Tout ce que vous avez toujours voulu savoir sur les clients Java http concurrents, asynchrones, sans oser demander !

Date : 9 avril 2015
Format : Conférence
Speakers : Fred Simon, Chief Architect chez JFrog

Fred travaille sur une librairie qui va bientôt être open-sourcée par JFrog.
JFrog est un éditeur de solutions de repository de binaires (Artifactory et Bintray). Le téléchargement et l'upload de gros fichiers sont vitaux. Ils ont du optimiser cette partie là. Il n'y avait pas de framework open source de concurrent download qui les ont satisfait.

Besoins fonctionnels :

- Téléchargement de fichiers en parallèle
- Téléchargement de fichiers en morceaux et réagencement
- Interruption / suspension / reprise de download
- Pouvoir notifier au client de la progression du download (feedback aux utilisateurs)
- Checksums caching : permet de ne pas télécharger plusieurs fois le fichier avec le même checksum (Gradle est bon à ce niveau)

Outils existants :

- **Java Downloader Manager** (JDM) : applet Java non utilisée par les développeurs dans la salle.
    - Pas de licence ? JFrog ne sait donc pas s'il peut le mettre dans Bintray ou non
    - Pas de site web ni de doc
    - Pas de sources
    - Plus une application qu'une librairie

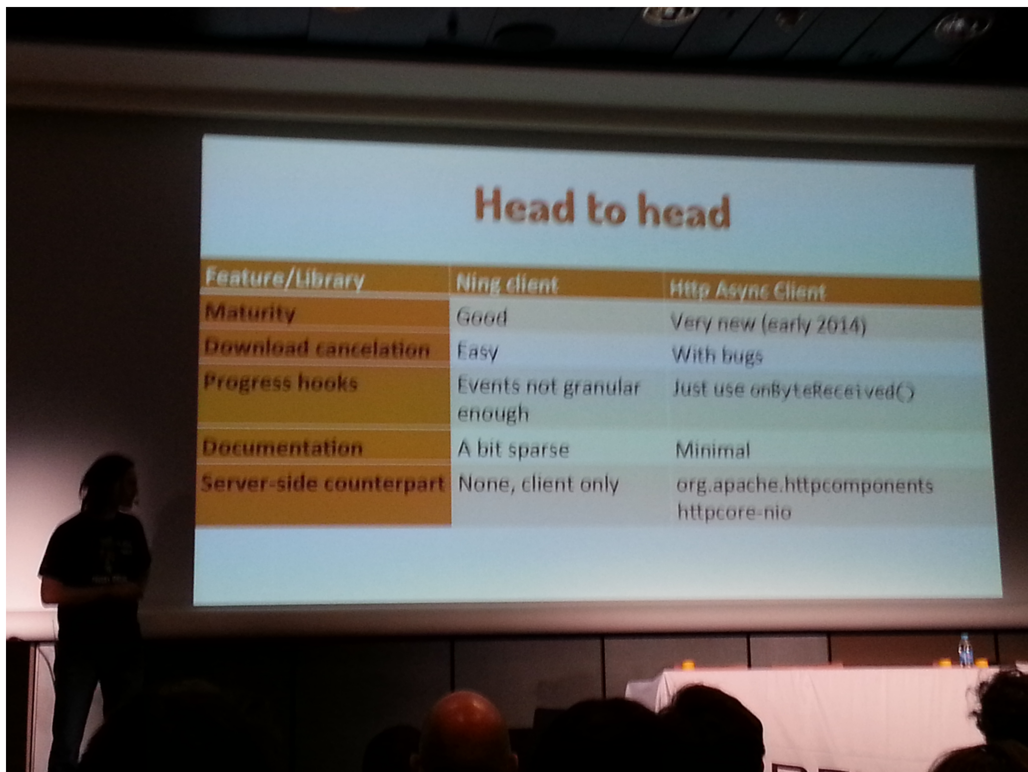
Comment implémenter un tel framework de download ?

Besoins techniques en termes d'I/O asynchrones non bloquantes et d'Event driven
Beaucoup de librairies NIO existent, mais principalement côté serveur.

Fred énumère différentes solutions avec leurs avantages/inconvénients :

- **URLConnection**
  - Fait partie du JDK
  - Problème de buffering mémoire rencontré au début de Maven et d'Ivy (grosse empreinte mémoire)
  - API minimaliste : proxy, SSL, double authentification …
  - Blocking streams
  - Pour ces raisons, la classe URLConnection n'est utilisée dans aucun outil développé par JFrog
- **Reactor avec RxJava** : un peu trop gourmand et lourd à mettre en place. Beaucoup de callbacks à mettre dans son code.
  - Pattern avec Concret Event Handler : les clients délèguent la lecture de fichiers au Reactor qui dispatchent
  - Basé sur NIO. Exemple de la boucle infinie du main thread qui sélectionne les événements et les dispatche.
  - Problème principal de RxJava :
    - Nécessite de maintenir l'état côté client.
- **Grizzly** : un peu vieux
- **Apache Mina** : dernière release en octobre 2012
- **Netty** : créé à partir de Mina
- **Ning Http Client** : utilisé énormément par Yahoo. JFrog a hésité à le prendre.
- **Apache HTTP Components / AsyncHttpClient** : retenu par JFrog

Tableau comparatif entre Ning et AsyncHttpClient :



JFrog utilise tous les mécanismes proposés par la spécification HTTP 1.1.
Ils ont du faire face à de nombreuses problématiques, notamment celle des redirect HTTP.

Fred termine sa présentation par une suite de questions / réponses :

Question 1 : valeur du content-length d'une réponse lorsqu'on utilise la compression (gzip) ? Non spécifié : au choix taille compressée ou non décompressée. Ou plus exactement la valeur la plus petite.

Question 2 : pourquoi lors d'une redirection vers un CDN (ex : Akamaï), tous les chunks repartent de zéro ? En HTTP, on peut choisir le range (en tête http HttpHeaders.RANGE). Nécessite une recopie du range à chaque fois.

Question 3 : combien de connexions simultanées entre un client et 1 serveur ? 2 connexions en HTTP 1.1. Aucun navigateur ne respecte cette norme (entre 6 et 13)

Question 4 : comment encoder une URL? chaque portion d'une URL doit être encodée différemment : nom de domaine, paramètre …. Ne pas utiliser java.net.URLEncoder. Privilégier les classes URIBuilder et URLEncodedUtils de Apache Commons http Client.

Question 5 : comment fermer correctement la socket ? le client Ruby ne ferme pas correctement ses sockets. Plusieurs techniques : server full close, server output half close, server input half close. Ne jamais attendre indéfiniment que le client ferme son socket (timeout)

Question 6 : comment écrire des fichiers par morceaux de manière concurrente ?

1. Soit écrire dans plusieurs fichiers pour les recombiner à la fin => trop lent
2. Ou soit écrire dans le même fichier à la bonne position => approche utilisée. Ne pas utiliser java.io.RandomAccessFile car il n'est pas asynchrone. Privilégier SeekableByteChannel de java.nio.

Question 7: Comment contourner le file locking ? C'est un problème au niveau de l'OS. Windows ne permet pas une écriture exclusive par plusieurs process (JVM). Deux solutions existent : Verrouillage de tout le fichier au niveau de l'OS ou verrouillage du dernier byte possible du fichier pour une approche optimisée.
La ce que j'ai compris, leur librairie utilise la technique bien connue du renommage .part à la fin du download.

Fred conclue sa présentation par un focus sur HTTP/2 qui s'avère très prometteur :

- Standardisation de Google SPDY
- Header compression
- Server push
- Multi-plexing
- Prioritization
- Content-length standardisé avec la taille compressée