wilson, 1998). En conséquence, dans le cas des systèmes embarqués, il est recommandable d'éviter tout usage de ces fonctions.

Le problème avec une pile SIP est encore plus grave, car il faut prévoir l'allocation et le prélèvement de nombreuses structures et chaînes de caractères, souvent de petite taille. Ce problème, qui s'ajoute à celui de la fragmentation, donne comme résultat une utilisation de la mémoire plutôt inefficace.

Le système d'exploitation MicroC/OS-II fournit des fonctions qui permettent de contourner partiellement les problèmes d'allocation de mémoire. Cependant, on a préféré bâtir une solution particulière. Cette décision s'appuie sur trois éléments principaux.

En premier lieu, le système MicroC/OS-II a été conçu pour optimiser le temps de réponse, ce qui comme on a déjà mentionné, n'est pas un élément prioritaire dans ce projet. En deuxième lieu, l'utilisation du système de MicroC/OS-II ne donne pas une réponse satisfaisante à la problématique de l'allocation de chaînes. En troisième lieu, l'utilisation d'un système de gestion de mémoire embarqué dans la pile, augmente sa portabilité. La solution

à cette problématique, dans le cadre de ce projet, a été de concevoir un ensemble de fonctions dédiées pour l'allocation de mémoire, exploitant les caractéristiques particulières des piles SIP et RTP conçues et avec une attention spéciale sur leur simplicité et l'efficacité dans l'utilisation de la mémoire. Complémentairement, la solution a inclus des fonctions spécialisées pour l'allocation de chaînes de caractères.

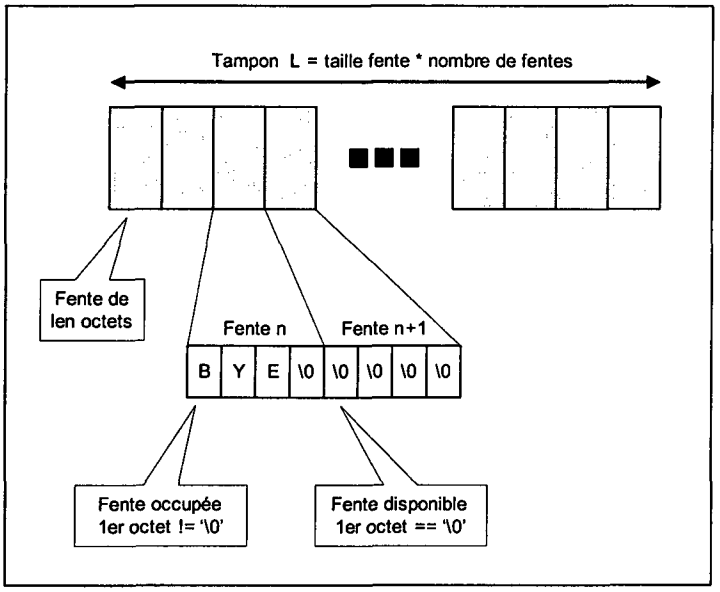

Figure 9.1: Allocation dynamique de chaînes de caractères

9.2 Allocation de chaînes de caractères

La solution trouvée à l'allocation de chaînes de caractères profite de la connaissance a priori de la structure du type de données à stocker. Dans le langage 'C', les chaînes de caractères sont stockées utilisant un caractère nul comme terminaison. Ceci signifie que les nuls sont traités comme des caractères spéciaux, qu'on ne trouve jamais à l'intérieur d'une chaîne. En conséquence, on peut se servir de cette particularité afin de signaler la disponibilité ou non d'un bloc de mémoire. À cet effet, un tampon de mémoire est divisé en fentes (*slots*) dont la longueur (*len*) est prédéterminée, comme on trouve dans la figure 9.1. La disponibilité ou l'indisponibilité d'une fente est déterminée par le contenu de son premier octet. Ainsi, une fente est considérée **occupée** lorsque le premier octet est un caractère non

nul. La détermination de la disponibilité d'une fente exige, par contre, l'analyse du dernier octet de la fente précédente afin de gérer la situation spéciale où la longueur de la chaîne précédente coïncide avec la longueur de la fente. La figure 9.3 illustre cette situation.

```
void      erLibStrStoIni(void);
erCHAR*   erLibStrSto(erCHAR* st);
erCHAR*   erLibnStrSto(erCHAR* st, erINT16U len);
erINT16U  erLibStrClr(erCHAR* st);
```

Figure 9.2: API du système d'allocation de chaînes de caractères

Lorsqu'on veut allouer une chaîne, les fonctions d'allocation cherchent séquentiellement une série de fentes libres consécutives capables de la stocker à partir du point de la dernière allocation. Cet algorithme d'allocation est appelé «*Next Fit*» et il a été choisi considérant le comportement de la pile SIP.

Il y a deux fonctions d'allocation : **erLibStrSto()**, qui permet d'allouer des chaînes finissantes avec un caractère nul, et **erLibnStrSto()**, qui permet d'allouer des sous-chaînes d'une longueur donnée.

Le système d'allocation de chaînes de caractères est initialisé automatiquement lors de la première utilisation. Il peut aussi être initialisé de manière explicite en appelant la fonction **erLibStrStoIni()**. La figure 9.2 montre l'API pour l'allocation des chaînes de caractères.

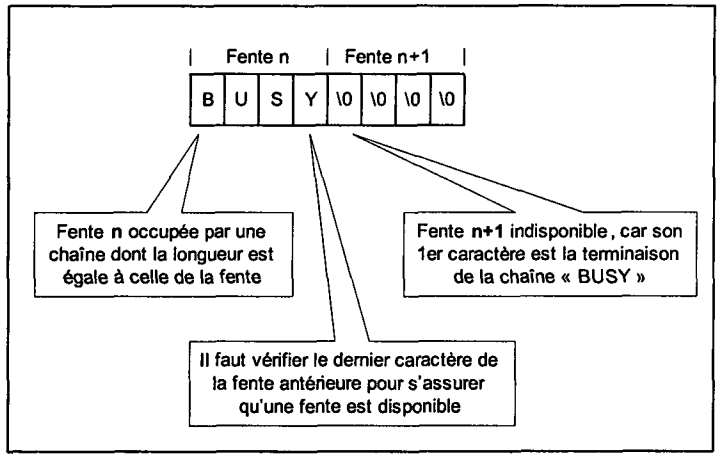

Figure 9.3: Cas particulier : la longueur de la fente est égale à celle de la chaîne

La détermination de la longueur optimale des fentes est importante, car elle a une incidence directe sur l'efficacité. De petites fentes amènent vers une efficacité plus élevée, mais au détriment de la vitesse d'exécution. La valeur choisie dans le projet est de 8 octets, ce qui permet d'allouer la plupart des mots en utilisant 1 ou 2 fentes.

9.3 Allocation de blocs de mémoire

Lorsqu'il s'agit de l'allocation de blocs de mémoire, contrairement au cas des chaînes de caractères, on ne peut faire aucune supposition quant à la nature des données. En conséquence, il est nécessaire d'utiliser un élément complémentaire. La solution trouvée utilise un tampon divisé en fentes et une table de contrôle où chaque position correspond à une fente du tampon.

La figure 9.4 montre l'usage de la table de contrôle. Un zéro dans une position de cette table indique que la fente associée dans la table d'allocation est disponible. Pourtant, une valeur différente de zéro indique que la fente est occupée. Cette valeur correspond au déplacement (*offset*) de l'index pour arriver à la fin du bloc courant, ce qui permet de parcourir la table rapidement. Une autre fois, l'algorithme d'allocation utilisé est le «*Next Fit*», c'est-à-dire que la recherche d'un bloc de mémoire commence au point de la dernière allocation.

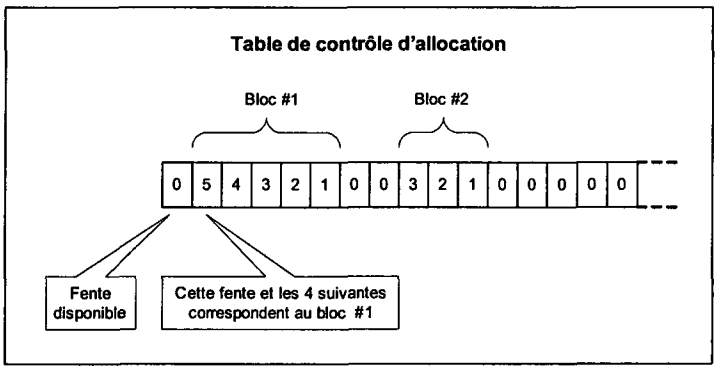

Figure 9.4: Description de la table de contrôle d'allocation de mémoire

De la même manière que dans le cas d'allocation de chaînes de caractères, la détermination de la taille des fentes est critique. Pour le projet, on a choisi une valeur de 32 octets par fente.

Une possible amélioration consiste à définir deux tas de mémoire avec deux tailles de fentes différentes. Une d'elles serait destinée à allouer de petits blocs dans de petites fentes. L'autre serait utilisée pour les blocs plus grands, avec de plus grosses fentes, alors qu'une fonction d'allocation se chargerait de déterminer dans quel tas serait assignée la demande de mémoire.

```
/**
 * @fn      void erTracef(const erCHAR *file, const erCHAR *fmt, ...)
 * @brief   trace function
 *
 * @param   file      source file identifier
 * @param   fmt       control format string
 * @return  void
 * @see
 **/
void erTracef(const erCHAR *file, const erCHAR *fmt, ...)
{
    va_list ap;

    erSchedLock();
    switch(erLogLevel){
        case 0:
            printf(" FATAL ERR ");
            break;
        case 1:
            printf(" ERROR.... ");
            break;
        case 2:
            printf(" WARNING.. ");
            break;
        case 3:
            printf(" INFO..... ");
            break;
        case 4:
            printf(" DEBUG.... ");
            break;
        case 5:
            printf(" TRACE.... ");
            break;
        default:
            printf(" LOGGING LEVEL ");
            return;
    }
    printf("%-16.16s: ",file);                    /* print source file            */
    va_start(ap, fmt);
    vprintf(fmt,ap);                              /* print variable argument list */
    va_end(ap);
    printf("\r\n");
    erSchedUnlock();
}
```

Figure 9.5: Fonction appelée pour les macros de TRACE

9.4 Système de traces

Le système inclut aussi des fonctions auxiliaires qui permettent de suivre en temps réel le comportement des fonctions d'allocation et de détecter les fuites de mémoire (*memory leaks*).

Dans le but de simplifier le déverminage et la mise en service des piles, la solution proposée inclut un système simple de traces. Il y a six niveaux de traces : «*Fatal Error*», «*Error*», «*Warning*», «*Info*», «*Debug*» et «*Trace*».

La macro **er**TRACE_LVEL, qui est définie au moment de la compilation, établit le niveau maximal de traces qui sera reporté. Le système de traces est activé ou désactivé de façon globale avec la macro **er**SIPTRACE. La Figure 9.5, montre la fonction appelée par la macro **er**TRACE.

Pour la conception, on a s'est inspiré du système de traces de la pile pjSip (). Parmi les principales améliorations à considérer dans une future itération, on trouve la possibilité d'activer et de désactiver les traces en temps réel et d'inclure un moyen pour rediriger les traces vers un autre terminal utilisant des datagrammes UDP.

9.5 Conclusion

Ce chapitre a servi à présenter les modules complémentaires les plus importants qui ont été conçus pendant le projet. Dans le cas des fonctions d'allocation, leur comportement a été évalué à l'aide des fonctions auxiliaires qui montrent l'état des tampons d'allocation. En ce qui concerne à l'algorithme d'allocation, «Next Fit», il exhibe de bons résultats avec le type d'applications développées. Dans le cas de RTP, où les exigences en matière d'allocation de mémoire sont plus exigeantes, aucun impact négatif sur la qualité de la voix n'a été observé.

Finalement, le système de traces a été utile à l'heure de déverminer les piles, surtout lors du débogage sur le CerfPDA, car l'utilisation de GDB *(GNU Debugger)* reste très limitée pour la configuration disponible à l'université.

TROISIÈME PARTIE

CONCLUSION

CHAPITRE 10
CONCLUSION

10.1 Bilan général

L'intérêt que déclenche la téléphonie sur IP est incontestable. Tout au long de ce projet, le nombre d'applications et d'implémentations des protocoles SIP et RTP n'ont pas cessé de grimper, ce qui confirme l'actualité du sujet de recherche au sein des communautés scientifiques et industrielles. En même temps, l'application du protocole SIP sur d'autres domaines, notamment la domotique et les systèmes M2M (*Machine to Machine*), commence à se consolider à partir des projets tels que HomeSip, en ouvrant de nouvelles possibilités pour SIP et pour d'autres projets, comme ceci, que ciblent des applications compactes sur des systèmes embarqués.

Le présent chapitre présente les conclusions. Il est structuré en trois sections, étant la première destinée à faire une analyse récapitulative du projet. La deuxième section expose les conclusions proprement dites, et finalement, la troisième partie explore les perspectives futures.

10.2 Analyse rétrospective du projet

L'objectif de cette section n'est pas de faire une analyse rétrospective rigoureuse du déroulement du projet, il s'agit plutôt de revenir sur quelques points qui ont eu une incidence importante sur les résultats finaux. Un premier point important à mentionner est que la définition des besoins du projet, faite pendant l'étape de planification, a permis de bien reconnaître les exigences du projet. Ainsi, la correcte identification des applications ciblées, la délimitation de la portée des implantations et l'analyse en profondeur des besoins du projet ont permis de cerner les caractéristiques essentielles requises et d'arriver à des ensembles de spécifications fonctionnelles et non-fonctionnelles consistantes et utiles. Ceci a eu une incidence notable sur la manière de coder et la structure générale des composants développés.

La portabilité des applications développées, considérée comme une condition importante à respecter, a eu un impact non négligeable sur le style de codage. En principe, le grade de portabilité d'une application est en relation directe avec son niveau de dépendance du matériel sous-jacent et du système d'exploitation. À cet égard, du côté du système d'exploitation, le critère suivi a été d'utiliser, autant que possible, d'une manière simple et directe, juste un petit ensemble de services de base disponibles dans presque tous les RTOS. Un critère similaire a été suivi au moment d'interagir avec les composants du matériel qui ne possèdent pas des abstractions dans le système d'exploitation. L'idée a été l'utilisation de ces composants de la manière plus standard possible, laissant de côté les caractéristiques particulières, parfois très attirantes, mais que normalement rendent plus difficiles les migrations vers d'autres plateformes. Des couches d'abstractions ont permis de regrouper et de cacher en arrière les particularités du système d'exploitation et du matériel..

À propos du RTOS, le système MicroC/OS-II avec la pile uC/TCP-IP de Micrium a prouvé être un bon choix pour ce type d'applications. Son utilisation a été simple et il n'a pas demandé des efforts particuliers. À cet égard, parmi les points les plus remarquables, on trouve la disponibilité du code source et sa description, très détaillée, qui se trouve dans le livre de Jean Labrosse (Labrosse, 2002). D'ailleurs, le fait qu'il soit gratuit pour des applications éducationnelles ou non commerciales est, sans doute, un autre point fort.

Du point de vue de la gestion, l'analyse rétrospective montre que la charge de travail a surpassé les estimations faites lors de la planification. En grandes lignes, les raisons identifiées sont trois : une sous-estimation du niveau de complexité de certains modules, la capacité de débogage très limitée par la plateforme matérielle et, finalement, le besoin de concevoir plusieurs modules additionnels, par exemple les pilotes audio et pour le contrôleur Ethernet du PDA.

10.3 Tests

La figure 10.1 montre le scénario utilisé pour les tests. Les PDAs ont des microphones et de haut-parleurs intégrés, par contre, pour la carte Zoom il faut utiliser un microphone et des écouteurs externes. Les sorties RS 232 des PDAs et de la carte Zoom sont connectées aux ordinateurs qui exécutent TeraTerm, un programme d'émulation des terminaux, afin d'interagir avec les programmes de test et de capturer les traces d'exécution. La capture

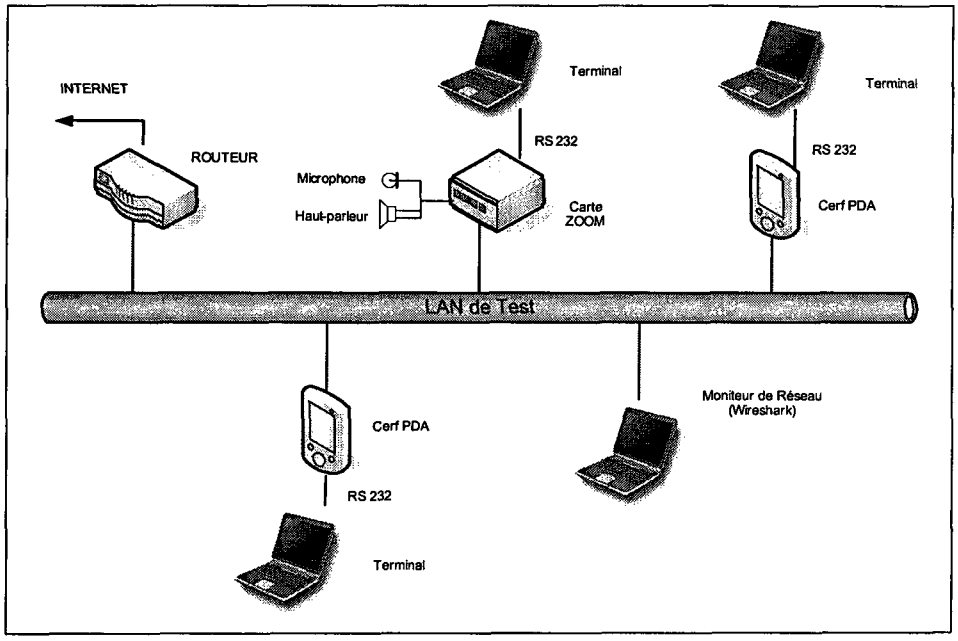

Figure 10.1: Disposition d'équipements utilisée pour les tests

et l'analyse des paquets SIP et RTP sur le réseau Ethernet sont faites avec le programme WireShark.

Le programme de test inclut un module moniteur, avec une interface en ligne de commande, qui permet d'établir et de finir une communication et de démarrer et d'arrêter l'envoie de paquets contenant de l'information audio (encodée avec un codeur G.711). Le moniteur possède aussi des options pour superviser des éléments clés, tels que l'utilisation des tampons pour l'allocation de mémoire, le niveau des tampons anti-gigue et l'exploration de la mémoire RAM.

Les tests réalisés ont permis de vérifier le fonctionnement global, l'utilisation de la mémoire et la structure des paquets capturés sur le réseau. Pendant les nombreuses communications vocales établies, l'appréciation subjective de la qualité de la voix a été très bonne et les délais introduits à l'intérieur de la pile RTP n'ont pas été perceptibles. Plusieurs communications de plus de 6 heures ont été établies afin de vérifier la stabilité et de détecter des problèmes de gestion de la mémoire. Par ailleurs, une légère différence de la fréquence d'échantillonnage audio, par rapport à la valeur nominale, dans la carte Zoom a permis de vérifier le comportement adaptatif du tampon anti-gigue.

10.4 Conclusions

L'implantation de protocoles complexes, comme SIP et RTP, donne toujours la possibilité de confronter une vaste diversité de défis et de problèmes qui embrassent plusieurs domaines de la conception logicielle et des communications. Cette section regroupe les conclusions tirées de cette implantation des protocoles SIP et RTP en ce qui concerne l'empreinte, la gestion de la mémoire, l'utilisation des machines à états finis, l'implantation de la couche de transport et la performance du système conçu.

Dans le chapitre 5, on a mis en relief l'importance d'atteindre une empreinte la plus petite possible à l'heure de développer des applications embarquées. Ainsi, c'est important d'évaluer les résultats du projet du point de vue de la taille finale du code compilé. Ces valeurs, inférieures à 75 ko pour SIP et à 14 ko pour RTP, sont très intéressantes et se trouvent parmi les plus petites si on compare avec d'autres implantations disponibles sur Internet. Cependant, dans le cas du protocole SIP, il faut considérer que la pile doit être complétée avec des développements complémentaires et que sa taille finale grandira. Il faut remarquer que ce n'est pas facile de faire une comparaison directe entre différentes implantations de SIP, car il y en a beaucoup des différences entre elles. Certaines piles, qui sont orientées vers des applications de communications d'envergure, sont très complètes et complexes. En conséquence, les facteurs de pondération sont nombreux et difficiles à établir. Par exemple, c'est fondamental de prendre en considération la plateforme matérielle ciblée. La taille du code destinée aux processeurs ARM sera significativement plus grande que celle générée pour des processeurs i386. Toutefois, on trouve, en Internet, que la taille des implantations disponibles se place dans une échelle allant d'un peu plus de 200 ko, pour les piles les plus compactes, jusqu'à plusieurs méga-octets, dans le cas des piles reSIProcate, SipX, Vocal ou Opal.

Par contre, dans le cas de RTP, l'empreinte est bien plus représentative étant donné que les alternatives d'implantation de ce protocole sont beaucoup plus limitées que dans le cas de SIP. Même en considérant que cette implantation doit être complétée avec l'incorporation de RTCP, la taille du code compilé est placée parmi les plus petites de ceux qui sont disponibles sur Internet.

L'importance de la gestion de la mémoire dynamique dans le cas des applications embarquées a été remarquée dans le chapitre 5. Cette problématique a été résolue d'une façon

simple, mais efficace : L'allocation des blocs de mémoires et l'allocation de chaînes de caractères ont été traitées séparément. Cette stratégie a permis de contourner le problème de la fragmentation de la mémoire tout en préservant un haut niveau d'efficacité en son utilisation. En particulier, l'idée derrière le sous-système d'allocation des chaînes de caractères consiste à fractionner un tas de mémoire en fentes de taille relativement petite (8 octets) et à exploiter la nature même des données à stocker afin de contrôler cette allocation. Il faut noter que l'utilisation de ces systèmes d'allocation a permis d'optimiser l'utilisation et de réduire la taille de la mémoire dynamique aux dépens de certaines concessions du côté de la vitesse d'allocation. Cependant, ces concessions ont eu un impact plutôt faible sur la performance globale.

Du côté du SIP, un de traits que particularise cette implantation est qu'elle cible un ensemble bien spécifique d'applications. Pour cette raison, elle a un profil très net, établie à partir de la définition des besoins du projet ce qui entraîne, en conséquence, un code compact et cohérent.

L'implantation des machines à états finis, un sujet cité souvent comme un point citrique dans les implantations du protocole SIP, a été résolue avec un procédé basé sur des tables de transitions avec des fonctions d'états très compactes et simples. Les temporisateurs associés au fonctionnement des MEFs ont été embarqués dans les mêmes structures de contrôle de telles machines, avec des gains en termes de performance et de taille du code. En définitive, la mise à point de ce module a été simple et rapide.

L'extension du concept de «dialogues SIP», défini pour le RFC 3261 (Rosenberg et al., 2002), a permis d'avoir un contexte unique à partir duquel c'est possible de régir tous les échanges de messages SIP, même ceux qui, selon la recommandation, n'établissent pas des dialogues. Cette approche a simplifié autant la génération de messages SIP que le gestionnaire de réception. La gestion de tels dialogues a été résolue à l'aide des machines à états finis.

La mise en œuvre de la couche de transport SIP s'écarte un peu des solutions traditionnelles. Dans ce projet, cette couche-ci s'occupe de la transmission et de la réception de messages d'une façon transparente, autrement dit, en faisant abstraction du contenu des messages. Ceci lui permet de se focaliser exclusivement sur les aspects liés au transport tandis que les tâches de conversion de format des messages SIP, soit l'analyse syntaxique ou *parsing* et la sérialisation, sont accomplies pour des couches de niveau plus haut. Cette solution

a permis de délimiter la portée de la couche de transport et de simplifier notablement sa conception.

Par ailleurs, parmi les aspects les plus remarquables de cette implantation de RTP, on trouve sa simplicité, son empreinte minimale, l'inclusion d'un tampon anti-gigue adaptatif et la réduction au minimum des mouvements de données dans la mémoire.

10.5 Perspectives futures

Dans le contexte actuel de convergence des réseaux et des services, les communications se trouvent dans un point tournant, étant SIP un des éléments clés au sein de ce changement. Dans ce scénario, il est possible d'envisager un champ de possibilités intéressant pour les composants développés dans ce projet pour certaines catégories de systèmes embarqués. C'est le cas, par exemple, de petits systèmes de communication ou des systèmes de contrôle réseautés, où l'importance de la taille du code et l'efficacité devient des facteurs cruciaux. Souvent, dans ce type de systèmes, le rôle des communications est complémentaire d'une autre fonction principale.

L'utilisation des piles SIP et RTP dans les applications mentionnées plus haut doit considérer, au préalable, l'intégration des parties et des protocoles complémentaires. Du côté de SIP, ces parties incluent des extensions de SIP, des éléments de sécurité et l'implantation des protocoles complémentaires, par exemple le protocole STUN (Rosenberg et al., 2003), qui sert à traverser les NATs (*Network Address Translation*)des coupe-feu. Pour les applications de VoIP, il sera aussi nécessaire d'intégrer le protocole SDP (Handley et al., 2006). Par ailleurs, et du côté de RTP, l'implantation de RTCP serait désirable, mais ce n'est pas essentiel.

QUATRIÈME PARTIE

ANNEXES

ANNEXE A
CHARGE UTILE (PT) POUR RTP

PT	Nom	Media Type	freq. echant. [Hz]	Canaux
0	PCMU	Audio	8000	1
1	reserved			
2	reserved			
3	GSM	Audio	8000	1
4	G723	Audio	8000	1
5	DVI4	Audio	8000	1
6	DVI4	Audio	16000	1
7	LPC	Audio	8000	1
8	PCMA	Audio	8000	1
9	G722	Audio	8000	1
10	L16	Audio	44,100	2
11	L16	Audio	44,100	1
12	QCELP	Audio	8000	1
13	CN	Audio	8000	1
14	MPA	Audio	90,000	1
15	G728	Audio	8000	*
16	DVI4	Audio	11,025	1
17	DVI4	Audio	22,050	1
18	G729	Audio	8000	1
19	reserved			
20-23	unassigened	Audio		
dynamic	G726-40	Audio	8000	1
dynamic	G726-32	Audio	8000	1
dynamic	G726-24	Audio	8000	1
dynamic	G726-24	Audio	8000	1
dynamic	G726-24	Audio	8000	1
dynamic	G726-16	Audio	8000	1
dynamic	G729D	Audio	8000	1
dynamic	G729E	Audio	8000	1
dynamic	GSM-EFR	Audio	8000	1
dynamic	L8	Audio	var.	var.
dynamic	RED	Audio		*
dynamic	VDVI	Audio	var.	var.

TABLEAU A.1: Types de charge utile (PT) pour des codifications audio (Schulzrinne and Casper, 2003)

PT	Nom	Media Type	clock rate [Hz]
24	unassigned	Video	
25	CelB	Video	90,000
26	JPEG	Video	90,000
27	unassigned	Video	
28	nv	Video	90,000
29	unassigned	Video	
30	unassigned	Video	
31	H261	Video	90,000
32	MPV	Video	90,000
33	MP2T	AV	90,000
34	H263	Video	90,000
35-71	unassigned	?	
72-76	reserved	N/A	N/A
77-95	unassigned	?	
96-127	dynamic	?	
dynamic	H263-1998	Video	90,000

TABLEAU A.2: Types de charge utile (PT) pour des codifications video (Schulzrinne and Casper, 2003)

BIBLIOGRAPHIE

(). site de pjsip : www.pjsip.org.

Abid, M. , Chebrou, A. , Favot, A. , Ellero, S. , and Koehler, S. (2006). Mise en œuvre d'une pile sip sur un système embarqué pour le contrôle de capteurs. *n.d.*

Berners-Lee, T. , Masinter, L. , and McCahill, M. (1994). *RFC 1738. Uniform Ressource Locators (URL).* IETF. Network Working Group.

Berners-Lee, T. , Fielding, R. , and Masinter, L. (2005). *RFC 3986. Uniform Resource Identifier (URI) : Generic Syntax.* IETF. Network Working Group.

Brandl, M. , Fransens, K. , Daskopoulos, D. , Dobbelsteijn, E. , Garroppo, R. , J.Janak, Juthan, J. , and Niccolini, S. (2004). *IP Telephony Cookbook.* Trans-European Research and Education Networking Association (TERENA).

Crocker, D. and Overell, P. (2008). *RFC 5234. Augmented BNF Syntax Specifications : ABNF.* IETF. Network Working Group.

Duric, A. and Andersen, S. (2004). *RFC 3952. Real-time Transport Protocol (RTP) Payload Format for internet Low Bit Rate Codec (iLBC) Speech.* IETF. Network Working Group.

ENSEIRB (s.d.). Welcome to the homesip project : Using the sip protocol for home automation, [en ligne]. URL `www.enseirb.fr/cosynux/HomeSIP`. (Page consultée le 5 mai 2008).

F.Groom and Groom, K. (2004). *The Basics of Voice over Internet Protocol.* International Engineering Consortium (IEC).

Goralski, W. and Kolon, M. (1999). *IP Telephony.* McGraw-Hill.

Hallock, J. (2004). A brief history of voip. *n. d.*, Document One - The Past.

Handley, M. , Schulzrinne, H. , Schooler, E. , and Rosenberg, J. (1999). *RFC 2543. SIP : Session Initiation Protocol.* IETF. Network Working Group.

Handley, M. , Jacobson, V. , and Perkins, C. (2006). *RFC 4566. SDP : Session Description Protocol.* IETF, Network Working Group.

ITU-T (1988). *Recommendation G.711 Pulse code modulation (PCM) of voice frecuencies.* G Series. ITU-T.

ITU-T (1996). *Recommendation H.323 (11/96). Visual telephone systems and equipment for local area networks which provide a non-guaranteed quality of service.* H Series. ITU-T.

ITU-T (2006a). *Recommendation G.723.1 Dual rate speech coder for communications transmitting at 5.3 and 6.3 kbit/s.* G Series. ITU-T.

ITU-T (2006b). *Recommendation H.323 Packet-based multimedia communications systems.* H Series. ITU-T.

ITU-T (2007). *Recommendation G.729 Coding speech at 8kbits using CS-ACELP.* G Series. ITU-T.

Johnston, A. , Donovan, S. , Cunningham, C. , and Summers, K. (2003). *RFC 3665. Session Initation Protocol (SIP) Basic Call Flow Examples.* IETF. Network Working Group.

Johnston, A. B. (2004). *Understanding the Session Initiation Protocol.* Artech House, 2nd edition.

Johnstone, M. and Wilson, P. (1998). The memory fragmentation problem : Solved ? *Proceedings of the 1st international symposium on Memory management*, -.

Kadionik, P. (2006). Le projet homesip : la domotique avec le protocole sip. *Linux Magazine.*

Labrosse, J. (2002). *MicroC/OS-II. The Real-Time Kernel.* CMP Books, 2nd edition.

Li, Q. (2003). *Real-Time Concepts for Embedded Systems.* CMP Books.

Mabilleau, P. (2001). Notes de cours gei 455 systèmes en temps réel. université de sherbrooke.

Newport-Networks (2008). Emergency call handling in voip networks. *n. d.*, -.

Ott, J. , Wenger, S. , Sato, N. , Burmeister, C. , and Rey, J. (2006). *RFC 4585. Extended RTP Profile for Real-time Transport Control Protocol (RTCP) - Based Feedback (RTP/AVPF).* IETF. Network Working Group.

Percy, A. (2005). Understanding latency in ip telephony. *Understanding Latency in IP Telephony. TelecomWorld.com*, -.

Perkins, C. (2006). *RTP. Audio and Video for the Internet.* Addison Wesley, 4th edition.

Peterson, J. (2004). *RFC 3853. S/MIME Advanced Encryption Standard (AES) Requirement for the Session Initiation Protocol (SIP).* IETF. Network Working Group.

Roach, A. B. (2002). *RFC 3265. Session Initiation Protocol (SIP)-Specific Event Notification.* IETF. Network Working Group.

Rosenberg, J. (2002). *RFC 3311. The Session Initiation Protocol (SIP) UPDATE Method.* IETF. Network Working Group.

Rosenberg, J. (2007). State of emergency : Voip and 911. *SIP Magazine*, -.

Rosenberg, J. and Schulzrinne, H. (2002a). *RFC 3262. Reliability of Provisional Responses in Session Initiation Protocol (SIP).* IETF. Network Working Group.

Rosenberg, J. and Schulzrinne, H. (2002b). *RFC 3263. Session Initiation Protocol (SIP) : Locating SIP Servers.* IETF. Network Working Group.

Rosenberg, J. and Schulzrinne, H. (2002c). *RFC 3264. An Offer/Answer Model with Session Description Protocol (SDP).* IETF. Network Working Group.

Rosenberg, J. , Schulzrinne, H. , Camarillo, G. , Johnston, A. , Peterson, J. , Sparks, R. , Handley, M. , and Schooler, E. (2002). *RFC 3261. SIP : Session Initiation Protocol.* IETF. Network Working Group.

Rosenberg, J. , Weinberger, J. , Huitema, C. , and Mahy, R. (2003). *RFC 3489. STUN - Simple Traversal of UDP Through NATs.* IETF. Network Working Group.

Schulzrinne, H. and Casper, S. (2003). *RFC 3551. RTP Profile for Audio and Video Conferences with Minimal Control.* IETF. Network Working Group.

Schulzrinne, H. , Casner, S. , Frederick, R. , and Jacobson, V. (1996). *RFC 1889. A Transport Protocol for Real-Time Applications.* IETF. Network Working Group.

Schulzrinne, H. , Casner, S. , Frederick, R. , and Jacobson, V. (2003). *RFC 3550. RPT : A Transport Protocol for Real-Time Applications.* IETF. Network Working Group.

Sinnreich, H. and Johnston, A. B. (2001). *Internet Communications Using SIP.* Wiley & Sons. New York.

Sloss, A. N. , Symes, D. , and Wright, C. (2004). *ARM System Developer's Guide.* Morgan Kaufmann.

Sun, D. G. and Kim, S. (2005). A kernel-level rtp for efficient support of multimedia service on embedded systems. *Computational Science and Its Applications. ICCSA 2005.*