# sBox 1.0

# Laboratoire réseau 2016

Par Cyril LORQUET et Louis SWINNEN

**Version 2016-03-16**

---



Mars 2016

# sBox : distributed source repository

HELMo Informatique – Laboratoire réseau 2016
Cyril LORQUET et Louis SWINNEN

## Introduction

L'objet de ce laboratoire est d'implémenter un système primitif de gestion de source *distribué*. Ainsi, nous souhaitons pouvoir, grâce à *un client* envoyer ou récupérer un projet vers le système. Le système, pour des raisons de sécurité, peut comporter une multitude de nœuds (au moins 1), qui se répliquent entre eux.

Dans cette version du système, nous *ne gérerons pas le retour à des versions précédentes* (voir options), seule la dernière version du projet est disponible.

## 1. Infrastructure

Pour l'infrastructure, nous vous demandons d'implémenter 3 éléments distincts :

- Le **client** qui est chargé de fournir une interface graphique à l'utilisateur, permettant de réaliser les opérations suivantes : créer un nouveau projet, envoyer un projet ou récupérer un projet. Le projet est matérialisé sous la forme d'un dossier à choisir par l'utilisateur, le client doit mémoriser la *version* actuelle du projet. Pour des raisons de facilité, le client créera une archive contenant tous les fichiers du projet et enverra ce fichier au serveur proxy.
- Le **serveur proxy** est un programme serveur qui aura pour tâche :
  - De dialoguer et répondre aux requêtes du client
  - De gérer les différents nœuds existants
    - Identifier les nœuds existants
    - Annoncer les dernières versions de chaque projet
  - D'implémenter un nœud primitif
- Le **nœud** est un programme serveur qui aura pour tâche :
  - De répliquer les informations depuis le serveur proxy ou depuis un autre nœud

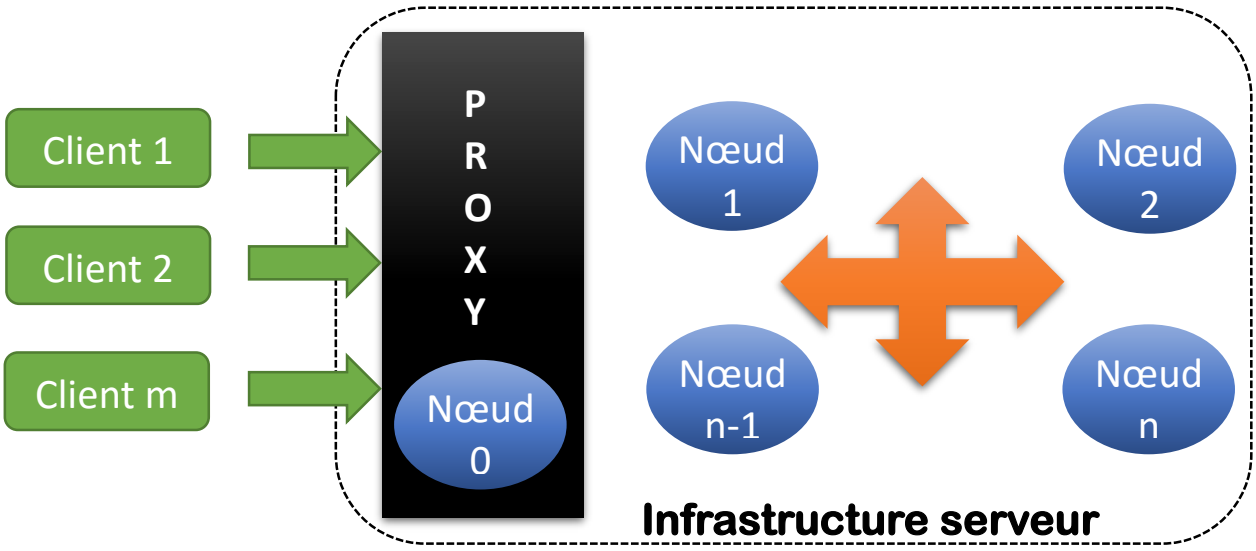



Figure 1 : Infrastructure serveur

## 2. Les différents cas d'utilisation

Dans la suite du document, nous allons aborder les différents cas d'utilisation de l'application. Ainsi, nous allons aborder **les échanges entre le client et le serveur proxy** dans un premier temps et ensuite, nous détaillerons **les échanges entre le serveur proxy et les nœuds**.

Pour mettre en place ce système, nous allons utiliser le *multicast* entre le serveur proxy et les différents nœuds. Une conséquence de ce choix est que les nœuds et le serveur proxy *doivent pouvoir dialoguer ensemble* (se trouver dans le même réseau, par exemple).

### 2.1 Le client vs le serveur proxy

Le programme client discute exclusivement avec le programme serveur **proxy**. En effet, la gestion des nœuds et de la réplication est réalisée par le programme proxy dans un second temps. Cette division vous permet d'implémenter ce premier cas d'utilisation indépendamment des autres.

#### 2.1.1 Survol

Pour ces échanges, le client utilisera exclusivement TCP (fiabilité de la connexion). Le programme client doit pouvoir soumettre 3 types de requête :

1. La **création d'un nouveau projet** dont l'identifiant est proposé par le client. Dans sa réponse, le serveur proxy mentionnera si l'identifiant est correct et retournera un nouveau de version initial (la version 0). Cette version est mémorisée.
2. La **soumission de la nouvelle version du projet** permettant de rendre cette version disponible à d'autres clients. Afin de limiter les échanges, le client est chargé de créer une archive[1] transmise au serveur, en même temps que le nouveau numéro de version. En cas de conflit (le numéro de version proposé est identique à celui présent sur le serveur), le dépôt est refusé.
3. Le **rapatriement de la version actuelle du projet** depuis le serveur proxy. Dans le cas, le serveur transmet l'archive contenant le projet au client, qui est chargé de décompresser celle-ci dans le dossier choisi par l'utilisateur. Le numéro de version du projet est également transmis au client qui doit mémoriser celle-ci.

La machine à états représentant les échanges entre un client et le serveur proxy peut se résumer comme suit, avec $S_0$ comme état initial et $S_5$ comme état final :

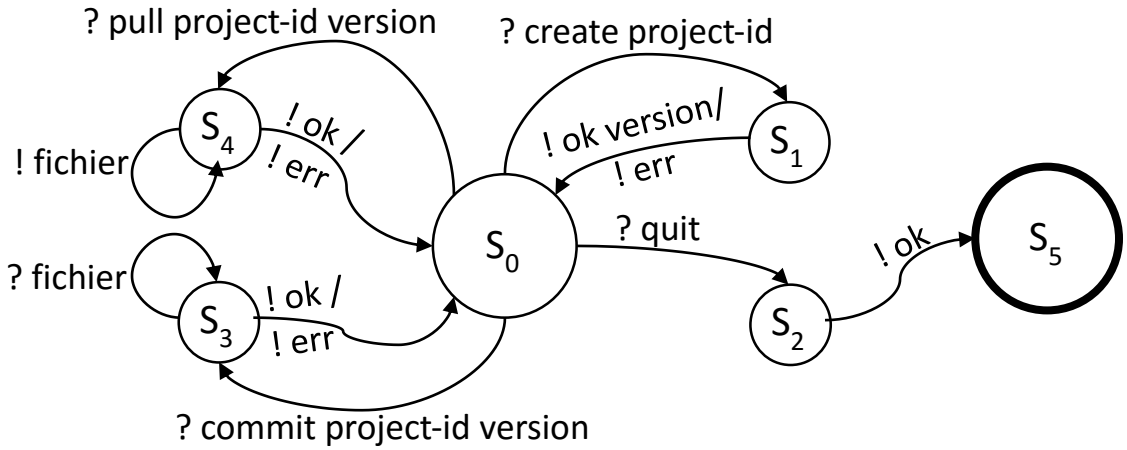


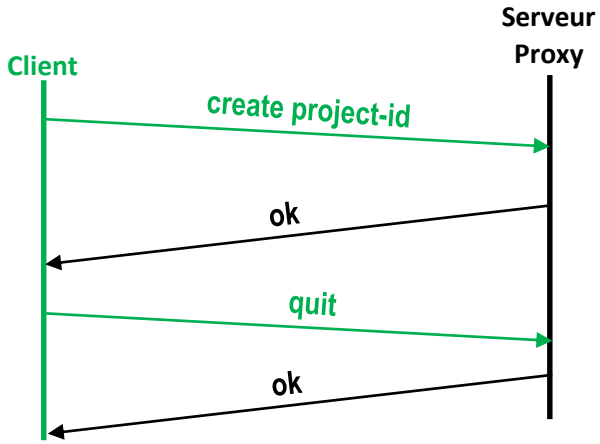

Figure 2 : Machine à états finis

[1] Les langages de programmation actuels proposent facilement la création de fichier ZIP, format que l'on vous demande de supporter dans ce système.

Comme nous pouvons le voir, les échanges entre le client et le serveur proxy sont très simples : il s'agit de simples requêtes / réponses. Nous allons, dans la suite, expliciter les échanges possibles.

### 2.1.2 Création d'un nouveau projet

La création, par un client, d'un nouveau projet répond au scénario suivant :




Le scénario présenté ne montre pas les erreurs qui peuvent survenir. Ainsi, *si un projet avec le même identifiant* existe déjà, un message d'erreur sera transmis au client (à la place du message *ok*, comme montré sur la figure 2). Ce message a pour but d'annoncer au serveur proxy qu'un nouveau projet va être créé. Le numéro de version 0 est automatiquement associé à ce projet. L'identifiant du projet est, dès ce moment, réservé sur le serveur proxy. Nous supposerons que le client choisit l'identifiant du projet. A charge pour le système de gérer les conflits.

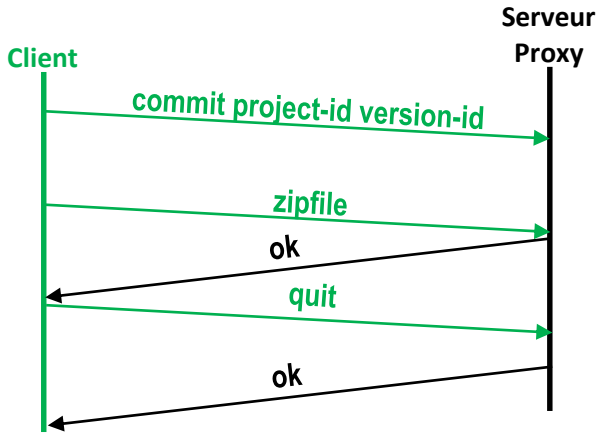
Le format des messages est exactement défini comme suit :

```
lettre_chiffre = [a-zA-Z]/[0-9]
id = 2*10(lettre_chiffre/"-"/"_")
espace = \d32
crlf = \d13\d10
message = 1*200(lettre_chiffre/espace)
```

```
version_id = 1*3[0-9]
create = "CREATE" espace id crlf
ok = "OK" [message] crlf
err = "ERR" [message] crlf
quit = "QUIT" crlf
```

### 2.1.2 Soumission d'une nouvelle version du projet

La soumission, par un client, d'une nouvelle version d'un projet existant, répond au scénario suivant :




Le scénario présenté ne montre pas les erreurs (*fichier corrompu, identifiant du projet inconnu, version en collision avec la version actuelle sur le dépôt, aucun nœud disponible, …*). La soumission s'effectue

par un message *commit* envoyé par le client au serveur. Ce message est directement suivi par l'envoi du fichier du projet (une archive ZIP, encodée en base 64[2] pour faciliter le transfert).

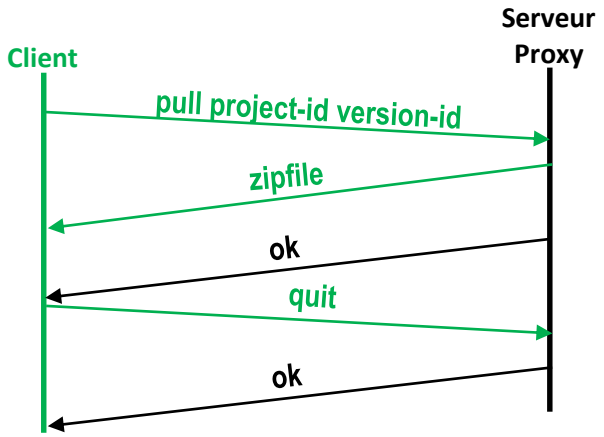
Le format des messages est exactement défini comme suit :

```
lettre_ou_chiffre = [a-zA-Z]/[0-9]
id = 2*10(lettre_chiffre/"-"/"_")
espace = \d32
crlf = \d13\d10
base64_ligne= 1*76(lettre_chiffre/
     "+" / "/" / "=")
message = 1*200(lettre_chiffre/espace)
```

```
version_id = 1*3[0-9]
zip_base64 = 0*(base64_ligne crlf)
commit = "COMMIT" espace id espace
     version crlf
zipfile = zip_base64 "###" crlf
ok = "OK" [message] crlf
err = "ERR" [message] crlf
quit = "QUIT" crlf
```

### 2.1.3 Rapatriement de la version actuelle du projet

Le rapatriement, par un client, de la version actuelle d'un projet existant répond au scénario suivant :




Le scénario présenté ne montre pas les erreurs (*fichier corrompu, version obsolète, identifiant du projet inconnu, aucun nœud disponible…*). Le rapatriement s'effectue par un message *pull* envoyé par le client au serveur. Ce message est directement suivi par l'envoi du fichier du projet (une archive ZIP, encodée en base 64 pour faciliter le transfert).

Le format des messages est exactement défini comme suit :

```
lettre_chiffre = [a-zA-Z]/[0-9]
id = 2*10(lettre_chiffre/"-"/"_")
espace = \d32
crlf = \d13\d10
base64_ligne= 1*76(lettre_chiffre/
     "+" / "/" / "=")
message = 1*200(lettre_chiffre/espace)
```

```
version_id = 1*3[0-9]
zip_base64 = 0*(base64_ligne crlf)
pull = "PULL" espace id espace
version_id crlf
zipfile = zip_base64 "###" crlf
ok = "OK" [message] crlf
err = "ERR" [message] crlf
quit = "QUIT" crlf
```

Dans cette version, le client doit être omniscient car il doit connaître l'identifiant et la version actuelle du projet qu'il souhaite rapatrier. Un message `QUERY` est prévu dans les options pour lever compte de cette limitation.

[2] L'encodage *base-64* est décrit ici : https://fr.wikipedia.org/wiki/Base64. Il est notamment utilisé dans les mails pour les pièces jointes. Dans cette implémentation, le fichier est encodé sous la forme de ligne de maximum 76 caractères, la fin du transfert est détectée par 3 caractères # sur une ligne seule (`###\r\n`)

## 2.2 Le serveur proxy et les nœuds

Le dialogue entre le serveur proxy et les nœuds est multiple : des messages multicast sont échangés ainsi que, pour la synchronisation des fichiers, des transferts unicast TCP directement entre les nœuds existants.

On peut, par faciliter, considérer que le serveur proxy héberge un nœud rudimentaire. Cette implémentation simplifie le passage des fichiers du serveur proxy vers les nœuds de dépôt. Il faut bien remarquer que le serveur proxy n'a pas vocation d'héberger les dépôts, il sert uniquement *de tampon* dans le dialogue avec un client.

### 2.2.1 Rôle du serveur proxy

Le serveur proxy effectue plusieurs tâches importantes :

- Répertorier les nœuds en cours d'exécution et détecter les nœuds inactifs ou déconnectés
- Annoncer, régulièrement, les projets et le numéro de version à destination de tous les nœuds
- Réceptionner, lors d'un *commit*, la nouvelle version d'un projet et démarrer la synchronisation vers les nœuds
- Récupérer, lors d'un *pull*, la dernière version d'un projet depuis un nœuds pour la transmettre au client

A l'exception d'échange de fichiers lors d'une synchronisation par exemple, tous les échanges entre le serveur proxy et les nœuds déployés s'effectuent en multicast. Pour ce faire, deux adresses multicast seront utilisées : une première adresse pour permettre aux nœuds de s'annoncer auprès du serveur proxy, une seconde utilisée pour localiser un nœud disposant de la dernière version d'un projet donné.

Dans la suite, nous utiliserons le nom multicast-1 pour désigner l'adresse utilisée par les nœuds pour s'annoncer et multicast-2 pour désigner l'adresse utilisée pour localiser un nœud disposant de la dernière version du projet.

#### *A. Répertorier les nœuds*

Pour détecter les nœuds qui sont actuellement en activité, chaque nœud démarré devra envoyer, *toutes les 15 secondes au moins*, un message ALIVE vers l'adresse multicast-1qui doit être écoutée par le serveur proxy.

Lorsque le serveur proxy reçoit le message ALIVE, il fixe l'indice d'activité du nœud identifié à 3. Toutes les 30 secondes, le serveur proxy décrémente tous les indices d'activité d'une unité. Ainsi, si un nœud devient indisponible, le serveur proxy pourra le déterminer.

Lorsqu'un nœud est indisponible ou déconnecté, aucun échange n'est effectué vers celui-ci.

Le format des messages est exactement défini comme suit :

```
espace = \d32
crlf = \d13\d10
```

```
noeud_id = 1*3[0-9]
alive = "ALIVE" espace noeud_id crlf
```

#### *B. Annonce des versions projets*

Le serveur proxy doit, *au moins toutes les 30 secondes*, annoncer les versions des différents projets à tous les nœuds en utilisant l'adresse multicast-2. Un nœud se doit d'écouter sur cette adresse afin de détecter lorsqu'une nouvelle version d'un projet devient disponible.

L'annonce régulière par le proxy permet, lorsqu'un nouveau nœud est démarré ou lorsqu'un nœud redevient actif, de connaître les noms des projets et leur dernière version. Ces dernières versions peuvent alors être rapatriées sur le nœud.

Le format des messages est exactement défini comme suit :

```
lettre_ou_chiffre = [a-zA-Z]/[0-9]
id = 2*10(lettre_ou_chiffre/"-"/"_")
espace = \d32
crlf = \d13\d10
```

```
version_id = 1*3[0-9]
info = "INFO" espace id espace
       version_id crlf
```

### *C. Synchronisation entre nœuds ou depuis le proxy vers un nœud*

Par simplicité, nous utiliserons le même mécanisme pour synchroniser un projet entre 2 nœuds ou entre le serveur proxy et un nœud (c'est la raison pour laquelle j'ai indiqué, précédemment, que le serveur proxy implémentait une version allégée d'un nœud).

La synchronisation nécessite **2 étapes** : localiser le nœud qui détient la dernière version du projet et le téléchargement du projet vers le nœud.

Dans la suite, nous appelleront émetteur, le nœud (ou le serveur proxy) qui détient la dernière version du projet souhaité et récepteur, le nœud qui souhaite se mettre à jour.

Le récepteur va envoyer un message *whohas* vers l'adresse multicast-2. Les nœuds détenant la version demandée du projet répondront par un message *provide*. Le récepteur choisira le premier nœud qui a répondu à sa requête comme émetteur.

Le format des messages est exactement défini comme suit :

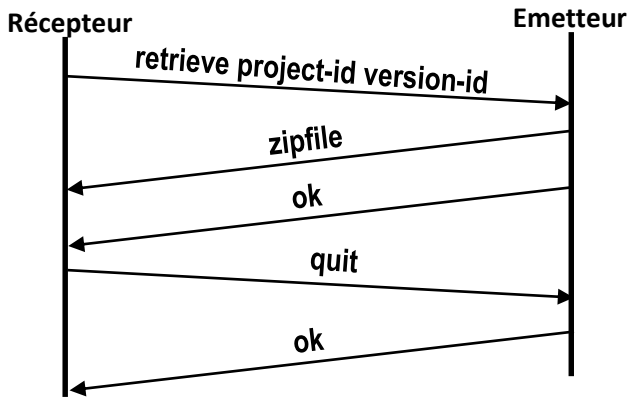
```
lettre_chiffre = [a-zA-Z]/[0-9]
id = 2*10(lettre_chiffre/"-"/"_")
espace = \d32
crlf = \d13\d10
port = 1*5[0-9]
```

```
version_id = 1*3[0-9]
whohas = "WHOHAS" espace id espace
         version_id crlf
provide = "PROVIDE" espace id espace
       version_id espace port crlf
```

Ensuite, le récepteur va établir une connexion TCP vers émetteur afin de démarrer le transfert du fichier. L'adresse du nœud émetteur est celle de l'expéditeur du message *provide*. Le port à utiliser pour le transfert est mentionné dans le message.

En effet, pour vous permettre de faire fonctionner plusieurs nœuds sur la même machine, le port de connexion (i.e. le port d'écoute) utilisé pour le transfert sera variable d'un nœud à l'autre. Il peut, par exemple, dépendre de l'identifiant du nœud.

Le transfert utilisera le scénario suivant :




Le scénario ne montre pas les erreurs éventuelles (fichier corrompu, identifiant du projet invalide, numéro de version incorrect, …). Le message *retrieve* permet de démarrer le transfert du projet dont la version est donnée en paramètre vers le récepteur.

Le format des messages est exactement défini comme suit :

```
lettre_chiffre = [a-zA-Z]/[0-9]
id = 2*10(lettre_chiffre/"-"/"_")
espace = \d32
crlf = \d13\d10
base64_ligne= 1*76(lettre_chiffre/
     "+" / "/" / "=")
message = 1*200(lettre_chiffre/espace)
```

```
version_id = 1*3[0-9]
retrieve = "RETRIEVE" espace id espace
             version_id crlf
zip_base64 = 0*(base64_ligne crlf)
zipfile = zip_base64 "###" crlf
ok = "OK" [message] crlf
err = "ERR" [message] crlf
quit = "QUIT" crlf
```

## 3. Quelques considérations techniques

Le serveur proxy doit être robuste, ainsi, sa configuration et son état (la liste des projets et des versions) doit être sauvegardé de manière permanente.

> Pour ce faire, vous pouvez utiliser des fichiers XML et sauvegarder l'information directement à l'intérieur. En Java, JAXB propose des opérations *Marshall* et *Unmarshall* directement depuis des POJO annotés.

Toutes les entrées (des utilisateurs, ou en provenance du réseau) seront validée à l'aide d'expressions régulières.

> En Java, nous vous conseillons de regarder les classes **Pattern** et **Matcher** pour implémenter les expressions régulières.

Le support du format ZIP et l'encodage en Base64 sont directement possibles depuis les environnements de programmation. Veillez cependant à bien respecter la grammaire : le fichier ZIP contenant le projet est encodé sur *plusieurs lignes textes* en utilisant Base64. Chaque ligne pouvant faire, *maximum* 76 caractères. La fin de la transmission est signalée par 3 caractères # sur une ligne seule (`###\r\n`).

En Java, il y a les classes **ZipOutputStream**, **Base64.Encoder** et **Base64.Decoder** qui sont intéressantes. Vous pouvez trouver sur Internet des exemples de création d'archive ZIP depuis l'environnement de programmation.

Chaque nœud, lorsqu'il est démarré, est identifié par un numéro et détient un répertoire de travail. Afin de faciliter le lancement de plusieurs nœuds, ces paramètres seront transmis par la ligne de commande.

Le nœud 0 désignera toujours le nœud primitif contenu dans le serveur proxy. Ce nœud primitif fait juste office de tampon. Lorsqu'un projet est reçu ou envoyé à un client, il est supprimé du serveur proxy. A ce titre, le serveur proxy ne réalise aucune autre synchronisation que celles nécessaires aux réponses des requêtes des clients.

## 3.1 Résumé de la grammaire

### 3.1.1 Entre un client et le serveur proxy

```
lettre_chiffre = [a-zA-Z]/[0-9]
id = 2*10(lettre_chiffre/"-"/"_")
espace = \d32
crlf = \d13\d10
base64_ligne= 1*76(lettre_chiffre/
     "+" / "/" / "=")
zip_base64 = 0*(base64_ligne crlf)
zipfile = zip_base64 "###" crlf
quit = "QUIT" crlf
message = 1*200(lettre_chiffre/espace)
version_id = 1*3[0-9]
pull = "PULL" espace id espace
     version_id crlf
commit = "COMMIT" espace id espace
     version crlf
create = "CREATE" espace id crlf
ok = "OK" [message] crlf
err = "ERR" [message] crlf
```

### 3.1.2 Entre 2 nœuds

```
lettre_chiffre = [a-zA-Z]/[0-9]
id = 2*10(lettre_chiffre/"-"/"_")
espace = \d32
crlf = \d13\d10
base64_ligne= 1*76(lettre_chiffre/
     "+" / "/" / "=")
zip_base64 = 0*(base64_ligne crlf)
zipfile = zip_base64 "###" crlf
info = "INFO" espace id espace
       version_id crlf
message = 1*200(lettre_chiffre/espace)
port = 1*5[0-9]
version_id = 1*3[0-9]
whohas = "WHOHAS" espace id espace
          version_id crlf
provide = "PROVIDE" espace id espace
        version_id espace port crlf
retrieve = "RETRIEVE" espace id espace
            version_id crlf
ok = "OK" [message] crlf
err = "ERR" [message] crlf
quit = "QUIT" crlf
noeud_id = 1*3[0-9]
alive = "ALIVE" espace noeud_id crlf
```

## 3.2 Les adresses IP et ports

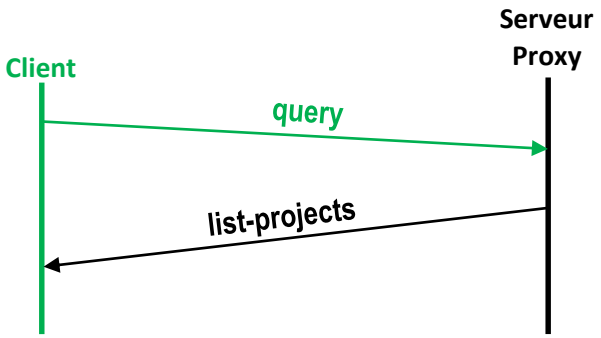
Dans ce projet, nous avons identifié 2 adresses IP multicast, nommées multicast-1 et multicast-2. Vous pouvez ainsi utiliser `225.0.<numeroLabo>.<numeroMachine>` et `225.1.<numeroLabo>.<numeroMachine>`. Ainsi, au labo IL41 sur la 1ère machine, cela donnerait `225.0.41.1` et `225.1.41.1`. A terme, tout le monde pourrait utiliser le même groupe multicast puisque le protocole est identique.

Pour les ports, vous pourriez utiliser le port `4000` pour les échanges entre le client et le serveur proxy, `5000` pour les échanges multicast (sur les 2 adresses identifiées) et `6000` pour les échanges de fichiers entre 2 nœuds.

## 3.3 Quelques options

On vous demande d'implémenter **deux options au moins**[3] :

- Cette version ne permet pas de garder un historique des versions. Implémenter la gestion des versions.
- Cette version ne permet pas de connaître les projets actuellement disponibles sur le système de gestion de source, pour y remédier, implémentez le message `QUERY` :




```
lettre_chiffre = [a-zA-Z]/[0-9]
id = 2*10(lettre_chiffre/"-"/"_")
espace = \d32
crlf = \d13\d10
```

```
version_id = 1*3[0-9]
query = "QUERY" crlf
list-projects = "LIST"
 espace 0*(id:version_id;) crlf
```

- Implémenter SSL dans le dialogue entre le client et le serveur Proxy
- Implémenter une authentification des utilisateurs au niveau du serveur proxy avec la possibilité de s'enregistrer et de se connecter.
- Implémenter un nœud particulier sauvegardant les projets et versions vers un service cloud comme Dropbox ou HubiC

# 4. Quelques liens utiles

- *ZIP : https://github.com/zeroturnaround/zt-zip*
- *JAXB : https://examples.javacodegeeks.com/core-java/xml/bind/jaxb-marshal-example/*

[3] Si vous avez d'autres idées que les options proposées ici, vous pouvez toujours les soumettre à votre professeur de laboratoire.