Université de Strasbourg
UFR Mathématiques et Informatique

Licence 3 Informatique
Année 2016/2017

Réseaux et Protocoles

Projet : partage de fichiers et table de hachage distribuée

BROUILLON - partie 2 et 3 à finir - version 1

Objectif : appréhender de manière simple ce qu'il se fait dans des outils de la vie quotidienne en recréant par nous-mêmes une version simplifiée.

Le projet se compose de deux parties qui peuvent être faites en parallèle : une bibliothèque de partage de fichiers (torrents) et une bibliothèque de gestion d'une table de hachage distribuée (DHT). Chaque bibliothèque aura son lot de programmes de test, afin de s'assurer du bon développement de chaque partie, sans régression. Les différents programmes ainsi créés devront être documentés (voir section 4).

Modalités

Les exercices sont à faire en groupe de 2 personnes. L'évaluation du travail se fera au dernier TP où vous devrez présenter votre travail.

Protip : un tracker d'une équipe devrait être compatible avec les clients d'une autre. Faites des tests entre groupes pour vérifier votre travail !

1 Partage de fichiers

Le partage de fichiers en réseau consiste en un ensemble d'acteurs et de messages. Les acteurs sont le « client » qui annonce et récupère des fichiers et le « tracker » qui fait la liaison entre les clients en fournissant une liste de clients (leur adresse IP et un port d'écoute) qui sont intéressés par le même fichier.

Les exercices suivants s'appuient sur les documents fournis en annexes section 5. Le contenu des messages est résumé dans le tableau 5.1.

1.1 Échauffement : hachage des fichiers

Il faut pouvoir identifier un fichier de manière unique, pour cela il faut créer une empreinte du fichier. L'algorithme à utiliser pour faire des empreintes est **SHA 256**. Il faut également pouvoir identifier des portions de fichiers, qu'on nommera « **chunks** ». Nous allons donc lire des chunks de 1 Mo et en produire des empreintes.

Le premier programme prend en paramètre un fichier, et créé une empreinte pour le fichier (lecture de tout le fichier) ainsi que pour chaque chunk lu.

```
$ hash-file ./musique.ogg
FILE HASH : 07123e1f482356c415f684407a3b8723e10b2cbbc0b8fcd6282c49d37c9c1abc
CHUNK 1 :   3407a723e10b2cbbc0b8fcd6282c49d37c9c107a3b7123e1f482356c415f6844
CHUNK 2 :   c03437c9c107a3b7123e1f4823507a723e10b2cbbb8fcd628415f6842c49d6c4
...
```

Il faut faire une bibliothèque avec les fonctions principales de hash de fichier et de découpage en chunks pour réutiliser ceci dans les futurs programmes.

1.2 Premier tracker, premier client

Faire deux programmes : premièrement un tracker qui sait juste lister des gens qui se connectent à lui pour annoncer un hash ou pour le récupérer (messages PUT et GET, voir en annexe), et un client. Le client peut annoncer un hash (fictif) et récupérer la liste des IP qui fournissent ce hash.

Premièrement, lancement du tracker.

```
$ ./q1-2_tracker localhost 9000     # lancement du tracker, port 9000
listening on ::1 port 9000...
```

Lancement du client qui annonce un fichier (via son hash) au tracker.

```
# client qui annonce un hash au tracker à l'adresse ::1 port 9000
# écoute des requêtes d'autres clients sur le port 20313

$ ./q1-2_client localhost 9000 20313 put c03437c9c
listening on 20313
put c03437c9c to ::1 port 9000
```

Lancement d'un client qui cherche un fichier, celui-ci va récupérer l'adresse IP et le port utilisés par le premier client (celui qui annonce le fichier).

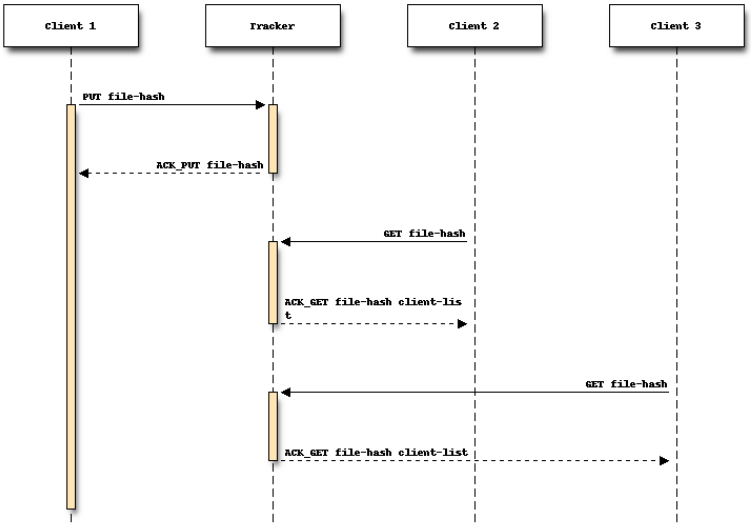
```
# client qui cherche un hash au tracker à l'adresse ::1 port 9000
# écoute des requêtes d'autres clients sur le port 21203

$ ./q1-2_client localhost 9000 21203 get c03437c9c
listening on 21203
get c03437c9c from ::1 port 9000
IP : ::1 port 20313
```

Enfin, on lance un nouveau client qui cherche le même fichier. Celui-ci devra récupérer l'adresse IP et le port utilisés par les deux premiers clients.

```
$ ./q1-2_client localhost 9000 21405 get c03437c9c
listening on 21405
get c03437c9c from ::1 port 9000
IP : ::1 port 20313
IP : ::1 port 21203
```

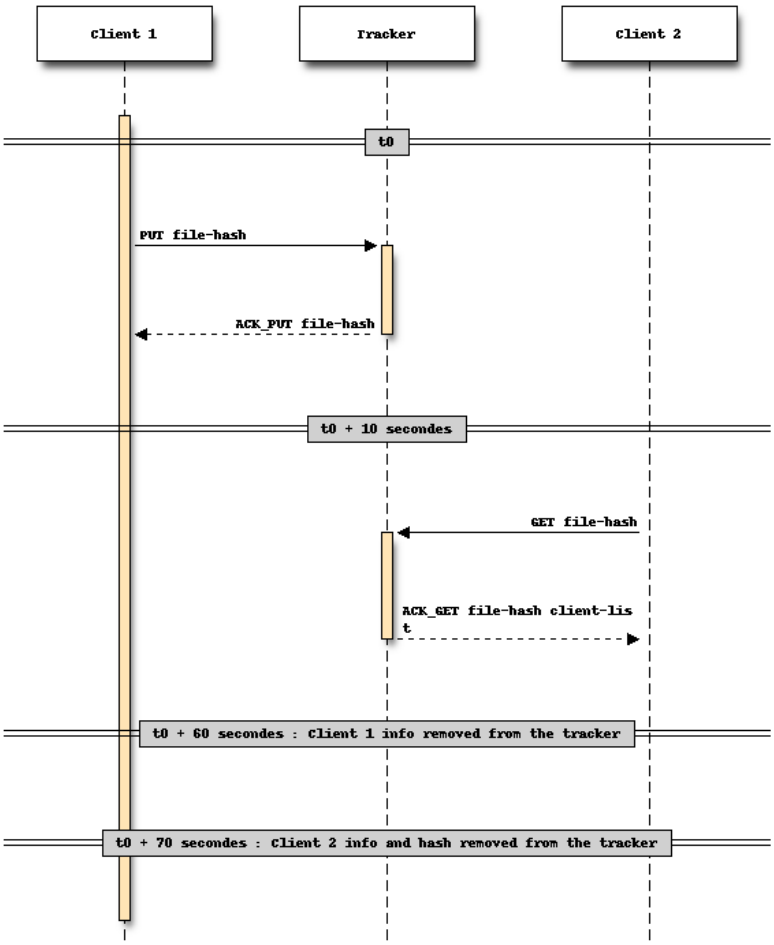
Voici une représentation de l'échange.



1.3 Péremption de la relation hash ⇔ client

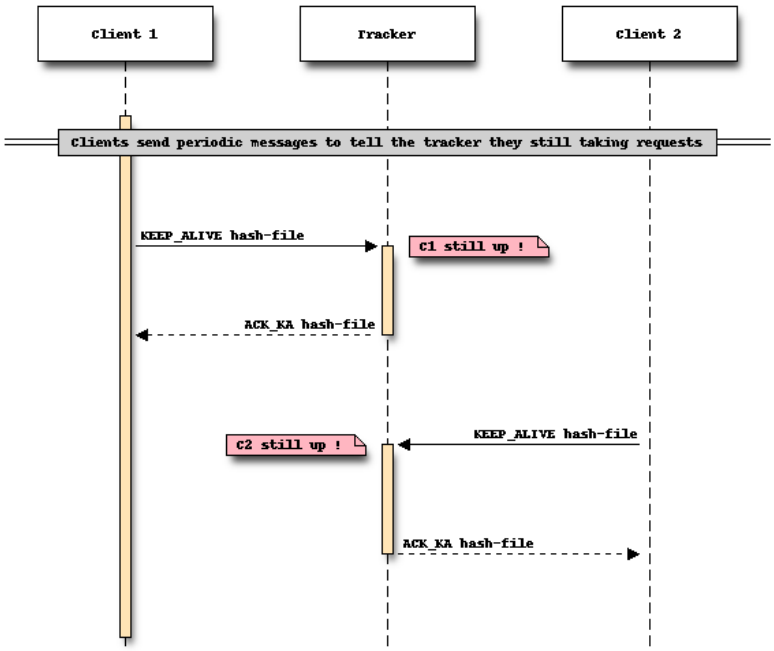
Dans cet exercice, on veut que les enregistrements d'adresses et de ports au niveau du tracker soient temporaires. Au bout d'un certain temps (configurable), il faut qu'on retire des enregistrements. Exemple avec un temps d'expiration de 60 secondes :

1. t0 : on lance un tracker et un client qui s'y connecte pour annoncer un hash, le tracker enregistre ce client
2. t0 + 10 secondes : on lance un client qui veut récupérer le hash, le tracker enregistre ce nouveau client
3. t0 + 60 secondes : le tracker supprime le premier client, le hash n'est désormais associé qu'au second client
4. t0 + 70 secondes : le tracker supprime le second client, le hash n'est plus associé à un client, il est supprimé du tracker



Représentation de ce scénario.

Pour que le tracker garde en mémoire les relations (hash ⇔ client) il faut que les clients envoient périodiquement un



message « KEEP_ALIVE ».

1.4 Communication client à client

Les clients doivent se parler entre eux et s'échanger des données. Pour cet exercice, un client va écouter sur un port et, pour un hash donné, envoyer la liste des chunks de celui-ci aux clients qui le réclament par un message « LIST ».

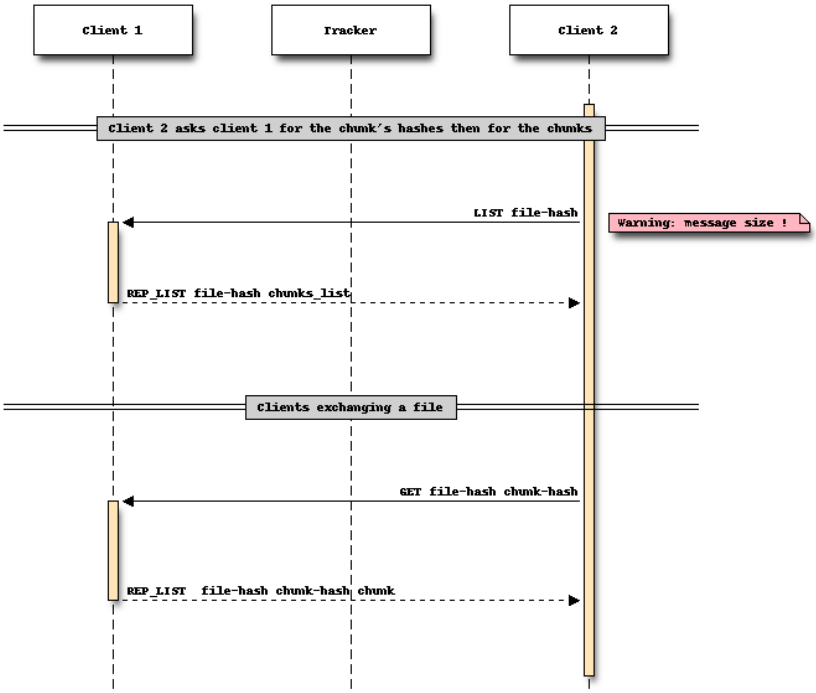
La fragmentation des messages n'est pas gérée en UDP, il faut gérer nous-mêmes la taille des messages. Un chunk représente jusqu'à 1 Mo de données, il faut le découper en fragments de 1 Ko maximum pour l'échange, voir **??**. Chaque message d'échange de données aura un indice du fragment de chunk envoyé, le nombre total de fragments à envoyer, et sera acquitté par un message qui renvoie l'indice du fragment reçu. Nous sommes en UDP, il faut **vérifier chaque chunk arrivé** par rapport à son hash, si le hash ne correspond pas au chunk reçu, on redemande le même chunk.

```
# client qui attend la connexion d'un autre au port 8000

$ ./q1-4_seeder 8000 ./musique.ogg
listening on 8000, file ./musique.ogg, hash ab8ac913f4c35
chunks [ 1cab93f438ac5 af4c35b8ac913 8ac9ab13f4c35 ]

# client qui se connecte à un autre à l'adresse localhost et port 8000
# pour récupérer un fichier dont le hash est ab8ac913f4c35
# le fichier sera enregistré dans "musique-telechargee.ogg"

$ ./q1-4_leecher localhost 8000 ab8ac913f4c35 musique-telechargee.ogg
connection to localhost port 8000, getting ab8ac913f4c35
recorded filename musique-telechargee.ogg
```



Représentation de ce scénario.

1.5 Communication entre plusieurs clients

Même exercice que le précédent, cette fois un client pourra se connecter à une liste de clients et téléchargera en parallèle plusieurs chunks provenant de clients différents. Il faut faire attention à ne pas demander plusieurs fois le même chunk.

Dès qu'un client a téléchargé un chunk, il doit savoir l'envoyer à un client qui lui demanderait. Il faut savoir montrer ce fonctionnement à l'aide d'un exemple.

Point de culture générale : dans les vrais clients Bittorrent, le téléchargement des « chunks » de données se fait par défaut aléatoirement. Cela permet d'être plus résilient à la coupure d'une source (le seeder) : si 5 personnes téléchargent toutes les mêmes premiers chunks depuis la source initiale et qu'il se déconnecte, il n'aura pas eu le temps de partager les derniers chunks.

1.6 Client et tracker complets

Maintenant que nous avons toutes les bases, faites un client et un tracker qui répondent à toutes les problématiques soulevées plus tôt.

Le client doit savoir :

— prendre en paramètre l'adresse et le port du tracker ainsi qu'une commande (« get hash fichier » ou « put fichier »)
— faire un hash du fichier et un hash par chunk
— envoyer le hash du fichier au tracker
— envoyer des messages « KEEP_ALIVE » au tracker pour qu'il garde en mémoire les relations hash ⇔ clients
— demander des morceaux (choisis aléatoirement) de fichiers à plusieurs clients
— répondre aux requêtes de plusieurs clients en parallèle
— bonus : gérer plusieurs fichiers à la fois

1.7 Des pistes d'amélioration

Donnez des pistes d'amélioration de ces programmes qui vont dans le sens de la simplification, de la résistance à une coupure réseau ou un crash d'un des programmes...

Bon courage !

2 Table de hachage distribuée

Une table de hachage distribuée (DHT) est un élément qui permet de s'affranchir d'un unique serveur qui aurait toute l'information et dont tout le monde serait dépendant. Une DHT permet donc de décentraliser une information : une liste de hash avec des valeurs associées. Nous allons en implémenter une version très simple.

Les formats et contenus des messages sont dans le tableau 2.3.

2.1 Avant de la rendre distribuée, faire une table de hachage

Le premier exercice consiste à faire un serveur unique gérant une table de hachage. Pour cela, il doit déjà répondre à deux types de requêtes : **GET** et **PUT**.

Le message PUT permet d'envoyer un hash et une valeur associée, et GET permet de la récupérer.

2.2 Connexions et déconnexions entre tables de hachage

Une table de hachage distribuée est constituée de plusieurs serveurs de table de hachage qui se connectent et se déconnectent entre eux. Il faut donc comprendre les messages **JOIN** et **LEAVE**. Chaque serveur de DHT aura un numéro d'identification (sur un octet), cela nous permettra de tous les identifier facilement. Lorsqu'un serveur se connecte à un autre, on lui envoi la liste des serveurs de la DHT dans l'ordre numérique.

Il faut différencier quel serveur gère quelle partie de la DHT, pour cela nous utiliserons le premier octet du hash et la liste des serveurs. Le premier octet des hash peut être compris entre 0-9 et a-z, ce qui donne 36 possibilités. Le premier serveur s'occupera des hash commençant par 0 jusqu'à $\frac{36}{nb_{serveurs}}$, le second de $\frac{36}{nb_{serveurs}}$ à $\frac{2\times36}{nb_{serveurs}}$ etc.

Explications sur les messages générés (message à tous les serveurs déjà enregistrés lors d'une connexion).

Explications sur les transferts liés à un serveur DHT qui se déconnecte. 1. LEAVE, 2. PUT de tous les hash + valeur

2.3 Keep Alive

Explications identiques aux Keep Alive de la partie Torrent.

Communication entre pairs			
Message	**Type**	**Valeur**	**Signification**
JOIN	200	identifiant	Rejoindre une DHT
ACK_{JOIN}	201	$liste_{serveurs}$	Envoi de la liste des serveurs de la DHT
LEAVE	202	identifiant	Partir d'une DHT
ACK_{LEAVE}	203		
PUT	204	hash + valeur	Ajouter un élément dans la table
ACK_{PUT}	205	hash + valeur	
GET	206	hash	Récupérer un élément de la table (voir 5.1)
KEEP_ALIVE	252	hash + valeur	Garder en mémoire une association hash $\Leftrightarrow$ valeur

FIGURE 1 – Contenu des messages échangés pour la DHT

3 Torrent et table de hachage distribuée

Vous avez un client et un tracker Bittorrent, et votre table de hachage distribuée fonctionne : on va joindre les deux pour faire un tracker décentralisé.

3.1 Tracker qui implémente la DHT

Rien ne change côté client, il se connecte toujours à un tracker pour effectuer des PUT ou des GET. Tout se passe côté Tracker.

Explications sur les interactions entre le tracker et le serveur de DHT.

1. tracker reçoit des messages des clients Bittorrent
2. s'il doit gérer le hash, tout est fait en local
3. sinon, messages de DHT (PUT, GET)

4 Consignes supplémentaires et conseils

Tout d'abord, vos programmes peuvent être vérifiés par Wireshark, utilisez cet outil autant que possible pour être sûr du contenu de vos messages.

TOUS LES PROGRAMMES DOIVENT ÊTRE DOCUMENTÉS. Sur un système d'exploitation, un programme ne doit pas être fourni sans un minimum d'explications. En guise d'entraînement, faites des pages de manuel pour chaque programme qui vous est demandé.

4.1 Écrire une page de manuel

Les pages de manuel de vos programmes comporteront au minimum :
— le nom du programme avec une petite description de son utilité
— le nom de l'auteur de la page de manuel
— la date de création de la page de manuel
— une description claire des différentes options possibles
— le synopsis pour savoir comment utiliser le programme
— les différents numéros de retour
— les erreurs et limitations de vos programmes
— et tout autre élément qui vous semblera pertinent

Faites `man 7 mdoc` pour apprendre à créer des pages de manuel, je vous conseille également de prendre exemple sur les pages de manuel déjà présentes sur votre système (dans `/usr/share/man/man1/` par exemple). Le nom de votre page de manuel sera le nom de votre programme suffixé par « .1 », le numéro « 1 » correspondant à la section dédiée aux manuels d'utilisation d'un programme (`man man` pour vous rafraîchir la mémoire).

4.2 La lisibilité

Le code doit être lisible et commenté. Un code non indenté vaut 0. Un code non commenté vaut 0. Les commentaires doivent être pertinents, et montrer que vous comprenez ce que vous faites. Un saut de ligne de temps en temps ne peut pas faire de mal.

Vous êtes libre d'organiser votre code comme bon vous semble, s'il vous paraît judicieux de séparer le code en plusieurs fichiers pour améliorer la lisibilité ou pour rendre votre code plus modulaire, n'hésitez pas !

Vous avez explicitement le droit de rendre vos sorties d'exécution plus jolies si vous le souhaitez, tant que cela ne nuit pas à la lisibilité du code et que c'est pertinent.

4.3 Trouver de l'aide

Les pages de manuel peuvent vous aider, de même que de nombreux sites. N'utilisez un moteur de recherche que si vous ne trouvez pas dans le manuel, vous en apprendrez plus ! ;)

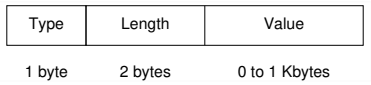
Si vous avez des questions, n'hésitez pas à envoyer un mail ! `p.pittoli CHEZ unistra.fr`

4.4 Le rendu

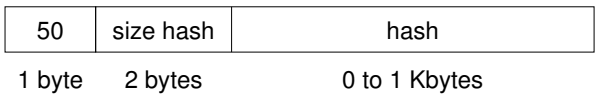
Le rendu doit se faire dans une archive au format `nom1-nom2_projet-rp.tar.gz`.

5 Annexes

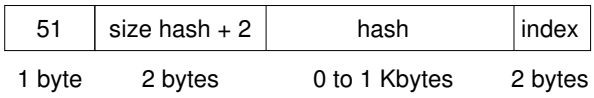
5.1 Formats des échanges de données



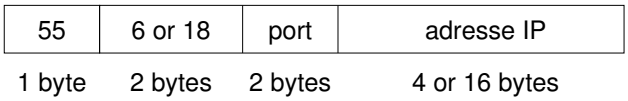
Format Type Length Value, utilisé dans nos exercices



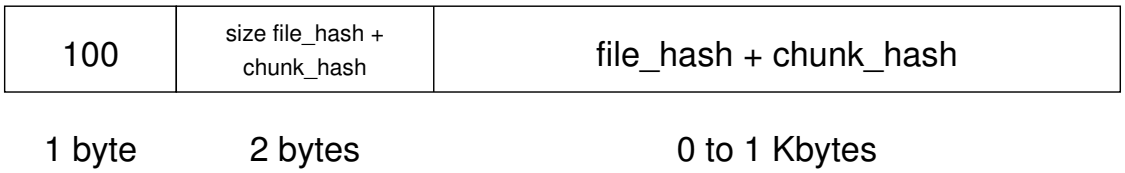
Format d'un « hash de fichier » dans un message



Format d'un « hash de chunk » dans un message



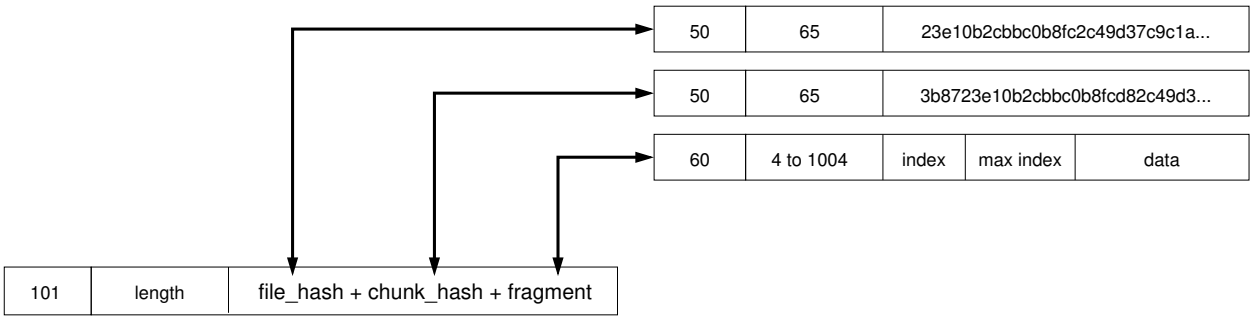
Format d'un client (IP + port) dans un message



Message « GET », client à client, demande d'un chunk

5.2 Diagramme de séquences

La figure 3 représente un scénario valable de fonctionnement de vos programmes.



Message « REP_GET », client à client, envoi d'un chunk

Communication entre pairs			
Message	**Type**	**Valeur**	**Signification**
GET	100	$hash_{fichier}$ + $hash_{chunk}$	Demander un chunk d'un fichier
REP_GET	101	$hash_{fichier}$ + $hash_{chunk}$ + chunk	Récupérer un chunk d'un fichier (voir 5.1)
LIST	102	$hash_{fichier}$	Récupérer la liste des chunks d'un fichier
REP_LIST	103	$hash_{fichier}$ + $liste_{chunks}$	Liste des chunks d'un fichier
Communication client $\Leftrightarrow$ tracker			
PUT	110	$hash_{fichier}$ + client	Indiquer qu'on possède un fichier
ACK_{PUT}	111	$hash_{fichier}$ + client	Confirmer l'ajout du client
GET	112	$hash_{fichier}$ + client	Demander la liste des clients qui possèdent ce fichier
ACK_{GET}	113	$hash_{fichier}$ + $liste_{clients}$	Liste des clients qui possèdent ce fichier
KEEP_ALIVE	114	$hash_{fichier}$	Annoncer au tracker qu'on gère toujours ce fichier
ACK_{KEEP_ALIVE}	115	$hash_{fichier}$	Acquittement du message KEEP_ALIVE
Debug sur le tracker			
PRINT	150		Liste les fichiers et les clients associés dans la console

FIGURE 2 – Contenu des messages échangés pour le torrent

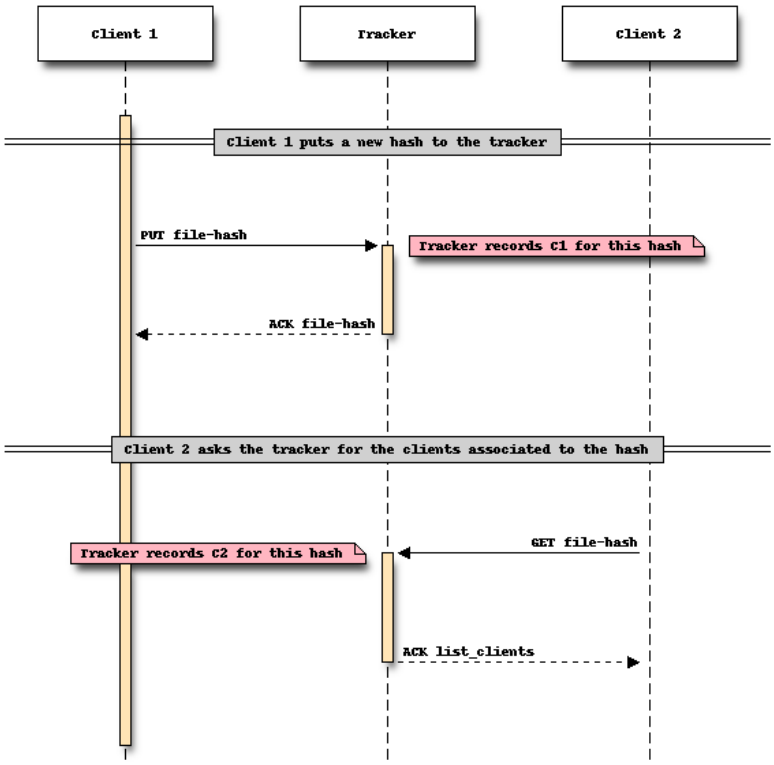


FIGURE 3 – Échange de messages, simple scénario