Exemple Java

1. Ajout du jar Kalima dans le projet

Pour tester l'exemple Java, il est nécessaire d'avoir quelques prérequis :

• Environnement de développement intégré IDE comme Eclipse. Lien de téléchargement : https://www.eclipse.org/downloads/packages/

• Un Kit de Développement Java JDK. Il faut au minimum la version 9 du JDK pour que l'exemple fonctionne.

Pour commencer, il faut inclure le jar Kalima.jar dans votre projet. Par exemple, sous Eclipse, placer le jar quelque part dans votre projet, puis ➔ Clic droit sur le jar ➔ Build Path ➔ Add to Build Path.

Vous avez maintenant accès à l'API Kalima dans votre projet.

2. Initialisations

Le code ci-dessous permet d'initialiser un certain nombre d'objet et de se connecter à la blockchain.

```
clonePreferences = new ClonePreferences(args[0]);
logger = clonePreferences.getLoadConfig().getLogger();

node = new Node(clonePreferences.getLoadConfig());
clone = new Clone(clonePreferences, node);

clientCallBack = new KalimaClientCallBack(this);

try {
	node.connect(null, clientCallBack);
} catch (IOException e) {

	logger.log_srvMsg("ExampleClientNode", "Client", Logger.ERR,
	"initComponents initNode failed : " + e.getMessage());
}

for(Map.Entry<String, KCache> entry : clone.getMemCaches().entrySet()) {
	clone.addListnerForUpdate(new
	ChannelCallback(entry.getValue().getCachePath()));
}
```

Le Node va être responsable de la connexion avec la Blokchain.

Le clone est responsable de la synchronisation des données en mémoire cache.

Le clientCallBack permet de réagir à l'ajout de nouvelles transactions dans la Blockchain (cf chapitre suivant).

On peut voir que l'on doit passer args[0] lors de la création de clonePreferences. En effet, vous devez lancer votre client en passant le chemin d'un fichier de configuration. On verra plus tard le contenu de ce fichier.

3. Callbacks

Comme on a pu le voir précédemment, on doit passer deux classes de callbacks à l'objet Node. Le serverCallback n'est pas utile pour un nœud ordinaire. Vous pouvez donc simplement créer une simple classe qui hérite de ServerCallback, sans rien mettre dans les méthodes.

Le ClientCallBack a plus d'importance en revanche, et nécessite quelques lignes obligatoires. Il vous permettra notamment de réagir à l'arrivée de nouvelles transactions. Pour commencer, créer une classe qui hérite de ClientCallback, puis ajouter les méthodes manquantes : Sur Eclipse, clique sur l'erreur (à gauche, à côté des numéros de ligne) ➔ « Add unimplemented methods ».

La fonction putData sera appelée à chaque nouvelle transaction reçue, vous devez au minimum y ajouter le code ci-dessous, et ensuite personnaliser votre code en fonction du comportement souhaité.

```
KMsg kMsg = KMsg.setMessage(msg);
client.clone.set(kMsg.getCachePath(), kMsg, true, false);
```

La fonction onConnectionChanged sera appelée à chaque connexion / déconnexion avec l'un des Notary Nodes. Vous devez à minima y insérer le code ci-dessous, et vous pouvez y ajouter du code en fonction de vos besoins.

```
client.clone.onConnectedChange( (status==Node.CLIENT_STATUS_CONNECTED) ? new
AtomicBoolean(true) : new AtomicBoolean(false), nioClient, false);
```

La fonction onNewCache est appelée à chaque fois qu'un nouveau Cache est créé dans notre Node. Tout les caches seront créés au début de la connexion, lors de la synchronisation. On peut créer des callbacks pour chaque mémoire Cache. Dans cet exemple, un callback a été crée pour gérer les smarts contracts. On souscrit alors à ce callback dans la fonction onNewCache :

```
client.getClone().addListnerForUpdate(new
SmartContractCallback(cachePath, client, contractManager));
```

4. Smarts Contracts (SmartContractCallback)

Les smarts contracts sont stockés sur git mais validés par la Blockchain Kalima. Toute la gestion de ces smarts contracts et intégrée dans l'API Kalima. Pour pouvoir exécuter des smarts contracts depuis notre Node, il suffit de fournir les informations de connexion (identifiant, mot de passe) d'un compte autorisé sur le répertoire git au sont stockés les smarts contracts.

Dans l'exemple, les identifiants sont demandés au démarrage de l'application.

```
contractManager.loadContract(GIT_URL, GIT_USERNAME, password,
kMsg.getKey(), kMsg.getBody());
```

Une fois chargé, un smart contract peut être exécuté :

```
private void runScript(KMsg kMsg) {
            String scriptPath = logger.getBasePath() +
"/git/KalimaContractsTuto" + kMsg.getCachePath() + ".js";
            try {
                  String result = (String)
contractManager.runFunction(scriptPath, "main", logger, kMsg,
client.getClone(), client.getNode());
                  logger.log_srvMsg("ExampleClientNode", "TableCallback",
Logger.INFO, "script result=" + result);
            } catch (Exception e) {
                  logger.log_srvMsg("ExampleClientNode", "TableCallback",
Logger.ERR, e);
            }
      }
```

Dans notre exemple, la fonction runScript est appelé à chaque fois qu'une nouvelle donnée arrive dans le Node. Le node va lancer le script portant le nom du cache path dans lequel on a reçu la donnée, s'il existe. Par exemple, si on reçoit une donnée dans /alarms/fire, le node va lancer le script KalimaContractsTuto/alarms/fire.js.

Les bindings permettent de passer des objets aux scripts. Dans cet exemple, nous passons un KMsg (le message reçu), un Logger, le clone et le node.

5. Fichier de configuration

Voici un exemple de fichier de configuration :

```
LedgerName=KalimaLedger
NODE_NAME=Node Client Example

NotariesList=62.171.131.154:9090,62.171.130.233:9090,62.171.131.157:9090,144.91
.108.243:9090
FILES_PATH=/home/rcs/jit/ClientExample
SerialId=PC1245Tuto
```

- LedgerName ➔N'est pas encore utilisé dans la version actuelle
- NODE_NAME ➔ Vous pouvez mettre quelque chose qui permet de reconnaître votre nœud
- NotariesList ➔ La liste des adresses et ports des notary, séparés par des virgules
- FILES_PATH ➔ C'est le chemin ou seront stockés les fichiers utiles à Kalima, ainsi que les logs
- serialId ➔ C'est un identifiant qui va permettre l'autorisation sur la blockchain au premier lancement du node client (fournis par Kalima Systems dans le cas d'un essais sur nos Notary)

6. Exécution du code

Pour tester votre projet, vous pouvez exécuter le code depuis Eclipse, ou depuis une console en ligne de commande. Il suffit de passer en paramètre, le chemin du fichier de configuration.

Exécution depuis Eclipse :

Dans Run ➔ Run Configurations ➔ Clic droit sur « Java Application » ➔ New Configuration.

- ⇨ Choisissez un nom pour la configuration.
- ⇨ Sous « Project » cliquez sur « Browse » et choisissez votre projet

- ⇨ Sous « Main class » cliquez sur « Search » et sélectionner la classe contenant la méthode Main que vous voulez lancer (ici : Client.java)
- ⇨ Sous l'onglet « Arguments », sous « Program arguments », donner le chemin du fichier de config (ici : etc/cfg/node.config)

Votre configuration est prête, vous pouvez l'exécuter.

Exécution en ligne de commande

Vous pouvez également générer le jar, puis l'exécuter en ligne de commande. Sur Eclipse, faites clic droit sur votre projet ➔ Export, Choisir Java ➔ Runnable Jar File ➔ Next.

Dans la fenêtre « Runnable JAR File Export », choisissez votre configuration sous « Launch Configuration », et choisissez une destination pour votre jar (ex : /Documents/git/KalimaTuto/TutoClient/etc/jar/TutoClient.jar), enfin cliquez sur « Finish ».

Ensuite, depuis la console :

```
cd /Documents/git/KalimaTuto/TutoClient/etc/
java -jar jar/TutoClient.jar cfg/node.config
```

7. Résultats

Le programme d'exemple se connecte à la Blockchain, puis envoi 10 messages (1/seconde). Le TTL (Time To Live) de ces messages est de 10, ce qui signifie que chaque message sera automatiquement supprimé au bout de 10 secondes (une transaction aura lieu sur la blockchain pour chaque suppression). Ainsi, si votre code est correct, que vous avez correctement configuré le fichier de configuration, et que votre appareil est bien autorisé sur la blockchain, vous devriez avoir quelque chose de similaire dans votre console au bout de 2 secondes :

```
GO
script /home/rcs/jit/KalimaExample/git/KalimaContractsTuto/sensors.js result=OK
putData cachePath=/sensors key=key0 body=21
script /home/rcs/jit/KalimaExample/git/KalimaContractsTuto/sensors.js result=OK
putData cachePath=/sensors key=key1 body=22
script /home/rcs/jit/KalimaExample/git/KalimaContractsTuto/sensors.js result=OK
putData cachePath=/sensors key=key2 body=23
script /home/rcs/jit/KalimaExample/git/KalimaContractsTuto/sensors.js result=OK
putData cachePath=/sensors key=key3 body=24
script /home/rcs/jit/KalimaExample/git/KalimaContractsTuto/sensors.js result=OK
putData cachePath=/sensors key=key4 body=25
script /home/rcs/jit/KalimaExample/git/KalimaContractsTuto/sensors.js result=OK
putData cachePath=/sensors key=key5 body=26
script /home/rcs/jit/KalimaExample/git/KalimaContractsTuto/sensors.js result=OK
putData cachePath=/sensors key=key6 body=27
script /home/rcs/jit/KalimaExample/git/KalimaContractsTuto/sensors.js result=OK
putData cachePath=/sensors key=key7 body=28
script /home/rcs/jit/KalimaExample/git/KalimaContractsTuto/sensors.js result=OK
putData cachePath=/sensors key=key8 body=29
script /home/rcs/jit/KalimaExample/git/KalimaContractsTuto/sensors.js result=OK
putData cachePath=/sensors key=key9 body=30
putData cachePath=/sensors key=key0 body=
putData cachePath=/sensors key=key1 body=
putData cachePath=/sensors key=key2 body=
```

```
putData cachePath=/sensors key=key3 body=
putData cachePath=/sensors key=key4 body=
putData cachePath=/sensors key=key5 body=
putData cachePath=/sensors key=key6 body=
putData cachePath=/sensors key=key7 body=
putData cachePath=/sensors key=key8 body=
putData cachePath=/sensors key=key9 body=
```

Au début le programme se connecte à la blockchain et une demande de snapshot est faite, ce qui permet à notre client de recevoir les données qu'il est autorisé à recevoir. Cela se fait relativement vite. Dans la classe principale Client.java, le programme est mis en attente pendant 2 secondes.

On affiche alors le message « Go ».

Ensuite, le client va envoyer 10 messages en 10 secondes. Les messages seront reçus par tous les nodes autorisés sur la cache path en question, dont le vôtre. Ainsi, vous devez voir dans les logs une ligne pour chaque message envoyé (lignes commençant par « `StoreLocal` »). Pour chaque message reçu dans /sensors, on lance le script reverse_string. Si vous avez répondez « Y » au début du programme, et que vous vous êtes correctement identifié sur git ensuite, vous devez voir dans les logs le route du script, qui affiche le body à l'envers (ex : 3olleh).

Enfin, les messages seront supprimés un à un, puisque le TTL a été configuré sur 10 secondes. Vous devez donc voir les transactions dans les logs (lignes commençant par « `StoreLocal remove` »).

S'il ne se passe rien après le « Go » il y'a plusieurs possibilités :

- Vous n'êtes pas autorisé sur la blockchain
- Vous avez fait une erreur dans le fichier de config
- Vous n'êtes pas connecté à Internet