Example C

Table of Contents

1. Library (lib)

This library contains all the headers of the library, as well as the static archive lib_Kalima-NodeLib.a. This archive will be used when creating the executable of the project. The headers will be used to call the methods that will be useful to us in our example.

2. Config file (etc/cfg)

```
LedgerName=KalimaLedger
NODE_NAME=Node Example
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=log/
PRIVATE_KEY_FILE=RSA/private.pem
PUBLIC_KEY_FILE=RSA/public.pem
BLOCKCHAIN_PUBLIC_KEY_FILE=RSA/public.pem
SerialID=louisTuto
```

Let's see the usefulness of the different configurations:

- LedgerName -> This is the name of the Ledger we are going to connect to. For the example it is not very useful.
- NODE_NAME -> This is the name of the node we create. It is also not very useful for our example.
- NotariesList -> This is the list of nodes to which we need to connect to communicatewith the blockchain tutorial. If you want to connect to another blockchain, just change the notaries here. Between each node, you have to put a space.
- FILES_PATH -> This is the directory in which you can find the log files. Din our example, the log folder will be created at launch in the directory
- KEY_FILES -> These are the paths of the different RSA encryption files. These files are necessary to communicate with the Blockchain. They will also be created automaticallyat the start of the project.
- SerialID -> This SerialID will serve as our identification with the blockchain. It must be authorized by the blockchain. This will probably be the only line you will have to modify to put the ID you want, please contact one of our administrators (jerome.delaire@kalima.io, tristan.souillard@kalima.io)

3. Makefile

```makefile
CC=gcc -pthread
CFLAGS=-W -Wall
LDFLAGS=
EXEC= Exec
INCLUDE=-I./lib/inc/KalimaCUtil -I./lib/inc/MQ2/netlib -I./lib/inc/MQ2/message -I./lib/inc/MQ2/nodelib -I./lib/inc/NodeLib
LIBRARY=libKalimaNodeLib.a

all: $(EXEC)

Exec:
	$(CC) $(INCLUDE) -o main.o -c src/main.c $(CFLAGS)
	$(CC) $(INCLUDE) -o main.run main.o -L. lib/$(LIBRARY) $(LDFLAGS)
	rm -rf main.o

clean:
	rm -rf *.o
	rm -rf main.run
	if [ -d log ]; then rm -rf log; fi

mrproper: clean
	rm -rf $(EXEC)
```

The makefile will allow us to create the executable to run the project. First it will create the objects from source file main. Finally, it will create the executable main.run from the object created before and the archive "lib_KalimaNodeLib.a" located in the lib directory.

To execute the makefile, you just must type "make" in your terminal (you have to be in the project path in your terminal).

To clear the objects, the executable and the logs, you just need to type "make clean".

4. Main project (src + inc)

```c
Node *node = create_Node("etc/cfg/config.txt", NULL);
if(node == NULL){
    printf("Error creating Node. Please verify the config file\n");
    return 0;
}
printf("Config loaded\n");
Connect_to_Notaries(node, NULL);
```

This is the start of the main function. It contains the Kalima Node initialization functions.

Here is the explanation on the useful Kalima function:

- create_Node : Will create a Kalima Node with information from the config file. The node is the main component which will allow communication with the Blockchain. The NULL parameter is because that's where you put the smart contract list when you are using smart contracts. To see how it works, use the C smart contract example.
- Connect_to_Notaries : Will start the connection to the Blockchain with the Node and Callback as a parameter.

When creating the node, we will also create a random deviceID that will be encrypted and written in the "DeviceID" directory that will be created in the location where you start the executable from. If this encrypted file already exists, it will not be recreated. The node will just decrypt the file and use the decrypted deviceID. This deviceID, along with the SerialID in your config file, will allow the Blockchain to identify our node and allow us to send data. An RSA directory will also be created containing a public key and a private key used to encrypt communications with the blockchain.

This example when launched will offer the user two options:

- Sending 10 default messages

```
void send_10_messages(Node *node){
    for(int i=0 ; i<10 ; i++) {
        uint8_t body_size = get_int_len(95+i);
        char body[body_size];
        snprintf(body, body_size+1, "%d", 95+i);
        char key[13];
        snprintf(key, 13, "%s%d", "temperature", i);
        put_msg_with_ttl(node->clone, "/sensors", 8, key, 12, body, body_size, 10);
        printf("Sending value %s to key %s on address : /sensors\n", body, key);
        sleep(1);
    }
}
```

This choice sends 10 predefined messages to the blockchain on the address "/sensors". The 10 messages will have a key ranging from temperature0 to temperature9 and a value ranging from 95 to 104. Each message will remain in the blockchain for 10 seconds before being automatically deleted.

- Fully configurable message

```
void send_modulable_message(Node *node){
    char temp;
    char choice[10] = {}, address[100] ={}, key[100] ={};
    scanf("%c", &temp);                            //Clear buffer
    while(strncmp(choice, "a", 1) != 0 && strncmp(choice, "d", 1) != 0){
        printf("Do you want to add (a) or delete (d) ?\n");
        fgets(choice, sizeof(choice), stdin);

    }
    printf("Type the address you want to interact with :\n");
    fgets(address, sizeof(address), stdin);
    strtok(address, "\n");
    printf("Type the key of you choice :\n");
    fgets(key, sizeof(key), stdin);
    strtok(key, "\n");

    if(strncmp(choice, "a", 1) == 0){
        char msg[100];
        printf("Type the message of you choice :\n");
        fgets(msg, sizeof(msg), stdin);
        strtok(msg, "\n");
        put_msg_default(node->clone, address, strlen(address), key, strlen(key), msg, strlen(msg));
    }
    if(strncmp(choice, "d", 1) == 0){
        remove_msg(node->clone, address, strlen(address), key, strlen(key));
    }
}
```

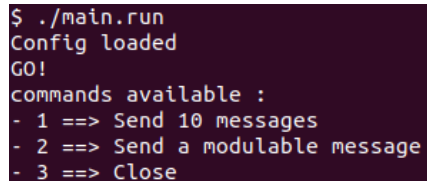Here we build the message from scratch.

1. First, the user is offered to choose between adding or deleting a message. Then we retrieve the address, the key and the message from the user's terminal. Finally, we send the message. Unlike the previous example, the message will remain here indefinitely.

5. Running the project

To run the example, simply open a terminal and move to the main folder (where your makefile is).

Then simply type "make" to launch the makefile seen above. This command will create the executable file "main.run" (as well as the DevID directory and the RSA directory) that will allow us to launch the program.

Finally, you must write "./main.run" to launch the program.

```
$ ./main.run
Config loaded
GO!
commands available :
- 1 ==> Send 10 messages
- 2 ==> Send a modulable message
- 3 ==> Close
```

When you reach this point, and you are sure you did all the correct steps to be authorized by the Blockchain, it means your Node has been successfully launched.

It also means that you received all the memcache data of the Blockchain and the Lua smart contract has been downloaded to the directory you specified in the main file (the contract directory will be created if it does not already exist).

You are now free to send data to the Blockchain using either of the options seen previously. If you receive a data with the criteria of your smart contracts, they will be loaded and used.

6. Possible issues

If your message does not appear in the blockchain, here are some possible errors:

- Verify that the SerialID in the config file is the same as the one entered in the blockchain.
- If you delete a DeviceID or RSA directory, the new calculated files will be different. It will therefore be necessary to redo the authorization process.
- If you have made changes on the hand that are not considered, make a "make clean" before redoing "make" so that all the changes are considered.
- If you have other problems, you can look at the logs when the program has problems and contact us.