POLYTECHNIQUE
MONTRÉAL

LE GÉNIE
EN PREMIÈRE CLASSE

Polytechnique Montréal

LOG8430

Architecture logicielle et conception avancée

Tp 1 : Analyse de Ring

Auteurs:
Ait Younes Mehdi,
Barbez Antoine,
Ouenniche Farouk,
Sierra Juan Raul,
Zins Pierre

Septembre 30, 2016

Contents

1 Introduction

Dans le cadre du cours LOG8430: Architecture logicielle et conception avancée, ce rapport présente le travail réalisé par la ***Scheitan*** team (équipe 5) lors de la première séance de travaux pratiques. Les travaux en laboratoire sont centrés autour de l'analyse, la rétro-conception et l'amélioration du logiciel libre Ring, logiciel de communication (vidéo et textuelle) développé par l'équipe de savoirfairelinux, qui sera développé plus en détail ci-après.

L'objectif de ce premier TP a été de définir et d'étudier d'une façon générale l'architecture de Ring au travers de cas d'utilisation. Plus clairement, notre travail a été d'isoler quatre cas d'utilisation classiques du logiciel Ring (Envois de messages, Réception de messages, etc) et de rétro-concevoir le logiciel autour de ces cas d'utilisation. Après une présentation du logiciel Ring et de son fonctionnement, nous vous présenterons, pour chacun des cas d'utilisation que nous avons choisit, une étude basée sur le modèle 4+1, c'est à dire quatre vues logiques basées sur un scénario (le cas d'utilisation choisi).

Chaque modèle est donc composé de:

- **Un scénario** qui définit la fonctionnalité étudiée et l'interaction entre les différents acteurs. Il est illustré par un diagramme UML de cas d'utilisation.
- **Une vue logique** illustrée par un diagramme de classe.
- **Une vue de développement** qui décrit les composants du système impliqués dans le cas d'utilisation.
- **Une vue de processus** qui traite de la dynamique du scénario en terme de communication entre les différentes classes, en s'appuyant sur un diagramme de séquence.
- **Une vue de physique** qui décrit l'aspect utilisation des ressources matérielles par un diagramme de déploiement.

2 Ring

2.1 Introduction

Ring est un logiciel libre pour communiquer facilement et de manière sécurisée entre deux ou plusieurs personnes.

Ring est développé par une petite équipe chez Savoir-faire Linux ainsi que par une communauté croissante de contributeurs. Ring est principalement découpé en 3 couches comme le montre la figure 1.

- Clients : qui représente les différents clients où le projet est distribué.
- LibRingClient (LRC) : qui représente le *wrapper*(ou code commun) entre les client et la librairie Ring.
- LibRing : Cœur de Ring ou encore le *daemon* représente l'essence même du projet.

Ring repose aussi sur quatre bibliothèques externes ayant chacune une fonction particulière :

- OpenDHT : Pour la récupération des données (P2P) via un annuaire distribué.
- GNUTLS : Pour la gestions des certificats.
- PJSIP : Pour la gestion des sessions de communication.
- FFMPEG et LIBAV : les ***codecs*** audio et vidéo (compression et décompression des paquets lors du transport)

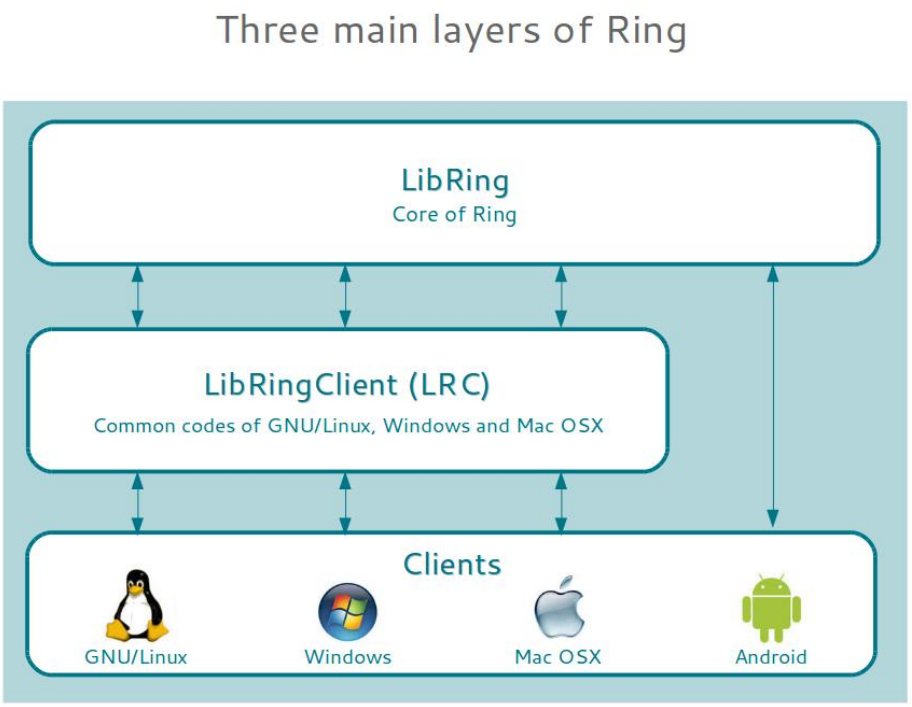


Figure 1: Les 3 principales couches de RING

2.2 Vue contextuelle

La vue contextuelle 2 est une représentation qui permet d'avoir une vision globale du projet et de ses principales relations, dépendances et interactions avec son environnement.

- VSC : Le principal système de version utilisé par l'équipe Ring est `Gerrit`[1](tous les changements effectué sur `Gerrit` sont ensuite automatiquement reporté sur le Github). Pour le système de *Issue Track*, l'équipe utilise `Tuleap`[2] .
- Langage : Les couches `Daemon et LRC` sont développées en `C++ 11`. Les clients sont développés selon leurs technologies.

[1] `https://www.gerritcodereview.com/`
[2] `https://tuleap.ring.cx/`

- Protocole : Les principaux protocoles de communication sont : `OpenDHT et ICE`. Ring propose aussi des communications via protocole `SIP`
- Plate-forme : Les différentes plate-formes où le projet est distribué.
- Licence : La licence utilisée par le projet est une licence `GNU General Public License 3`.
- Intégration Continue (CI) : L'équipe Ring utilise `Jenkins` comme outils de CI.
- Communication : Un *chanel* `IRC`[3] est disponible sur les serveurs de `FreeNode`.
- Contributeurs : Ring étant un projet libre, tout le monde peut contribuer. Néanmoins les principaux contributeurs sont l'équipe Ring de chez **Savoir Faire Linux**.

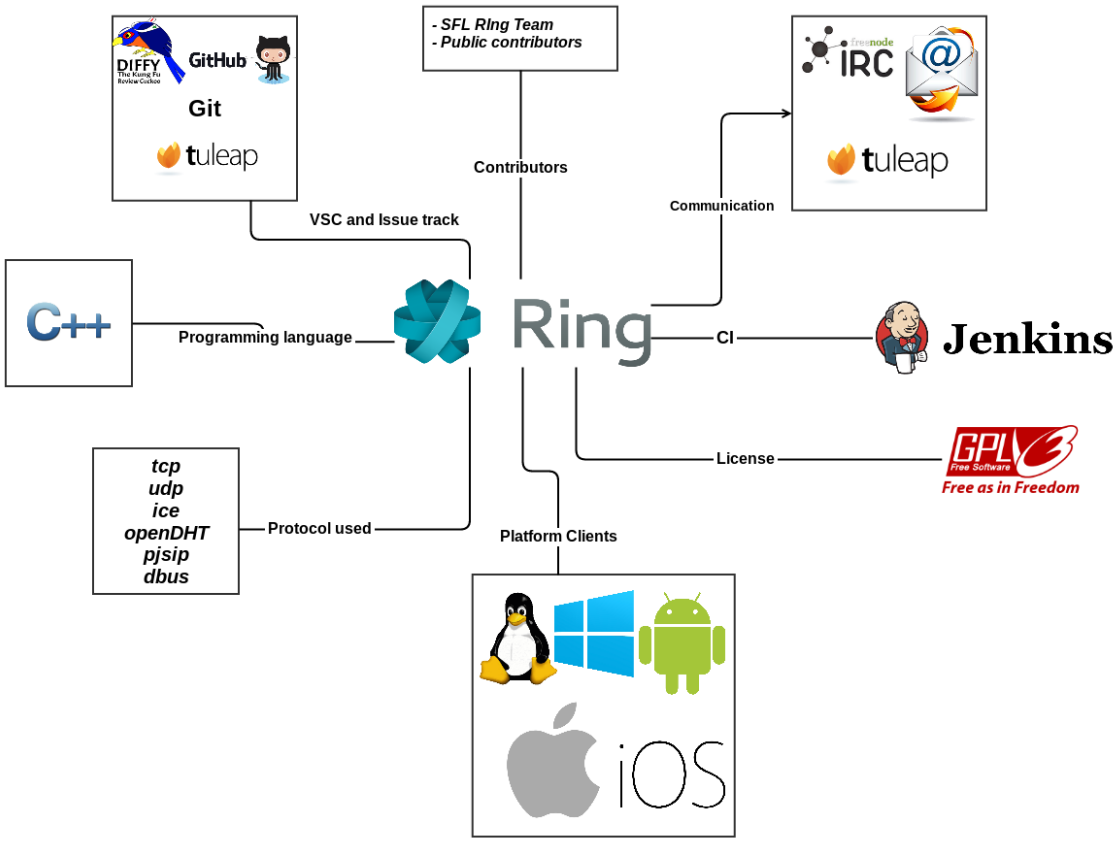


Figure 2: Vue contextuelle du projet Ring

[3] `https://webchat.freenode.net/`

3 Études

Ring étant déjà divisé en 3 couches (`Client, LRC, Daemon`), il aurait été difficile d'avoir une bonne vision du projet si nous avions effectué notre analyse sur l'ensemble du projet. Nous avons donc choisi la couche du `Daemon` comme principal domaine d'étude. Cette couche a été choisie car elle représente le **cœur** même du projet, il est donc très intéressent d'approfondir l'analyse de cette couche.

Lors de cette étude nous avons choisi comme scénarios des opérations qu'un utilisateur peux effectuer lors de l'utilisation de Ring. L'acteur choisi dans cette étude est l'**utilisateur**. Nous considéré 5 scénarios d'étude venant du Global Use Case(3) de Ring:

- **2x** envoi de message : **1x** avec un compte `SIP` et **1x** avec un `RingAccount`.
- 1x Réception d'un message.
- 1x Passage d'appel.

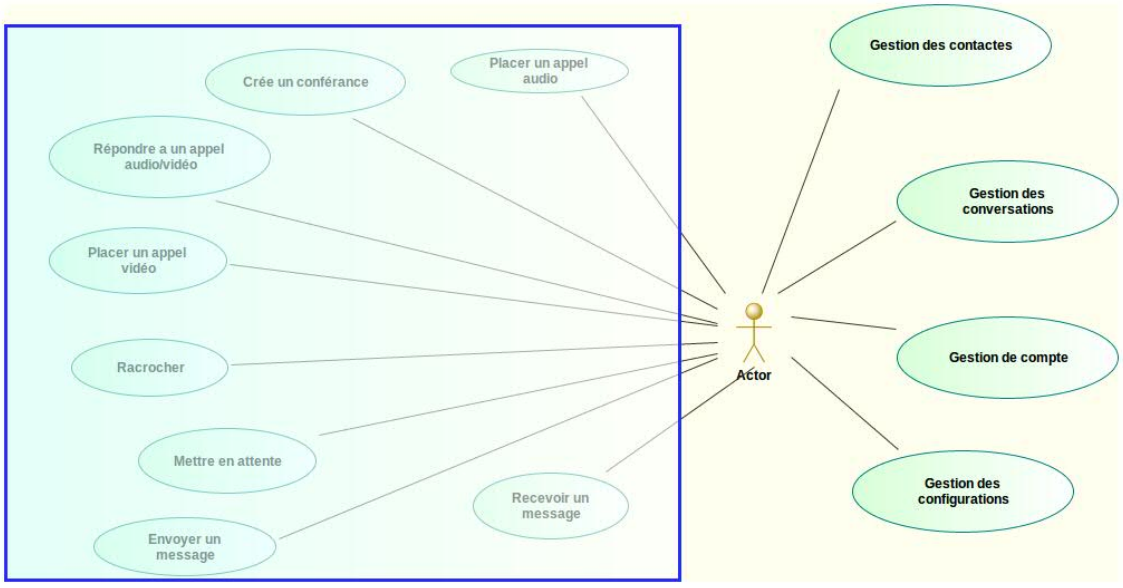


Figure 3: Use case général des principales actions de l'application Ring

3.1 Scénario 1 : Envoi de message via un compte Ring

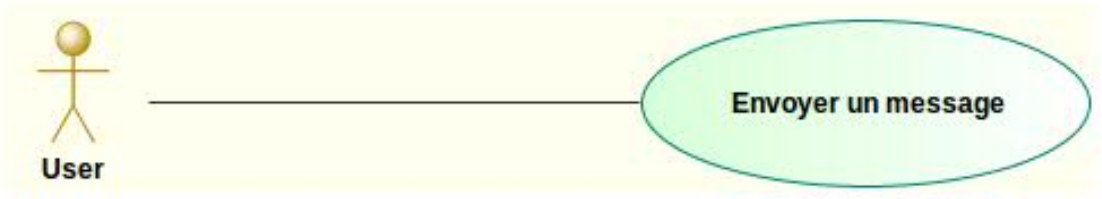


Figure 4: Use case du scénario : Envoyer un message via un compte Ring.

3.1.1 Vue Logique

Diagramme de classe :

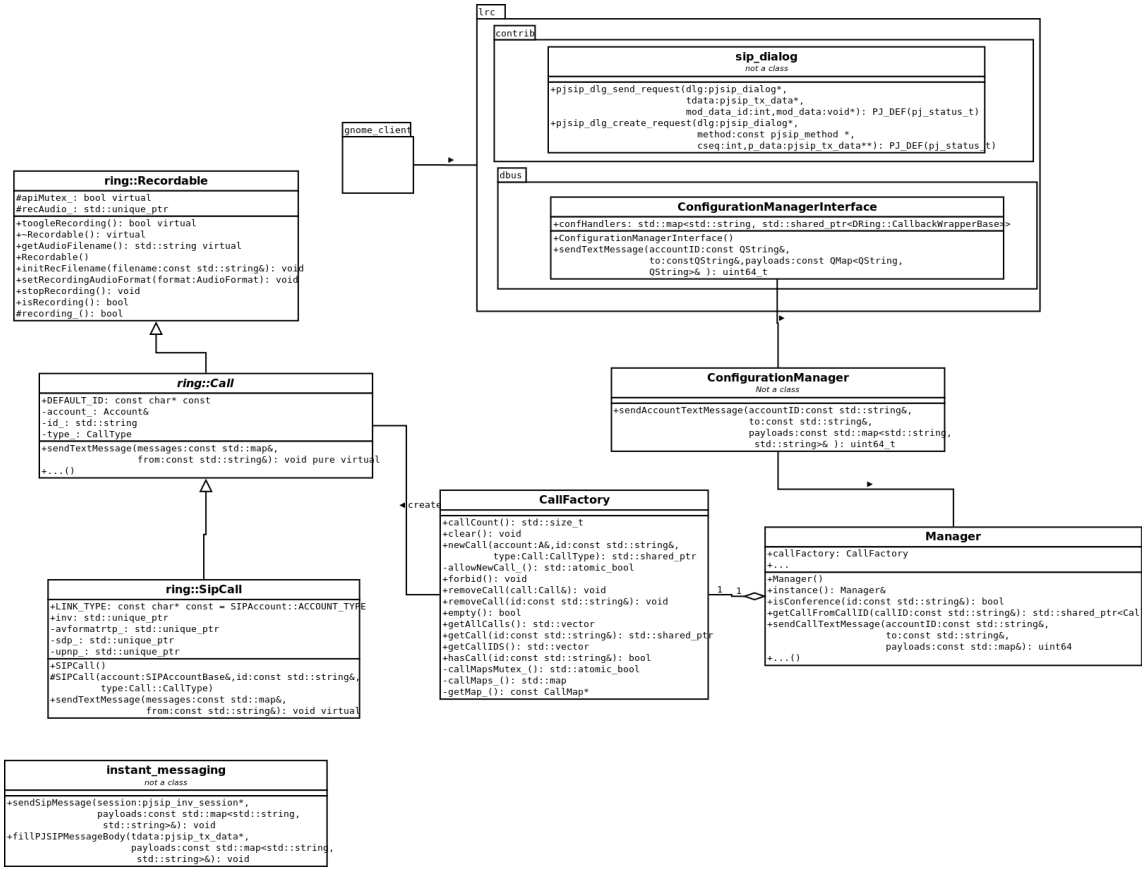


Figure 5: Diagramme de classe du scénario : Envoi de message via un compte Ring.

Pour les diagrammes de classes, nous nous sommes concentrés sur les classes ayant un rôle important pour chaque scénario. Une des difficultés, était le fait qu'un certain nombre d'éléments n'étaient pas des classes. Par exemple, l'élément CallManager n'est pas une vraie classe C++. Il consiste en un ensemble de fonctions regroupées dans un fichier "callmanager.cpp". Nous avons représenté ces éléments comme des classes, car cela permettait de rendre les diagrammes plus compréhensibles. Nous avons précisé dans les diagrammes de classes, qu'il ne s'agissait pas de classe ("not a class") Lors de l'envoi d'un message avec un compte ring, le point d'entrée dans le daemon se trouve au niveau du ConfigurationManager. Le lien lrc-daemon se fait au travers du ConfigurationManagerInterface (lrc) et du ConfigurationManager (daemon). Ce dernier est directement lié au Manager, qui est un singleton. Le manager fera ensuite appel à son callFactory afin de créer un SIPCall. La classe SIPCall dérive de la classe abstraite Call. Le SIPCall, enverra ensuite le message à l'aide de l'élément instant_messaging, qui lui fera utiliser le sip_dialog du lrc.

3.1.2 Vue Développement

Diagramme de composants : Figure 6

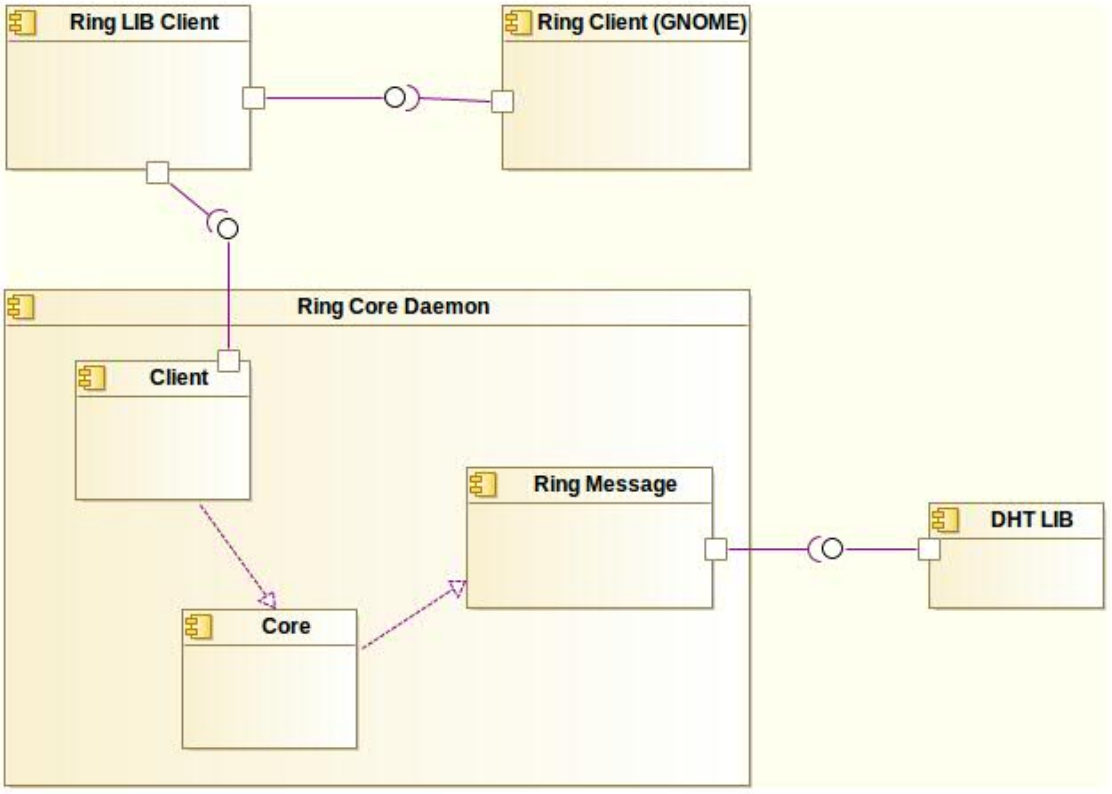


Figure 6: Diagramme de composant pour le scénario d'envoi de message avec un RingAccount

Justifications/ explications :

- **Ring Client (GNOME)** : Ce composant représente le client d'où le service est demandé. Dans notre cas, l'utilisateur doit pouvoir envoyer un message à un autre utilisateur via son compte **Ring**. Le client GNOME a besoin du service qui peux lui permettre d'envoyer un message. Nous allons appeler le service : *Send-Message*. Le composant `Ring LIB Client` est un composant qui permet de faire la liaison entre les différents clients et le cœur de Ring. Dans le cas du client GNOME, Le composant `Ring LIB Client` va utiliser un wrapper `QT` afin de pouvoir fournir le service *Send-Message* au client.
- **Ring Core Daemon** : Le point d'entrée de ce composant avec Le composant `Ring LIB Client` est le composant `Client`. La principale classe de ce composant est le `ConfigurationManager`. Celui-ci va fournir le service *Send-Message* en faisant appel au contrôleur du `Daemon` et aux autres composants/classes nécessaire : `Core`. Le composant `Core` est le système qui va

s'occuper de charger le service demandé au composant `Client`. Ce système est composé de plusieurs classes, principalement `Manager, Account, AccountFactory`.

Le composant `Ring Message` est le composant qui va construire le message et l'envoyer. Ce composant est principalement constitué des classes `RingAccount et MessageEngine`. Le composant `Ring Message` a directement besoin des services fournis par la librairie DHT (représentée par le composant `DHT LIB`) afin de finaliser le service *Send-Message* et de pouvoir envoyer le message au destinataire.

3.1.3 Vue processus

Diagramme de séquence:

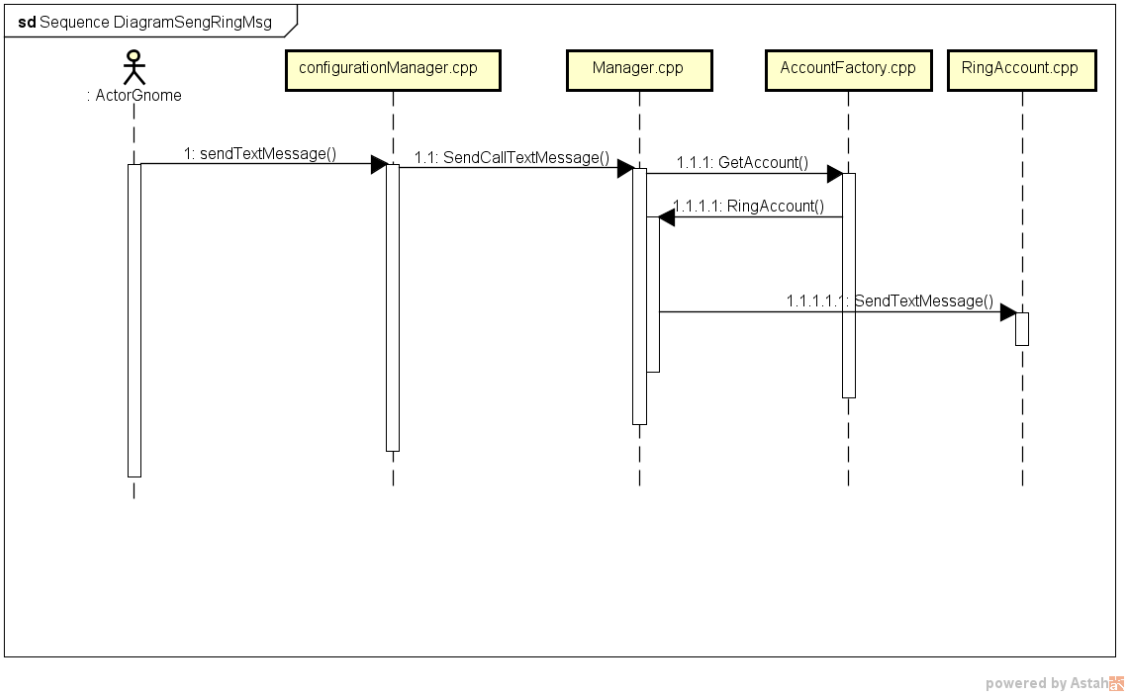


Figure 7: Diagramme de séquence pour l'envoi de message via un compte SIP

Justifications/ explications : Le diagramme ci-dessus décrit la séquence de l'envoi d'un message via un compte Ring. Le point d'entrée dans ce cas est le ConfigurationManager.cpp. Il aura besoin du callID, message et émetteur(***from***) afin de transmettre le message au Manager.cpp. Ce dernier récupère un RingAccountId afin de transmettre le message depuis RingAccount.cpp.

3.1.4 Vue physique

Diagramme de déploiement :

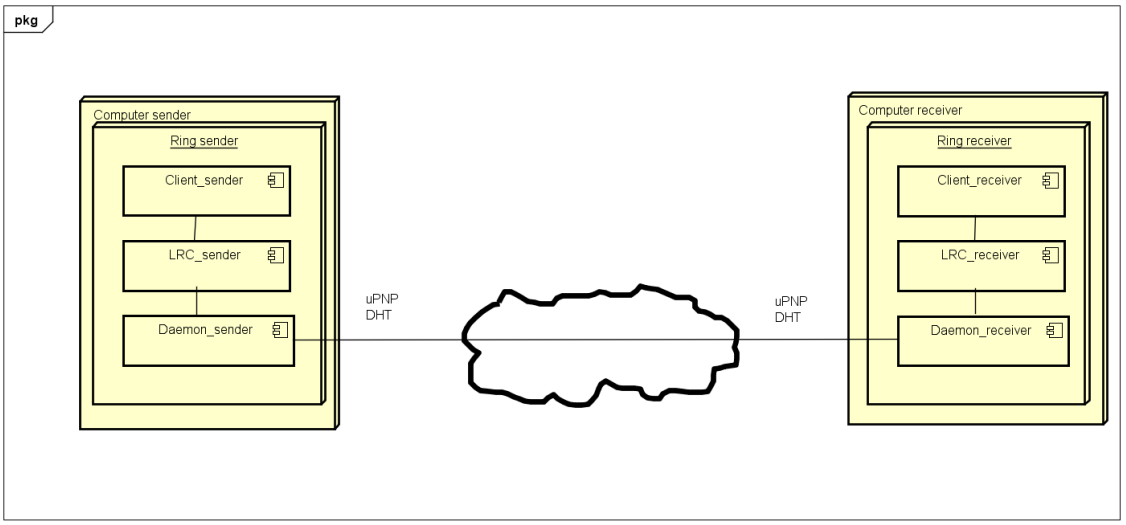


Figure 8: Diagramme de déploiement pour l'envoi de message via un compte Ring

Le diagramme de déploiement montre les deux ordinateurs qui établissent la communication pour envoyer un message en utilisant un compte Ring. Tous les clients embarquent de base le lrc et le daemon. La connexion point à point est faite en utilisant le protocole DHT.

3.2 Scénario 2 : Envoi de message via un compte SIP

Figure 9: Use case du scénario : Envoyer un message via un compte SIP

3.2.1 Vue Logique

Diagramme de classe :

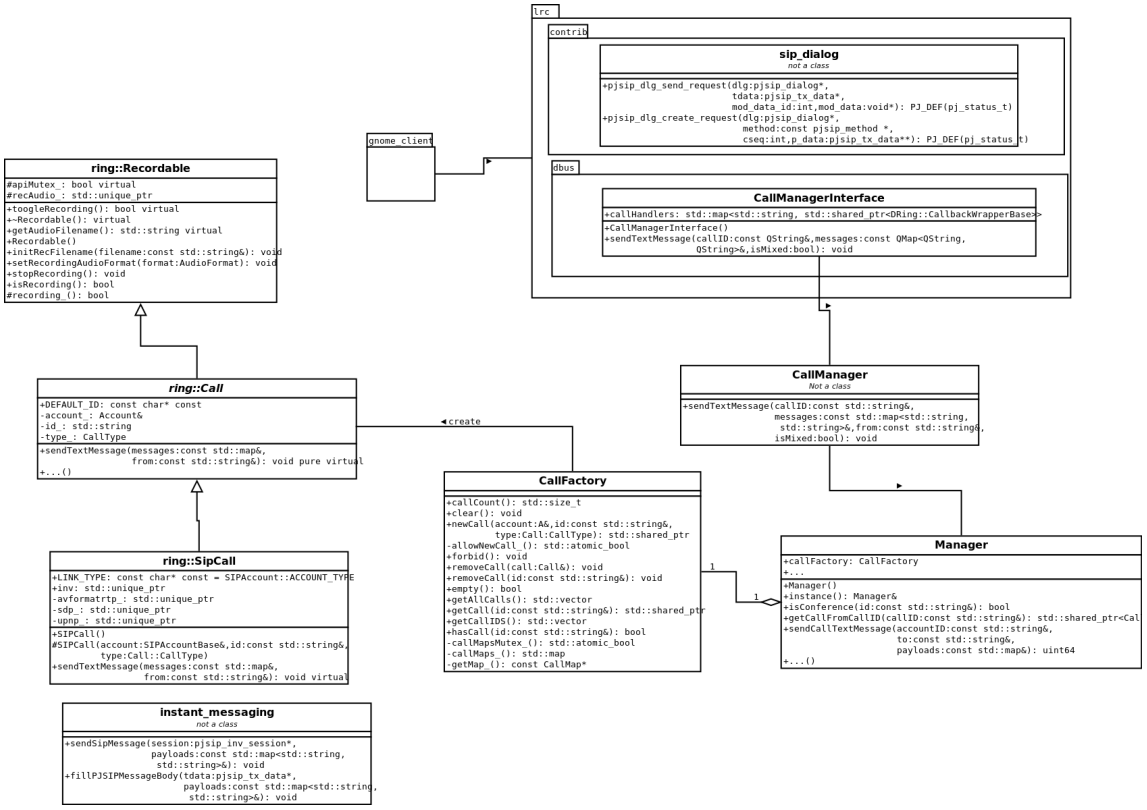


Figure 10: Diagramme de classe du scénario : Envoi de message via un compte SIP

Lors d'un envoi de message via un compte SIP, le principe est le même que l'envoi avec un compte Ring. La seule différence est au niveau du point d'entrée sur le daemon. Le lien lrc-daemon se fait au travers du CallManagerInterface (lrc) et du CallManager (daemon)

3.2.2 Vue Développement

Diagramme de composants : Figure 11

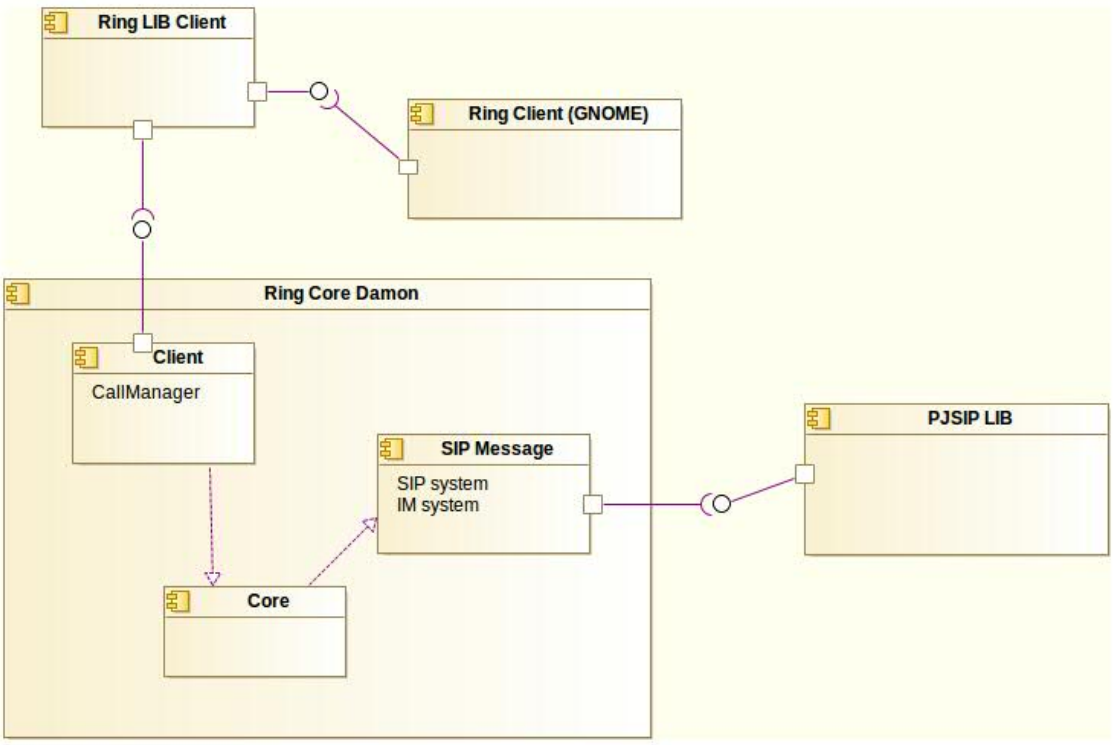


Figure 11: Diagramme de composant pour le scénario d'envoi de message avec un compte SIP

Justifications/ explications : Ce scénario est très similaire à celui du scénario 1 (Envoi de message via un compte RING) même au niveau du diagramme de composant. Les principales différences sont décrites dans les points ci-dessous : (Note le service demandé dans ce cas est toujours : *Send-Message*)

- **Ring Core Daemon** : L'une des principales différences est le point d'entrée au niveau du composant `Client`. La classe utilisée dans ce cas est le `CallManager` et non pas `ConfigurationManager`. Au niveau du composant `Core`, il joue toujours le même rôle mais avec des classes différentes : `Manager, Call, CallFactory`. (Note : la classe `Manager` devrait techniquement être présente dans tous les scénarios possibles car elle représente le contrôleur du projet.)
- **SIP Message** : Comme le composant `Ring Message` du précédent scénario, le composant `SIP Message` permet de construire le message et de l'envoyer. Deux principales différences : **SIP système** qui va s'occuper de la gestion de la communication SIP et **IM système** qui va se charger de la construction du corps du message. Le composant `SIP Message` utilise la librairie **PjSIP**

(représentée dans ce diagramme par `PJSIP LIB`) afin d'établir la communication SIP et d'envoyer les messages.

3.2.3 Vue processus

Diagramme de séquence:

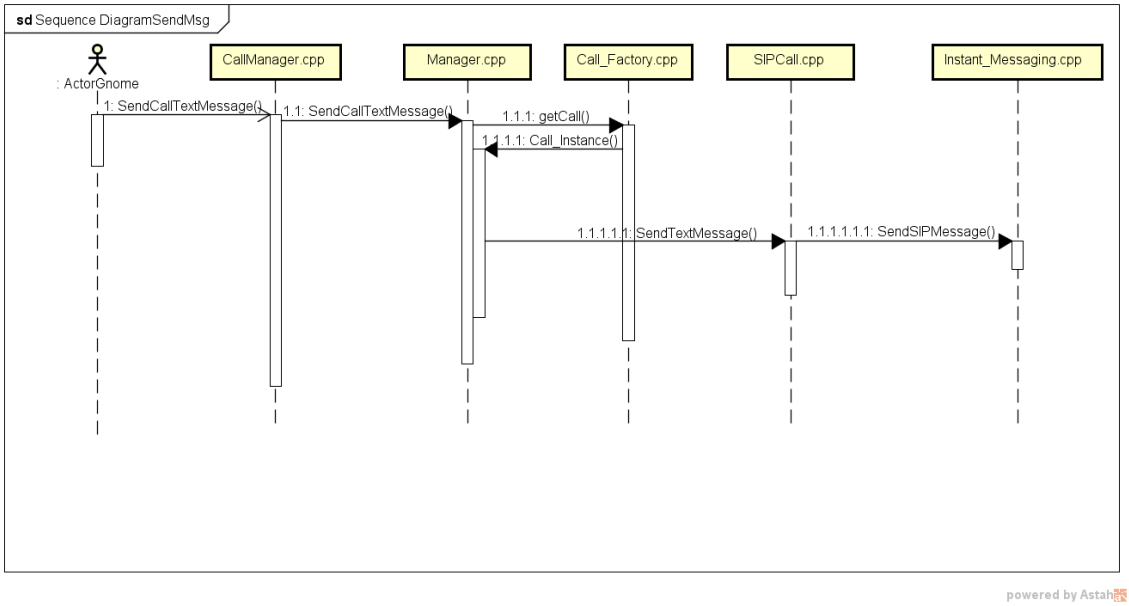


Figure 12: Diagramme de séquence pour l'envoi de message via un compte SIP

Justifications/ explications : Le diagramme ci-dessus décrit la séquence de l'envoi d'un message via un compte SIP.Le point d'entrée dans ce cas est le CallManager.cpp. Il aura besoin du callID, message et émetteur(from) afin de transmettre le message au Manager.cpp. Ce dernier récupère un SIPid afin de transmettre le message au destinataire.

3.2.4 Vue physique

Diagramme de déploiement :

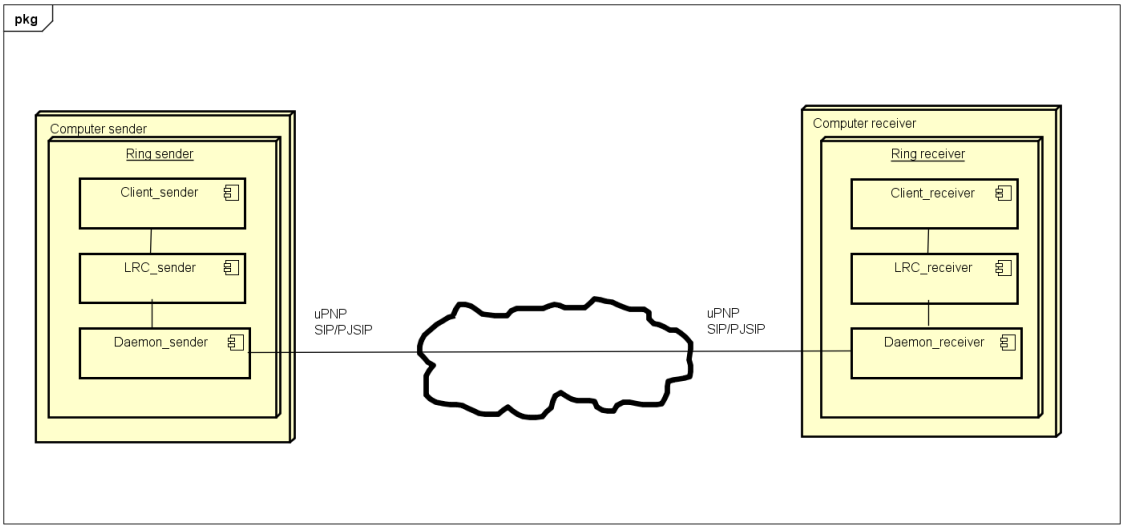


Figure 13: Diagramme de déploiement pour l'envoi de message via un compte SIP

Le diagramme de déploiement montre les deux ordinateurs qui établissent la communication pour envoyer un message en utilisant le protocole SIP. Tous les clients embarquent de base le lrc et le daemon. La connexion point à point est faite en utilisant les protocoles de communication SIP et PJSIP.

3.3 Scénario 3 : Réception d'un message

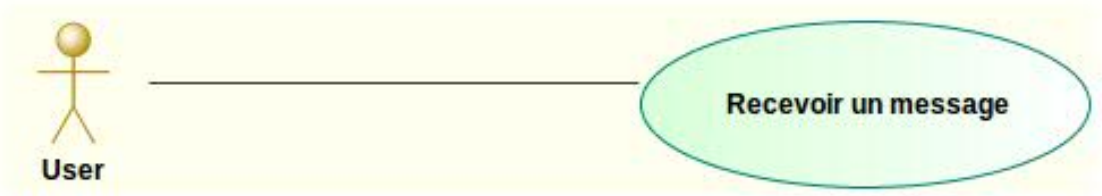


Figure 14: Use case du scénario :Recevoir un message

3.3.1 Vue Logique

Diagramme de classe :

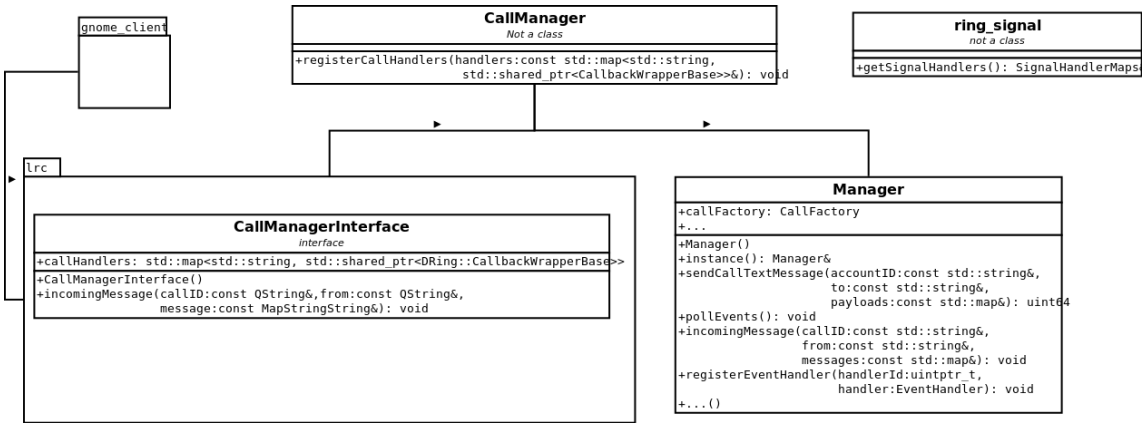


Figure 15: Diagramme de classe du scénario : Réception d'un message via un compte Ring

Pour la réception d'un message, la classe Manager est l'élément central qui va détecter un message entrant. Le CallManager enregistrera des Handlers, qui seront appelés lorsque le Manager aura détecté un message entrant. Le lien avec le lrc est fait au niveau du CallManagerInterface

3.3.2 Vue Développement

Diagramme de composants :

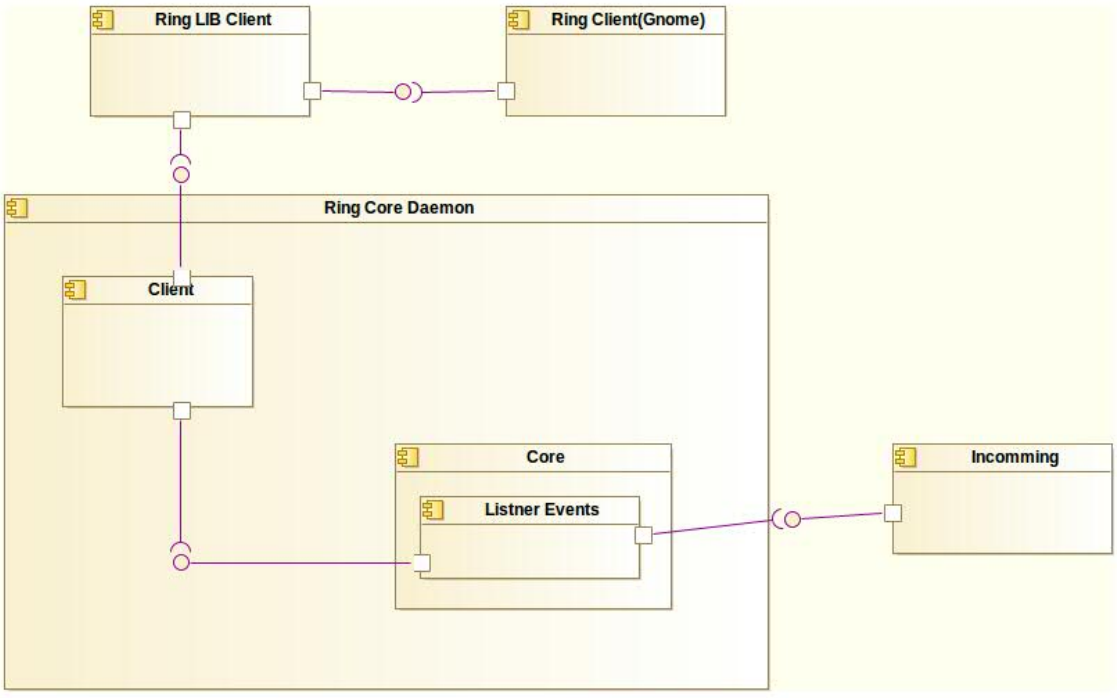


Figure 16: Diagramme de composant pour la réception d'un message.

Justifications/ explications : Dans le cas du scénario de réception de message nous allons plus parler d'*Event* que de service. Au niveau du diagramme 16 les demis-cercles représentent plus un processus d'**écoute** qu'une dépendance. Le point d'entrée dans ce scénario est le composant `Incomming`

- **Incomming** : Ce composant représente le point d'entrée de différents types d'évenements dans notre cas on gère la réception d'un message text.
- **Core** : Au niveau du core, le composant `Listner Events` est constamment en écoute afin d'intercepter les évenements. La classe principale qui héberge ce processus est la classe `Manager`.
- **Client** : Le client va s'occuper d'envoyer le signale aux couches supérieurs qui va notifier l'utilisateur qu'un message viens d'être reçu. La classe principale de ce composant est `Ring_signal`.

3.3.3 Vue processus

Diagramme de séquence:

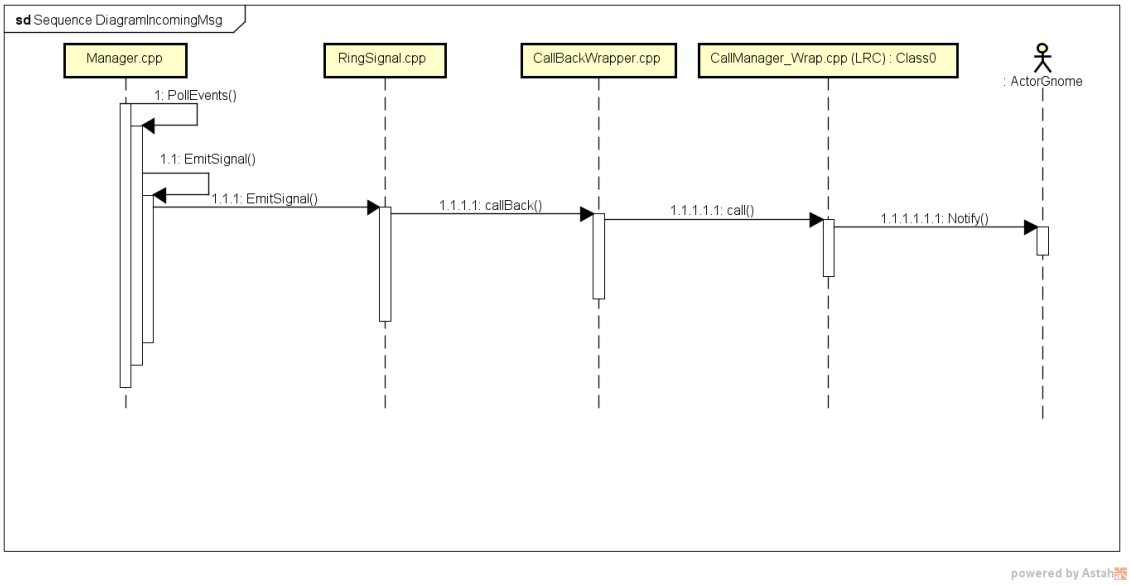


Figure 17: Diagramme de séquence pour la réception d'un message.

Justifications/ explications : Le diagramme ci-dessus décrit la séquence de la réception d'un message. Le manager se met préalablement en écoute sur les ***callBacks*** des différents signaux (dans ce cas c'est le signal d'***incoming message***). De plus le manager boucle infiniment dans la méthode pollEvents. En cas d'une réception d'un signal, le manager notifie ***RingSignal.cpp*** afin de notifier le ***callBackWrapper.cpp*** qui se charge de notifier les couches antérieurs (Couche LRC et client).

3.3.4 Vue physique

Diagramme de déploiement :

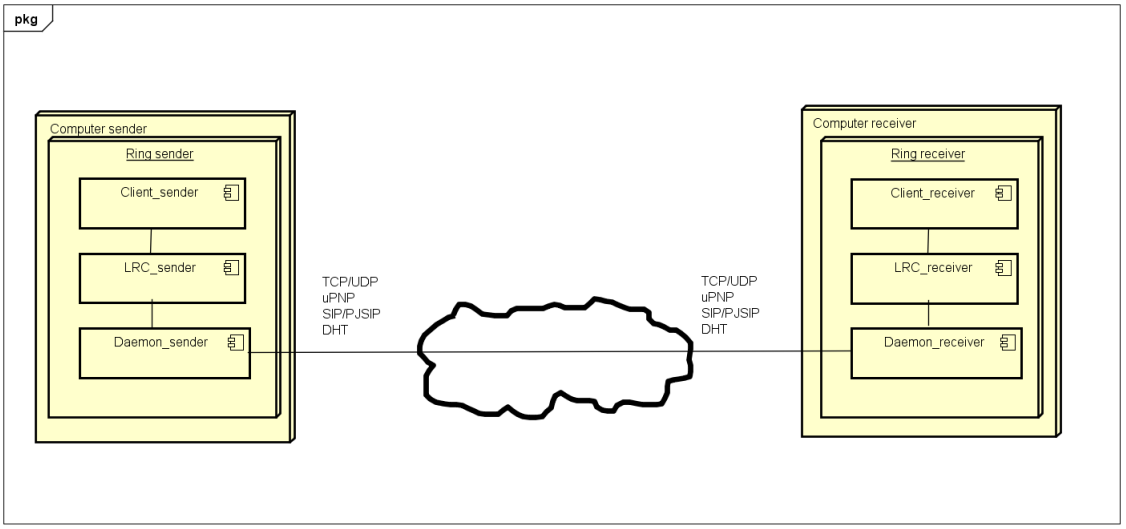
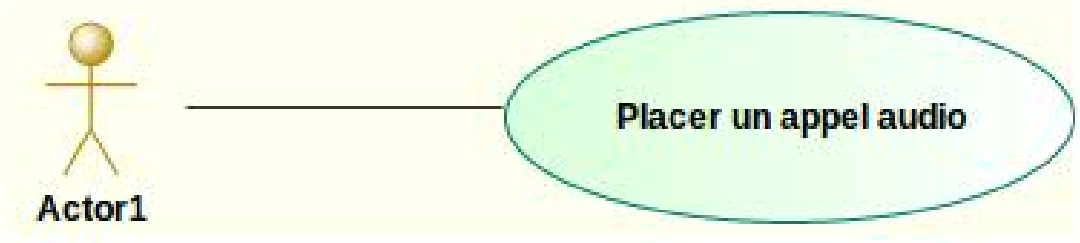


Figure 18: Diagramme de déploiement pour la réception d'un message.

Le diagramme de déploiement montre la réception d'un message. Tous les clients embarquent de base le lrc et le daemon. La connexion point à point est faite en utilisant les protocoles uPnP, SIP/PJSIP et DHT.

3.4 Scénario 4 : Effectuer un appel via un compte Ring



11

Figure 19: Use case du scénario : Effectuer un appel via un compte Ring

3.4.1 Vue Logique

Diagramme de classe :

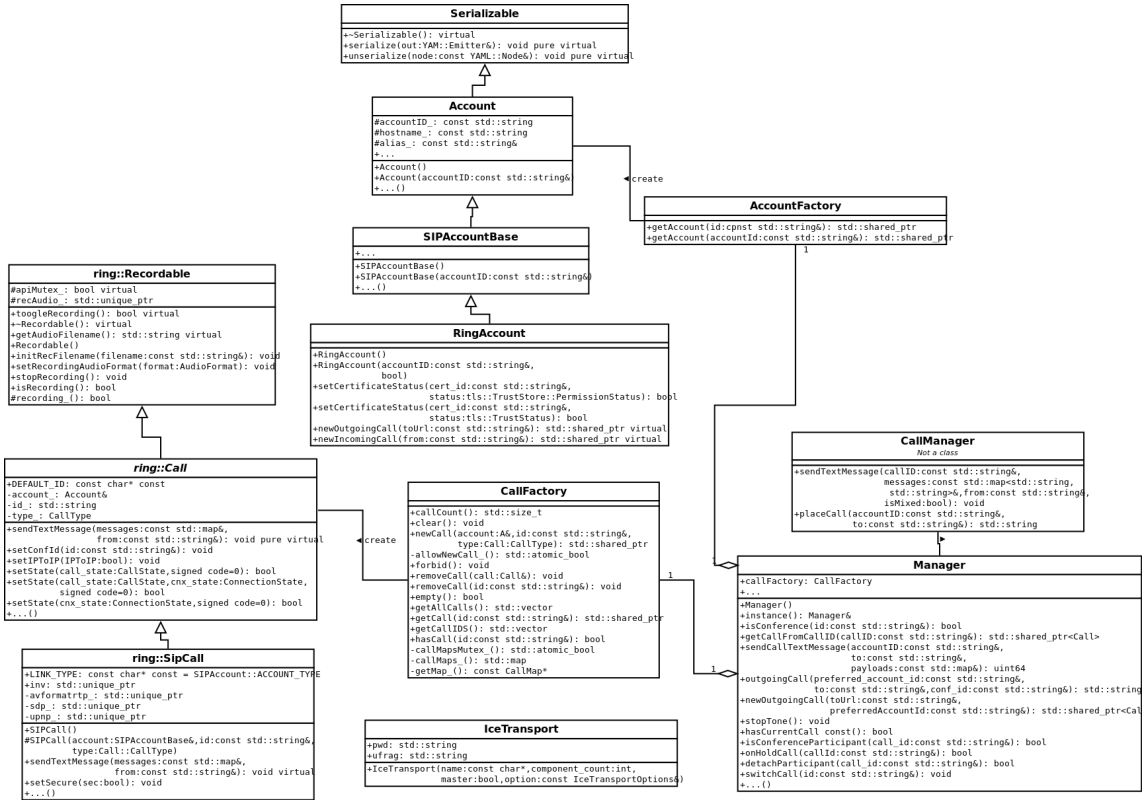


Figure 20: Diagramme de classe du scénario : Effectuer un appel via un compte Ring

Pour effectuer un nouvel appel, le point d'entrée du daemon est le CallManager. Puis le Manager avec son AccountFactory va pouvoir créer un RingAccount. Ce dernier dérive de SIPAccountBase qui lui dérive de Account. C'est le RingAccount qui va pouvoir créer l'appel avec la méthode "newOutgoingCall()".

3.4.2 Vue Développement

Diagramme de composants : Figure 21

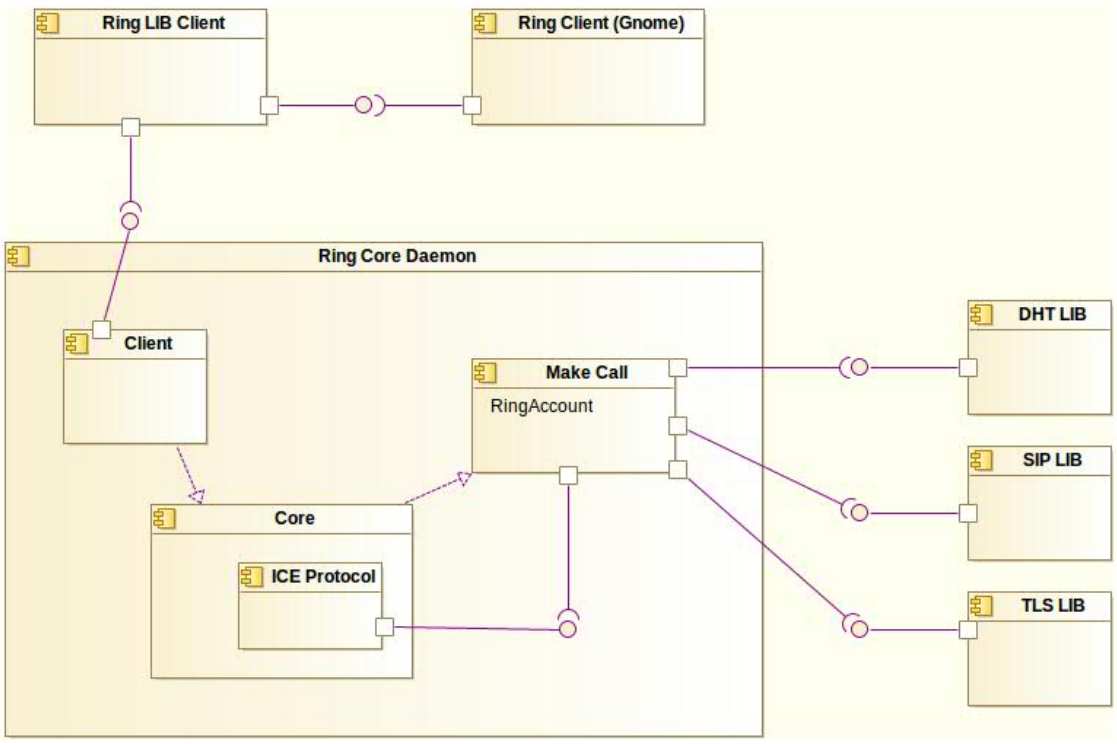


Figure 21: Diagramme de composant pour le scénario de passage d'appel via un compte Ring

Justifications/ explications : Le service demandé dans ce scénario est de pouvoir passer un appel (faire sonner le destinataire). Nous appellerons ce service : *Place-Call*. Les composants `Ring Client(Gnome)) et Ring LIB Client` sont les mêmes, la seule différence est la demande du service qui change. Ci-dessus les explications plus en détails des principaux composants :

- **Ring Core Daemon** : Le point d'entrés se trouve toujours dans le composant `Client` avec la classe `CallManager` comme principal acteur. Au niveau du composant `Core`, on retrouve toujours la classe `Manager` comme éternel contrôleur. En plus de la classe `Manager` on retrouve : `Account, AccountFactory` et le composant `ICE protocole` qui se chargera d'établir la communication. Le composant `Core` va faire appel au composant `Make Call` qui est principalement constitué de la classe `RingAccount`. Le composant `Make Call` va se charger de finaliser le service : récupérer les informations sur le destinataire, établir la communication **P2P**, initialiser le protocole de communication **ICE**, envoi de données via le DHT, sécurisation de la communication... et enfin passer l'appel. Afin de pouvoir réaliser les opérations citées précédemment,

le composant `Make Call` fait appel aux librairies **DHT, SIP** et **ICE protocole**(définit dans le composant `Core`)

3.4.3 Vue processus

Diagramme de séquence:

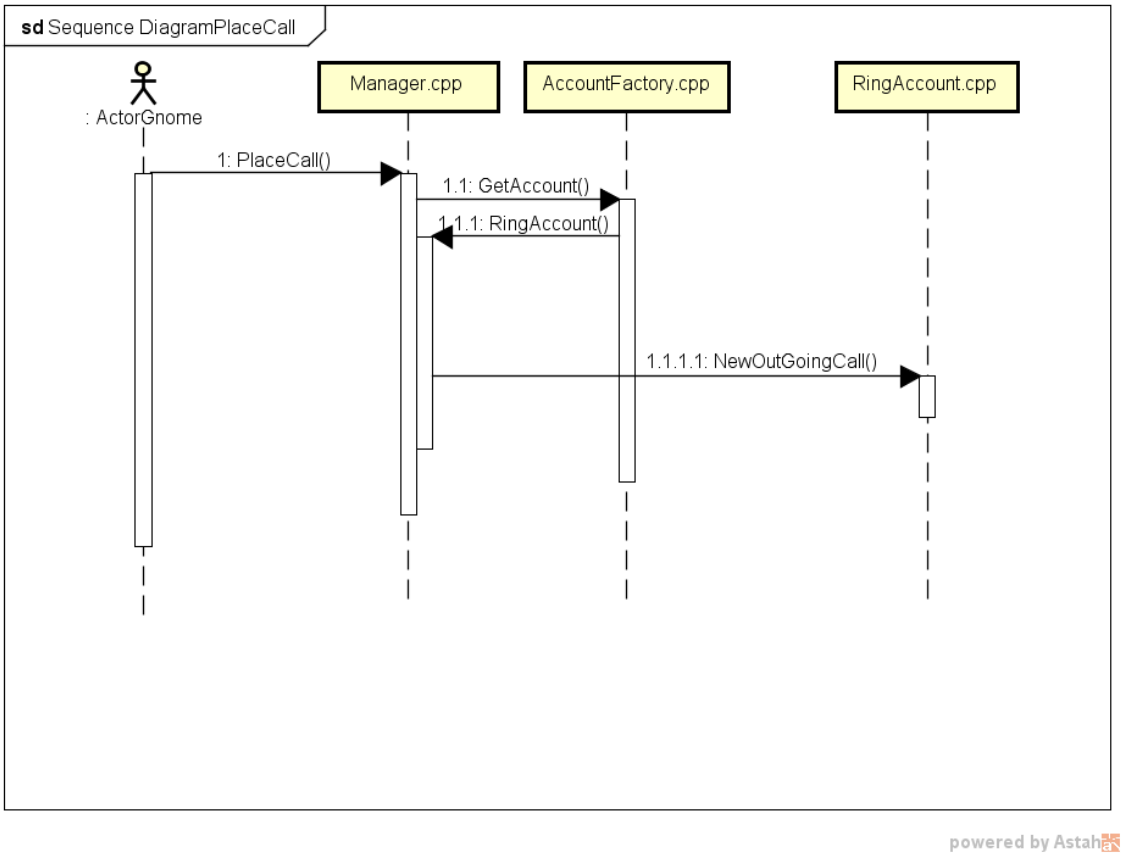


Figure 22: Diagramme de séquence pour effectuer un appel via un compte Ring.

Justifications/ explications : Le diagramme ci-dessus décrit la séquence de la réception d'un appel. Dès que le Manager reçois l'appel de la fonction ***PlaceCall*** depuis les couches antérieurs (couche LRC), il récupère un ***RingAccountID*** afin d'appeler la méthode ***NewOutGoingCall*** qui permettra de placer l'appel.

3.4.4 Vue physique

Diagramme de déploiement :

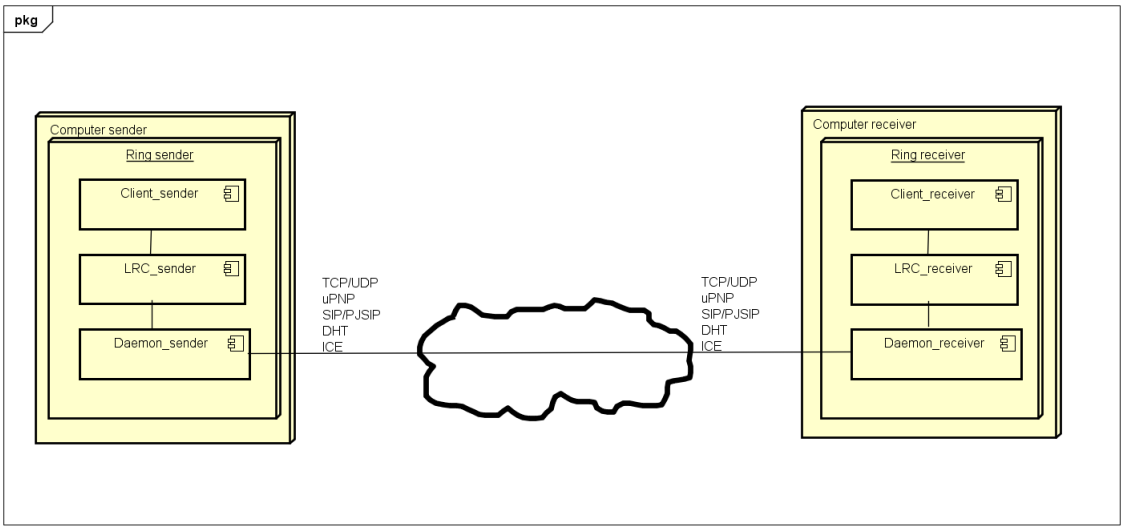


Figure 23: Diagramme de déploiement pour effectuer un appel via un compte Ring.

Le diagramme de déploiement montre l'établissement d'un appel entre deux ordinateurs en utilisant Ring. La connexion point à point est faite en utilisant les protocoles uPnP, SIP/PJSIP ou DHT et ICE.

4 Conclusion

Lors de cette première phase d'analyse nous avons pu voir les principaux acteurs (au niveau classes et fonctions) du projet Ring.

Lors de ce projet nous nous sommes focalisés principalement sur l'analyse du ***daemon*** qui représente à nos yeux l'une des parties les plus intéressantes du projet. Durant notre analyse nous avons pu remarquer l'apparition répétée de certaines classes (*Manager, (Account/Call)_Factory, CallManager*) sur les quatre scénarios étudiés, ce qui peux être éventuellement notre point d'entrée pour la prochaine étape du TP.

Le projet Ring reste néanmoins grand et complexe (*plus complexe!*), notre équipe est donc en constante analyse du projet afin de mieux le prendre en main et de le comprendre. Les diagrammes présents dans ce document ne représentent qu'une partie du projet. Il reste encore énormément d'endroits à visiter et à analyser afin de pouvoir fournir une représentation complète et correcte du projet.