Block Exchange Protocol v1

Claudio Sousa

2017

Conception de protocoles réseau

HEPIA

Contents

1 Introduction

Ce document est le résultat de l'étude du protocole *Block Exchange Protocol*[1] (BEP), développé et implémenté par Syncthing.[2] 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 séquence
- Le diagramme de classes des messages échangées

Dans le deuxième chapitre, *BEP Client*, on décrit le mini-projet consistant à implémenter une partie du 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 a transiter d'état et les actions liées à ces transitions. On décrit également les différents scenarios d'exécution qui peuvent amener le noeud a 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 $\frac{condition}{action}$.

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

Conceptuellement, la machine d'états est essentiellement réactive, réagissant à des événements déclenchés par la couche du dessus (l'application) ou à la réception d'un messages d'un noeud BEP pair. Dans notre diagramme, nous proposons une représentation séquentielle afin de tenir en compte la dimension du temps. Pour ce faire, nous utilisons des variables globales qui conditionnent le comportement de la machine à état des actions. Aussi, certaines actions que nous appelons lors des transitions n'ont pas de comportement spécifié dans ce document car leur comportement est implémenté par la couche du dessus. En general elles mettent à jour des variables globales qui vont conditionner le comportement de la machine à états.

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.

[1] Block Exchange Protocol v1 (2017)
[2] Syncthing (2017)

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*.

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: determine le temps d'attente maximal depuis le dernier message envoyé au pair, avant l'envoi du message ping (*heartbeat*). Le protocole[3] 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 a 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 messages 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.

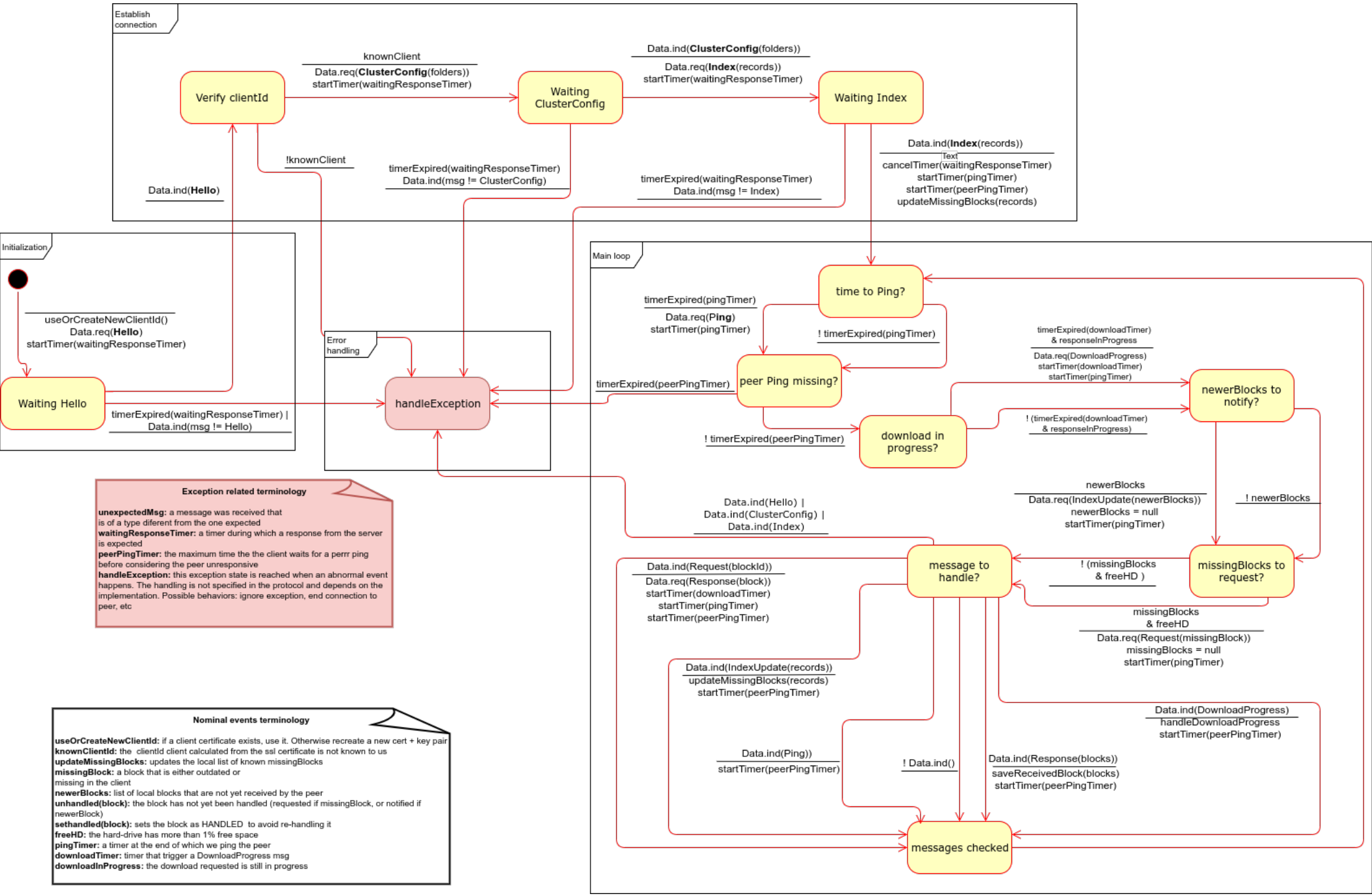
Quelques exemples:

- 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.

[3] Block Exchange Protocol v1 (2017)

2.1.7 Schema


Establish connection
Verify clientId
knownClient
Data.req(ClusterConfig(folders))
startTimer(waitingResponseTimer)
Waiting ClusterConfig
Data.ind(ClusterConfig(folders))
Data.req(Index(records))
startTimer(waitingResponseTimer)
Waiting Index
Data.ind(Index(records))
Text
cancelTimer(waitingResponseTimer)
startTimer(pingTimer)
startTimer(peerPingTimer)
updateMissingBlocks(records)
!knownClient
timerExpired(waitingResponseTimer)
Data.ind(msg != ClusterConfig)
timerExpired(waitingResponseTimer)
Data.ind(msg != Index)
Data.ind(Hello)
Initialization
useOrCreateNewClientId()
Data.req(Hello)
startTimer(waitingResponseTimer)
Waiting Hello
timerExpired(waitingResponseTimer) |
Data.ind(msg != Hello)
Error handling
handleException
Main loop
time to Ping?
timerExpired(pingTimer)
Data.req(Ping)
startTimer(pingTimer)
! timerExpired(pingTimer)
peer Ping missing?
timerExpired(peerPingTimer)
! timerExpired(peerPingTimer)
download in progress?
timerExpired(downloadTimer)
& responseInProgress
Data.req(DownloadProgress)
startTimer(downloadTimer)
startTimer(pingTimer)
newerBlocks to notify?
! (timerExpired(downloadTimer)
& responseInProgress)
newerBlocks
Data.req(IndexUpdate(newerBlocks))
newerBlocks = null
startTimer(pingTimer)
! newerBlocks
Data.ind(Hello) |
Data.ind(ClusterConfig) |
Data.ind(Index)
message to handle?
missingBlocks to request?
! (missingBlocks
& freeHD)
Data.ind(Request(blockId))
Data.req(Response(block))
startTimer(downloadTimer)
startTimer(pingTimer)
startTimer(peerPingTimer)
missingBlocks
& freeHD
Data.req(Request(missingBlock))
missingBlocks = null
startTimer(pingTimer)
Data.ind(IndexUpdate(records))
updateMissingBlocks(records)
startTimer(peerPingTimer)
Data.ind(DownloadProgress)
handleDownloadProgress
startTimer(peerPingTimer)
Data.ind(Ping))
startTimer(peerPingTimer)
! Data.ind()
Data.ind(Response(blocks))
saveReceivedBlock(blocks)
startTimer(peerPingTimer)
messages checked
Exception related terminology
unexpectedMsg: a message was received that is of a type diferent from the one expected
waitingResponseTimer: a timer during which a response from the server is expected
peerPingTimer: the maximum time the the client waits for a perrr ping before considering the peer unresponsive
handleException: this exception state is reached when an abnormal event happens. The handling is not specified in the protocol and depends on the implementation. Possible behaviors: ignore exception, end connection to peer, etc
Nominal events terminology
useOrCreateNewClientId: if a client certificate exists, use it. Otherwise recreate a new cert + key pair
knownClientId: the clientId client calculated from the ssl certificate is not known to us
updateMissingBlocks: updates the local list of known missingBlocks
missingBlock: a block that is either outdated or missing in the client
newerBlocks: list of local blocks that are not yet received by the peer
unhandled(block): the block has not yet been handled (requested if missingBlock, or notified if newerBlock)
sethandled(block): sets the block as HANDLED to avoid re-handling it
freeHD: the hard-drive has more than 1% free space
pingTimer: a timer at the end of which we ping the peer
downloadTimer: timer that trigger a DownloadProgress msg
downloadInProgress: the download requested is still in progress

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 ne faisant pas 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 envoi le message *Hello* au noeud pair(pour plus de détails sur les messages, voir chapitre Diagramme 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 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 réussit 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 maintient 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 envoi 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 atteindre 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 envoi 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 atteint 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*. Ces 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 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 jusqu'à synchroniser de synchroniser tous les blocks qui n'existent pas chez tous les pairs dans leur version la plus récente. On sera a l'écoute aussi de nouveaux messages notifiant des nouvels 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 envoit 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 de 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 blocks qu'on a pas encore et il y a suffisamment d'espace de 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 envoi les blocks manquants.
   - **startTimer(downloadTimer):** on se rappelle de vérifier plus tard si des messages de *Download-Progress* doivent être envoyées.
   - **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(Pint):** 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.

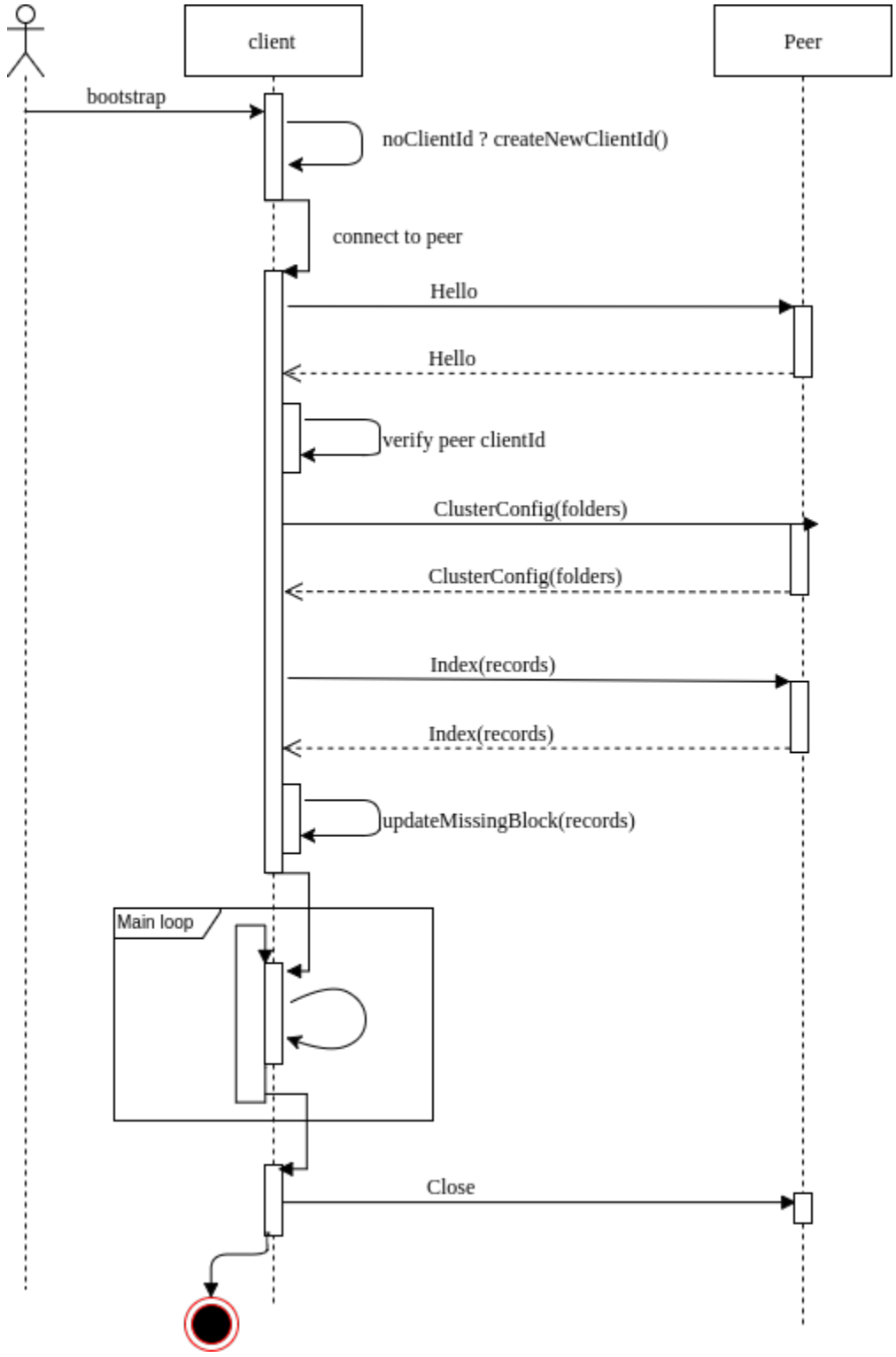


Figure 1: Diagramme de séquence - connect to peer

2.2 Diagrammes de séquence

2.2.1 Phase initialle

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 la 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 choisit de montrer les différentes actions qui ont lieu sous forme de réaction à un événement. Exemples d'événements: 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 lorqu'on n'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.

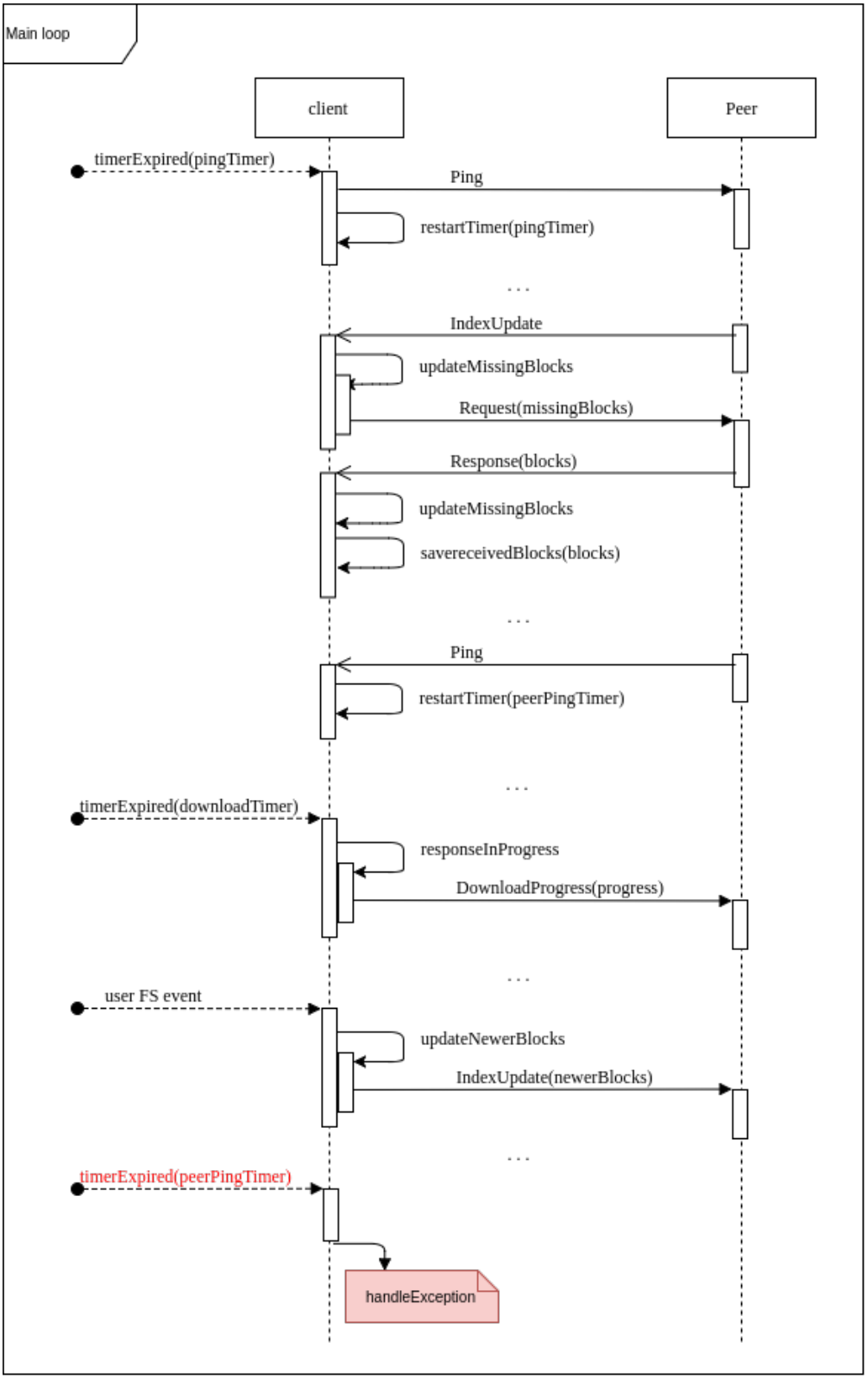


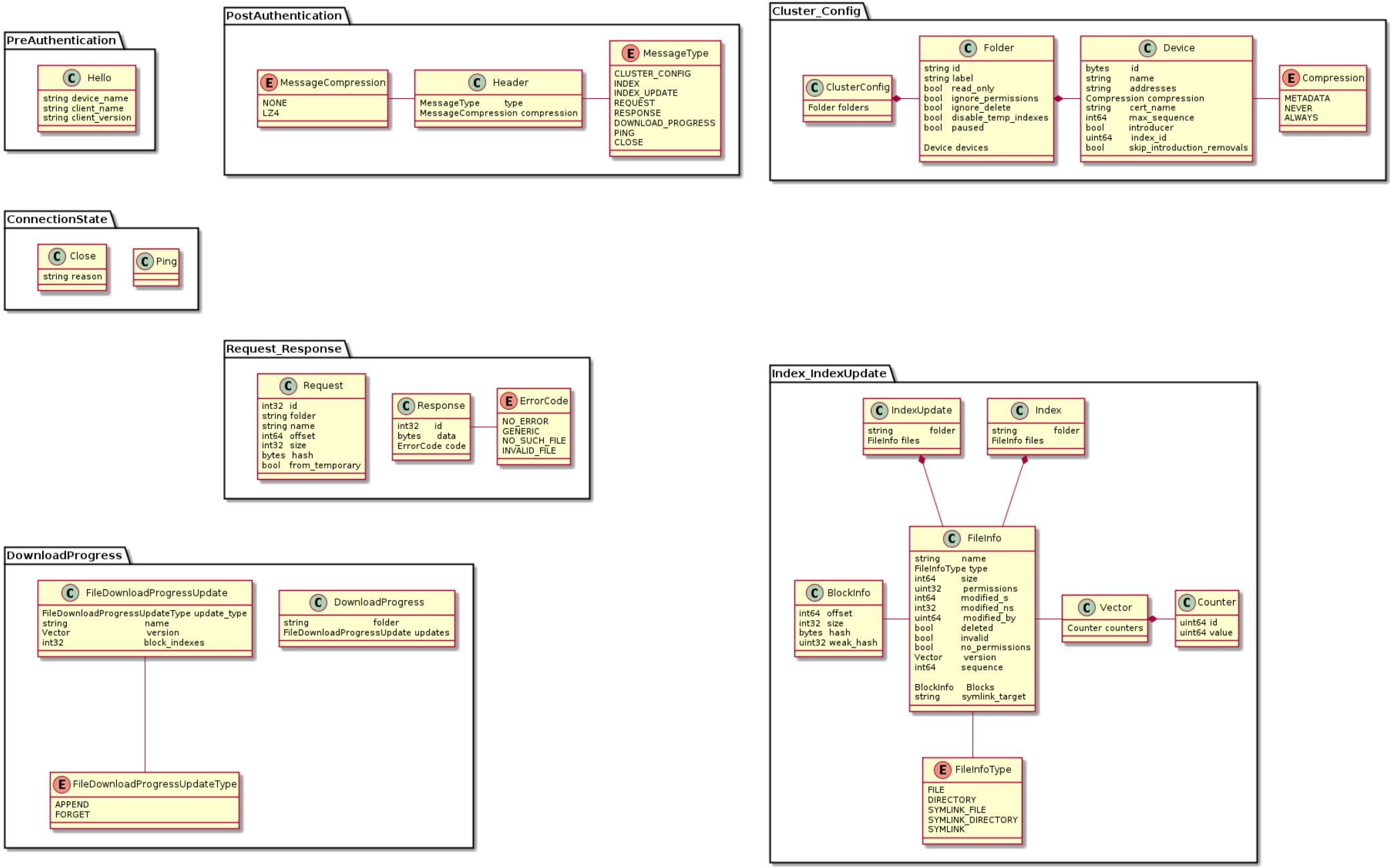
Figure 2: Diagramme de séquence - main loop

2.4 Diagramme de classe des Messages

La page du protocol BEP[4] décrit textuellement le rôle de chaque message et leurs attributs. A cette description nous n'avons pas de mots à rajouter, mais proposons dans ce chapitre une réprésentation graphique en complément.

Le diagramme de classes des différentes méssages échangées est dessiné ci-dessous. Dans ce schéma nous montrons une vue de l'ensemble des messages, les classes et enumérations utilisées, leurs dépendences et leur *packaging* fonctionel.

[4]Syncthing (2017)


PreAuthentication
Hello
string device_name
string client_name
string client_version
PostAuthentication
MessageCompression
NONE
LZ4
Header
MessageType type
MessageCompression compression
MessageType
CLUSTER_CONFIG
INDEX
INDEX_UPDATE
REQUEST
RESPONSE
DOWNLOAD_PROGRESS
PING
CLOSE
Cluster_Config
ClusterConfig
Folder folders
Folder
string id
string label
bool read_only
bool ignore_permissions
bool ignore_delete
bool disable_temp_indexes
bool paused
Device devices
Device
bytes id
string name
string addresses
Compression compression
string cert_name
int64 max_sequence
bool introducer
uint64 index_id
bool skip_introduction_removals
Compression
METADATA
NEVER
ALWAYS
ConnectionState
Close
string reason
Ping
Request_Response
Request
int32 id
string folder
string name
int64 offset
int32 size
bytes hash
bool from_temporary
Response
int32 id
bytes data
ErrorCode code
ErrorCode
NO_ERROR
GENERIC
NO_SUCH_FILE
INVALID_FILE
Index_IndexUpdate
IndexUpdate
string folder
FileInfo files
Index
string folder
FileInfo files
FileInfo
string name
FileInfoType type
int64 size
uint32 permissions
int64 modified_s
int32 modified_ns
uint64 modified_by
bool deleted
bool invalid
bool no_permissions
Vector version
int64 sequence
BlockInfo Blocks
string symlink_target
BlockInfo
int64 offset
int32 size
bytes hash
uint32 weak_hash
Vector
Counter counters
Counter
uint64 id
uint64 value
FileInfoType
FILE
DIRECTORY
SYMLINK_FILE
SYMLINK_DIRECTORY
SYMLINK
DownloadProgress
FileDownloadProgressUpdate
FileDownloadProgressUpdateType update_type
string name
Vector version
int32 block_indexes
DownloadProgress
string folder
FileDownloadProgressUpdate updates
FileDownloadProgressUpdateType
APPEND
FORGET

3 Bep client

Nous avons implémenté une partie du protocole BEP dans un client nommé *BepClient* qui offre quelques fonctionalitées de base BEP. Ces fonctionnalités sont disponibles en tant qu'executable en ligne de commande (CLI), mais aussi en tant que librairie. Cette dernière pourrait être utilisée par une application souhaitant communiquer avec un server BEP sans avoir à re-implémenter le protocole.

L'énoncé établi quelques limitations:

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

Le client offre les les fonctionalitées suivantes: #. Genèrer le *client-id* depuis le certificat #. Se connecter à un client BEP et lister les dossiers partagés (*shares*) #. Lister tous les fichiers d'un dossier partagé #. Télecharger un fichier partagé

La philosophie derrière notre implémentation est celle de proposer un outil simple qui implémente le protocol BEP avec simplicité. Le résultat est un script court qui se veut facile à maintenir. Il peut être utilisé par un script pour créer des taches automatisées.

3.1 Diagramme de séquence

Les fonctionalitées proposées par notre *BepClient* sont incrémentales et ceci est illustré dans le diagramme de séquence montré en figure 3.

Lorqu'on veut obtenir la liste des dossiers partagés par un noeud BEP, le programme exécute les sections *1 Connect* et *2 Get list of shares* du diagramme de séquence. Si on veut obtenir la liste des fichiers dans un dossier partagé, le programme execéte les section 1 et 2, comme précédèment, suivi de la section *3 Get share file list*.

Finalment, lorqu'on désire télécharger un document le programme exécute toutes les sections: 1, 2, 3 et *4 Download a file*.

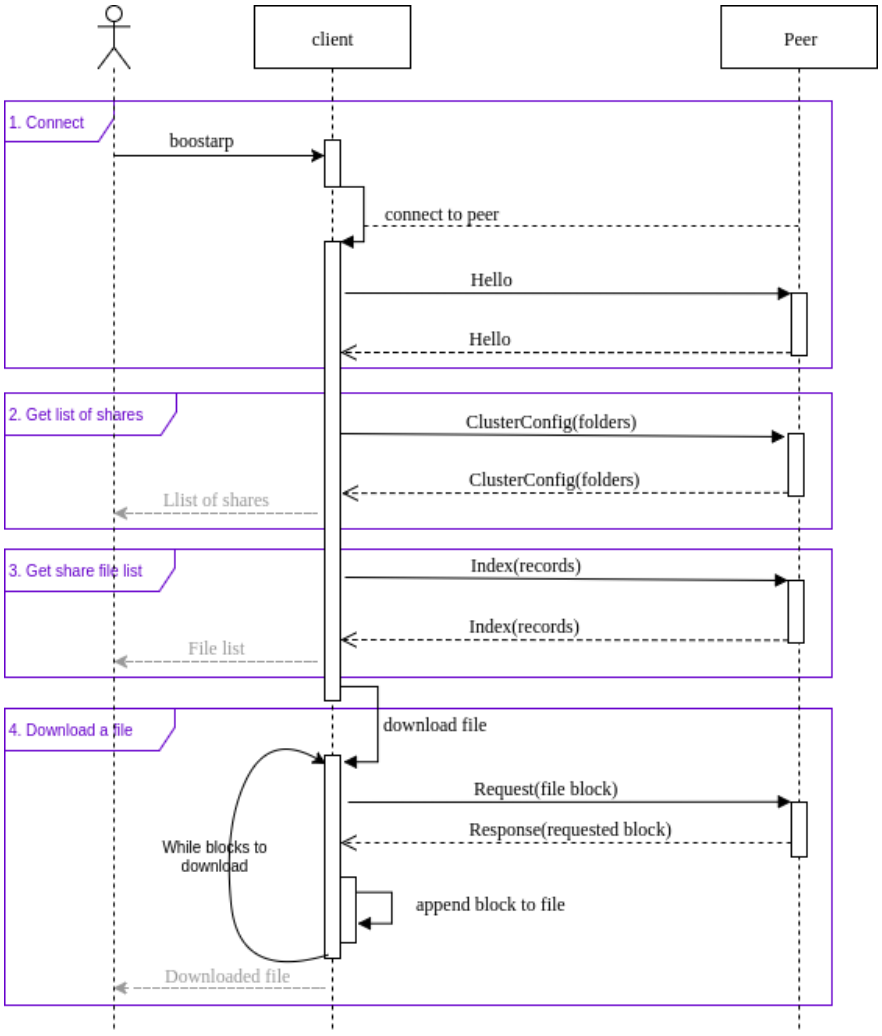


Figure 3: BepClient sequence diagram

3.2 Diagramme de classes

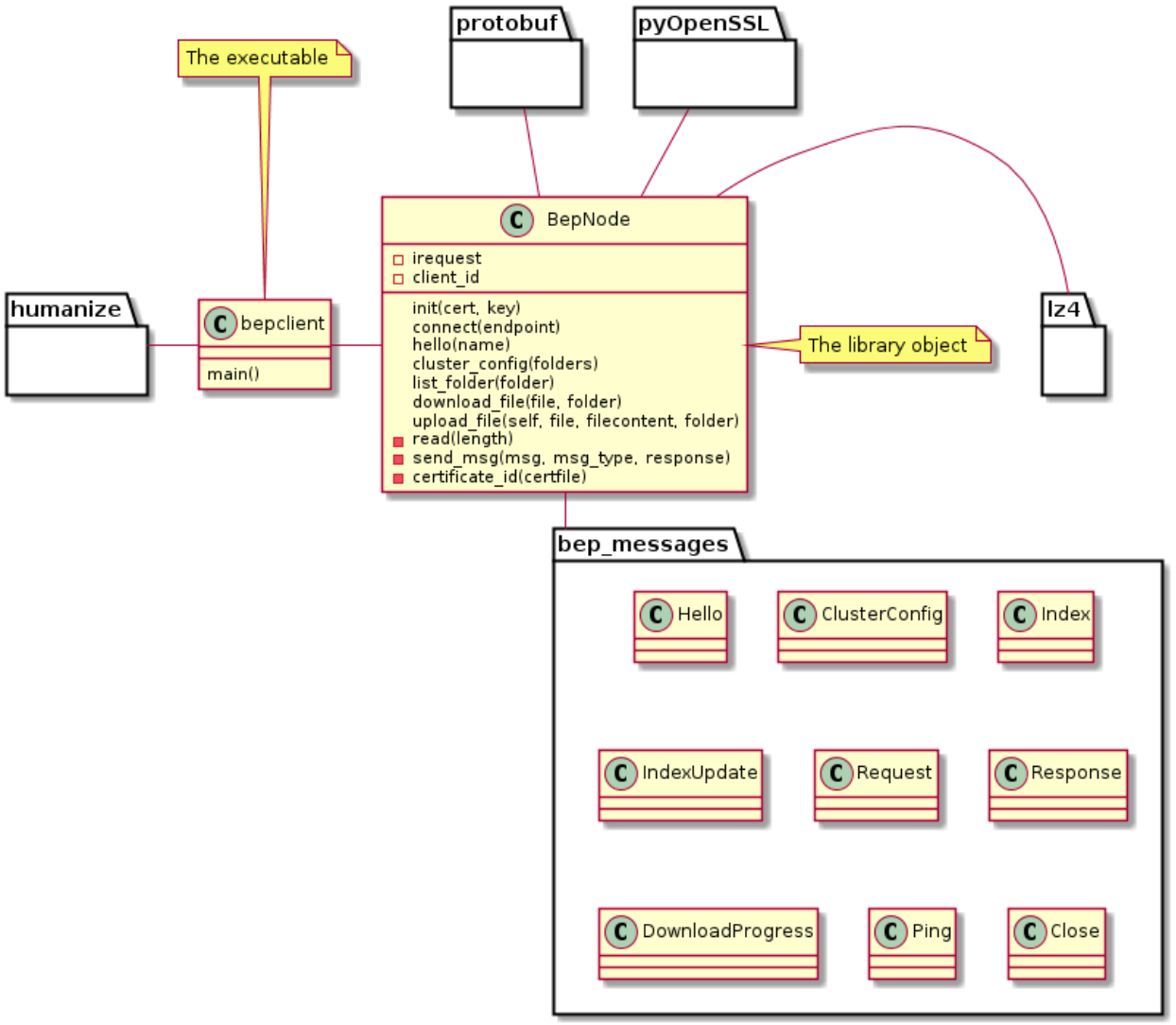


Figure 4: BepClient class diagram

La classe *BepNode* est au coeur de l'implémentation. Elle implémente le protocol BEP et utilise les classes de message générées depuis la définition protobuf. Cette classe utilise des packages tiers pour la serialization, connection SSL et décompression. Ceci est visible dans la figure 4.

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

3.3 Utilisation

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

```
$> ./bepclient.py -h
Bep client, can be used to download files from a BEP node.

Usage:
  bepclient.py [options] (showid | connect <host> [share <share_id> [download <remotefile> <localfile>]]

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 /tmp/plistlib.py
  bepclient.py -h | --help

Options:
  --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].
  -h --help          Show this screen.
```

3.3.1 Exemples

Montrer l'id du certificat utilisé:

```
$> bepclient.py showid
Client id: HG3DI2F-JKKVY3Z-HL5ZCWN-FH53M35-CMGFGE5-WAPGTV6-5SBWC6W-4VZSFA
```

Montrer les folders d'un noeud pair:

```
$> ./bepclient.py connect 129.194.186.177
Connected to: redbox
Shared folders: 3
        -  (facile)
        -  (hyperfacile)
        -  (moins_facile)
```

Montrer les fichiers d'un folder:

```
$> ./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:

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

Références

Block Exchange Protocol v1. 2017. “Block Exchange Protocol V1.” https://docs.syncthing.net/specs/bep-v1.html.

Syncthing. 2017. “Syncthing.” https://syncthing.net.