École Nationale Supérieure des Télécommunications de Bretagne
Projet d'Ingénieur. Printemps 2005.

mardi 28 juin 2005

RAPPORT TECHNIQUE

(version finale)

Projet P14

Système P2P de partage de bookmarks

Auteurs :

- Céline ABI ZEID DAOU
- Stéphane NDONGNA NDOUNIA
- Hai-Nam NGUYEN
- Amélien RETOURNÉ

Destinataires :

- M. Michel PLU (industriel)
- M. Serge GARLATTI (encadrant)
- M. Roger WALDECK (encadrant)

En partenariat avec france telecom R&D

La gestion de bookmarks est rentable si l'information est bien gérée et réutilisée. Notre projet conçoit un système pair-à-pair permettant d'avoir le meilleur résultat dans le partage de bookmarks.

The bookmarks management is useful if information is well organized and reused. Our project elaborates a peer-to-peer system bringing the best results in bookmarks sharing.

TABLE DE MATIERE

1. Introduction

Rédacteur : Stéphane. Relecteur : Hai-Nam.

La recherche de ressources Web devient de plus en plus difficile sur le WWW dont la structure est très complexe et le contenu est très hétérogène. Devant l'immensité de ce dernier, les moteurs de recherche ne sont pas toujours satisfaisants et parfois la recherche d'informations sur le Web est considérée comme une perte de temps. C'est pourquoi la gestion de bookmarks est très efficace et permet de faciliter la localisation de sources d'informations sur la toile. En plus, il est très utile et important d'exploiter les relations entre les utilisateurs du Web et, par la même occasion, leur permettre de collaborer ensemble pour faciliter cette tâche de gestion. La collaboration se traduit par le partage de bookmarks entre eux afin d'améliorer la recherche et l'identification de ressources pertinentes sur internet. Ainsi, ce n'est plus l'utilisateur qui navigue d'une information à une autre mais c'est l'information qui navigue d'un utilisateur à un autre.

Notre projet vise à concevoir et prototyper un système logiciel P2P dans lequel les utilisateurs s'échangent des URLs de ressources Web catégorisées et stockées dans leurs bases personnelles. Il sera développé en utilisant des logiciels libres et des protocoles et formats de données standards et déjà existants comme Jabber et RDF. Ceci va faciliter l'intégration et la compatibilité avec d'autres systèmes.

Ce document va présenter les besoins du client c'est-à-dire les spécifications fonctionnelles et contraintes du logiciel à réaliser ; puis le fonctionnement et notre utilisation de Jabber, RDF/Jena, et la base de données ; ensuite la modélisation logicielle ; et enfin l'interface homme machine.

2. Besoins du client

Rédacteur : Céline. Relecteur : Hai-Nam.

Le but de ce projet est de réaliser, comme son titre l'indique, une réplication de bases personnelles de bookmarks. En effet il s'agit de réaliser un système de *partage de bookmarks* entre utilisateurs dont chacun dispose d'une base de données personnelle contenant les bookmarks qui l'intéressent. Un projet similaire, connu sous le nom de « Web of people » a été développé par le service R&D de France Télécom. Il favorise un système *distribué* de connaissances permettant à l'utilisateur de retrouver plus facilement et plus efficacement les informations désirées. Les nœuds du système sont les utilisateurs et les informations (bookmarks) navigueront entre ces utilisateurs, contrairement au web traditionnel où les utilisateurs naviguent d'une information à l'autre pour pouvoir retrouver celle qui les intéresse.

2.1. Spécifications

La réplication d'une base de données permet d'avoir des copies d'informations de cette base dans toutes les autres (dont les utilisateurs appartiennent à une liste de diffusion de celle-ci, notion détaillée plus tard). La réplication se fait soit d'une façon asynchrone (réplication paresseuse) soit d'une manière synchrone. Dans les deux cas, toute modification dans une base, si celle-ci est connectée, sera diffusée immédiatement. Pourtant la communication se passe parfois en mode asynchrone puisque la base (ou les bases) réceptrice(s) dans ce cas est (sont) déconnectée(s) pendant le partage, elle(s) recevra et traitera la modification une fois re-connectée.

Expliquons comment sont stockées les informations et comment elles sont gérées.

En fait, toutes les données sont représentées par le format RDF. Celui-ci définit un statement comme étant le triplet formé par un objet, une relation (ou propriété) et une valeur que prend l'objet : (O R V). Lorsqu'il y a une modification d'un triplet d'une base, ce qui implique modification d'une donnée dans la base, la réplication permet aux autres bases personnelles de détecter cette modification.

Les données dans la base sont organisées sous la forme d'*étiquettes* (topic) et de *revues* (post). Les étiquettes représentent les thèmes qui intéressent les utilisateurs. Elles contiennent plusieurs références (revues) concernant ce thème. La revue est ainsi associée à un titre, à une URL, à un commentaire et à une ou plusieurs étiquettes.

Il est important de savoir faire la différence entre URL[1] et URI[2]. Une URI doit permettre d'identifier une ressource de manière permanente, même si la ressource

[1] Uniform Resource Locator en Anglais, c'est littéralement une « repère uniforme de ressource »
[2] Uniform Resource Identifier en Anglais, soit littéralement « identifiant uniforme de ressource »

est déplacée ou supprimée. Une URL, informellement appelée *adresse Web*, permet de préciser un endroit ou location sur le WWW. Dans notre projet, les URLs sont utilisées pour indiquer les adresses Web et les URIs sont attachées aux étiquettes, aux revues et aux utilisateurs.

On peut créer autant d'étiquettes que l'on veut, celles-ci sont hiérarchiques, il existe des étiquettes et des sous-étiquettes. Ainsi cette structure d'étiquettes et de sous-étiquettes s'appelle taxonomie. Chaque utilisateur a sa propre taxonomie dans sa base personnelle. Prenons un exemple :

```
a1 Réseaux
        a11 connexions
             a111 évolution
        a12 protocoles
b1 Internet
```

Pour permettre le partage de revues, on associe à chaque étiquette une *liste de diffusion* (ldf) qui contient tous les utilisateurs intéressés par cette étiquette et qui veulent recevoir des revues concernant cette étiquette. Donc on peut définir la relation suivante : toute revue « r » est accessible par un utilisateur « u » si et seulement si « u » appartient à la liste de diffusion du topic « t » tel que « t » appartient aux étiquettes de « r » :

→ Accessible(r, u) ssi u ∈ ldf(t) et t ∈ étiquettes(r).

Les sous-étiquettes héritent par défaut de la liste de diffusion de l'étiquette dont elles dérivent, mais il est possible de modifier cette liste héritée en supprimant ou ajoutant des utilisateurs. Toutes modifications seront diffusées vers le bas c'est-à-dire tout au long de la hiérarchie par l'héritage traditionnel. Chaque gestionnaire de la base peut suivre son principe de gestion d'héritages qui est local à chaque base.

Ainsi la base personnelle d'un utilisateur contient ses revues et étiquettes personnelles ainsi que les étiquettes dont celui-ci appartient à la liste de diffusion.

Un utilisateur peut supprimer une étiquette personnelle ou une étiquette reçue. S'il s'agit une étiquette dont il n'est pas le propriétaire, une notification sera transmise aux abonnés de cette étiquette afin de leur permettre d'abonner directement à la source de cette étiquette. En plus, une fois que l'utilisateur reçoit une revue qui l'intéresse, il doit pouvoir laisser une copie de cette revue dans sa base. En effet, il garde toutes les infos nécessaires comme le titre, les commentaires, les URLs de cette revue mais aussi il ajoute sa propre annotation et il lui attribue des étiquettes personnelles.

Il est intéressant de pouvoir trier les annotations par date (similaire à Outlook) et d'avoir aussi la possibilité de communiquer (par chat, mail ….) avec la personne qui a envoyé une étiquette.

Pour effectuer la diffusion de revues, on va se servir du flux RSS qui représentera une source d'URLs appartenant à un utilisateur virtuel. Il faut pouvoir

analyser le flux pour en tirer l'URL associée et le message commentaire (si le flux ne contient pas d'URLs, on mettra une URL aléatoire). Pour analyser le flux RSS, on peut utiliser Informa[3]. Une fois le flux est reçu, il sera analysé pour en tirer les informations précédentes, il faut lui associer une étiquette et *un utilisateur virtuel*. De ce fait, il est possible de permettre à un utilisateur de s'ajouter à la liste de diffusion de l'étiquette concernée par ce flux.

2.2. Contraintes

On rappelle que la base personnelle d'un utilisateur contient ses étiquettes personnelles et les étiquettes dont il appartient à la liste de diffusion. Quand une modification plus spécialement une suppression d'une étiquette se produit, celle-ci est répliquée dans les bases où l'étiquette est normalement diffusée. Toutefois l'utilisateur qui recevait des revues de cette étiquette supprimée garde toujours ses annotations personnelles qu'il a déjà créées à partir de celles-ci. Il est également en mesure de re-créer cette étiquette et annoncer comme sa propre étiquette.

Quand l'utilisateur consulte ses annotations, il consulte sa propre base, il n'y a pas de langage de requête à distance.

Le protocole utilisé dans la couche de communication et qui permet d'échanger des messages entre les bases personnelles est le XMPP. Celui-ci permet l'intégration de logiciels libres de communication et la gestion des utilisateurs et de leur présence. Pour ceci, on doit comprendre le fonctionnement du Jabber et pouvoir l'utiliser avec le langage Java qu'on a choisi (il y a sans doute plein d'autres langages qui peuvent être utilisés).

La structure des données des bases personnelles ainsi que les messages échangés sont représentés par le format RDFS modélisé par un schéma spécifique. Celui-ci définit des classes et des propriétés que l'on peut utiliser dans notre système pour décrire les données et les actions prises sur celles-ci.

Notons aussi qu'il faut développer une IHM (Interface d'Humain Machine : interface utilisateur) pour effectuer les tests fonctionnels du système. Celle-ci peut être développée soit à partir d'une application Java stand-alone, soit à partir d'un serveur web permettant de consulter la base personnelle à l'aide d'un navigateur.

[3] Site web du projet : http://informa.sourceforge.net/

3. Jabber

Rédacteur : Céline. Relecteur : Amélien.

La réplication de bases personnelles de données ne peut pas être réalisée sans avoir implémenté un protocole de communication assurant le transfert de messages entre les bases des différents utilisateurs. Ainsi au niveau de la couche communication, le protocole XMPP de Jabber va pouvoir être utilisé. Celui-ci est un protocole appartenant aux logiciels libres. Basé sur la technologie XML pour le format des données, il permet aux utilisateurs Jabber d'envoyer et de recevoir des messages en temps réel. En fait, Jabber est un système de messagerie et de présence instantanées (Instant Messaging : IM) similaire aux systèmes MSN, ICQ, Yahoo et AIM.

3.1. Bénéfices de Jabber

Le choix de Jabber comme protocole de messagerie instantanée dans notre projet a été mûrement réfléchi. Plusieurs raisons ont poussé notre client à choisir ce système et nous allons en citer quelques-unes :

- Libre : Les protocoles Jabber sont libres, gratuits, publics et simples à comprendre ;
- Standardisé : L'Internet Engineering Task Force (IETF) a formalisé le protocole dont le noyau utilise XML et ainsi l'a baptisé sous le nom XMPP et édité ses caractéristiques dans les RFC 3920 et 3921 ;
- Décentralisé : Aucune organisation ou groupe n'a le droit de contrôler le système. Par conséquent, n'importe qui peut créer son propre serveur Jabber, l'utiliser et former son propre réseau ;
- Extensible : Ayant XML dans sa base de développement, Jabber permet à n'importe quelle personne d'étendre ses fonctionnalités ;
- Interopérabilité : Les utilisateurs Jabber ont la possibilité de communiquer avec des utilisateurs appartenant à d'autres systèmes de messagerie.

3.2. Principe et fonctionnement de Jabber

Le fonctionnement du système Jabber est très similaire à celui de la messagerie normale (email). En fait, les deux systèmes se basent sur le même principe de communication qui est assurée par des serveurs distribués et des clients spécialisés. Le client se connecte à un serveur, reçoit les messages de l'utilisateur, envoie des messages à d'autres utilisateurs connectés au même serveur ou à d'autres serveurs d'un autre domaine. Cependant, le système Jabber diffère des systèmes de messagerie classiques du fait qu'il permet la délivrance de messages en temps réel. La notion de temps réel est très importante et elle est possible parce que le serveur Jabber est capable de savoir si l'utilisateur est connecté ou non (online ou offline). Les contacts de l'utilisateur peuvent aussi le savoir s'ils sont autorisés à le faire par cet utilisateur. Cette information connue sous le nom de « présence » renforce le concept de messagerie instantanée.

Il faut préciser aussi deux autres caractéristiques du système Jabber. La première consiste sur l'utilisation de XML pour le format des données. La seconde assure des adresses uniques basées sur DNS dont la forme est similaire aux adresses email user@host.

3.2.1. Architecture de Jabber

Le système Jabber suit l'architecture client/serveur. Celle-ci oblige à passer par une connexion vers un serveur afin de transférer les données d'un client à un autre. Actuellement, il est possible à 2 clients d'établir une connexion directe (peer-to-peer) pour le transfert de données et spécialement pour le transfert de fichiers.

Le client Jabber établit une connexion TCP avec le serveur Jabber à travers le port 5222. La connexion entre les serveurs se fait à travers le port 5269. La connexion client/serveur n'est rompue qu'à la fin de la session de l'utilisateur.

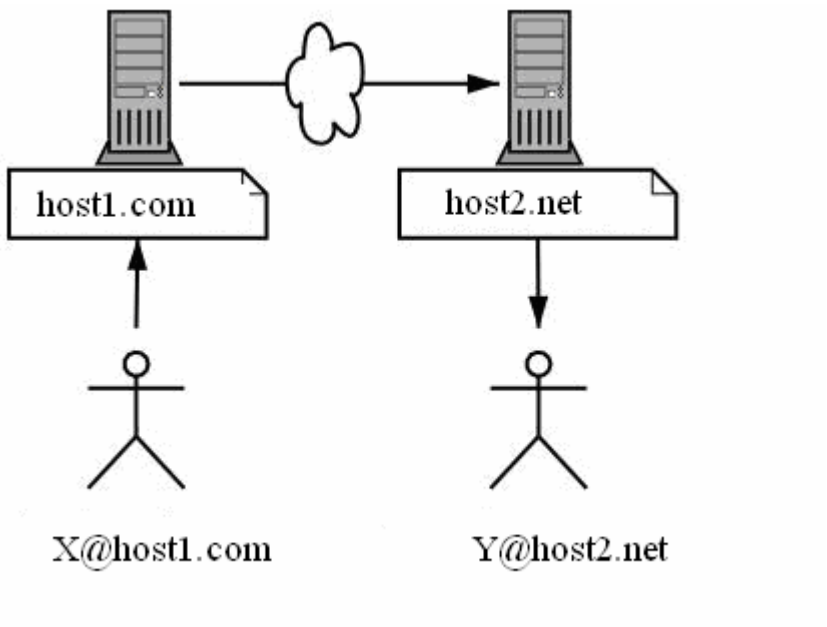


Figure 1. Architecture de Jabber

Puisque la connexion est toujours active durant la durée de la session d'un utilisateur U, le serveur délivre immédiatement le message destiné à U si celui-ci est connecté. Sinon, le serveur stocke le message et le délivre une fois que l'utilisateur U se reconnecte.

Pour résumer le rôle du serveur et celui du client, nous pouvons dire :

a) Le serveur permet de remplir trois tâches principales :

- Gérer les connexions vers les clients et communiquer avec eux.
- Communiquer avec d'autres serveurs.
- Coordonner ses différents composants.

En fait, le serveur peut supporter plusieurs fonctionnalités comme l'authentification, la registration, les listes des contacts des utilisateurs, le stockage des messages,… En plus, grâce à la flexibilité de Jabber il peut supporter l'ajout de nouveaux services et composants comme des interfaces vers d'autres systèmes de messagerie.

b) Le client doit pouvoir :

- Communiquer avec un serveur en établissant, avec lui, une connexion TCP.
- Parser et interpréter les données XML qu'il reçoit.
- Comprendre les différents types des données (message et présence).

3.2.2. Format des données en XML

La technologie XML est à la base du système Jabber. Ceci permet de structurer les données et de rendre le protocole plus extensible. Quand un client se connecte à un serveur, deux flots de XML sont établis, un à chaque direction. Ainsi toutes les communications entre le client et le serveur se produisent sur ces deux flots suivant le format XML. Voici un exemple de message envoyé de X@host1.com vers Y@host2.net.

```
<message from ='X@host1.com' to ='Y@host2.net'>
    <subject> Rendez-vous </subject>
    <body> Bonjour, comment ça va? </body>
</message>
```

Cet exemple est très simple à comprendre. Il indique l'émetteur, le récepteur, le sujet et le contenu d'un message. Grâce à XML, il est possible d'étendre la structure et d'ajouter d'autres fragments aux messages, permettant ainsi l'ajout d'autres fonctionnalités comme le RSS et autres.

3.2.3. L'adressage Jabber

Dans un réseau Jabber, différentes entités ont besoin de communiquer. Ces entités représentent les serveurs, les utilisateurs, les groupechat,…Ainsi les adresses Jabber appelés Jabber IDs (JID) sont utilisées pour pouvoir identifier les différentes entités afin de permettre le routage des données entre elles.

Chaque entité du réseau possède un JID unique qui permet de l'identifier. Ce dernier est facile à mémoriser. En plus, il est flexible de façon à donner la possibilité à l'utilisateur Jabber de communiquer avec des utilisateurs appartenant à d'autres systèmes d'IM.

Le JID est formé de plusieurs éléments : un nœud, un domaine et une ressource suivant le format suivant : [noeud@]domaine[/ressource]

Le premier identifiant est le domaine qui correspond au serveur Jabber auquel l'utilisateur est connecté.

Le second identifiant correspond au nœud qui représente l'utilisateur en question.

Le troisième est la ressource qui est un identifiant optionnel. En fait, celle-ci correspond à des équipements ou à des localisations appartenant à un même utilisateur.

Par exemple, X@host1.com/work et X@host1.com/home.

De cette façon, l'utilisateur a la possibilité d'établir plusieurs connexions simultanées avec le serveur Jabber.

3.3. Son application au niveau du logiciel.

Après avoir compris le fonctionnement de Jabber, il fallait choisir une API qui permet d'intégrer Jabber dans notre programme. Notons bien que notre programme est complètement écrit en Java. Ainsi, l'API choisie est en Java aussi. En se référant sur les différentes librairies qui existent sur le site officiel de Jabber, nous avons choisi l'API smack (version 1.5). Celle-ci supporte toutes les fonctionnalités nécessaires à notre projet. En plus, elle est simple à utiliser.

Le travail à faire au niveau de cette partie concerne alors la programmation réseaux. Les fonctionnalités nécessaires sont l'ouverture d'une connexion avec le serveur Jabber, l'envoi de messages à certains utilisateurs, la réception de messages et la fermeture de la connexion. Le code correspondant est écrit dans une seule classe « Jabber.java ».

La première fonctionnalité est traduite au niveau de la méthode « connect » qui permet en premier lieu d'ouvrir une connexion avec le serveur désiré à travers le port concerné (5222 dans notre cas). En second lieu, une procédure d'authentification se produit pour permettre à l'utilisateur de se logger au niveau de son serveur. Évidemment cette opération ne peut réussir si l'utilisateur n'a pas déjà créé son propre compte ou ne possède pas un JID sur le serveur correspondant.

L'opération d'envoi de messages se produit chaque fois qu'il y a un certain changement (modification, ajout ou suppression d'un topic/post) au niveau d'une base personnelle d'un utilisateur appartenant au système de notre projet. Elle est réalisée par la méthode « sendMessage » qui permet de saisir le récepteur, le sujet et le contenu du message afin de l'envoyer vers la bonne destination sur la connexion déjà établie.

L'opération de réception de messages se produit évidemment en réponse à la précédente.

Au niveau de la programmation, celle-ci est fortement liée au concept du thread. Un thread va permettre à cette fonctionnalité de tourner sans interruption et parallèlement avec d'autres processus. De cette façon, l'utilisateur est toujours informé lorsqu'un nouveau message lui arrive. La réception de messages est traduite au niveau de la fonction « receiveMessage ».

Enfin, il est toujours possible à l'utilisateur de se déconnecter. La fermeture de la connexion est réalisée par la fonction « disconnect ».

4. RDF/Jena

Rédacteur : Amélien. Relecteur : Céline.

4.1. Définition de RDF/Jena

4.1.1. Définition de RDF/RDFS

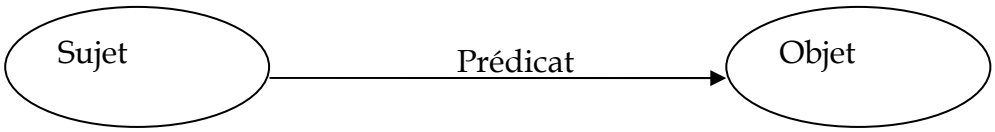
RDF est un graphe orienté et étiqueté dont la syntaxe est écrite en XML. Il s'agit d'un formalisme utilisé pour représenter les propriétés d'une ressource et les valeurs de ces propriétés ; il contient donc trois types d'objets : les ressources, les propriétés et les valeurs. Une ressource peut être une page HTML, une partie d'une page HTML, un ensemble de pages, un objet ou toute entité qui peut être accédée par un identificateur (URI). Il est possible de créer des données instances de RDF basées sur des schémas multiples provenant de sources multiples. Ces schémas descriptifs peuvent être écrits eux-mêmes en RDF : c'est le RDFS.

L'entité élémentaire du modèle RDF est un triplet [*sujet*, *prédicat*, *objet*] qui représente une description (ou méta-donnée) sur la ressource.

- *Sujet* est la **ressource** que l'on veut décrire
- *Prédicat* est une **propriété** de la ressource
- *Objet* est la **valeur** pour la propriété

4.1.2. Les triplets en RDF

Le triplet [sujet, prédicat, objet] est traduit ainsi en RDFS :



4.1.3. La syntaxe en RDF

*Le triplet [**sujet**, **prédicat**, **objet**] est traduit ainsi en syntaxe XML :*

```
<rdf:Description about=sujet>
    <prédicat>objet</prédicat>
</rdf:Description>
```

4.1.4. Définition de Jena

Jena est une API Java très performante qui permet notamment de jouer le rôle de parseur (analyseur syntaxique) de document RDF. Ainsi, Jena permet de manipuler des données écrites en RDF et d'accéder à l'information individuelle contenue dans ces données. Jena est un programme open source.

Jena possédait donc toutes les qualités que l'on recherchait pour le projet, et c'est tout naturellement que notre choix s'est porté sur cette API.

4.2. Utilisation de RDF dans l'environnement du projet

4.2.1. Le schéma RDFS du modèle de Web Of People :

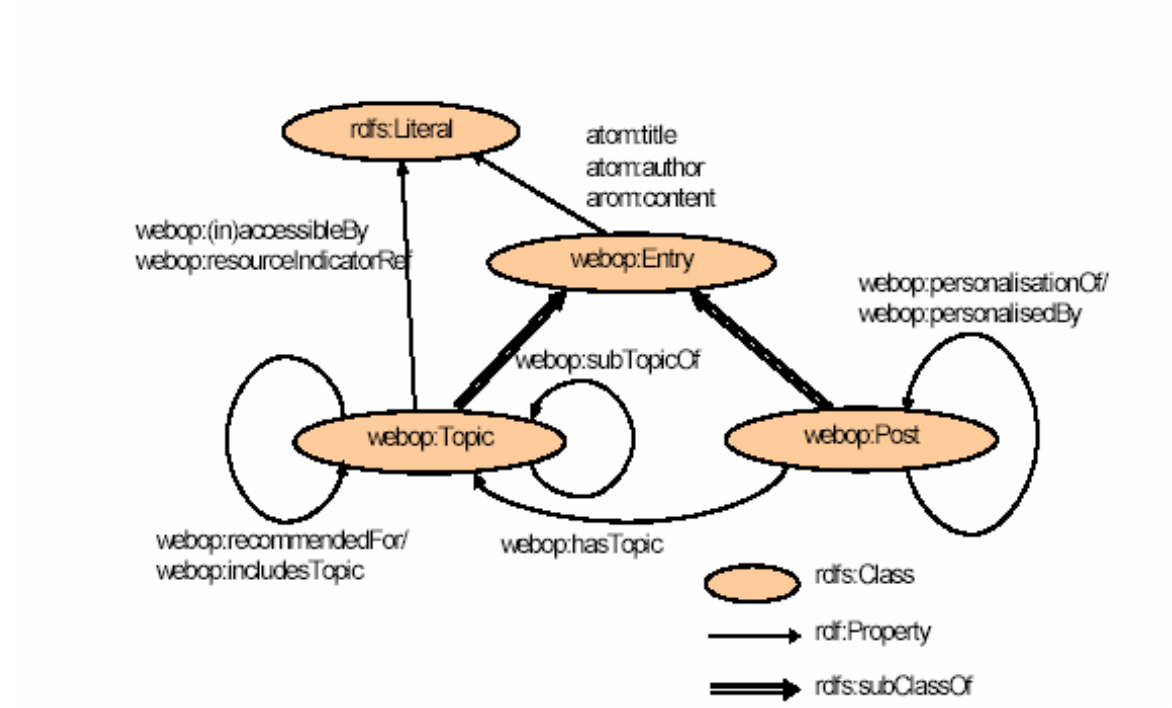


Figure 2. Le schéma RDFS du modèle de Web Of People

Ce schéma RDFS a été la base de notre réflexion sur la partie RDF. En effet, le Web Of People ou Webop nous a permis de déduire tous les triplets que nous devions utiliser afin d'exécuter les différentes actions possibles pour l'utilisateur.

En annexe est mis le fichier RDF décrivant ce schéma RDFS.

4.2.2. Les triplets déduits du schéma RDFS

Les actions sont divisées en 3 groupes : création, suppression et modification (suppression, puis création). Il nous suffit de détailler seulement les actions de type « création », pour celles du type « suppression », c'est un triplet « inverse » (c'est-à-dire en remplaçant dans le même triplet l'évènement de création par l'évènement de suppression).

- Créer une étiquette :

```
[ webop:Resource, atom:title, rdfs:Literal ]
[ webop:Resource, atom:author, rdfs:Literal ]
[ webop:Resource, atom:content, rdfs:Literal ]
```

- Créer une revue :

```
[ webop:Resource, atom:title, rdfs:Literal ]
```

```
[ webop:Resource, atom:author, rdfs:Literal ]
[ webop:Resource, atom:content, rdfs:Literal ]
[ webop:Post, webop:resourceRef, rdfs:Literal ]
```

- attacher une revue à une étiquette :

```
[ webop:Post, webop:hasTopic, webop:Topic ]
```

- attacher une étiquette à une (autre) étiquette :

```
[ webop:Topic, webop:includesTopic, webop:Topic ]
[ webop:Topic, webop:subTopicOf, webop:Topic ]
```

- attacher une annotation à une revue :

```
[ webop:Resource, atom:title, rdfs:Literal ]
[ webop:Resource, atom:author, rdfs:Literal ]
[ webop:Resource, atom:content, rdfs:Literal ]
[ webop:Post, webop:resourceRef, rdfs:Literal ]
[ webop:Post, webop:personalisationOf, webop:Post ]
[ webop:Post, webop:personalisedBy, webop:Post ]
```

- abonner un utilisateur :

```
[ webop:Topic, webop:accessibleBy, rdfs:Literal ]
```

- exclure un utilisateur :

```
[ webop:Topic, webop:inaccessibleBy, rdfs:Literal ]
```

- attacher une annotation à une étiquette: non réalisé

Il y à également quelques triplets déduits de ce Schéma RDF qui n'ont pas été utilisés :

```
[ webop:Topic, rdfs:subClassOf, webop:Resource ]
[ webop:Post, rdfs:subClassOf, webop:Resource ]
[ webop:Topic, webop:resourceIndicatorRef, rdfs:Literal ]
[ webop:Topic, webop:recommendedFor, webop:Topic ]
```

4.3. Utilisation de Jena dans le projet

4.3.1. Pourquoi avoir utilisé Jena

Jena a joué un rôle central dans notre application, puisque nous l'avons utilisé en tant que parseur et créateur RDF. En fait, Jena nous a permis de créer un objet à partir d'un fichier RDF, c'est son rôle de parseur RDF, et par opération inverse de créer un fichier RDF à partir d'un objet, ici c'est le rôle de créateur RDF.

Grâce à cette API, nous disposions d'objet à la place de document RDF, ce qui nous a permis par la suite d'effectuer facilement toutes les actions nécessaires pour la gestion de bookmarks.

4.3.2. Comment utiliser Jena ?

Jena est une API relativement simple à utiliser : dans un premier temps Jena lit le document RDF grâce à un *ByteArrayInputStream*. Ensuite Jena crée un objet de type *Model* qui contient tous les triplets. Ces derniers sont contenus dans des objets *Statement*. Le Model permet de récupérer la liste de ces *Statement*, ainsi il nous a fallu traiter chacun de ces *Statement*. En fait, nous avons créé la structure de donnée *Triple* qui correspondait plus à l'idée que nous nous faisions des triplets au sens RDF.

En effet, le type Triple contient les sujets, prédicats et objets, c'est donc un format qui nous est propre et qui correspond exactement aux triplets RDF classiques. Ceci a permis d'améliorer considérablement la gestion des triplets et la lisibilité du programme.

Puis nous avons du adapter et convertir les *Statement* en *Triple* car les premiers sont relativement complexes à utiliser et gérer de manière brute, c'est pourquoi les seconds répondaient plus à nos exigences de conception. Par la même occasion nous avons pu définir et créer des objets de type *Message* permettant quant à eux de gérer et référencer les messages transmis par Jabber (ceux qui contiennent les données RDF).

5. Base de données

Rédacteur : Hai-Nam. Relecteur : Amélien, Céline.

Comme dans de nombreuses applications, nous sauvegardons des données dans le disque lors de la fermeture du logiciel et nous les reprenons lors du démarrage du programme. Au cours du lancement, toutes les données sont stockées dans la mémoire principale (sous formes d'objets Java) et sont accessibles directement.

Afin de simplifier la gestion, dans un premier temps, nous sérialisons les objets puis nous les mettons dans la base sous forme d'une paire <nom, valeur>. Nous ne pouvons pas sauvegarder l'objet principal (ici l'objet de classe Main) parce qu'il contient lui-même un objet qui gère la base de données (et donc gère la sauvegarde). Nous avons défini une classe qui contient toutes les données à sauvegarder (information de l'utilisateur, les listes des étiquettes, des revues et des abonnés, une liste des messages RDF reçus mais pas encore traités). Finalement, pour qu'un objet soit sérialisable, sa classe doit implémenter l'interface `Serializable` et ses attributs sont sérialisables.

```
class Container implements Serializable {
    private static final long serialVersionUID =
15646132573L;
    protected User owner;
    protected Collection<Topic> topics;
    protected Collection<Post> posts;
    protected Collection<User> users;
    protected Collection<RdfMessage> messages;
}
```

Pour la liberté de choix de la gestion de base de données, une interface est définie :

```
package p14.util;
public interface Database {
    public abstract void initialisation();
    public abstract void close();
    public abstract Object loadObject(String name);
    public abstract void saveObject(String name, Object
value);
}
```

Trois implémentations de cette interface ont été envisagées par des raisons multiples : avec SQLite (`DatabaseSQLite`), avec XML (`DatabaseXML`) et avec fichier binaire (`DatabaseFile`).

5.1. SQLite

SQLite est un Système de Gestion de Bases de Données Relationnel (SGBDR) écrit en C. Il est très petit, compact, rapide, ne demande aucune configuration et stocke une base de données complète dans un fichier.

Malheureusement il est écrit en C et requiert donc des distributions différentes sous différentes plateformes (Windows, UNIX, Linux…). Un wrappeur et un pilote JDBC pour Java sont disponibles[4] mais ils sont encore en phase de test[5]. En plus, notre programme ne demande pas beaucoup d'activités avec la base de données, nous la sollicitons d'une manière relativement simple. Une implémentation a pourtant été mise en place. La sérialisation et désérialisation sont faits par `ObjectOutputStream.writeObject()` et `ObjectInputStream.readObject()`, respectivement.

5.2. XML

XML est la solution universelle pour le stockage et le transfert des informations. La lecture est gérée par SAX et DOM s'occupe de l'écriture. Puisqu'un document XML est un fichier texte, les objets sérialisés sont encodés en Base64[6]. L'implémentation de Base64 dans Java est réalisée par Robert Harder[7], qui fait presque tout de travail : `Base64.decodeToObject(String s)` pour récupérer l'objet d'origine et `Base64.encodeObject(Serializable o)` pour sérialiser l'objet sérialisable `o`.

Toutes les valeurs sont stockées dans un fichier XML unique, elles seront toutes lues et mises dans une table de hachage[8] afin de les réutiliser ultérieurement quand nous aurons besoin d'autres valeurs.

5.3. Fichier binaire

En utilisant des méthodes traditionnelles dans Java, nous pouvons stocker *chaque objet* sérialisé dans *un* fichier binaire. Cela pourrait poser des problèmes si nous avons plusieurs objets à sauvegarder.

Dans notre programme, des données de revue, d'étiquette et d'utilisateur ont des relations entre eux. Il nous faut sauvegarder le tout dans un objet sérialisable, c'est donc une seule chaîne de caractères à stocker. De ce fait, nous avons des avantages quand nous la gérons directement par un fichier :

---

[4] Site officiel : http://www.ch-werner.de/javasqlite/

[5] Jusqu'à 28 mai 2005, support pour SQLite3 est vraiment expérimenté, testé sous Linux seulement

[6] Un codage de l'information utilisant 64 caractères, choisis pour être disponible sur la majorité des systèmes

[7] Site officiel : http://iharder.net/base64

[8] Une table de hachage est un tableau dans lequel on accède à un élément à l'aide d'une *clé*. Chaque clé correspond à un seul élément.

- Utiliser directement les méthodes prévues et fournies par Sun dans Java ;
- Passer outre à l'encodage en Base64 ;
- Être indépendant de la plateforme.

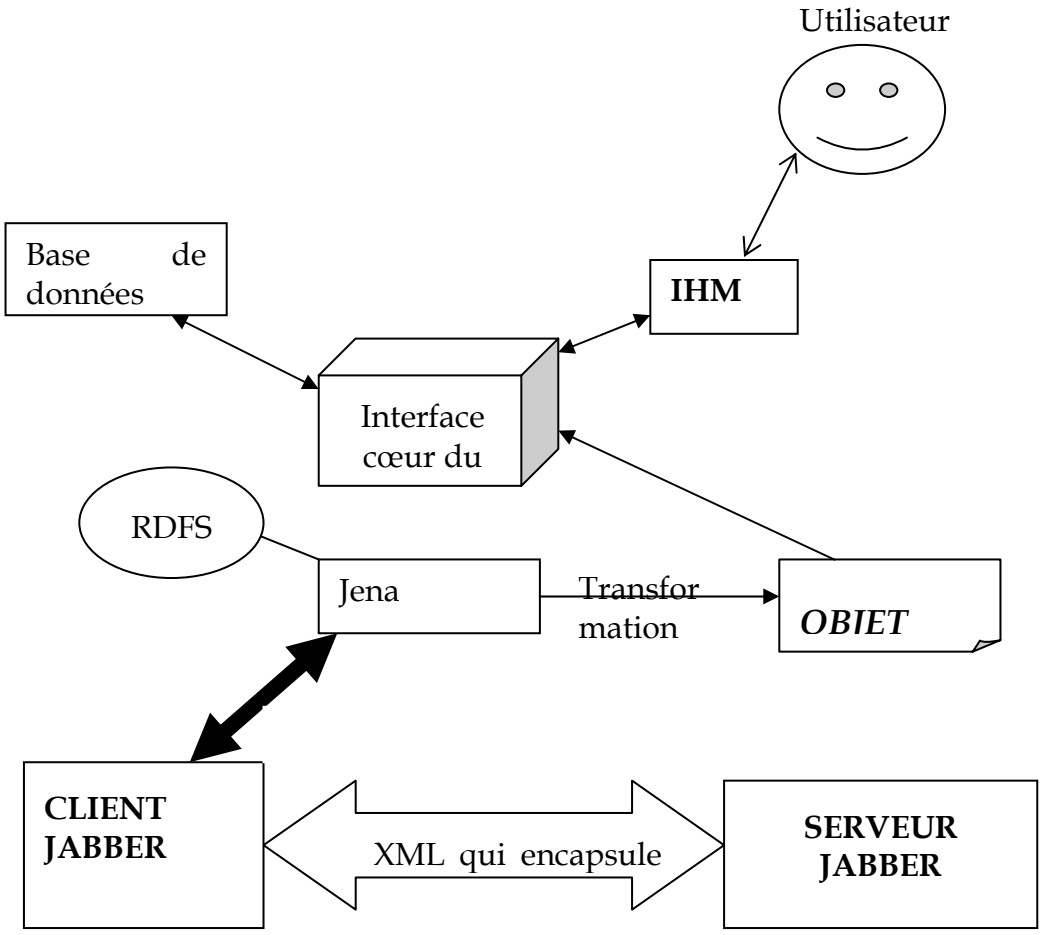
L'implémentation avec fichier est utilisée par défaut.

6. Modélisation

Rédacteur : Stéphane. Relecteur : Céline.

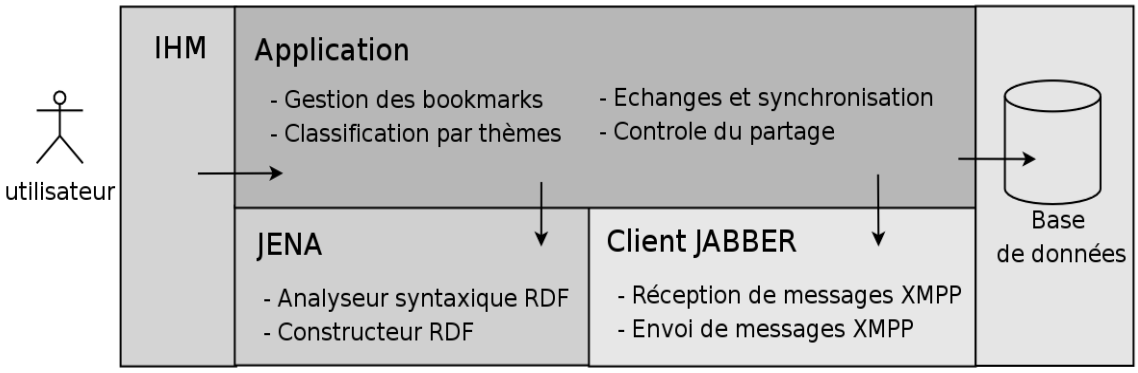
6.1. Architecture générale

Les spécifications fonctionnelles nous ont amené à définir l'architecture logicielle ci-dessous :



C'est une représentation générale des entités qui entrent en jeu pour le fonctionnement du système. Elle traduit ceci :

1. Un utilisateur interagit avec le cœur du système Java via l'interface graphique ;
2. Le cœur Java manipule des objets (que nous détaillerons par la suite) qu'il construit à partir de données stockées dans la base personnelle ou provenant du parseur RDF Jena ;
3. Jena sert aussi à construire les messages RDF qui seront expédiés à d'autres utilisateurs via un client Jabber/XMPP.



Le cœur logiciel contient toute la logique applicative du système, il est constitué de classes contrôleur en charge de la gestion effective des échanges de bookmarks et la synchronisation des bases personnelles. C'est ici qu'on retrouve les classes du modèle comme les revues, les étiquettes (thèmes), et les utilisateurs.

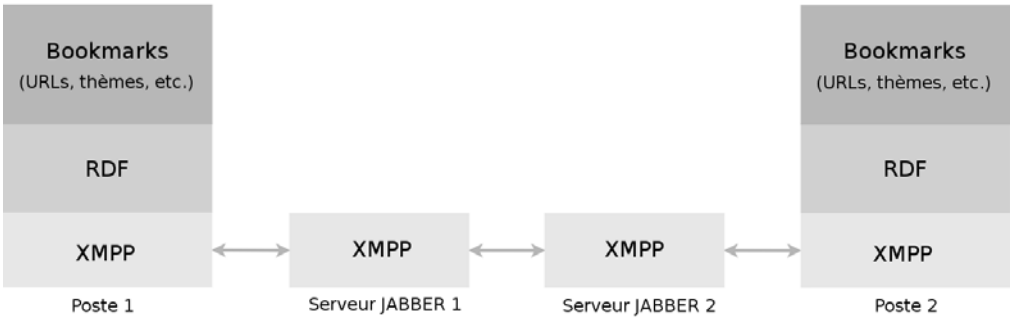
Le formalisme RDF est utilisé pour représenter :

- les bookmarks (revues) et leurs propriétés (URL, auteur).
- les étiquettes et leur hiérarchie (nom, étiquette parente).
- les ajouts, suppression, et mise à jour éventuelles des revues, et la hiérarchie des étiquettes.

C'est la bibliothèque Jena que nous utilisons pour effectuer l'analyse syntaxique à la réception, et la construction à l'envoi, de messages RDF.

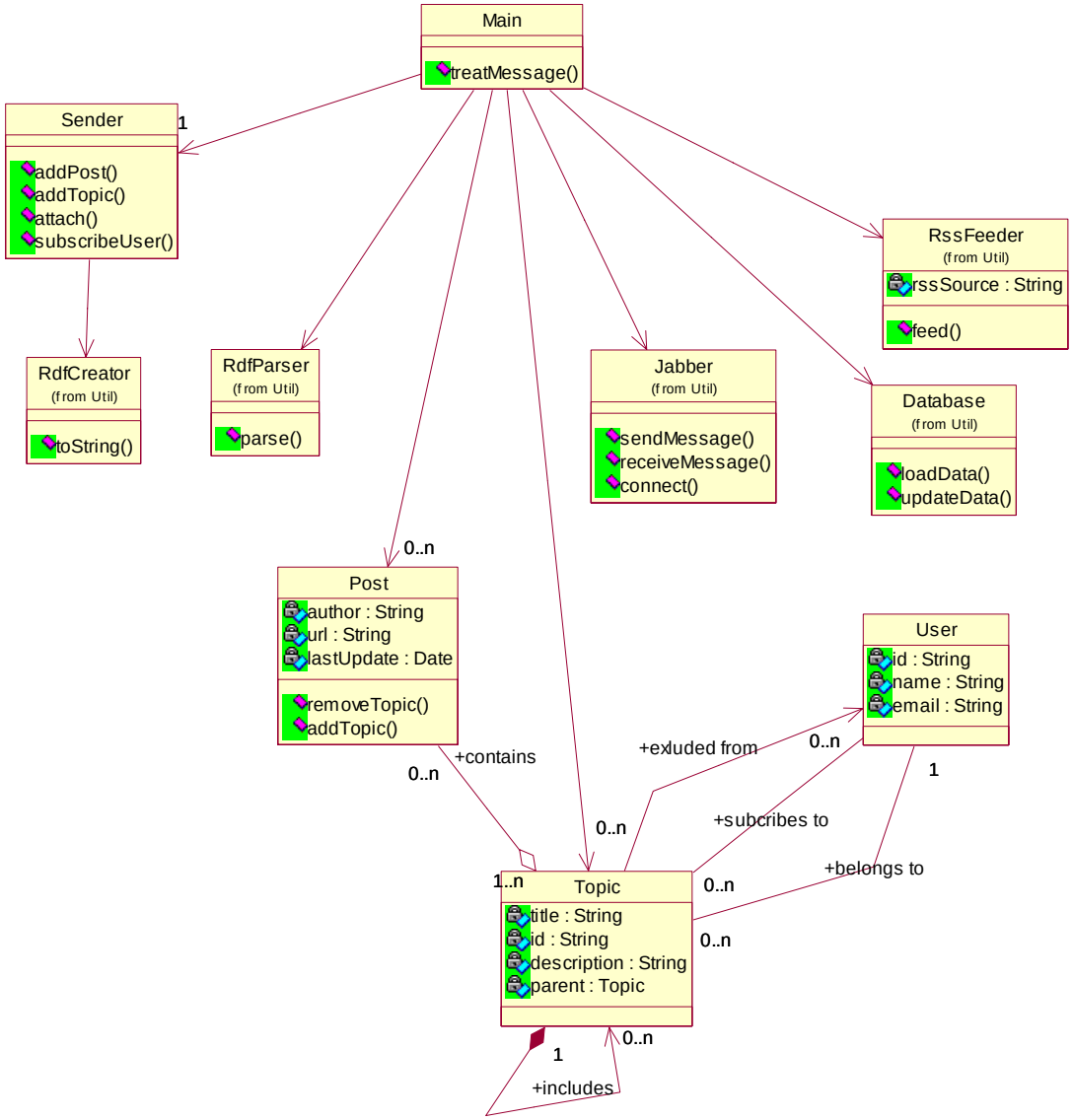
Un client Jabber est utilisé pour l'envoi et la réception effective des messages textes préalablement formatés en RDF. La propriété peer-to-peer et l'asynchronisme des échanges sont donc réalisés grâce au système Jabber (et le protocole XMPP) qui gère déjà cet aspect.

L'architecture en couche du système d'acheminement des bookmarks est représentée sur la figure ci-dessous.



6.2. Diagramme de classes

Les classes principales du cœur logiciel sont représentées sur ce diagramme de classes UML.



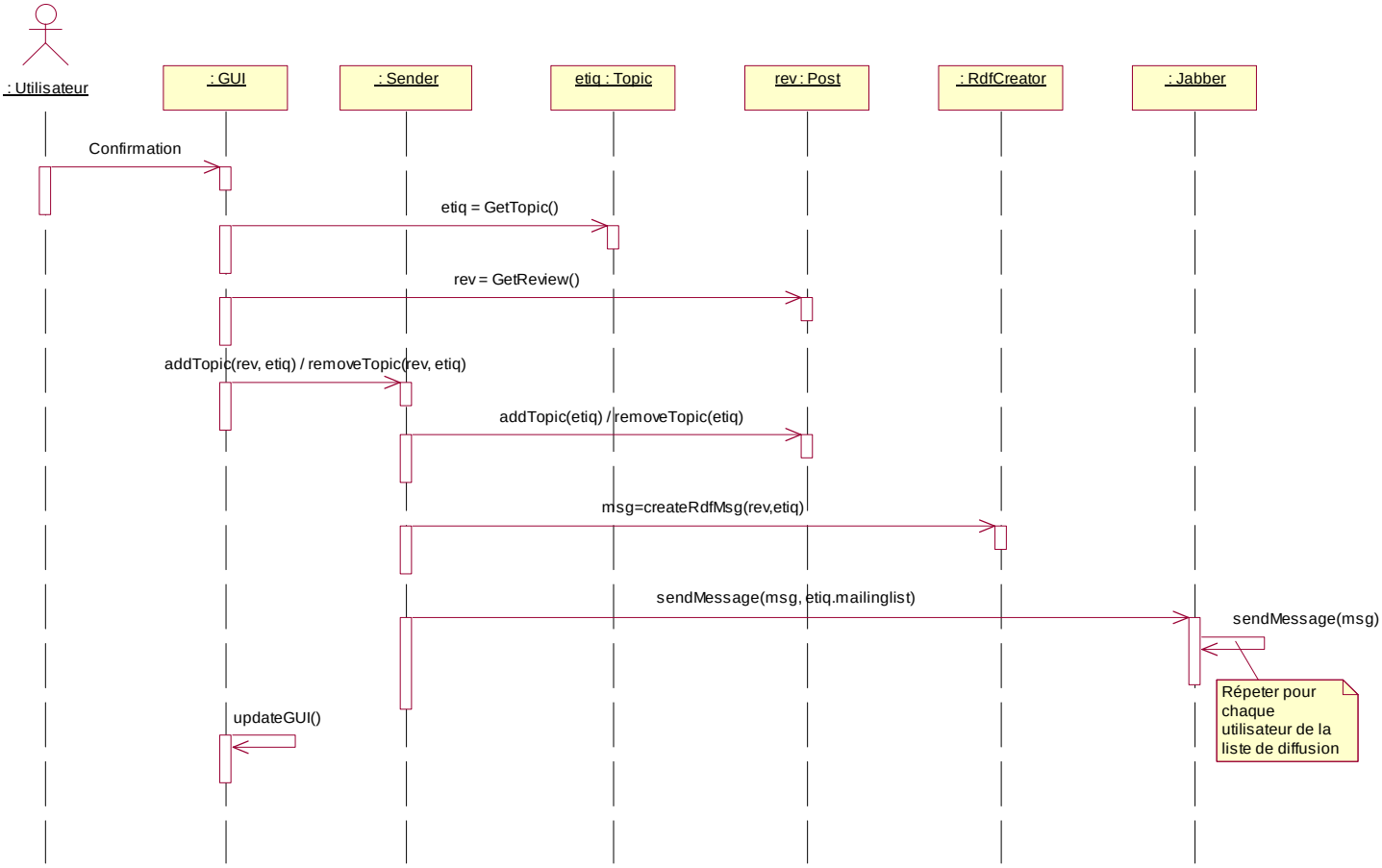
Les classes *User*, *Post*, et *Topic* permettent de modéliser respectivement les utilisateurs, les revues (bookmarks) et les étiquettes (thèmes). Ce sont les classes métier du modèle.

Les classes *Jabber*, *RDFParser*, *RDFCreator*, *Database* et *RSSFeeder* modélisent les outils que nous avons décrits dans les parties précédentes. Elles réalisent respectivement la communication XMPP, l'analyse et la construction de messages RDF, le stockage des données dans la base personnelle, et l'acquisition de flux RSS.

Les classes *Main* et *Sender* gèrent l'échange contrôlé et la synchronisation des bookmarks et étiquettes. *Main* exécute les actions associées aux messages entrants, et *Sender* celles générant des messages sortants.

Afin de bien comprendre la dynamique entre toutes ces classes, examinons un cas d'utilisation du système à travers 2 diagrammes de séquence.

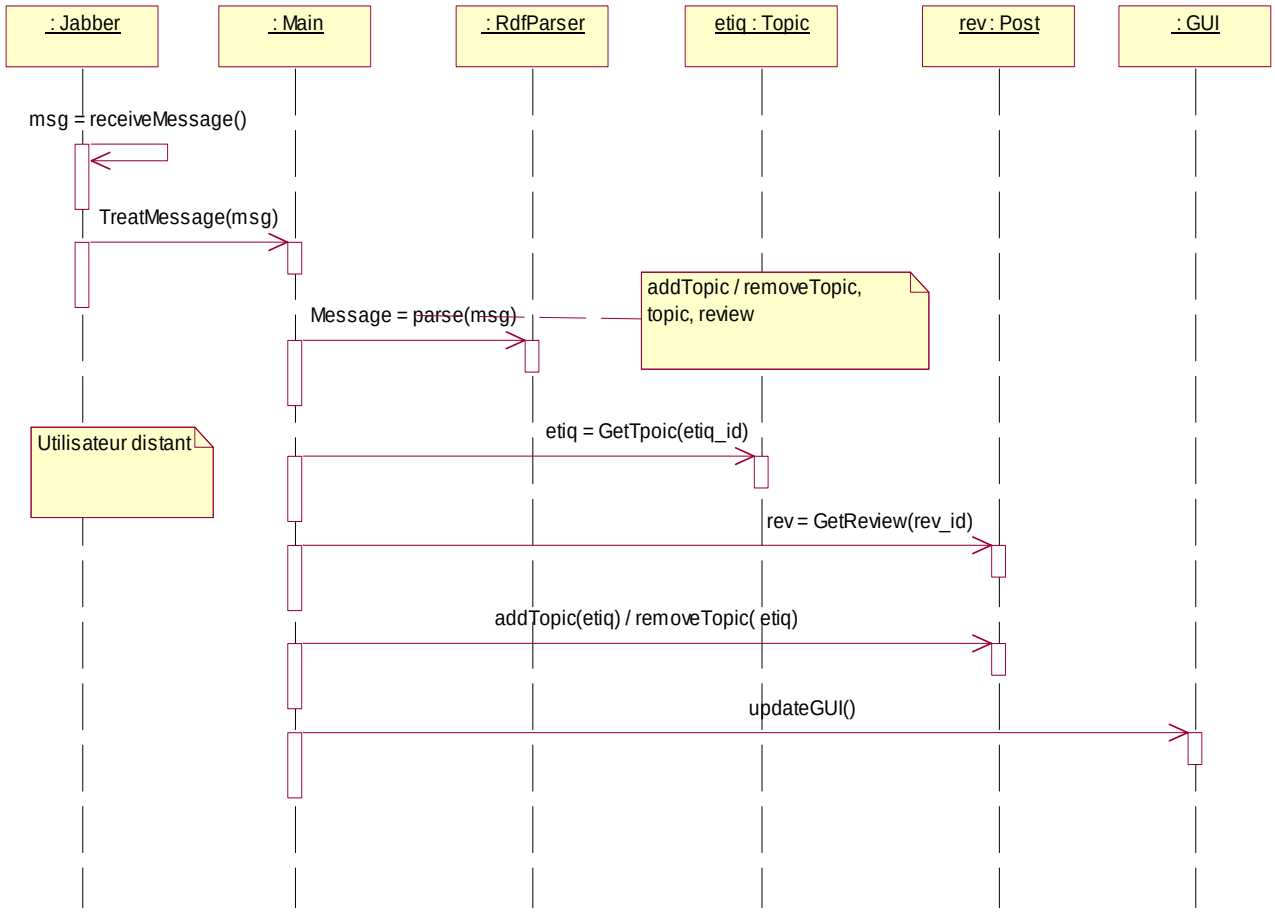
6.3. Diagramme de séquence « Attacher/Détacher une revue à une étiquette » (utilisateur local)



Ici un utilisateur attache/détache une étiquette à une revue.

1. Après confirmation au niveau de l'IHM, les objets *etiq* et *rev* (représentant respectivement l'étiquette et la revue sélectionnées par l'utilisateur) sont construits
2. C'est l'objet *Sender* qui se charge d'exécuter l'action Attacher/Détacher.
3. L'étiquette est attachée/détachée à la revue en local.
4. Un message RDF associé est créé via un objet *RDFCreator*
5. Ce message est envoyé à tous les utilisateurs de la liste de diffusion de l'étiquette via l'objet *Jabber*.
6. l'IHM est mise à jour à la fin de l'opération.

6.4. Diagramme de séquence « Attacher/Détacher une revue à une étiquette » (utilisateur distant)



Ce diagramme montre les opérations sur le site distant à la réception du message précédent.

7. L'objet *Jabber* reçoit le nouveau message et l'envoie à l'objet *Main* pour traitement.
8. L'objet *Main* fait recours à l'objet *RDFParser* pour analyser le message RDF afin de déterminer l'action à effectuer, en l'occurrence Attacher/Détacher l'étiquette à la revue.
9. Les objets *rev* et *etiq* concernés sont instanciés, et l'étiquette est attachée/détachée à la revue en local.
10. L'IHM est enfin mise à jour.

7. Interface homme machine

Rédacteur : Hai-Nam. Relecteur : Amélien.

L'interface homme machine n'est pas le but principal de notre projet, elle sert plutôt à faire des tests interactifs variants au lieu des longs scénarios de test ennuyeux.

C'est pourquoi nous avons tenté de faire l'IHM avec le plugin *VEP* (Visual Editor Project) de l'Eclipse. La plupart des composants sont dans la bibliothèque *Swing*.

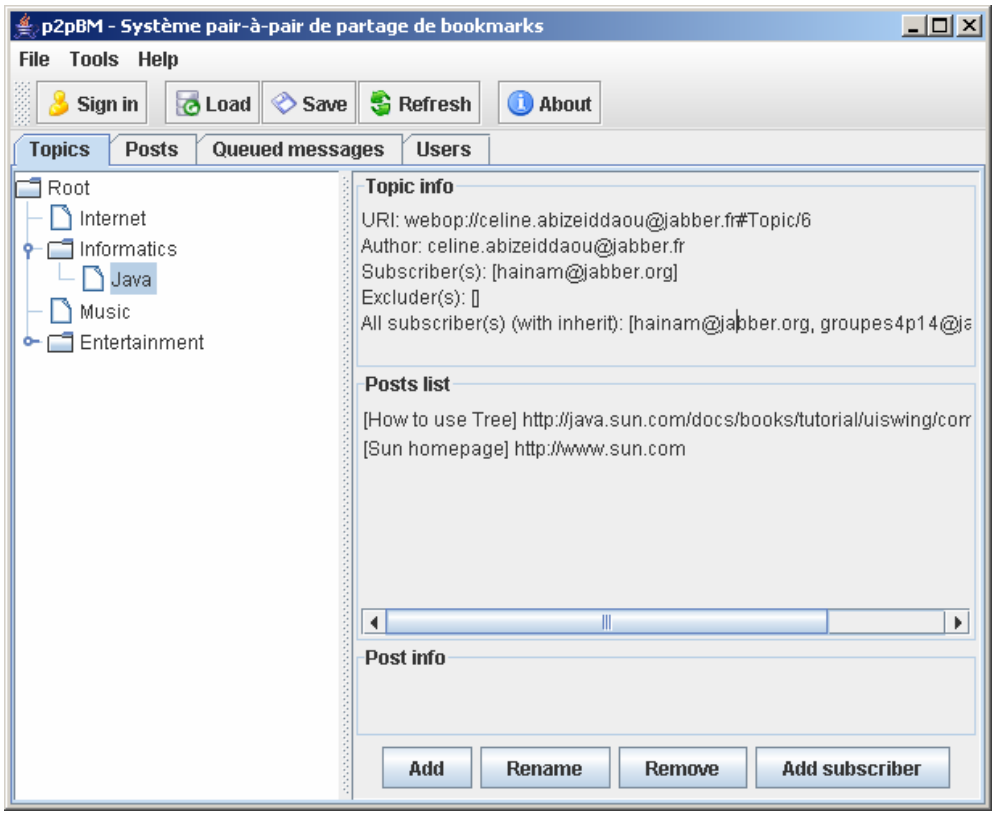


Figure 3. L'interface homme machine

L'interface comprend une barre d'outils (avec des boutons qui expriment bien leurs fonctions) et un panneau de 4 onglets :

- Topics : afficher l'arborescence des étiquettes, pour chaque étiquette nous affichons son information, une liste des revues qui y sont attachées. Il y a aussi des boutons qui permettent d'ajouter/supprimer/renommer une étiquette ou d'inscrire un utilisateur à une étiquette.
- Posts : afficher toutes les revues dans la base. Nous pouvons trier par plusieurs critères (par nom, par URL, par date, par propriétaire…)
- Queued messages : des messages à attendre une décision. Dans la plupart de temps, ce sont les messages concernant des modifications d'une étiquette, de réorganisation des revues… Les messages essentiels (comme

l'ajout d'une revue, modification d'une revue...) sont traités automatiquement sans demander avis.

- Users : la liste des utilisateurs qui sont abonnés dans notre base de données. Pour chacun nous attachons une liste des étiquettes à laquelle ils sont inscrits.

8. Conclusion

Rédacteur : Hai-Nam. Relecteur : Hai-Nam.

Fondé sur l'architecture weblogs du projet *Web of People*, notre projet utilise pourtant une approche totalement différente de ce dernier : l'approche « *pair à pair* ».

À ce stade du développement, nous avons achevé l'objectif de la conception du prototypage du système et nous avons mis en place des scénarios de tests, ainsi qu'une interface graphique. De plus, la réplication des bases personnelles est parfaitement effectuée dans toutes les activités principales. Enfin, la structure de revue et d'étiquette est bien gérée.

Suite au désengagement d'un membre du groupe (nous n'étions donc plus que 4), nous n'avons pas pu intégrer le flux RSS au système.

En utilisant les composants libres dans notre application, nous avons fait en sorte de faciliter le travail pour les successeurs du développement de ce système. Dans ce projet, nous avons appris à travailler en groupe, à utiliser et maîtriser CVS/Eclipse. Nous avons également eu une idée sur le fonctionnement des grands projets industriels.

Enfin, il est évident que de nombreux points doivent être retravaillés pour la prochaine version, dont on peut citer (non exhaustivement) :

- Des plugins permettant d'exporter la base sous forme des triplets, d'importer /d'exporter vers des bookmarks d'Internet Explorer et de Mozilla, support des flux RSS ;
- L'amélioration de la gestion de la base de données (la sérialisation/désérialisation n'est pas efficace) ;
- Le système de requête sur RDF (dixit RDQL avec Jena) permettant de travailler directement sur les triplets ;
- Une interface d'homme machine plus conviviale, multi langue ;
- L'amélioration des flux XML entre les utilisateurs.

9. Bibliographie

[1] M. Plu, L. Agosto, P. Bellec, W. Van De Velde. The Web of People: a dual view on the WWW. Twelfth International World Wide Web Conference, Budapest, 24-26 Mai, 2003.

[2] T-A. Ta, J-M. Saglio, M. Plu. An architecture based on semantic weblogs for exploring the Web of People
http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-107/paper3.pdf

[3] H. Stuckenschmidt, F. van Harmelen. Information Sharing on the Sematic Web. Springer, 2005

[4] W.Wright, D.Moore. Jabber and lightweight languages do the trick. Jabber Developer's Handbook (SAMS, 2003)
http://blog.csdn.net/jiangtao/archive/2004/04/08/268.aspx

[5] Jabber. http://www.jabber.org/about/overview.shtml.

[6] Smack. http://www.jivesoftware.org/smack/

[7] Shelley Powers. Practical RDF. O'Reilly, First edition, July 2003, ISBN 0-596-00263
http://www.oreilly.com/catalog/pracrdf/index.html

[8] RDF. http://www.w3.org/RDF/Validator/

[9] Raptor RDF Parser Toolkit. http://librdf.org/raptor/

[10] Jena Tutorial. http://www.hpl.hp.com/semweb/doc/tutorial/

[11] Jena. http://www.xml.com/pub/a/2001/05/23/jena.html

[12]. The Atom Syndication Format.
http://www.atomenabled.org/developers/syndication/atom-format-spec.php

[13] SQLite. http://www.sqlite.org/

[14] JDBC. http://www.ch-werner.de/javasqlite/

[15] Base64. http://iharder.net/base64

[16] D. Brookshier, D. Govoni, N. Krishnan, J-C. Soto. JXTA: Java™ P2P Programming. Sams Publishing, March 2002, ISBN 0-672-32366-4

[17] Eclipse. http://www.eclipsetotale.com

10. Annexes

10.1. Architecture du logiciel pour N utilisateurs

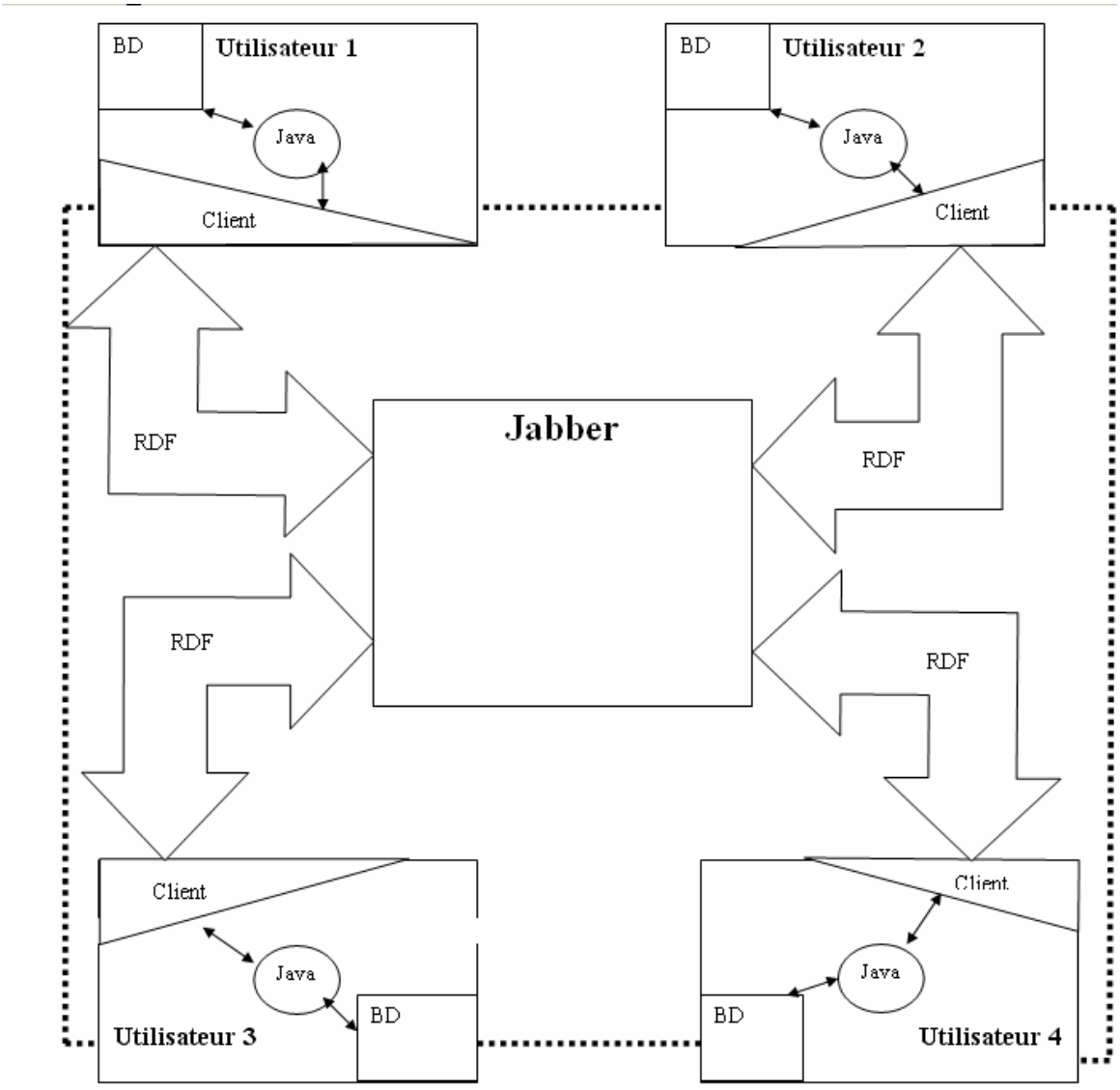


Figure 4. Architecture du logiciel pour N utilisateurs

10.2. Le schéma Webop

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#" >

<rdfs:Class rdf:about="http://purl.org/webop/1.0/Resource"
/>
```

```
<rdfs:Class rdf:about="http://purl.org/webop/1.0/Topic">
  <rdfs:subClassOf
rdf:resource="http://purl.org/webop/1.0/Resource"/>
</rdfs:Class>

<rdfs:Class rdf:about="http://purl.org/webop/1.0/Post">
  <rdfs:subClassOf
rdf:resource="http://purl.org/webop/1.0/Resource"/>
</rdfs:Class>

<rdf:Property
rdf:about="http://purl.org/webop/1.0/subTopicOf">
  <rdfs:domain
rdf:resource="http://purl.org/webop/1.0/Topic"/>
  <rdfs:range
rdf:resource="http://purl.org/webop/1.0/Topic"/>
</rdf:Property>

<rdf:Property
rdf:about="http://purl.org/webop/1.0/hasTopic">
  <rdfs:domain
rdf:resource="http://purl.org/webop/1.0/Post"/>
  <rdfs:range
rdf:resource="http://purl.org/webop/1.0/Topic"/>
</rdf:Property>

<rdf:Property
rdf:about="http://purl.org/webop/1.0/includesTopic">
  <rdfs:domain
rdf:resource="http://purl.org/webop/1.0/Topic"/>
  <rdfs:range
rdf:resource="http://purl.org/webop/1.0/Topic"/>
</rdf:Property>

<rdf:Property
rdf:about="http://purl.org/webop/1.0/includedBy">
  <rdfs:domain
rdf:resource="http://purl.org/webop/1.0/Topic"/>
  <rdfs:range
rdf:resource="http://purl.org/webop/1.0/Topic"/>
</rdf:Property>

<rdf:Property
rdf:about="http://purl.org/webop/1.0/recommendedFor">
  <rdfs:domain
rdf:resource="http://purl.org/webop/1.0/Topic"/>
  <rdfs:range
rdf:resource="http://purl.org/webop/1.0/Topic"/>
```

```
</rdf:Property>

<rdf:Property
rdf:about="http://purl.org/webop/1.0/deliveredFor">
  <rdfs:domain
rdf:resource="http://purl.org/webop/1.0/Topic"/>
  <rdfs:range
rdf:resource="http://purl.org/webop/1.0/Topic"/>
</rdf:Property>

<rdf:Property
rdf:about="http://purl.org/webop/1.0/subscribesTo">
  <rdfs:domain
rdf:resource="http://purl.org/webop/1.0/Topic"/>
  <rdfs:range
rdf:resource="http://purl.org/webop/1.0/Topic"/>
</rdf:Property>

<rdf:Property
rdf:about="http://purl.org/webop/1.0/personalisationOf">
  <rdfs:domain
rdf:resource="http://purl.org/webop/1.0/Post"/>
  <rdfs:range
rdf:resource="http://purl.org/webop/1.0/Post"/>
</rdf:Property>

<rdf:Property
rdf:about="http://purl.org/webop/1.0/hasPersonalisation">
  <rdfs:domain
rdf:resource="http://purl.org/webop/1.0/Post"/>
  <rdfs:range
rdf:resource="http://purl.org/webop/1.0/Post"/>
</rdf:Property>

<rdf:Property
rdf:about="http://purl.org/webop/1.0/subjectIndicatorRef">
  <rdfs:domain
rdf:resource="http://purl.org/webop/1.0/Topic"/>
  <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-
rdf-syntax-ns#Resource"/>
</rdf:Property>
<rdf:Property
rdf:about="http://purl.org/webop/1.0/resourceRef">
  <rdfs:domain
rdf:resource="http://purl.org/webop/1.0/Post"/>
  <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-
rdf-syntax-ns#Resource"/>
</rdf:Property>
```

```
<rdf:Property
rdf:about="http://purl.org/webop/1.0/accessibleBy">
  <rdfs:domain
rdf:resource="http://purl.org/webop/1.0/Topic"/>
  <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-
schema#Literal"/>
</rdf:Property>

<rdf:Property
rdf:about="http://purl.org/webop/1.0/inaccessibleBy">
  <rdfs:domain
rdf:resource="http://purl.org/webop/1.0/Topic"/>
  <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-
schema#Literal"/>
</rdf:Property>

<rdf:Property
rdf:about="http://purl.org/webop/1.0/referencingAllowed">
  <rdfs:domain
rdf:resource="http://purl.org/webop/1.0/Topic"/>
  <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-
schema#Literal"/>
</rdf:Property>

<!-- Vocabulary of message -->

<rdfs:Class rdf:about="http://purl.org/webop/1.0/Message"
/>

<rdf:Property rdf:about="http://purl.org/webop/1.0/from">
  <rdfs:domain
rdf:resource="http://purl.org/webop/1.0/Message"/>
  <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-
schema#Literal"/>
</rdf:Property>

<rdf:Property rdf:about="http://purl.org/webop/1.0/to">
  <rdfs:domain
rdf:resource="http://purl.org/webop/1.0/Message"/>
  <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-
schema#Literal"/>
</rdf:Property>

<rdf:Property
rdf:about="http://purl.org/webop/1.0/feedbackTo">
  <rdfs:domain
rdf:resource="http://purl.org/webop/1.0/Message"/>
```

```
  <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-
schema#Literal"/>
</rdf:Property>

<rdf:Property rdf:about="http://purl.org/webop/1.0/text">
  <rdfs:domain
rdf:resource="http://purl.org/webop/1.0/Message"/>
  <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-
schema#Literal"/>
</rdf:Property>

<rdf:Property rdf:about="http://purl.org/webop/1.0/auth">
  <rdfs:domain
rdf:resource="http://purl.org/webop/1.0/Message"/>
  <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-
schema#Literal"/>
</rdf:Property>

<rdf:Property rdf:about="http://purl.org/webop/1.0/body">
  <rdfs:domain
rdf:resource="http://purl.org/webop/1.0/Message"/>
  <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-
rdf-syntax-ns#Bag"/>
</rdf:Property>

<rdfs:Class rdf:about="http://purl.org/webop/1.0/Event" />

<rdf:Property
rdf:about="http://purl.org/webop/1.0/createdStmt">
  <rdfs:domain
rdf:resource="http://purl.org/webop/1.0/Event"/>
  <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-
rdf-syntax-ns#Statement"/>
</rdf:Property>

<rdf:Property
rdf:about="http://purl.org/webop/1.0/deletedStmt">
  <rdfs:domain
rdf:resource="http://purl.org/webop/1.0/Event"/>
  <rdfs:range rdf:resource="http://www.w3.org/1999/02/22-
rdf-syntax-ns#Statement"/>
</rdf:Property>
</rdf:RDF>
```

Annexe I : expressions des besoins

Diagramme de cas d'utilisation étiquette

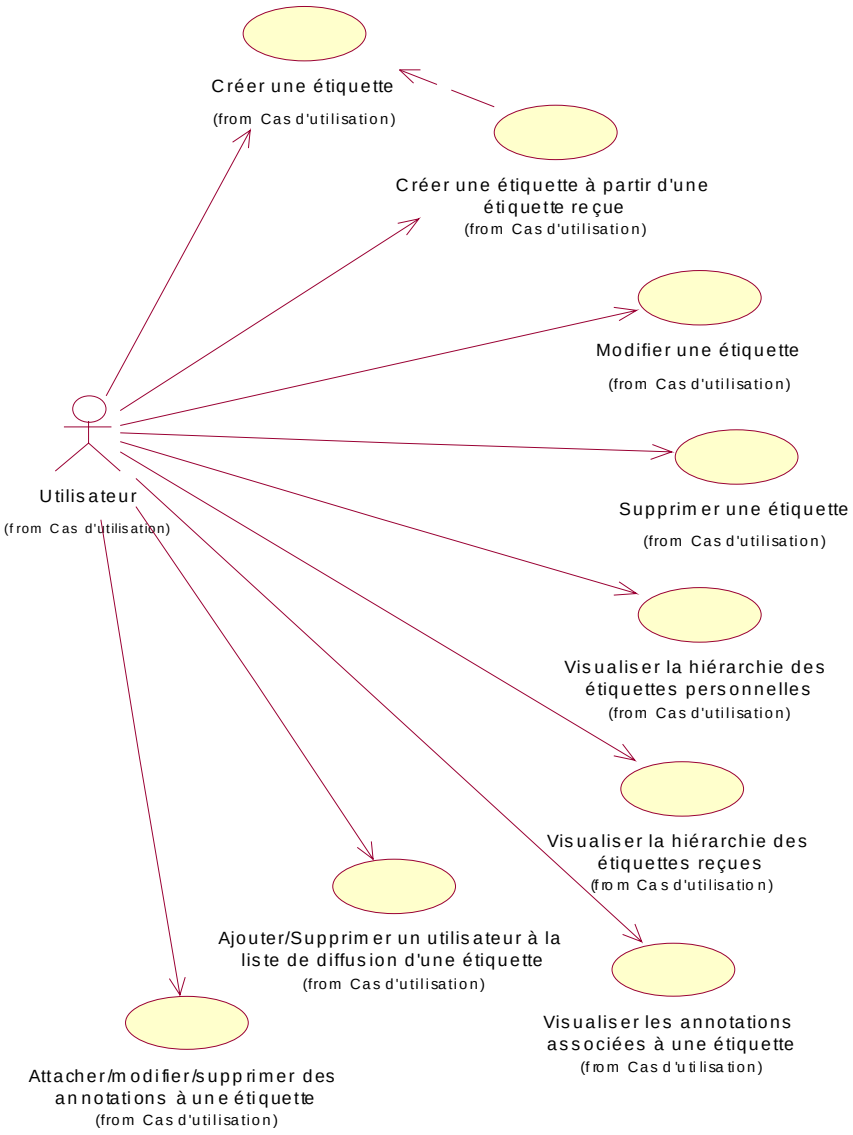
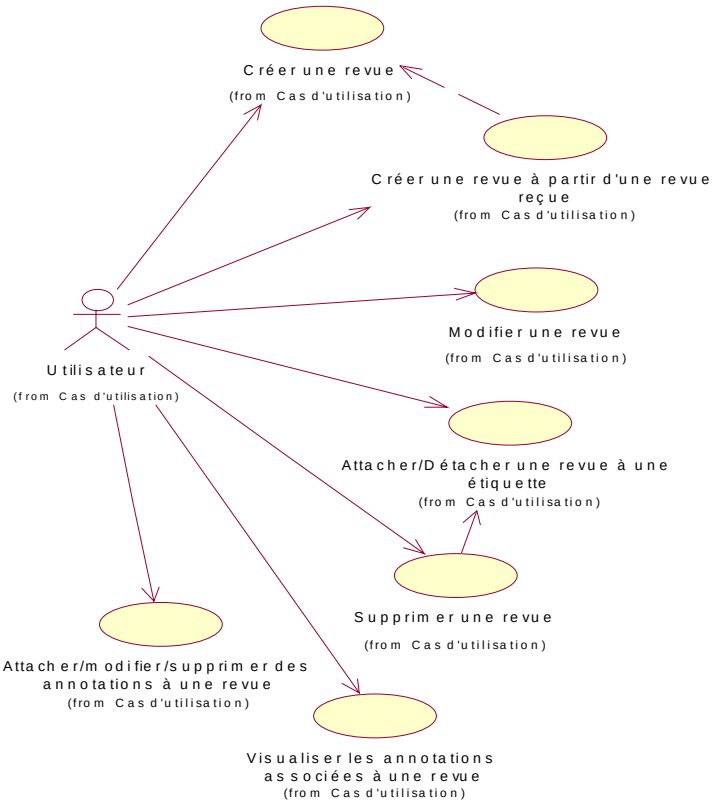
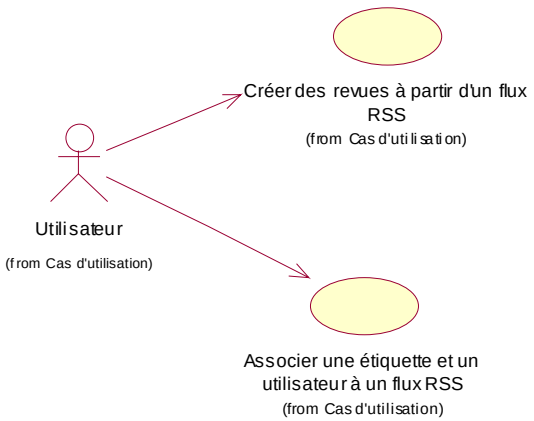
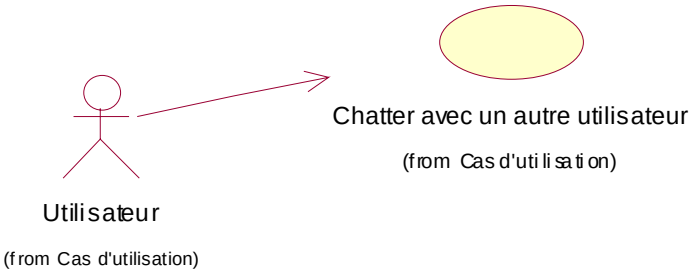


Diagramme de cas d'utilisation revue



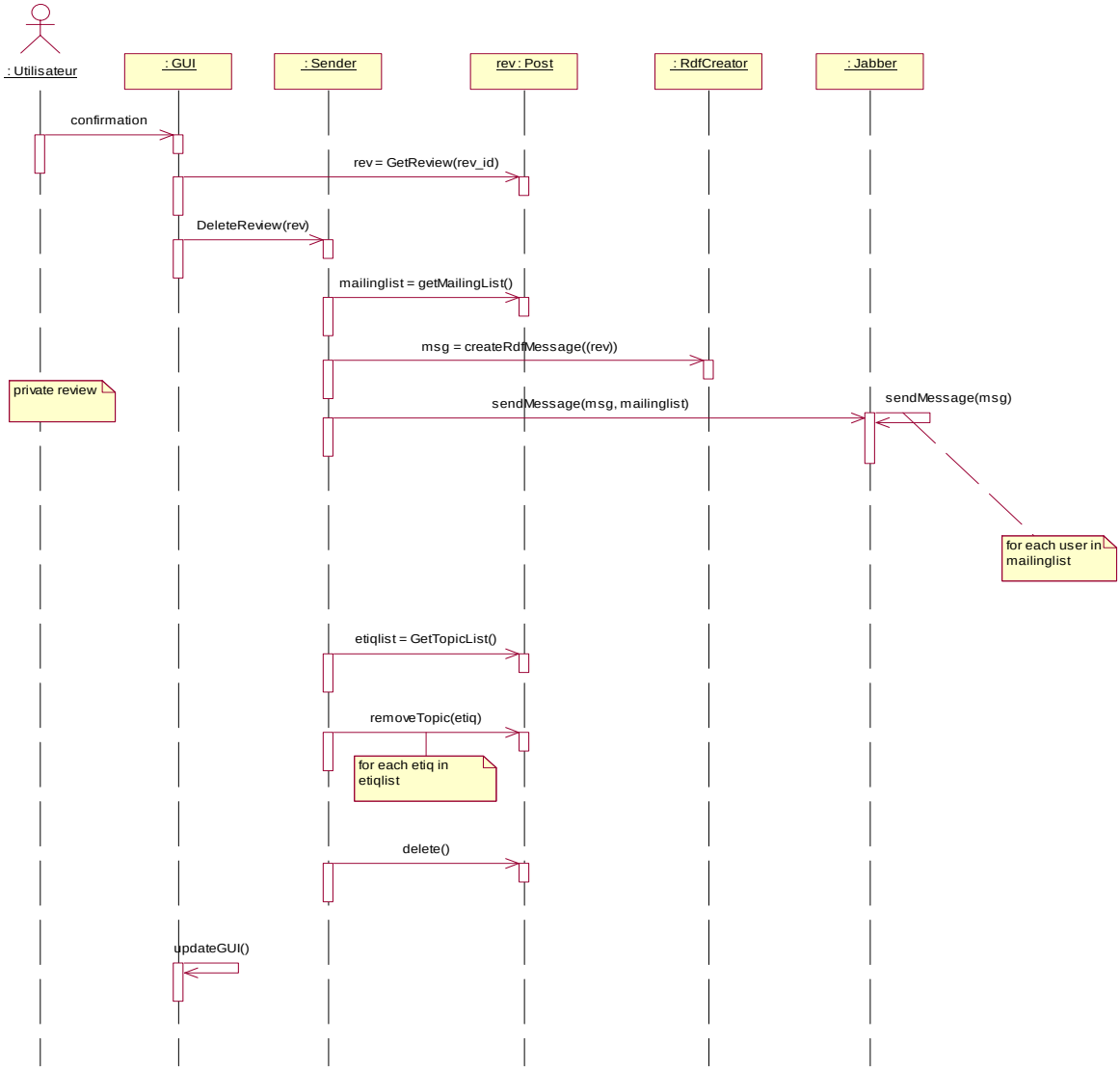
Autres cas d'utilisations



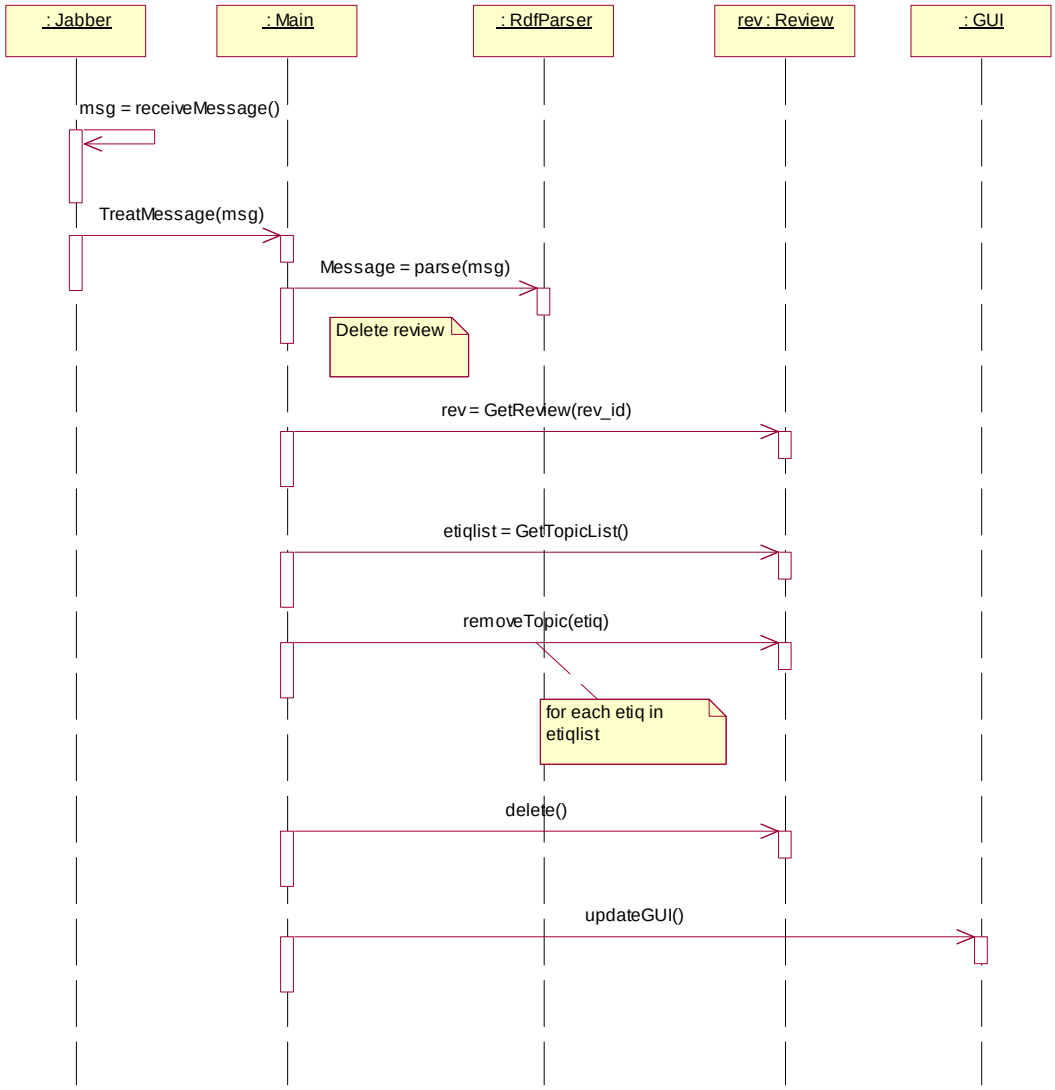


10.3. Diagrammes de séquence

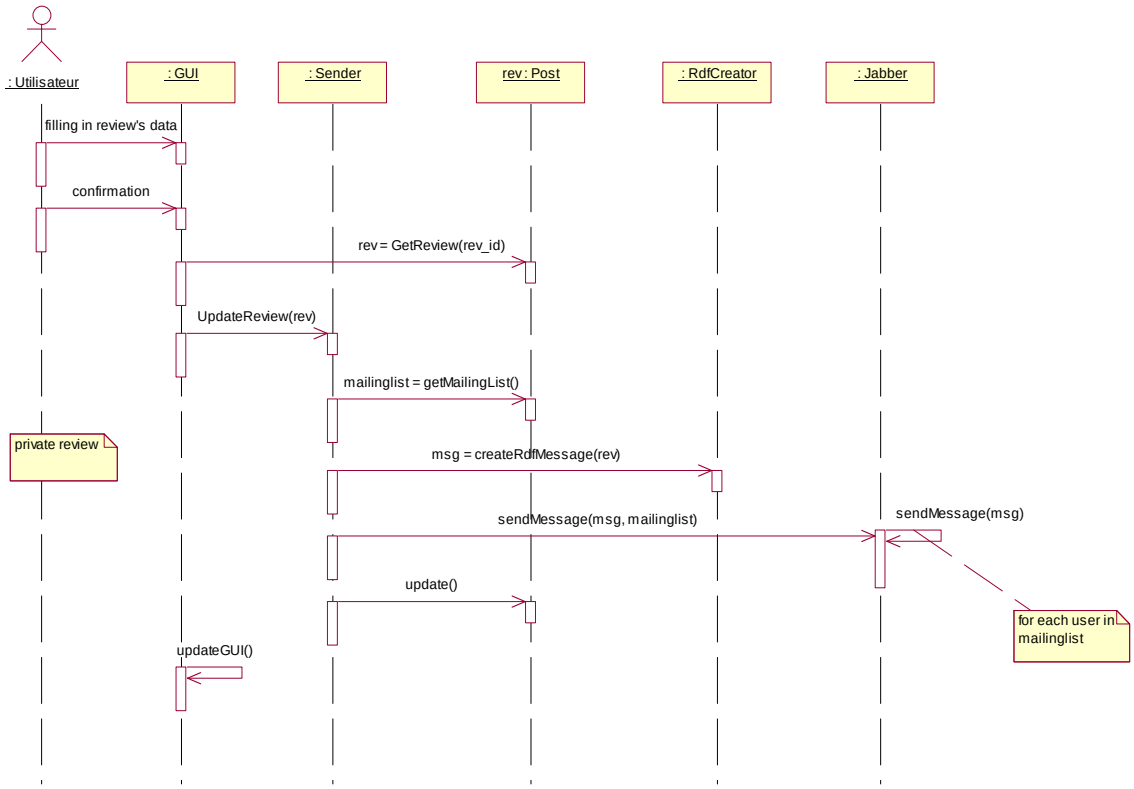
10.3.1. Supprimer une revue (1)



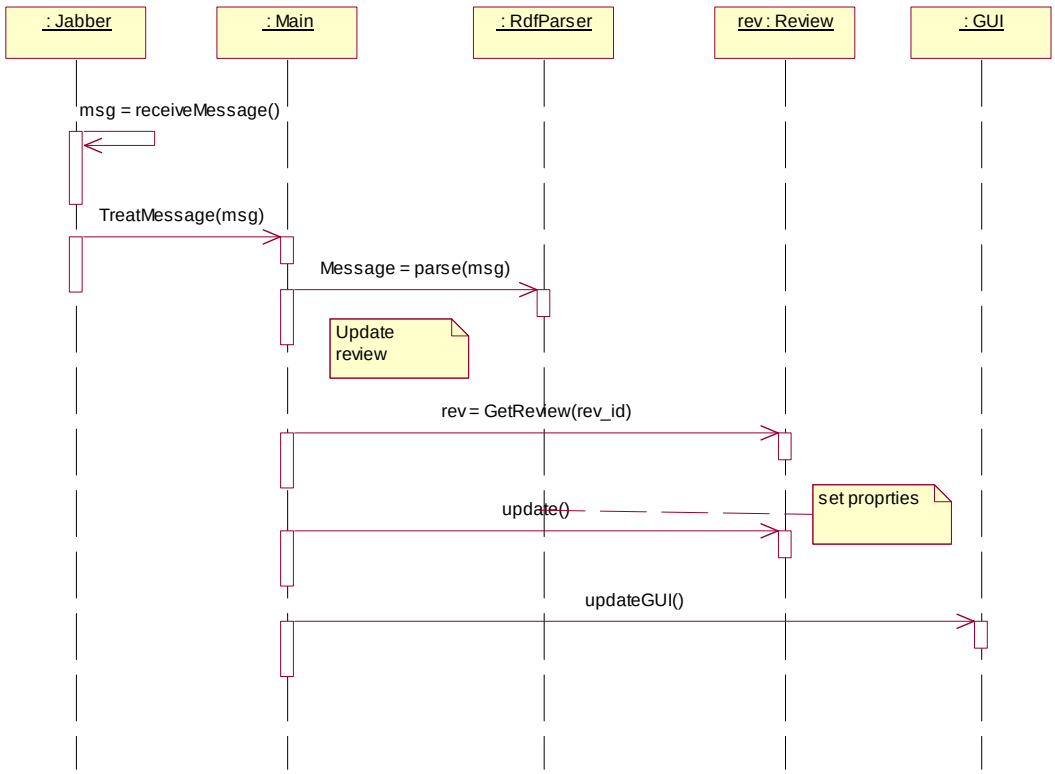
10.3.2. Supprimer une revue (2).



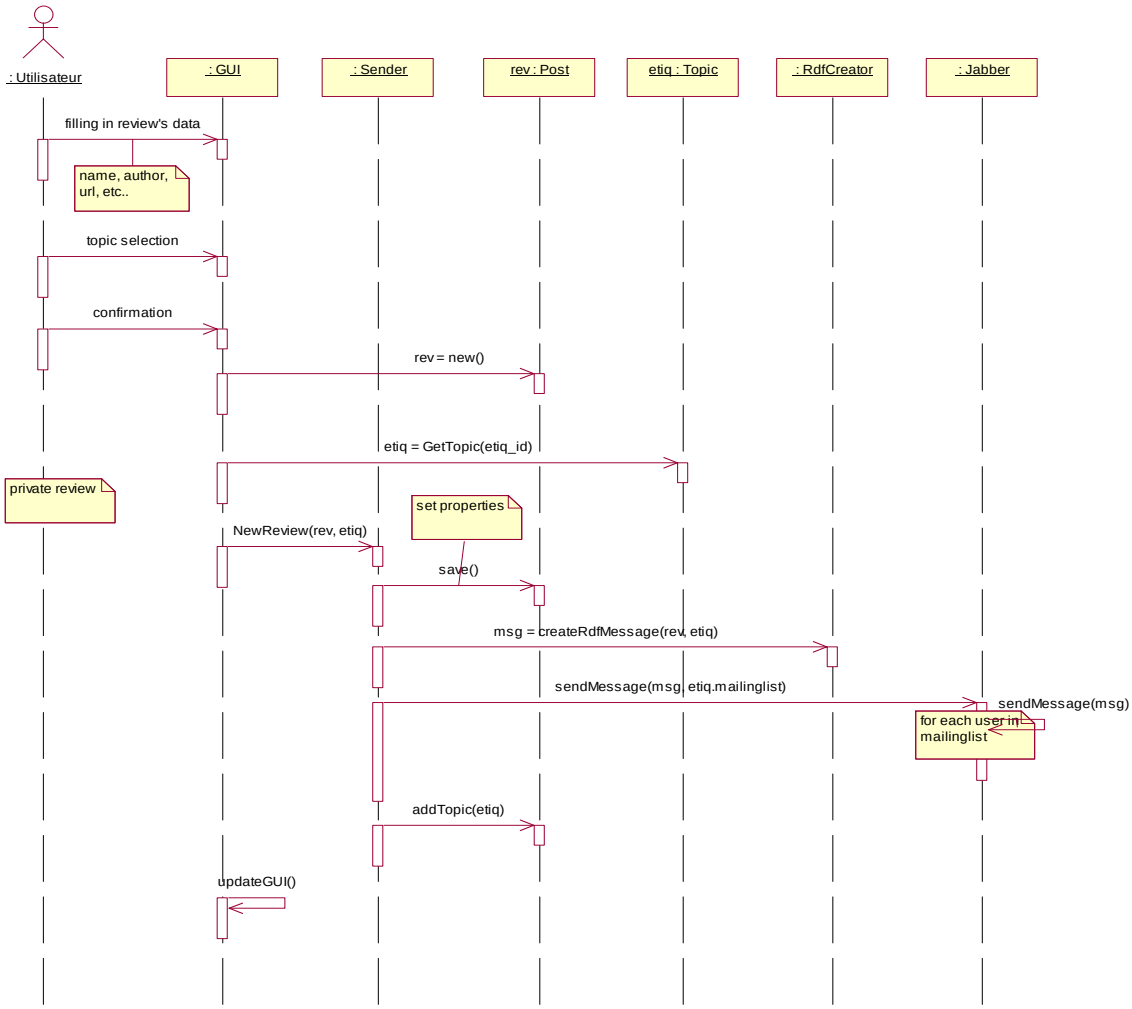
10.3.3. Modifier une revue (1)



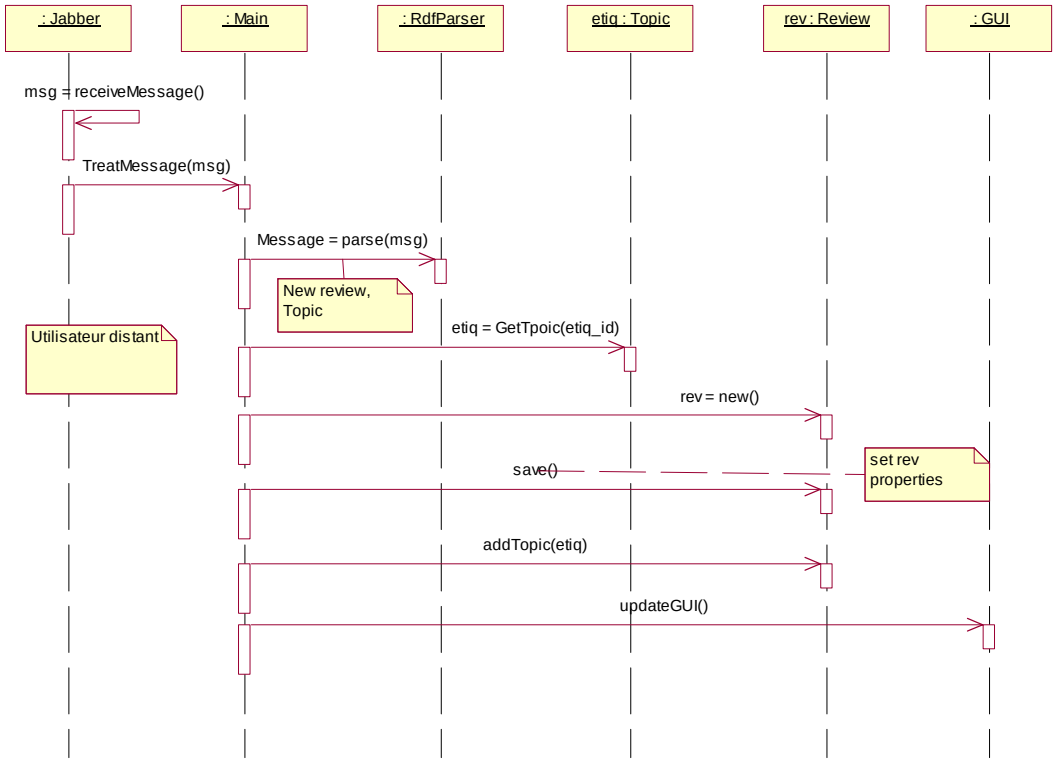
10.3.4. Modifier une revue (2)



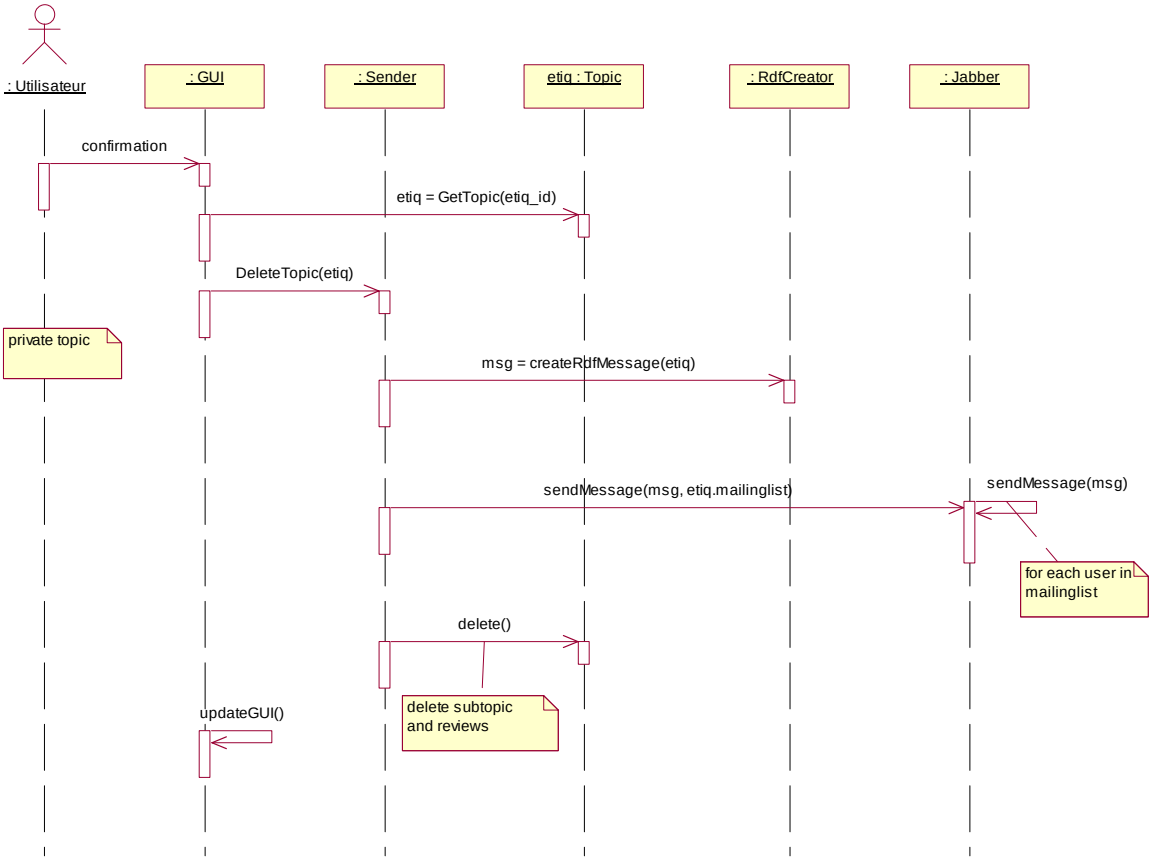
10.3.5. Créer une revue (1)



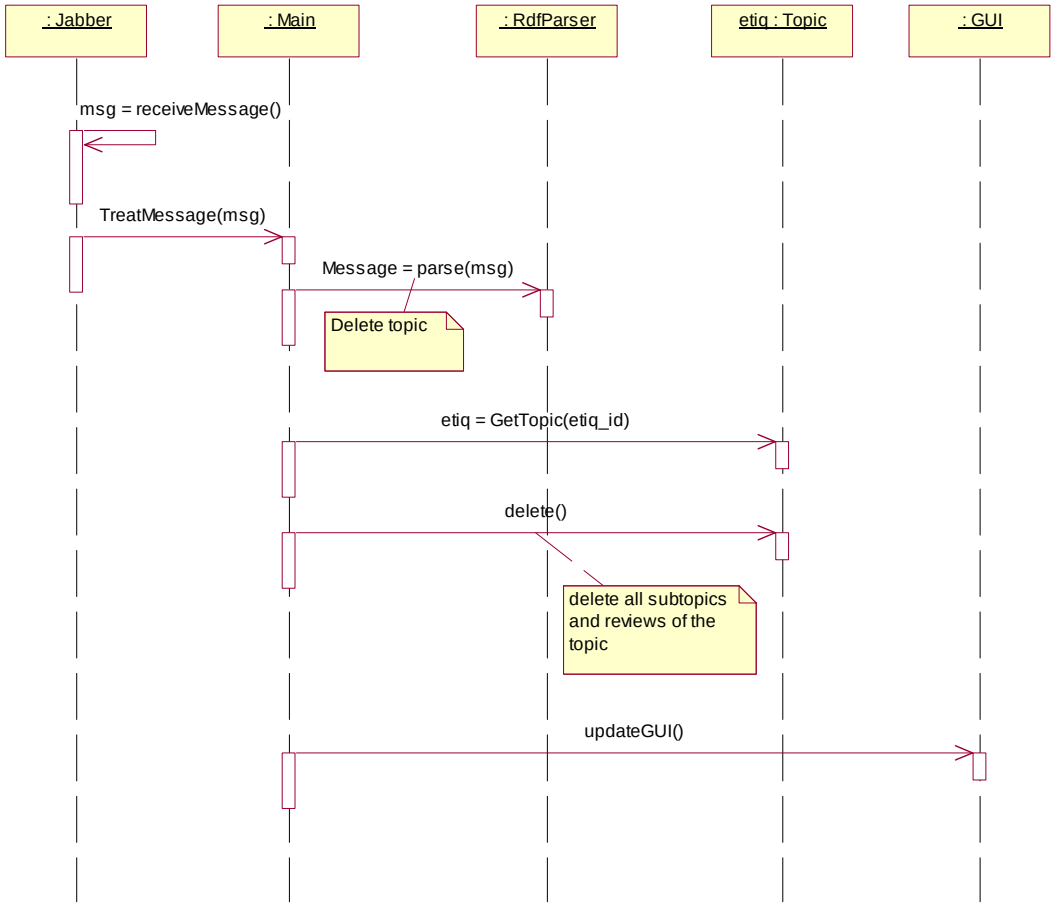
10.3.6. Créer une revue (2)



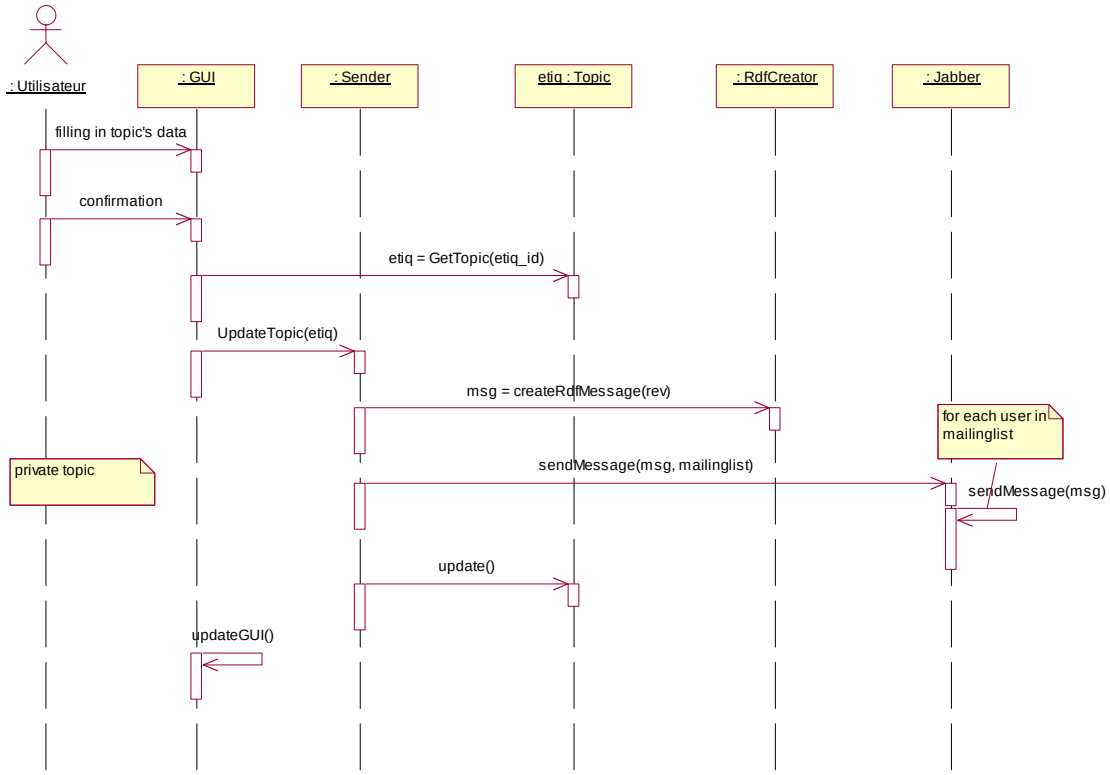
10.3.7. Supprimer une étiquette (1)



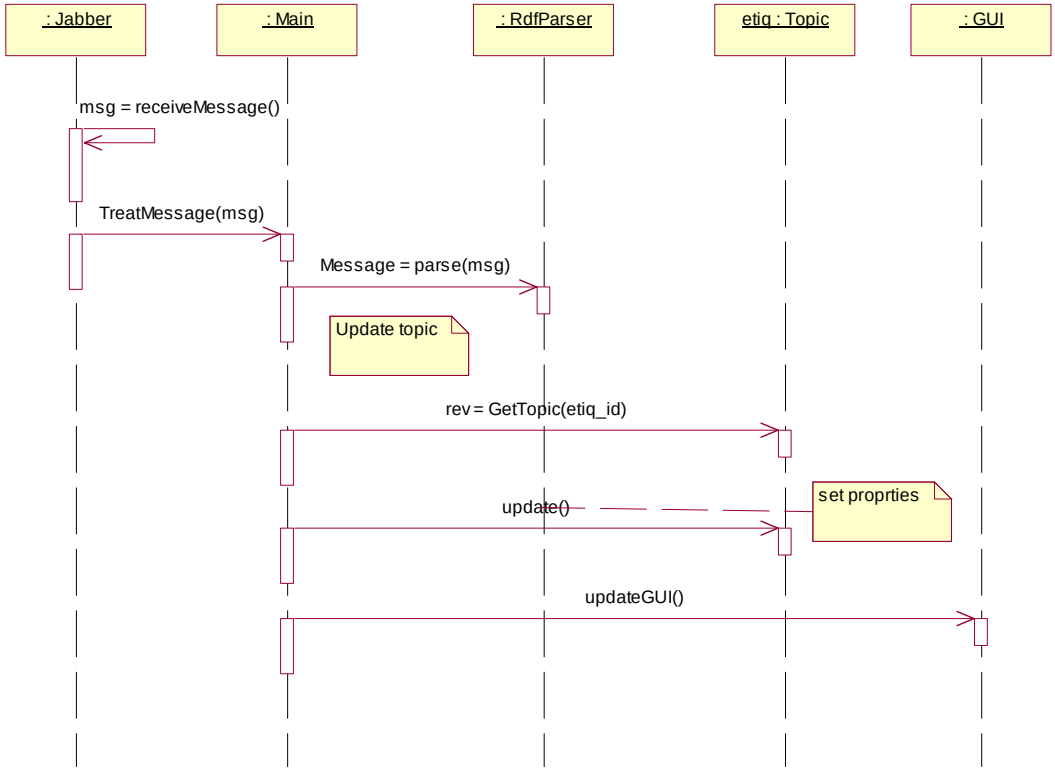
10.3.8. Supprimer une étiquette (2)



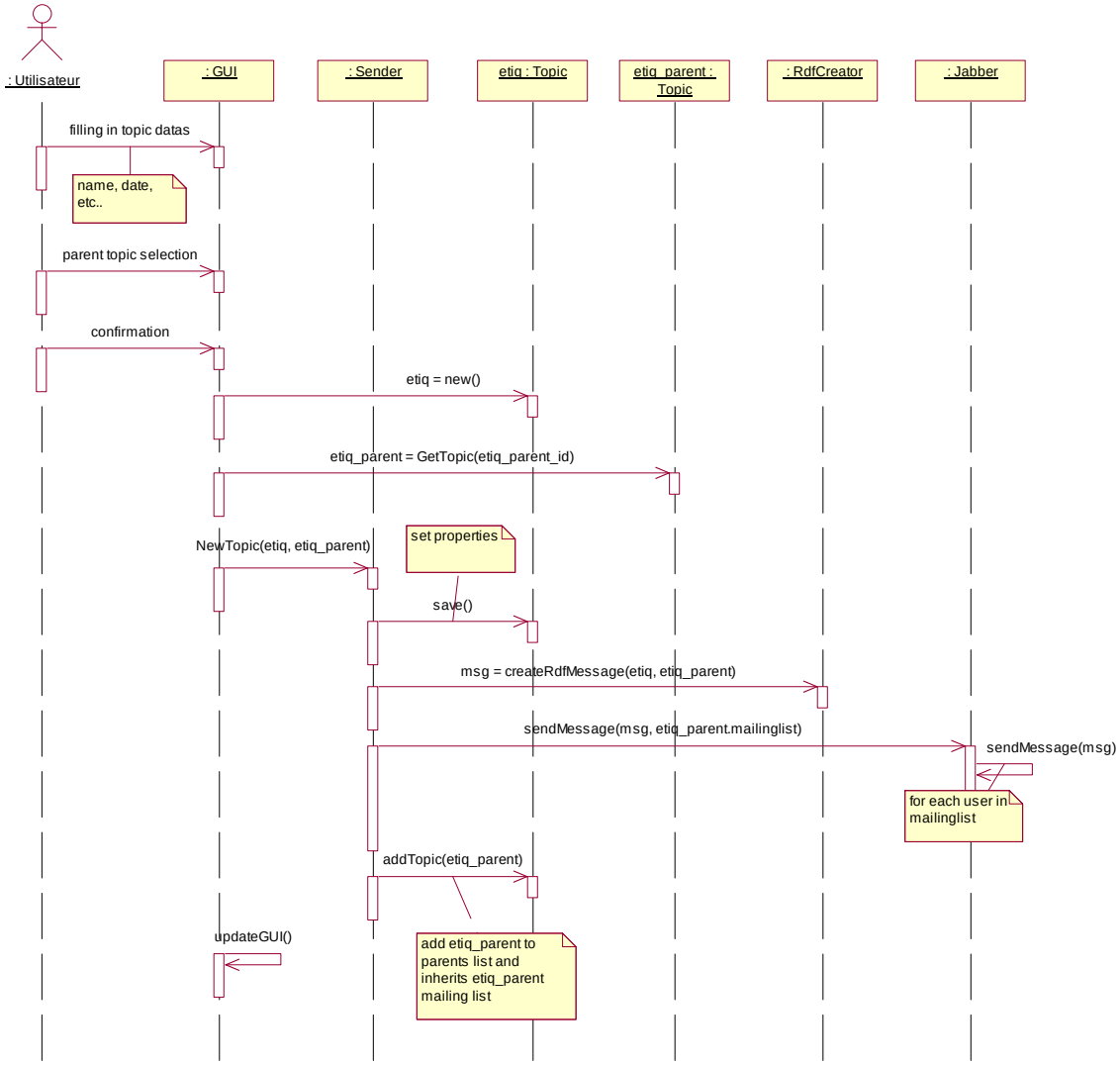
10.3.9. Modifier une étiquette (1)



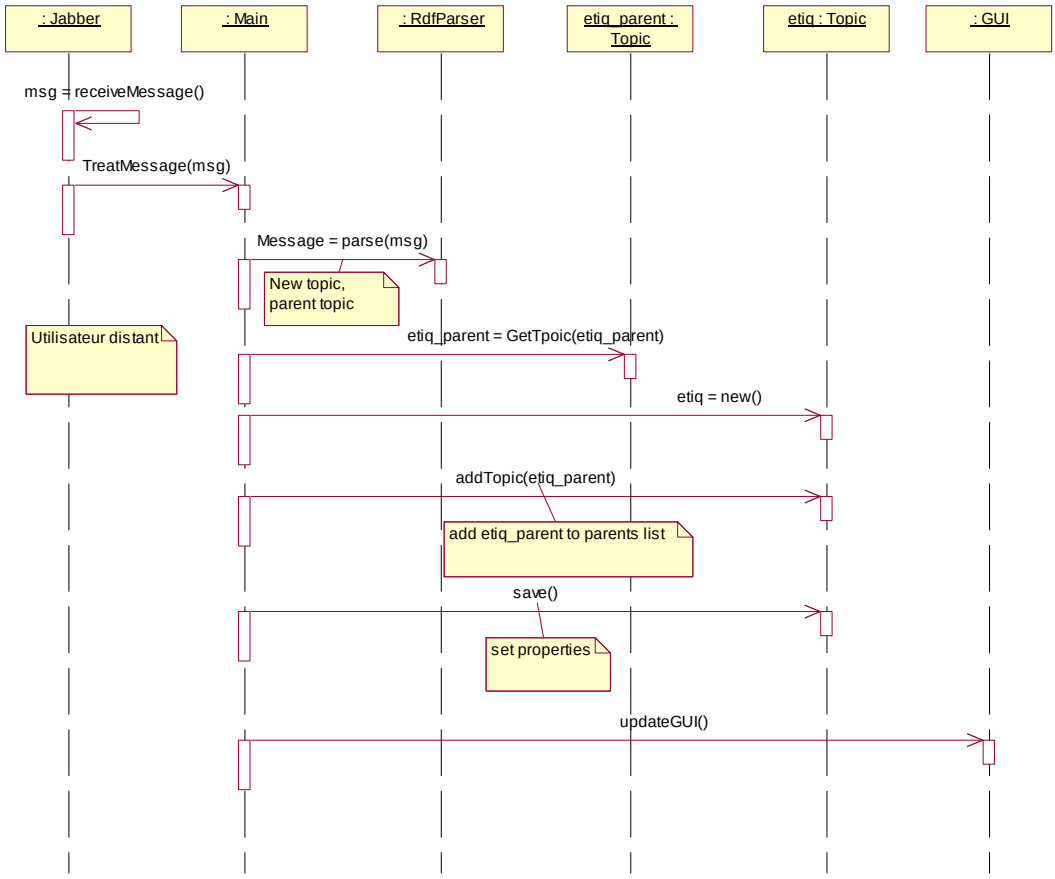
10.3.10. Modifier une étiquette (2)



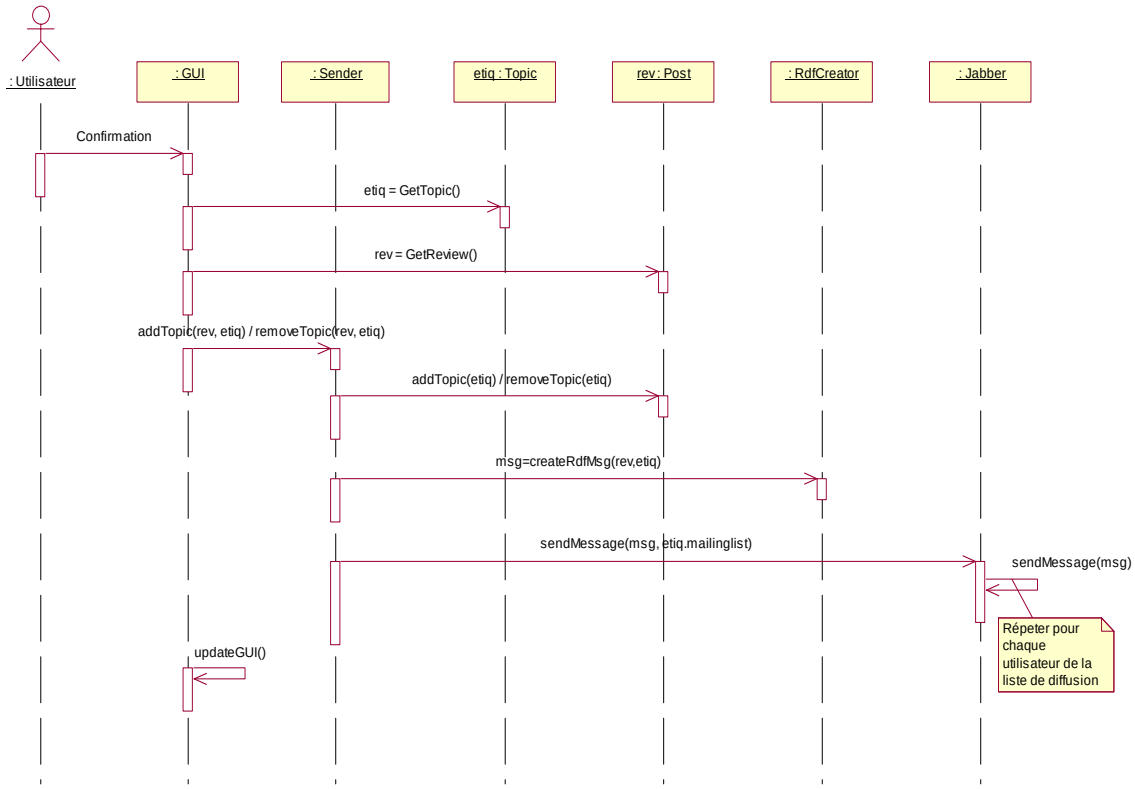
10.3.11. Créer une étiquette (1)



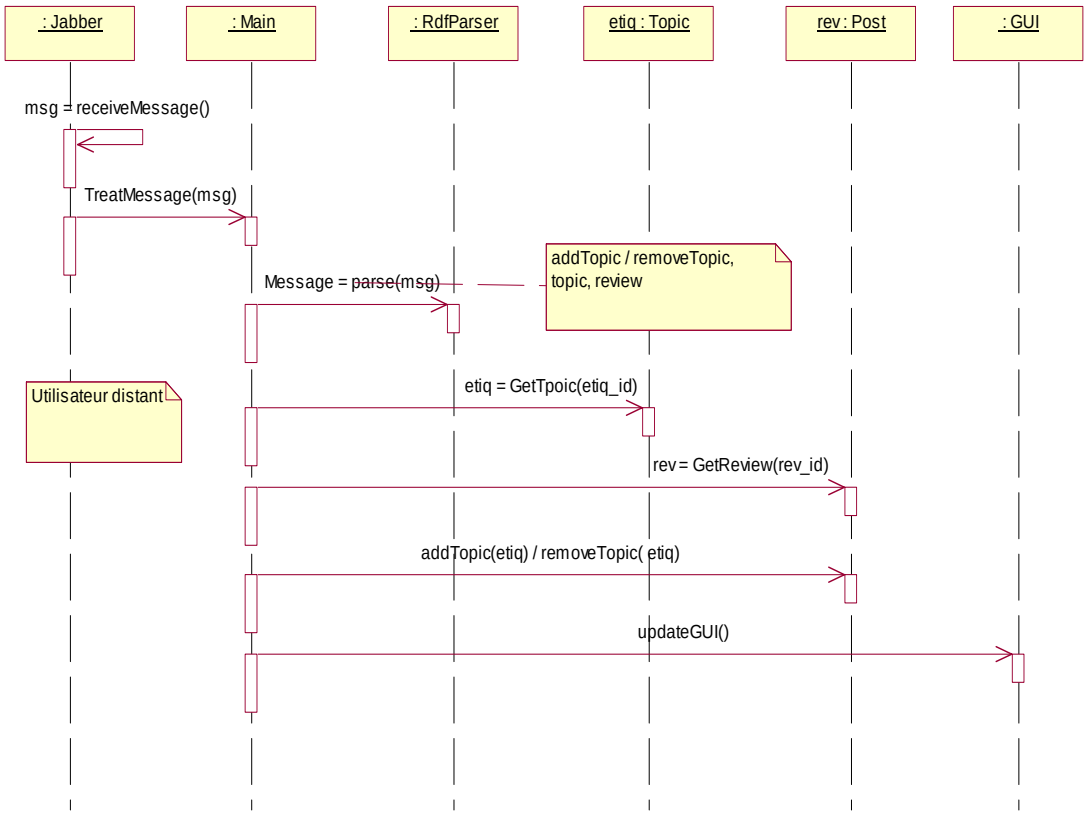
10.3.12. Créer une étiquette (2)



10.3.13. Attacher/détacher un revue à une étiquette (1)



10.3.14. Attacher/détacher une revue à une étiquette (2)



10.3.15. Associer un utilisateur à une étiquette