Exemple Java

Table des matières

1. Prérequis pour l'installation

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 est conseillé d'avoir au minimum la version 11 du jdk pour que l'exemple fonctionne correctement.

Pour commencer, vérifiez que le jar Kalima.jar est bien inclus dans votre projet (dans le dossier /libs). Ensuite, faîtes Clic droit sur le jar ➔ Build Path ➔ Add to Build Path. Cela devrait déjà être fait, mais verifies.

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

2. Explication du code

a. Initialisations

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

```
public Client(String[] args) {
	clonePreferences = new ClonePreferences(args[0]);
	logger = clonePreferences.getLoadConfig().getLogger();
}


public void initComponents(String[] args){
	node = new Node(clonePreferences.getLoadConfig());
	clone = new Clone(clonePreferences, node);

	clientCallBack = new KalimaClientCallBack(this, gitUser, gitPassword);

	try {
		node.connect(null, clientCallBack);
	} catch (IOException e) {
		logger.log_srvMsg("ExampleClientNode", "Client", Logger.ERR,
"initComponents initNode failed : " + e.getMessage());
	}
}
```

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.

Enfin, le properties va nous permettre de récupérer des informations depuis le fichier config. Nous verrons plus tard des précisions en expliquant le fichier config.

b. 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. Dans le code ici, nous avons donc une classe qui hérite de ClientCallback. Les méthodes qui ne nous intéressent pas sont laissées vides. Les fonctions qui nous intéressent pour cet exemple sont putData, onConnectionChanged et onNewCache.

La fonction putData sera appelée à chaque nouvelle transaction reçue. Il doit donc y avoir au minimum le code ci-dessous, et ensuite ajouter des détails selon ce que vous souhaitez réaliser.

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

La fonction onConnectionChanged sera appelée à chaque connexion / déconnexion avec l'un des Notary Nodes. Il doit donc y avoir au minimum le code ci-dessous, et ensuite ajouter des détails selon ce que vous souhaitez réaliser.

```
client.getClone().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. Tous 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, gitUser, gitPassword));
```

c. Smarts Contracts (SmartContractCallback)

Les smarts contracts sont stockés sur git mais validés par la Blockchain Kalima. Toute la gestion de ces smarts contracts est 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 où sont stockés les smarts contracts.

Si vous souhaitez ajouter votre propre smart contract, il vous faudra alors nous contacter pour que nous puissions l'ajouter sur le git privé validé par la BlockChain Kalima.

Dans l'exemple, les identifiants sont demandés au démarrage de l'application et on renseigne ces informations avec le code ci-dessous.

```
contractManager.loadContract(GIT_SERVER + GIT_HOST, gitUser, gitPassword,
kMsg.getKey(), kMsg.getBody());
```

Une fois chargé, un smart contract peut être exécuté :

```
private void runContract(KMsg kMsg) {
	if(client.getNode().getSyncingKCaches().get(kMsg.getCachePath()))
		return;
	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);
		System.out.println("script " + scriptPath + " result=" + result);
	} catch (Exception e) {
		logger.log_srvMsg("ExampleClientNode", "TableCallback",
Logger.ERR, e);
	}
}
```

Dans notre exemple, la fonction runContract 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.

3. Fichier de configuration

Voici un exemple de fichier de configuration :

```
LedgerName=KalimaLedger
NODE_NAME=Node Exemple

NotariesList=167.86.124.188:9090,62.171.130.233:9090,62.171.131.157:9090,144.91
.108.243:9090
FILES_PATH=/home/rcs/jit/KalimaExample
SerialId=PCTuto
#WATCHDOG=600000
SEND_TIMEOUT=10000
```

- 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)

Pour chaque appareil, il faut changer le serialId. Il est donc très important de le vérifier lorsque vous lancez sur un nouvel appareil.

- WATCHDOG ➔ Le watchdog ici est commenté, il ne nous est pas utile pour notre exemple. Le watchdog est utile en cas de connexion prolongée avec la Blockchain. Il correspond à un temps (ici, 600000 = 600 secondes = 10 minutes). En fait toutes les 10 minutes, on va vérifier que la connexion est toujours active. On peut ensuite par exemple faire un smart contract qui nous enverra un mail si la connexion est perdue. C'est donc utile pour vérifier l'état du réseau.
- `SEND_TIMEOUT` ➔Correspond au temps maximal considéré pour l'envoi d'un message. Si au bout du temps indiqué, le message n'est pas bien reçu, on réessaye d'envoyer (ici 10 secondes).

4. 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.

a. 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)

Il y a de fortes chances que la seule chose que vous ayez à modifier est l'argument. Tout le reste devrait déjà être fait au lancement. Si ce n'est pas le cas, suivez toutes les instructions.

Votre configuration est maintenant prête, vous pouvez l'exécuter.

b. 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/KalimaJavaExample/etc/jar/TutoClient.jar), enfin cliquez sur « Finish ».

Ensuite, depuis la console :

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

5. Résultats

Lorsque l'on exécute le programme, la première chose que l'on voit est :

```
Do you want use Smart Contracts ? (Y/n)
```

En fonction du choix, vous allez choisir si vous exécutez ou non un Smart Contract.

Une fois ce choix fait, vous aurez le menu suivant qui s'affichera :

```
What example do you want to use?
 - Send 10 messages to /sensors (with ttl 10): 1
 - Send/Delete 1 message of your choice: 2
```

Nous vous proposons pour l'instant 2 exemples possibles, en fonctions de votre choix (1 ou 2), voici ce que vous aurez :

a. Exemple 1

Le programme d'exemple 1 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 :

Si au moment du choix de l'utilisation du Smart Contract, vous avez choisi « n » :

```
putData cachePath=/alarms/fire key=BEGINMEMCACHE body=/alarms/fire
[…]
putData cachePath=/Kalima_Password_json key=ENDMEMCACHE
body=/Kalima_Password_json
GO
putData cachePath=/sensors key=key0 body=95
putData cachePath=/sensors key=key1 body=96
putData cachePath=/sensors key=key2 body=97
putData cachePath=/sensors key=key3 body=98
putData cachePath=/sensors key=key4 body=99
putData cachePath=/sensors key=key5 body=100
putData cachePath=/sensors key=key6 body=101
putData cachePath=/sensors key=key7 body=102
putData cachePath=/sensors key=key8 body=103
putData cachePath=/sensors key=key9 body=104
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=
```

Si vous avez choisi « y », vous aurez :

```
putData cachePath=/alarms/fire key=BEGINMEMCACHE body=/alarms/fire
[…]
putData cachePath=/Kalima_Password_json key=ENDMEMCACHE
body=/Kalima_Password_json
GO
script /home/rcs/jit/KalimaExample/git/KalimaContractsTuto/sensors.js result=OK
putData cachePath=/sensors key=key0 body=95
script /home/rcs/jit/KalimaExample/git/KalimaContractsTuto/sensors.js result=OK
putData cachePath=/sensors key=key1 body=96
script /home/rcs/jit/KalimaExample/git/KalimaContractsTuto/sensors.js result=OK
putData cachePath=/sensors key=key2 body=97
script /home/rcs/jit/KalimaExample/git/KalimaContractsTuto/sensors.js result=OK
putData cachePath=/sensors key=key3 body=98
script /home/rcs/jit/KalimaExample/git/KalimaContractsTuto/sensors.js result=OK
putData cachePath=/sensors key=key4 body=99
script /home/rcs/jit/KalimaExample/git/KalimaContractsTuto/sensors.js result=OK
putData cachePath=/sensors key=key5 body=100
script /home/rcs/jit/KalimaExample/git/KalimaContractsTuto/sensors.js result=OK
putData cachePath=/sensors key=key6 body=101
script /home/rcs/jit/KalimaExample/git/KalimaContractsTuto/sensors.js result=OK
putData cachePath=/sensors key=key7 body=102
script /home/rcs/jit/KalimaExample/git/KalimaContractsTuto/sensors.js result=OK
putData cachePath=/sensors key=key8 body=103
script /home/rcs/jit/KalimaExample/git/KalimaContractsTuto/sensors.js result=OK
putData cachePath=/sensors key=key9 body=104
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=
```

Le programme se lance réellement une fois que « Go » s'affiche.

Dans notre exemple, 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 sensors.js. Si vous avez répondez « Y » au début du programme, et que vous vous êtes correctement identifié sur git ensuite, vous devriez voir dans les logs les résultats du script.

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` »).

b. Exemple 2

L'exemple 2 vous permettra d'envoyer ou de supprimer un message sur un cache path. Ici nous choisissons, tous les paramètres qui composent la commande d'envoi de message vers la BlockChain. Ainsi, vous devrez choisir vous-même le cache path, la clé et la valeur du message que vous souhaitez. Ainsi, après le Go vous aurez :

```
Do you want to add (a) or delete (d)?
a
Type the cachePath you want to interact with :
/sensors
Type the key of your choice :
test
Type the value of your choice :
hello
[key : test / value : hello] added to cachePath : /sensors
```

L'exemple ci-dessus enverra la clé « test » avec la valeur « hello » sur le cache path « /sensors » de la Blockchain. Cela correspond ici à un ajout. Si on décide de supprimer un élément à la place, les étapes seront les mêmes mais la valeur ne sera pas demandée (on supprime en fonction de la clé, la valeur n'est pas importante).

À noter que si l'on ajoute une clé qui existe déjà avec une valeur différente, cela remplacera l'ancienne clé.

6. Erreurs possibles

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
- Il arrive que votre device ne soit plus accepté par la Blockchain. Il faut donc le refaire valider.