API Java Kalima

Prérequis

Pour utiliser l'API Java Kalima, il est recommandé d'avoir préalablement lu la documentation API_Kalima et d'avoir installé le JDK java dans sa version 11 au minimum.

Configuration

L'api Kalima est fournie sous la forme d'un JAR, il en existe deux versions :

- Kalima.jar ➔ Contient les éléments nécessaires pour créer un nœud Kalima capable de se connecter sur une Blockchain Kalima, d'effectuer des transactions sur la blockchain, et de réagir de manière évènementielle à la création de nouvelles transactions (cela permet donc de créer des smarts contrats Java)
- KalimaSC.jar ➔ Inclue en plus le ContractManager qui offre la possibilité de lancer des smarts contrats Javascripts

Pour utiliser l'api Kalima dans votre projet, il vous suffit donc d'inclure le jar de votre choix dans vos dépendances et n'oubliez pas non plus d'ajouter le chemin de votre fichier de configuration (dans notre cas, etc/cfg/node.config) aux variables de configuration d'exécution.

Implémentation

Connexion à une blockchain Kalima

Dans cette section nous allons voir comment mettre en place le minimum nécessaire pour créer un nœud java et le connecter à une blockchain Kalima.

Fichier de configuration

Pour initialiser un nœud Kalima, il faut lui fournir quelques informations qui sont chargées depuis un fichier de configuration, exemple :

```
SERVER_PORT=9100
NotariesList=167.86.103.31:8080,5.189.168.49:8080,173.212.229.88:8080,62.171.153.36:8080,167.86.124.188:8080
FILES_PATH=/home/rcs/jit/KalimaJavaExample
# CHANGE IT WITH THE SERIALID GRANTED AFTER ADMINISTRATIVE VALIDATION
SerialId=JavaExample
```

- SERVER_PORT : Chaque nœud Kalima est composé de plusieurs « clients », et d'un « serveur ». Même si dans la majorité des cas, la partie serveur ne sera pas utilisée pour un nœud client Kalima, on doit spécifier un port pour le serveur. On peut choisir le port que l'on veut, en prenant soin de choisir un port qui n'est pas déjà utilisé sur la machine.

- NotariesList : Ce paramètre permet de définir la liste des Notary Nodes sur lesquels on veut connecter notre nœud. La liste ci-dessus permet de se connecter à la blockchain dédiée aux tutoriels.
- FILES_PATH : Indique le dossier dans lequel seront stockés les fichiers nécessaires au fonctionnement du nœud. On y trouvera notamment les logs de l'application.
- SerialId : Pour que la connexion aboutisse, votre nœud doit être autorisé sur la blockchain. Pour initialiser la connexion, un administrateur Kalima doit vous créer une autorisation temporaire (valable 5 minutes). Cette autorisation temporaire se fait par le biais du SerialId. On peut autoriser un nœud sur une liste d'adresses, en lecture ou en écriture. Le nœud aura donc accès aux transactions de toutes les adresses sur lesquelles il est autorisé en lecture ou en écriture, en revanche il pourra créer de nouvelles transactions uniquement sur les adresses sur lesquelles il est autorisé en écriture et pour obtenir un serialId, merci de contacter un de nos administrateurs (jerome.delaire@kalima.io, tristan.souillard@kalima.io)

Clone

Pour créer un nœud Kalima, il suffit de créer un Clone et d'appeler la fonction connect :

```
ClonePreferences clonePreferences = new ClonePreferences(configFilePath);
clone = new Clone(clonePreferences);
clone.connect();
```

ClonePreferences permet de charger le fichier de configuration du nœud. Son constructeur prend un paramètre de type String, qui correspond au chemin du fichier de configuration.

Clone va initialiser tous les composants nécessaires au nœud Kalima, et permettra par la suite d'accéder aux données, de créer des transactions, etc. Son constructeur prend un seul paramètre : Le ClonePreferences.

La fonction connect permet de connecter le nœud à la blockchain. Nous verrons par la suite que l'on peut lui passer un paramètre.

Créer une transaction

Pour créer une transaction, il faut à minima renseigner deux paramètres :

- L'adresse : L'adresse sur laquelle la transaction sera créé. L'adresse peut par exemple correspondre à un wallet. Dans nos tutoriaux, les adresses sont volontairement représentées comme des chemins que l'on pourrait avoir dans un système de fichiers (exemple : /alarms/fire).
- La clé : La clé unique de la transaction

Une transaction est bien entendu immuable, mais comme expliqué la documentation Kalima_API, il faut bien distinguer les transactions des valeurs courantes dans Kalima. Si pour une adresse donnée, on a une valeur courante avec la clé « temperature », cette valeur courante peut être remplacé si on crée une nouvelle transaction avec la même clé, et peut être supprimé si on crée une transaction de suppression (body vide) avec cette même clé.

Il existe ensuite plusieurs fonctions dans **l'objet Clone** pour créer des transactions :

```
boolean put(String address, String key, byte[] body)
boolean put(String address, String key, byte[] body, int ttl)
boolean remove(String address, String key)
```

Le body correspond au contenu de la transaction. La transaction peut être sous n'importe quelle forme (Chaine de caractères, JSON…), il suffit de la convertir en tableau de bytes par la suite.

Le TTL (Time To Live) correspond à la durée de vie de la transaction en secondes. Si le TTL vaut 10 par exemple, une transaction de suppression sera automatiquement créée sur la blockchain sur cette adresse et pour cette clé.

Les deux fonctions put permettent de créer des transactions dites d'ajout (ou d'édition) et la fonction remove permet de créer une transaction de suppression (c'est pourquoi elle ne prend pas de body en paramètre).

Exemples

Envoi d'une température à l'adresse /sensors :

```
clone.put(
   "/sensors",
   "temperature",
   String.valueOf(temperature).getBytes()
);
```

Suppression de cette température :

```
clone.remove(
   "/sensors",
   "temperature"
);
```

Récupérer les valeurs courantes

Récupérer une valeur precise

Il est possible de récupérer une valeur courante sur une adresse en connaissant sa clé unique, via la fonction **get** de l'objet Clone. Cette fonction retourne un objet de type KMsg, via lequel on peut retrouver l'adresse, la clé, le contenu de la transaction par exemple.

Récupérer toutes les valeurs sur une adresse

On peut également parcourir toutes les valeurs sur une adresse, en parcourant la mémoire cache correspondante à cette adresse. Pour cela on peut récupérer la mémoire cache via la fonction **getMemCache** de l'objet Clone.

Exemples

Pour afficher le contenu sous forme de string de la température précédemment créée :

```
KMsg kMsg = clone.get("/sensors", "temperature");
if(kMsg != null) {
        System.out.println(new String(kMsg.getBody()));
} else {
        System.out.println("/sensors/temperature not found");
}
```

Pour afficher tout le contenu de l'adresse /sensors :

```
MemCache memCache = (MemCache) clone.getMemCache("/sensors");
if(memCache == null) {
        System.out.println("Address /sensors not found");
        return;
}

for(Map.Entry<String, KMessage> entry : memCache.getKvmap().entrySet()) {
        KMsg kMsg = KMsg.setMessage(entry.getValue());
        System.out.println("KEY=" + entry.getKey() + " BODY=" + new String(kMsg.getBody()));
}
```

Réagir à des événements

Il existe certains callbacks que l'on peut implémenter pour réagir à certains événements comme la création d'un nouveau cache ou encore l'arrivée de nouvelles transactions.

Pour cela il existe deux interfaces que l'on peut implémenter :

- ClientCallback
- MemCacheCallback

ClientCallback

L'interface ClientCallback contient plusieurs callbacks que l'on peut implémenter :

- onNewVersion : Ce callback est appelé lorsqu'un changement de version de la blockchain est détecté. Ce callback peut permettre par exemple d'automatiser la mise à jour des nœuds lors d'une mise à jour conséquente de la blockchain.
- onNewCache : Ce callback est appelé lorsqu'une nouvelle mémoire cache est créée. Chaque nœud est autorisé sur une liste d'adresse. Au démarrage, le nœud va se connecter à la blockchain, mettre en place une communication cryptée avec celle-ci, puis se synchroniser
- onCacheSynchronized : Ce callback est appelé lorsqu'un cache est synchronisé, c'est-à-dire lorsque les valeurs courantes d'une adresse ont été reçue au démarrage du node. Cela permet d'attendre certaines données avant de lancer certaines actions par exemple.
- onReject : Ce callback est appelé lorsque votre nœud tente de se connecter à la blockchain mais n'a pas d'autorisation. Il permet par exemple d'afficher un message pour prévenir l'utilisateur.

MemCacheCallback

On peut ajouter un MemCache callback par adresse pour réagir à l'arrivée de nouvelles transactions.

L'interface MemCacheCallback contient les callbacks suivants :

- getAddress : Doit retourner l'adresse correspondante
- putData : Ce callback est appelé à l'arrivé de nouvelle donnée en cache (ajout, mise à jour d'une valeur courante)
- removeData : Ce callback est appelé lorsqu'une valeur courante est supprimée

Exemple

Pour exemple, nous avons les adresses suivantes :

- /sensors : Contiendra des valeurs de capteurs, comme des températures par exemple
- /alarms/fire : Contiendra des alarmes incendies

On souhaite qu'une alarme incendie soie crée lorsqu'une température dépasse 100°C.

On souhaite également afficher dans la console l'arrivée de nouvelles températures ainsi que l'arrivée d'alarmes incendie.

Lorsque l'on relance notre node, on ne souhaite pas que les données déjà présentes soit traités à nouveau, pour éviter les doublons d'alarmes.

On commence par implémenter deux MemCacheCallbacks différents (un pour l'adresse /sensors, un pour l'adresse /alarms/fire) :

```
public class SensorsCallBack implements MemCacheCallback{

	private String address;
	private Clone clone;

	public SensorsCallBack(String address, Clone clone) {
		this.address = address;
		this.clone = clone;
	}

	@Override
	public void putData(KMessage kMessage) {
		KMsg msg = KMsg.setMessage(kMessage);
		String key = msg.getKey();
		System.out.println("new sensor value key=" + key + " body=" +
new String(msg.getBody()));
		if(key.equals("temperature")) {
			int temperature = Integer.parseInt(new
String(msg.getBody()));
			if(temperature >= 100) {
				clone.put("/alarms/fire", "temperature",
("Temperature too high: " + temperature + " °C").getBytes());
			}
		}
	}

	@Override
	public void removeData(KMessage kMessage) {
```

```
	}

	@Override
	public String getAddress() {
		return address;
	}
}
```

```
public class AlarmsCallback implements MemCacheCallback {

	private String address;

	public AlarmsCallback(String address) {
		this.address = address;
	}

	@Override
	public String getAddress() {
		return address;
	}

	@Override
	public void putData(KMessage kMessage) {
		KMsg kMsg = KMsg.setMessage(kMessage);
		System.out.println("new alarm key=" + kMsg.getKey() + " body="
+ new String(kMsg.getBody()));
	}

	@Override
	public void removeData(KMessage kMessage) {

	}
}
```

Comme on peut le voir, nos deux implémentations sont relativement simples. SensorsCallback se contente d'afficher le body des messages reçues, puis, si la clé de la donnée reçue est « temperature », il va contrôler la température, et créer une nouvelle transaction à l'adresse /alarms/fire si la température est supérieure ou égale a 100°C.

AlarmsCallback se contente d'afficher le body des données reçues.

Ces callbacks doivent être passés au clone via la fonction **addMemCacheCallback**. Cependant, il faut que les caches correspondants existent déjà dans le clone.

On va donc implémenter un ClientCallback. On peut alors instancier SensorsCallback et AlarmsCallback dans onNewCache ou bien dans onCacheSynchronized. Dans notre cas, on prendra la seconde option, car on ne veut pas traiter les données déjà présentes lorsque l'on démarre notre nœud :

```
public class KalimaClientCallback implements ClientCallback {

	private Clone clone;

	public KalimaClientCallback(Clone clone) {
		this.clone = clone;
	}

	@Override
	public void onCacheSynchronized(String address) {
		if(address.equals("/sensors")) {
			clone.addMemCacheCallback(new SensorsCallBack(address,
clone));
		} else if(address.equals("/alarms/fire")) {
			clone.addMemCacheCallback(new AlarmsCallback(address));
		}
	}

	@Override
	public void onNewCache(String address) {}

	@Override
	public void onNewVersion(int majver, int minver) {}

	@Override
	public void putRequestData(SocketChannel socketChannel, KMessage
kMessage) {}

	@Override
	public void onReject(SocketChannel arg0) {
		System.out.println("You are not authorized on this
Blockchain.");
		System.out.println("Please contact an administrator");
		System.exit(-1);
	}

}
```

Il ne reste plus qu'à passer notre ClientCallback au clone. Cela se fait par le biais de la fonction connect :

```
clone.connect(new KalimaClientCallback(clone));
```