Kalima C# API

Prerequisite

To use the Kalima C# API, it is recommended that you have previously read the documentation API_Kalima.

Configuration

The Kalima C# API is provided as a DLL.

To use the Kalima api in your project, simply include la dll in your dependencies.

Implementation

Connecting to a Kalima blockchain

In this section we will see how to set up the minimum necessary to create a C# node and connect it to a Kalima blockchain.

Configuration file

To initialize a Kalima node, you must provide it with some information that is loaded from a configuration file, for example:

```
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/KalimaCSharpExample

# CHANGE IT

SerialId=CSharpExample
```

- SERVER_PORT: Each Kalima node is composed of several "clients", and a "server". Although in the majority of cases, the server part will not be used for a Kalima client node, a port for the server must be specified. You can choose the port you want, taking care to choose a port that is not already used on the machine.
- NotariesList: This parameter allows you to define the list of Master Nodes on which we want to connect our node. The list above allows you to connect to the blockchain dedicated to tutorials.
- FILES_PATH: Specifies the folder in which the files necessary for the operation of the node will be stored. In particular, you will find the logs of the application.
- SerialId: For the connection to succeed, your node must be authorized on the blockchain. To initiate the connection, a Kalima administrator must create a temporary authorization (valid for 5 minutes). This temporary authorization is done through the SerialId. One can allow a node on a list of addresses, read or write. The node will therefore have access to transactions from all addresses on which it is allowed to read or write, but it will be able to create new transactions only on the addresses on which it is authorized to write and to obtain a serialId, please contact one of our administrators (jerome.delaire@kalima.io, tristan.souillard@kalima.io)

Clone

To create a Kalima node, simply create a Clone and call the connect function:

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

ClonePreferences is used to load the node configuration file. Its constructor takes a parameter of type String, which corresponds to the path of the configuration file.

Clone will initialize all the necessary components for the Kalima node, and will subsequently allow access to data, create transactions, etc. Its constructor takes a single parameter: The ClonePreferences.

The connect function is used to connect the node to the blockchain. We will see later that we can pass it a parameter.

Create a transaction

To create a transaction, you must at least fill in two parameters:

- Address: The address will be at which the transaction will be created. For example, the address can correspond to a wallet. In our tutorials, addresses are deliberately represented as paths that one could have in a file system (example: /alarms/fire).
- The key: The unique key of the transaction

A transaction is of course immutable, but as explained in the documentation Kalima_API, it is necessary to distinguish transactions from current values in Kalima. If for a given address, we have a current value with the key "temperature", this current value can be replaced if we create a new transaction with the same key, and can be deleted if we create a deletion transaction (empty body) with this same key.

There are then several functions in the Clone object to create 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)
```

The body corresponds to the content of the transaction. The transaction can be in any form (String, JSON...), just convert it into an array of bytes afterwards.

TTL (Time To Live) is the lifetime of the transaction in seconds. If the TTL is worth 10 for example, a deletion transaction will be automatically created on the blockchain on this address and for this key.

The two put functions allow you to create so-called add (or edit) transactions and the remove function allows you to create a delete transaction (which is why it does not take a body as a parameter).

Examples

Sending a temperature to /sensors:

```
clone. put(
   "/sensors",
   "temperature",
Encoding.ASCII.GetBytes(temperature.toString())
);
```

Removal of this temperature:

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

Retrieve current values

Retrieve a precise value

It is possible to retrieve a current value on an address by knowing its unique key, via the get function of the Clone object. This function returns an object of type KMsg, through which we can find the address, the key, the content of the transaction for example.

Retrieve all values on an address

One can also browse all the values on an address, browsing the corresponding cache memory at that address. To do this, you can retrieve the cache memory via the getMemCache function of the Clone object. You can then browse the cache memory via the navigate function which takes as a parameter an interface that you must implement.

Examples

To display the contents as a string of the previously created temperature:

```
KMsg kMsg = clone. get("/sensors", "temperature");
if(kMsg != null) {
        Console.WriteLine( System.Text.Encoding.Default.GetString(kMsg. getBody()));
} else {
        Console.WriteLine("sensors/temperature not found");
}
```

To view the contents of the /sensors address :

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

memCache.navigate(new PrintKmsg());
```

And the implementation of PrintKmsg:

```
public class PrintKmsg: MemCache. NextKMsg
    {
       public void invoke(KMsg kmsg)
       {
Console. WriteLine("KEY=" + kmsg. getKey() + " BODY=" + System. Text. Encoding.
Default. GetString(kmsg. getBody()));
       }
    }
```

React to events

There are some callbacks that can be implemented to react to certain events such as the creation of a new cache or the arrival of new transactions.

For this there are two interfaces that can be implemented:

- ClientCallback
- MemCacheCallback

ClientCallback

The ClientCallback interface contains several callbacks that can be implemented:

- onNewVersion: This callback is called when a blockchain version change is detected. This callback can make it possible, for example, to automate the update of nodes during a consequent update of the blockchain.
- onNewCache: This callback is called when a new cache is created. Each node is allowed on an address list. At startup, the node will connect to the blockchain, set up encrypted communication with it, and then synchronize
- onCacheSynchronized: This callback is called when a cache is synchronized, that is, when the current values of an address were received when the node started. This makes it possible to wait for certain data before launching certain actions for example.

MemCacheCallback

You can add one MemCache callback per address to react to the arrival of new transactions.

The MemCacheCallback interface contains the following callbacks:

- getAddress: Must return the corresponding address
- putData: This callback is called when new cached data arrives (adding, updating a current value)
- removeData: This callback is called when a current value is deleted

Example

For example, we have the following addresses:

- /sensors: Will contain sensor values, such as temperatures for example
- /alarms/fire: Will contain fire alarms

We want a silk fire alarm to be created when a temperature exceeds 100°C.

We also want to display in the console the arrival of new temperatures as well as the arrival of fire alarms.

When we restart our node, we do not want the data already present to be processed again, to avoid duplicate alarms.

We start by implementing two different MemCacheCallbacks (one for the /sensors address, one for the /alarms/fire address):

```
namespace KalimaCSharpExample
{
    class SensorsCallback : MemCacheCallback
    {

        private String address;
        private Clone clone;

        public SensorsCallback(String address, Clone clone)
        {
            this.address = address;
            this.clone = clone;
        }
        public string getAddress()
        {
```

```
            return address;
        }

        public void putData(KMessage kMessage)
        {
            KMsg msg = KMsg.setMessage(kMessage);
            String key = msg.getKey();
            String body =
System.Text.Encoding.Default.GetString(msg.getBody());
            Console.WriteLine("new sensor value key=" + key + " body=" +
body);
            if(key.Equals("temperature"))
            {
                int temperature = (int) Int64.Parse(body);
                if(temperature >= 100)
                {
                    clone.put("/alarms/fire", "temperature",
Encoding.ASCII.GetBytes("Temperature too high: " + temperature + " °C"));
                }
            }
        }

        public void removeData(KMessage kMessage){}
    }
}
```

```
namespace KalimaCSharpExample
{
    class AlarmsCallback : MemCacheCallback
    {

        private String address;

        public AlarmsCallback(String address)
        {
            this.address = address;
        }
        public string getAddress()
        {
            return address;
```

```
        }

        public void putData(KMessage kMessage)
        {
            KMsg kMsg = KMsg.setMessage(kMessage);
            Console.WriteLine("new alarm key=" + kMsg.getKey() + " body=" +
System.Text.Encoding.Default.GetString(kMsg.getBody()));
        }

        public void removeData(KMessage kMessage)
        {

        }
    }
}
```

As we can see, our two implementations are relatively simple. SensorsCallback simply displays the body of the messages received, then, if the key of the data received is "temperature", it will control the temperature, and create a new transaction at the address /alarms/fire if the temperature is greater than or equal to 100 ° C.

AlarmsCallback simply displays the body of the received data.

These callbacks must be cloned via the addListenerForUpdate function. However, the corresponding caches must already exist in the clone.

We will therefore implement a ClientCallback. You can then instantiate SensorsCallback and AlarmsCallback in onNewCache or in onCacheSynchronized. In our case, we will take the second option, because we do not want to process the data already present when we start our node:

```
namespace KalimaCSharpExample
{
      public class KalimaClientCallBack : ClientCallback
      {

            private Client client;
            private Logger logger;

            public KalimaClientCallBack (Client client)
            {
                  this. client = client;
                  this. logger = client. getLogger();
```

```
            }

            public void putRequestData(SocketChannel ch, KMessage msg) {}

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

            public void onNewCache(String address) {}

            public void onCacheSynchronized(string address)
            {
                  if (address. Equals("/sensors"))
            {
                        customer. getClone(). addListnerForUpdate(new
SensorsCallback(address, client. getClone()));
                  } else if (address. Equals("/alarms/fire"))
            {
                        customer. getClone(). addListnerForUpdate(new
AlarmsCallback(address));
            }

            }
    }
}
```

All that remains is to pass our ClientCallback to the clone. This is done through the connect function:

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