

# Block Exchange Protocol v1

Claudio Sousa

2017

Conception de protocoles réseau

HEPIA

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Block Exchange Protocol</b>	<b>3</b>
2.1	Diagramme d'états . . . . .	3
2.1.1	<i>Actions et conditions communes</i> . . . . .	3
2.1.2	Actions . . . . .	3
2.1.3	Conditions . . . . .	3
2.1.4	Les timers . . . . .	4
2.1.4.1	Timers d'exception . . . . .	4
2.1.5	Variables . . . . .	4
2.1.6	État d'exception <i>handleException</i> . . . . .	4
2.1.7	Schema . . . . .	4
2.1.8	Les block et leurs états . . . . .	6
2.1.8.1	Block <i>Initialization</i> . . . . .	6
2.1.8.1.1	État <i>Waiting Hello</i> . . . . .	6
2.1.8.2	Block <i>Establish connection</i> . . . . .	6
2.1.8.2.1	État <i>Verify deviceId</i> . . . . .	6
2.1.8.2.2	État <i>Waiting ClusterConfig</i> . . . . .	6
2.1.8.2.3	État <i>Waiting Index</i> . . . . .	7
2.1.8.3	Block <i>Main loop</i> . . . . .	7
2.1.8.3.1	État <i>time to Ping?</i> . . . . .	7
2.1.8.3.2	État <i>peer Ping missing?</i> . . . . .	8
2.1.8.3.3	État <i>download in progress?</i> . . . . .	8
2.1.8.3.4	État <i>newerBlocks to notify?</i> . . . . .	8
2.1.8.3.5	État <i>missingBlocks to request?</i> . . . . .	8
2.1.8.3.6	État <i>message to handle?</i> . . . . .	9
2.1.8.3.7	État <i>messages checked</i> . . . . .	9
2.2	Diagrammes de séquence . . . . .	11
2.2.1	Phase initiale . . . . .	11
2.3	<i>Main loop</i> . . . . .	11
2.4	Diagram de classe des Messages . . . . .	11
<b>3</b>	<b>Bep client</b>	<b>14</b>
3.1	Class diagramme . . . . .	14
3.2	Utilisation . . . . .	15
3.2.1	Exemples . . . . .	15
<b>4</b>	<b>Références</b>	<b>16</b>

# 1 Introduction

Ce document est le résultat de l'étude du protocole *Block Exchange Protocol*<sup>1</sup> (BEP), développé et implémenté par Syncthing<sup>2</sup>. L'étude se décompose en deux chapitres:

Le premier chapitre, *Block Exchange Protocol*, décrit l'analyse faite sur le protocole du même nom. On modélise le protocole depuis trois approches différentes:

- Le diagramme d'états
- Le diagramme de séquences
- Le diagramme de classes des messages échangées

Dans le deuxième chapitre, *BEP Client*, on décrit le mini-projet consistant à implémenter une partie le protocole décrit au premier chapitre.

## 2 Block Exchange Protocol

### 2.1 Diagramme d'états

Dans ce diagramme d'états on montre en détail le cas nominal d'exécution: les différents états dans lesquels un noeud BEP peut se trouver, les conditions qui peuvent l'amener à transiter d'état et les actions liées à ces transitions. On décrit également les différents scénarios d'exécution qui peuvent amener le noeud à transiter vers un état d'exception.

La syntaxe utilisée est basée sur celle vue en cours, en particulier les notes sur les transitions ont la forme  $\frac{\text{condition}}{\text{action}}$ .

Le diagramme proposé ici respecte la contrainte forte que, pour chaque état, une seule condition de transition ne peut être vraie à la fois. Ceci est important pour avoir un comportement d'exécution prévisible.

#### 2.1.1 Actions et conditions communes

On définit ici quelques *actions* et *conditions* communes utilisées à plusieurs endroits du diagramme d'état. Les actions et conditions qui apparaissent une seule fois dans le diagramme sont décrites dans leur état respectif.

#### 2.1.2 Actions

**Data.req:** demande à la couche en dessous (couche transport) d'envoyer le message passé en paramètre.

Exemple: *Data.req(Hello)* pour envoyer un message Hello.

**startTimer:** démarre le timer spécifié. Si le timer est en exécution, il est redémarré.

**cancelTimer:** annule l'exécution du timer passé paramètre.

#### 2.1.3 Conditions

**timerExpired:** le timer spécifié a expiré.

**Data.ind:** un message du type spécifié a été reçu et son type est celui spécifié en paramètre.

Exemples:

- *Data.ind(Hello)* est vrai si le prochain message dans le buffer de réception est de type *Hello*.
- *Data.ind(msg != Hello)* est vrai si le prochain message dans le buffer est de type différent de *Hello*.

---

<sup>1</sup>@bep

<sup>2</sup>@syncthing

### 2.1.4 Les timers

Les timers décrits ici permettent de rajouter la dimension du temps dans le protocole. La valeur de leur temps n'est pas toujours précisée car dépendante de l'implémentation.

**pingTimer:** détermine le temps d'attente maximal depuis le dernier message envoyé au pair, avant d'envoi du message ping (*heartbeat*). Le protocole<sup>3</sup> spécifie que la valeur de ce timer est de 90s.

**downloadTimer:** vérifie si un *download* a toujours lieu afin de notifier la progression le cas échéant.

#### 2.1.4.1 Timers d'exception

Lorsque ces timers expirent, un événement d'exception à lieu et la machine d'états passe à l'état *handleException*.

**waitingResponseTimer:** détermine le temps maximal d'attente de réception d'un message.

**peerPingTimer:** de valeur supérieure pingTimer, ce timer compte le temps depuis la dernière réception d'un message de la part du noeud pair.

**downloadTimer:** mesure la fréquence à laquelle des message DownloadProgress doivent être envoyés, si nécessaire.

### 2.1.5 Variables

Lors de l'exécution de la machine d'états, quelques variables globales maintiennent des information de synchronisation.

**newerBlocks:** cette variable représente tous les nouveaux blocks qui n'existent que localement et qui n'ont pas encore été annoncés au server. Souvent, ils résultent d'une modification du fichier effectuée par l'utilisateur (modification, ajout, suppression de fichier).

**missingBlocks:** cette variable représente tous les nouveaux blocks qui existent chez le noeud pair mais pas chez nous, et dont le contenu n'a pas encore été demandé par un message *Request*.

### 2.1.6 État d'exception *handleException*

L'exécution de la machine d'état tombe dans cet état particulier lorsqu'un événement non attendu à lieu.

Quelques exemples:

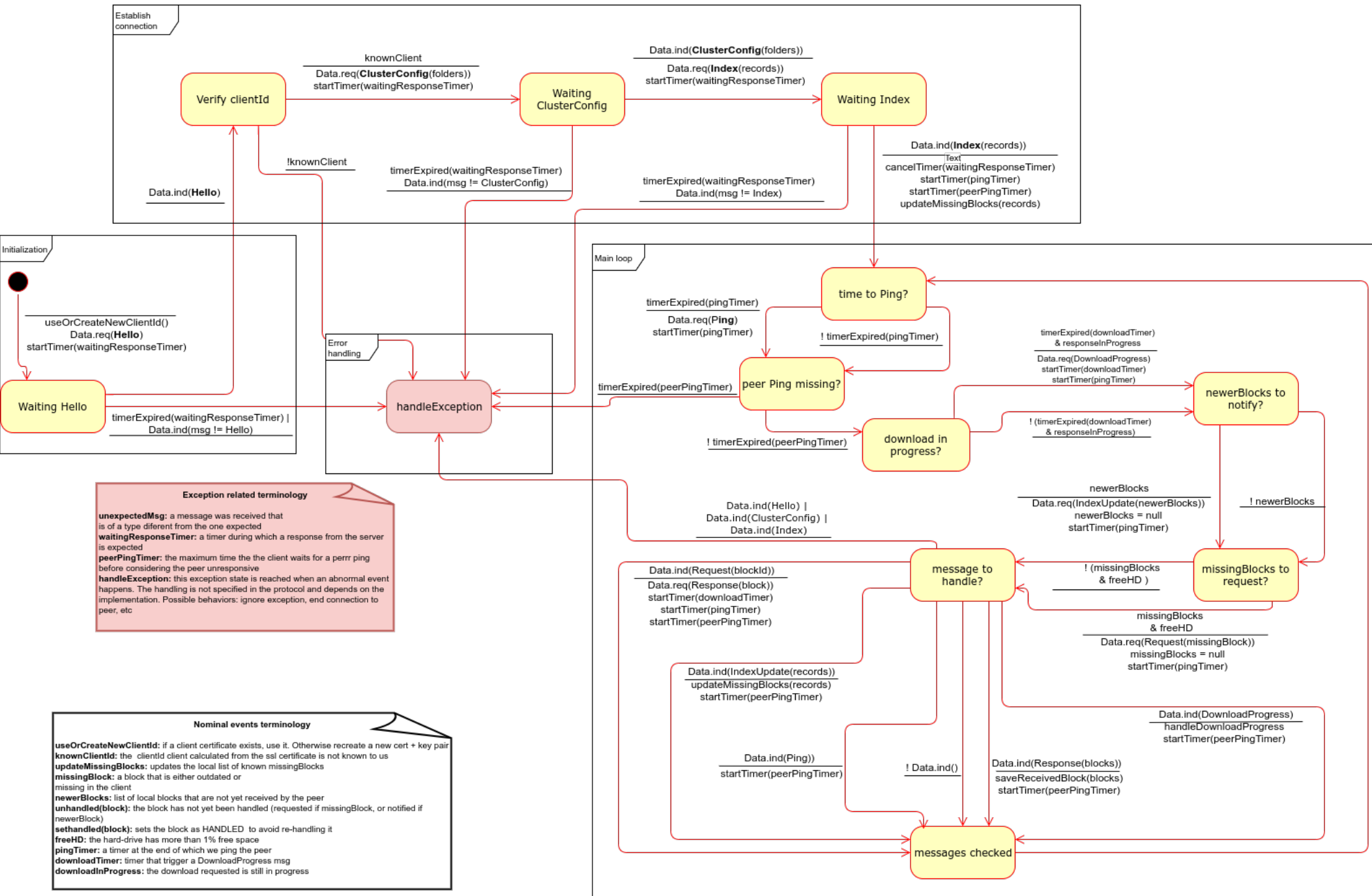
- On reçoit un message de type différent de *Hello* suite à l'envoi de notre *Hello*
- On ne reçoit pas de réponse à notre message *Hello*

On spécifie dans notre diagramme d'états les conditions qui nous amènent dans cet état d'exception, mais on ne spécifie pas le traitement qui a lieu dans cet état. On considère que le choix du traitement dépend de l'implémentation.

### 2.1.7 Schema

---

<sup>3</sup>@bep



## 2.1.8 Les block et leurs états

### 2.1.8.1 Block *Initialization*

Dans ce block, le client est initialisé et essaye de joindre le noeud pair.

#### Actions initiales, exécutées sans condition

- **useOrCreateNewClientId:** bien que pas partie strictement du protocole, cette étape est cruciale car elle initialise, si besoin, la clé publique et le certificat à être utilisés par le client. L'identifiant du client est une information dérivée directement du certificat public.
- **Data.req(Hello):** On envoie le message *Hello* au noeud pair (pour plus de détails sur les messages, voir chapitre [Diagram de classe des Messages](#)).
- **startTimer(waitingResponseTimer)**

#### 2.1.8.1.1 État *Waiting Hello*

Après l'envoi du message *Hello*, le client reste dans cet état jusqu'à ce qu'une de deux conditions soit remplie.

##### Conditions de sortie:

1. **Data.ind(Hello):** on reçoit le *Hello* du noeud pair, on passe à l'état [Verify deviceId](#) du prochain block
2. **timerExpired(waitingResponseTimer) | Data.ind(msg != Hello):** condition d'exception, a lieu si on ne reçoit pas de message dans le temps alloué (*waitingResponseTimer*) ou qu'on reçoit un message de type non attendu (différent de *Hello*). Le client passe à l'état [handleException](#)

### 2.1.8.2 Block *Establish connection*

Après avoir réussi à attendre le noeud pair dans le block précédent, le client va ici essayer d'établir une connexion et échanger l'états de leurs folders.

#### 2.1.8.2.1 État *Verify deviceId*

Après avoir échangé les messages *Hello*, le client va vérifier que le pair est un client connu en calculant le *deviceId* du pair (depuis son certificat utilisé pour établir la connexion SSL) et en vérifiant que le *deviceId* obtenu est dans la liste des pairs auxquels le client fait confiance.

Les détails concernant le maintien de la liste des *clientIds* connus ne fait pas partie du protocole et dépendra de l'implémentation.

##### Conditions de sortie:

1. **knownDevice:** le *deviceId* du pair est reconnu comme valide.

##### Actions:

- **Data.req(ClusterConfig(Folders)):** on envoie le message *ClusterConfig* contenant les informations des *Folders* partagés
  - **startTimer(waitingResponseTimer)**
2. **!knownDevice:** condition d'exception, vérifiée si le *deviceId* calculé est inconnu.

#### 2.1.8.2.2 État *Waiting ClusterConfig*

Suite à l'envoi du message *ClusterConfig*, le client doit attendre la réception du message du même type de la part du noeud pair.

##### Conditions de sortie:

1. **Data.ind(ClusterConfig(Folders))**: on reçoit le message attendu avec les informations des *Folders* partagés

**Actions:**

- **Data.req(Index(Records))**: le client envoie le message *Index* contenant les informations des *Blocks* connus
  - **startTimer(waitingResponseTimer)**
2. **timerExpired(waitingResponseTimer) | Data.ind(msg != ClusterConfig)**: condition d'exception, à lieu si on ne reçoit pas de message dans le temps alloué (*waitingResponseTimer*) ou qu'on reçoit un message de type non attendu (*ClusterConfig*)

#### 2.1.8.2.3 État *Waiting Index*

Suite à l'envoi du message *Index*, le client attend un message *Index* du noeud pair.

**Conditions de sortie:**

1. **Data.ind(Index(Records))**: on reçoit le message attendu avec les informations des blocks partagés

**Actions:**

- **cancelTimer(waitingResponseTimer)**: on n'attend plus une réponse immédiate.
  - **startTimer(pingTimer)**: on veut se rappeler quand envoyer le ping au pair.
  - **startTimer(peerPingTimer)**: on veut savoir quand l'attente d'un message de la part du noeud pair à expiré.
  - **updateMissingBlocks(records)**: on compare les *records* envoyés avec ceux reçus, afin de mettre à jour la variable *missingBlocks*. Ceux blocks seront demandés au pair ultérieurement.
2. **timerExpired(waitingResponseTimer) | Data.ind(msg != Index)**: condition d'exception, à lieu si on ne reçoit pas de message dans le temps alloué (*waitingResponseTimer*) ou qu'on reçoit un message de type non attendu (*Index*)

#### 2.1.8.3 Block *Main loop*

Ce block contient de la boucle d'exécution principale du programme. Dans les blocks précédents, la connexion fut bien établie avec le client, et chaque noeud a échangé l'état de leur *Folders* et *Records*. Dans ce block on itérera sans fin afin de synchroniser tous les blocks qui n'existent pas chez tous les pairs dans leur version la plus récente. On sera à l'écoute aussi de nouveaux messages notifiant des nouveaux records chez le pair, de demandes de *push* de blocks manquants chez le pair, de messages *Response* à nos messages *Request*, etc.

##### 2.1.8.3.1 État *time to Ping?*

Lors de cet état on vérifie si le timer de notre *Ping* a expiré.

**Conditions de sortie:**

1. **timerExpired(pingTimer)**: il est temps d'envoyer un message *Ping*

**Actions:**

- **Data.req(Ping)**
  - **startTimer(pingTimer)**: on veut se rappeler de quand renvoyer le ping au pair.
2. **!timerExpired(pingTimer)**: on ne fait rien, on passe à l'état suivant

#### 2.1.8.3.2 État *peer Ping missing*?

Lors de cet état on vérifie si le timer du *Ping* du noeud pair a expiré.

Conditions de sortie:

1. **!timerExpired(peerPingTimer):** on ne fait rien, on passe à l'état suivant.
2. **timerExpired(peerPingTimer):** le pair n'a pas envoyé de message dans le temps alloué, on passe à l'état d'exception.

#### 2.1.8.3.3 État *download in progress*?

Lors de cet état on vérifie si des messages *Réponse* sont encore en envoi.

Conditions de sortie:

1. **timerExpired(downloadTimer) & responseInProgress:** si un message *Response* initié plutôt est toujours en envoi, on envoi un message *DownloadProgress* pour notifier le pair du progrès.

Actions:

- **Data.req(DownloadProgress):** envoi l'état de progrès du download
  - **startTimer(downloadTimer)**
  - **startTimer(pingTimer)**
2. **!(timerExpired(downloadTimer) & responseInProgress):** on ne fait rien, on passe à l'état suivant

#### 2.1.8.3.4 État *newerBlocks to notify*?

On vérifie dans cet état si notre client a des nouveaux blocks dont il doit notifier le pair.

Conditions de sortie:

1. **newerBlocks:** il y a des nouveaux blocks chez nous dont le pair ne connaît pas encore l'existence.

Actions:

- **Data.req(IndexUpdate(newerBlocks)):** on notifie le pair que des nouveaux blocks existent chez nous.
  - **newerBlocks = null:** on marque qu'il n'y a plus de newerBlocks
  - **startTimer(pingTimer)**
2. **!newerBlocks:** on ne fait rien, on passe à l'état suivant

#### 2.1.8.3.5 État *missingBlocks to request*?

On vérifie dans cet état si on connaît des nouveaux blocks existant seulement chez le noeud pair dont on a pas encore fait la demande.

Conditions de sortie:

1. **missingBlocks & freeHD:** le pair a des block qu'on a pas encore et il nous avons suffisamment d'espace disque libre (client Syncthing exige 1% d'espace libre minimal)

Actions:

- **Data.req(Request(missingBlocks)):** on demande au noeud pair de nous envoyer les blocks manquants.
- **missingBlocks = null:** on marque qu'il n'y a plus de missingBlocks.
- **startTimer(pingTimer)**



2. **!(missingBlocks & freeHD)**: on ne fait rien, on passe à l'état suivant.

#### 2.1.8.3.6 État *message to handle*?

On vérifie dans cet état si un message a été reçu et on le traite le cas échéant.

**Conditions de sortie:**

1. **!Data.ind()** pas de message à traiter, on passe à l'état suivant.
2. **Data.ind(Request(missingBlock))**: le pair nous fait la demande de blocks qu'il n'a pas.

**Actions:**

- **Data.req(Response(missingBlocks))**: on envoie les blocks manquants.
- **startTimer(downloadTimer)**: on se rappelle de vérifier plus tard si des messages de *DownloadProgress* doivent être envoyés.
- **startTimer(pingTimer)**
- **startTimer(peerPingTimer)**

3. **Data.ind(DownloadProgress)**: on reçoit la notification du progress d'un download

**Actions:**

- **handleDownloadProgress**: notification utilisateur ? Dépendra de l'implémentation.
- **startTimer(peerPingTimer)**

4. **Data.ind(IndexUpdate(records))**: on reçoit la notification que des nouveaux records existent chez le pair

**Actions:**

- **updateMissingBlocks**: on met à jour la variable `missingBlocks` qui contient les blocks manquant chez nous.
- **startTimer(peerPingTimer)**

5. **Data.ind(Response(blocks))**: on reçoit la réponse à une *request* précédente

**Actions:**

- **saveReceivedBlocks(blocks)**: on sauvegarde les nouveaux blocks reçus.
- **startTimer(peerPingTimer)**

6. **Data.ind(Ping)**: on reçoit le *Ping*

**Actions:**

- **startTimer(peerPingTimer)**

7. **Data.ind(Hello) | Data.ind(ClusterConfig) | Data.ind(Index)**: on reçoit un message auquel on ne s'attend pas, on passe à l'état d'exception

#### 2.1.8.3.7 État *messages checked*

État symbolique, atteint après la gestion d'un message reçu (ou son absence).

On boucle vers l'état initial du block (*time to Ping?*) sans autre condition.

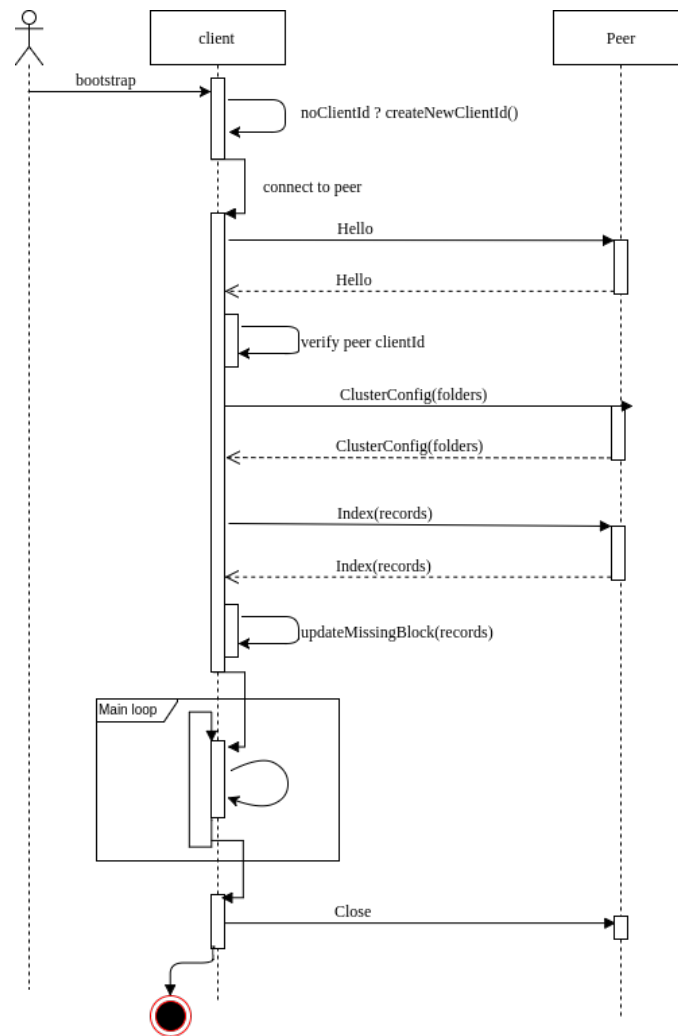


Figure 1: Diagramme de séquence - connect to peer

## 2.2 Diagrammes de séquence

### 2.2.1 Phase initiale

Le diagramme de séquence de la figure 1 montre les différentes échanges qui ont lieu lors de la phase initial de connection entre deux noeuds BEP (nommés ici *client*, et *Peer*).

Le déroulement est assez linéaire et décrit le cas nominal de ce qui se passe lors des blocks *Initialization* et *Establish connection* du diagramme d'états. En quelques mots, une fois la connection établie entre les deux noeuds, les messages suivantes sont échangées:

- *Hello*: contenant le nom et numéro de version du client
- *ClusterConfig*: avec l'énumération des folders qui sont partagés avec le noeud
- *Index*: information sur les fichiers connus du noeud et leur version

Ce diagramme finit par le block *Main loop*, où le programme restera pendant toute la durée de l'exécution.

Lors de son arrêt, le programme peut envoyer le message *Close*, sans aucun contenu, pour informer le noeud pair de la fermeture de connection.

### 2.3 *Main loop*

La figure 2 montre le diagramme de séquence décrivant ce qui se passe dans block *Main loop*.

Ici nous avons choisis de montrer les différentes actions qui ont lieu sous forme de réaction à un événement. Exemples d'événements sont: un timer expire, un message est reçu du noeud pair, un événement du système de fichiers a lieu.

On exemplifie un cas exception: la gestion du timer *peerPingTimer* qui a lieu lorsqu'on a pas reçu de message de la part du pair depuis trop longtemps. Cet événement nous amène sur l'état *handleexception* dont le traitement dépend de l'implémentation.

Par soucis de simplicité graphique, chaque traitement n'est représenté qu'une seule fois dans sa symétrie, soit dans sa version serveur, soit dans sa version client

## 2.4 Diagram de classe des Messages

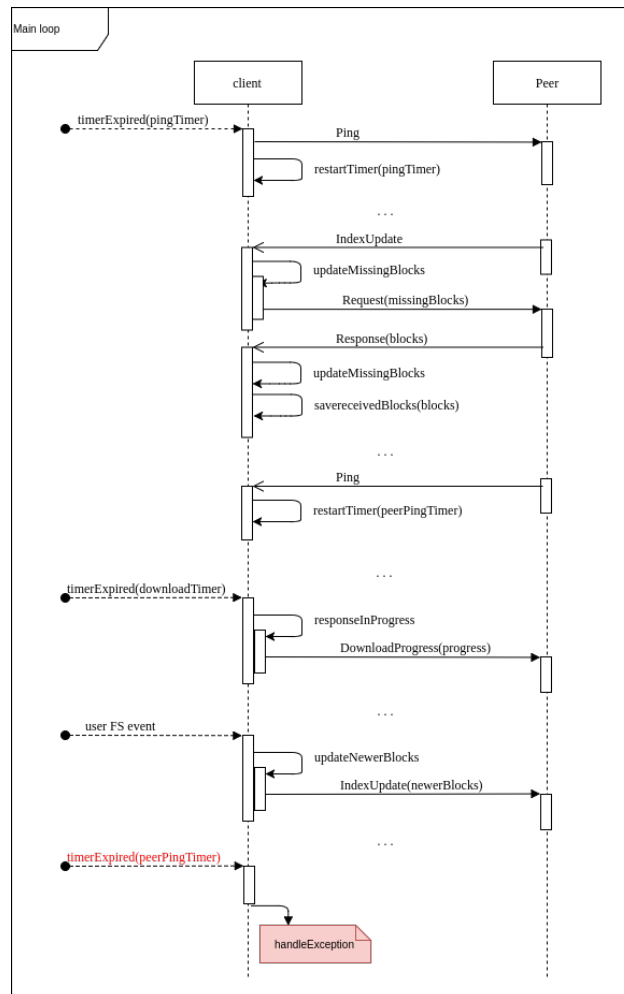
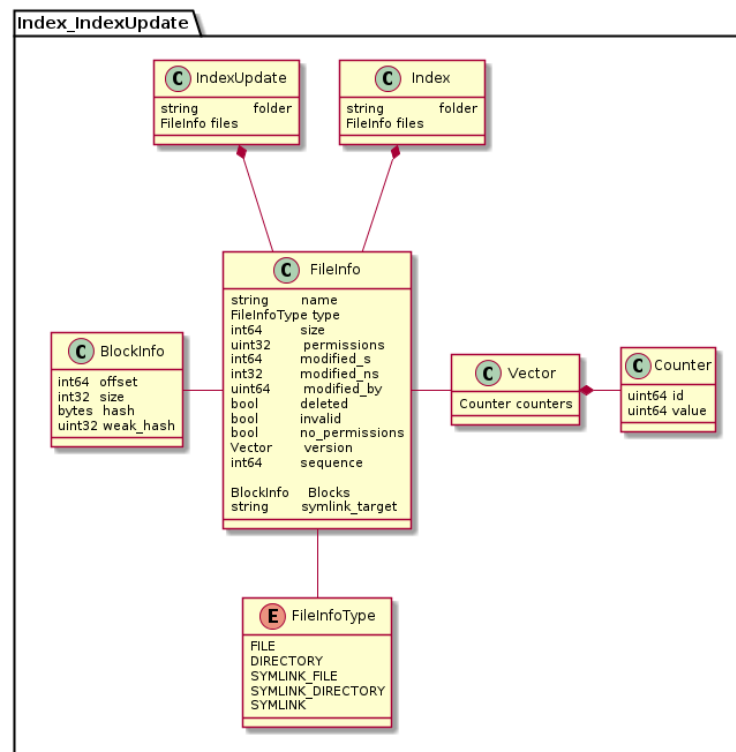
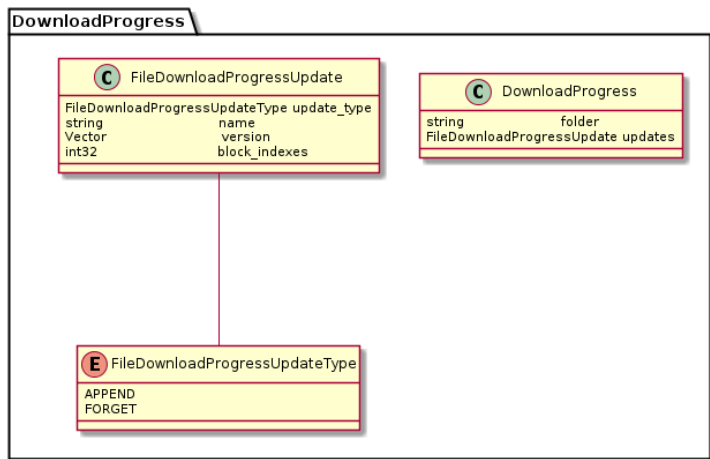
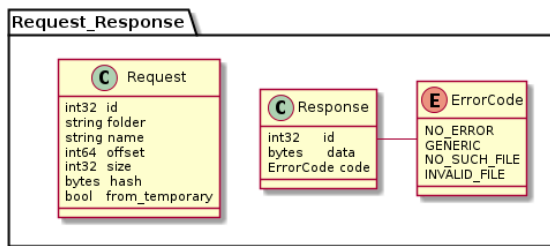
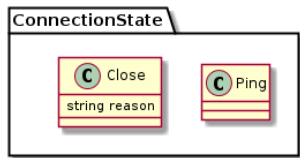
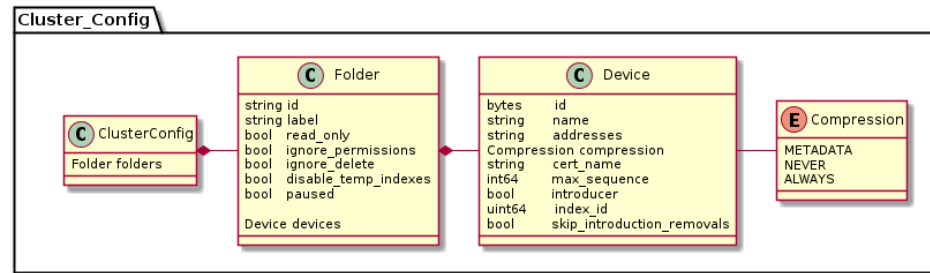
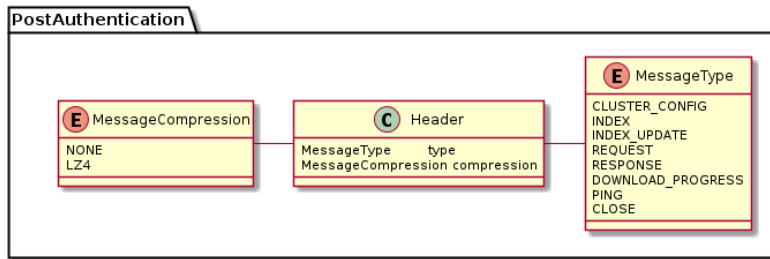
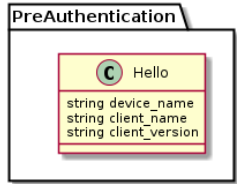


Figure 2: Diagramme de séquence - main loop



### 3 Bep client

Nous avons implémenté une partie du protocole BEP dans un client nommé *Bep client* qui offre quelques fonctionnalités de base BEP. Ces fonctionnalités sont disponibles en tant que exécutable en ligne de commandes, mais aussi en tant que librairie. Cette dernière pourrait être utilisée par une application souhaitant communiquer avec un serveur BEP sans avoir à re-implémenter le protocole.

L'énoncé établit quelques limitations:

- la synchronisation se fait avec un seul noeud Syncthing.
- on suppose qu'on connaît l'IP du noeud Syncthing, on ne fera pas de découverte de protocole Global/Local Discovery

#### 3.1 Class diagramme

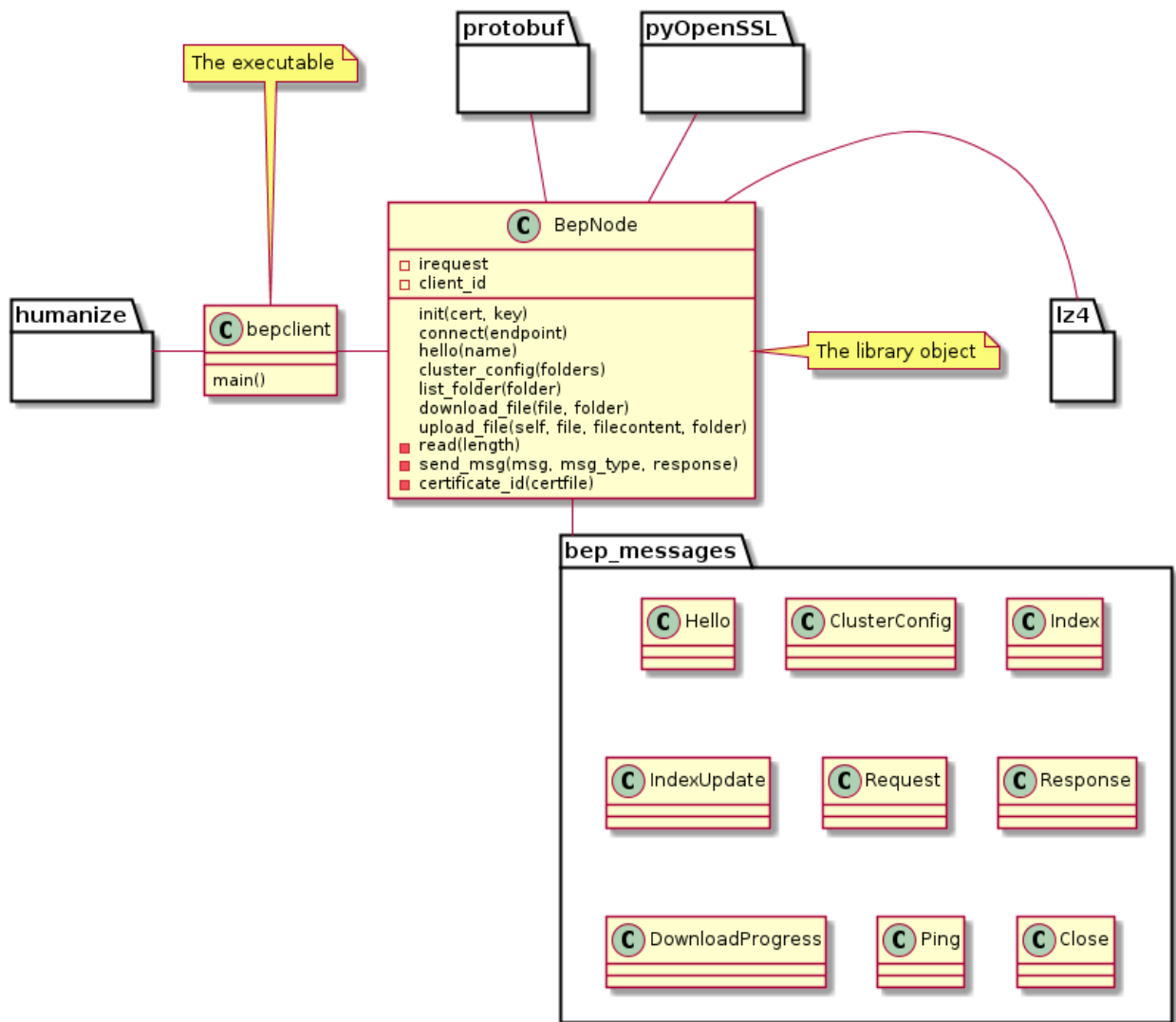


Figure 3: bep client

La classe *BepNode* est au coeur de l'implémentation. Elle implémente le protocole BEP et utilise les classes

me message générées depuis la définition protobuf. Cette classe utilise des packages pour la serialization, connection SSL et décompression.

L'exécutable *bepclient* offre le *Command Line Interface* (CLI) sur la fonctionnalité du *BepClient*.

## 3.2 Utilisation

L'exécutable *bepclient* à l'interface suivant:

```
$> python bepcient.py -h
```

Bep client, can be used to download & upload files to a BEP node.

Usage:

```
bepclient.py [options] (showid | connect <host> [share <share_id> [(download|upload) <file>]])
```

Examples:

```
bepclient.py [options] showid
bepclient.py [options] connect 129.194.186.177
bepclient.py [options] connect 129.194.186.177 share hyperfacile
bepclient.py [options] connect 129.194.186.177 share hyperfacile download plistlib.py
bepclient.py [options] connect 129.194.186.177 share hyperfacile upload filename.py
bepclient.py -h | --help
```

Options:

```
-h --help          Show this screen.
--key=<keyfile>    Key file [default: config/key.pem].
--cert=<certfile>  Certificate file [default: config/cert.pem].
--port=<port>      Host port [default: 22000]
--name=<name>      The client name [default: Claudio's BEP client].
```

### 3.2.1 Exemples

Montrer l'id du certificat utilisé:

```
$> python bepcient.py showid
```

Client id: HG3DI2F-JKKVY3Z-HL5ZCWN-FH53M35-CMGFGE5-WAPGTV6-5SBWC6W-4VZSFA

Montrer les folders d'un noeud pair:

```
$> python ./bepclient.py connect 129.194.186.177
```

Connected to: redbox

Shared folders: 3

- (facile)
- (hyperfacile)
- (moins\_facile)

Montrer les fichiers d'un folder:

```
$> python ./bepclient.py connect 129.194.186.177 share hyperfacile
```

Connected to: redbox

Folder 'hyperfacile' files:

- platform.py	size: 51.4K	modified: Nov 02	blocks: 1
- platform.pyc	size: 36.8K	modified: Nov 02	blocks: 1
- plistlib.py	size: 14.8K	modified: Nov 02	blocks: 1
- plistlib.pyc	size: 18.7K	modified: Nov 02	blocks: 1

Télécharger un fichier:

```
$> python ./bepclient.py connect 129.194.186.177 share hyperfacile download plistlib.py
Connected to: redbox
File plistlib.py downloaded
```

Upload un fichier:

```
$> python ./bepclient.py connect 129.194.186.177 share hyperfacile upload test.py
Connected to: redbox
File test.py uploaded
```

## 4 Références