

Example C

Table of Contents

1.	Library (lib)	2
2.	Config file (etc/cfg)	2
3.	Makefile	3
4.	Main project (src + inc)	3
1.	Main	3
2.	LuaFunctions	4
3.	Callback	5
5.	Running the project	7
6.	Possible issues	8

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 communicate with 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. In 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 automatically at 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

```
HOST_CC=gcc
CC=$(HOST_CC) -pthread
CFLAGS=-W -Wall
LDFLAGS=-lm -ldl
EXEC= Exec
INCLUDE=-I./inc -I./lib/inc/KalimaUtil -I./lib/inc/MQ2/netlib -I./lib/inc/MQ2/message -I./lib/inc/MQ2/nodelib -I./lib/inc/NodeLib -I./lib/inc/ContractManager
LIBRARY=libKalimaNodeLib.a

all: $(EXEC)

Exec:
$(CC) $(INCLUDE) -o LuaFunctions.o -c src/LuaFunctions.c $(CFLAGS)
$(CC) $(INCLUDE) -o callback.o -c src/callback.c $(CFLAGS)
$(CC) $(INCLUDE) -o main.o -c src/main.c $(CFLAGS)
$(CC) -o main.run *.o -L. lib$(LIBRARY) $(LDFLAGS)
rm -rf *.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 files LuaFunctions, callback and main. Finally, it will create the executable main.run from the objects 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)

1. Main

```
ContractList *list = new ContractList("LuaScripts/", 1);
Node *node = create_Node("etc/cfg/config.txt", (void*)list);
printf("Config loaded\n");
ClientCallback* callback = set_callback();
Connect_to_Notaries(node, callback);
```

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

Here is the explanation on the useful Kalima function:

- new_ContractList : Will create a SmartContract list which will contain every Lua smartcontracts present in the indicated directory. It will also create a watcher which will monitor the said directory to check if a contract is added, modified or deleted and will update the list accordingly. The second parameter specifies whether the contracts are encrypted or not (1: encrypted, 0: not encrypted).
- 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.
- set_callback : Will initiate the client callback which we will see in the explanations on the callback.c file.
- 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[5];
        snprintf(key, 5, "%s%d", "key", i);
        put_msg_with_ttl(node->clone, "/sensors", 8, key, 4, body, body_size, 10);
        sleep(1);
    }
}
```

This choice sends 10 predefined messages to the blockchain on the address `"/sensors"`. The 10 messages will have a key ranging from `key0` to `key9` 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.

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.

2. LuaFunctions

LuaFunctions is the file where we will put the C functions that we want to use inside the contracts. Those functions follow a simple format:

```
int LuaGetBody(lua_State*L){
    void* Kmsg_ptr = lua_touserdata(L,1);
    KMsg* msg = (KMsg*) Kmsg_ptr;
    char* body = (char*)getBody(msg);

    lua_pushstring(L,body);
    free(body);
    return 1;
}
```

As a reference, this function is called in the contract using : "body = LuaGetBody(kmsg)"

The "lua_touserdata(L,1)" specifies the data received is the first parameter inside the function call in the contract (here kmsg). It will be received as a pointer.

Then we call the classic getBody function and push it in the contract with the same method seen earlier.

Finally, the "return 1" is very important. If you want the function to return data to the script you need to put "1". If no data is returned, you can put "0"

Here is an example :

```
int LuaPutLog(lua_State*L){
    void* node_ptr = lua_touserdata(L,1);
    Node* node = (Node*) node_ptr;
    char* log = (char*)lua_tostring(L,2);
    log_srvMsg(node->config->Files_Path, "Main", "Log", INFO, log);
    return 0;
}
```

This function only writes a log but return no data, so we put a "return 0".

3. Callback

The callback is useful to allow the user to do some actions based on what is happening inside functions in the library without having to modify the said library.

In the callback, we have 5 possible actions located in different parts of the library:

- c_send : Isn't really used for now.
- onNewAddress : Same.
- onAddressSynchronized : Same.
- onReject : Will write a log if the join request is refused by the Blockchain.
- PutData : Is the useful callback function for now. It will act based on data received from the Blockchain. You can change it to do whatever you want with the data received. We'll explain what is being done in this example.

```
Client *client = (Client*)client_ptr;
KMsg *kmsg = setMessage(Kmessage);
ContractList *contract_list = (ContractList*)client->node->contract_list;
char* address = (char*)getAddress(kmsg);

if(getAddressSize(kmsg)==0)
    return;
```

This part will turn the pointers into usable structure and create a Kmsg based on the Kmessage received from the Blockchain. We'll then get the address from the Kmsg (Address of the data received from the Blockchain).

```

if(contract_list->iscrypted == 1){
    if(strncmp(address, "/Kalima_Scripts", getAddressSize(kmsg)) == 0){
        char* key = (char*)getKey(kmsg);
        char* file;
        set_string(key, strlen(key), (void*)&file);
        strtok(file, ".");
        char* filetype = strtok(NULL, "0");
        if(key != NULL && filetype != NULL && strcmp(filetype, "lua", 3) == 0){
            char* url = (char*)getProp(kmsg, "downloadURL");
            log_srvMsg(client->node->config->Files_Path, "Contract", "Manager", ERR, "Curl request ...");
            curl_req(url, key, contract_list->Contract_path);
            log_srvMsg(client->node->config->Files_Path, "Contract", "Manager", ERR, "Curl request done");
            free(url);
            int i;
            for(i=0; i<nb_of_elements(contract_list->List); i++){
                Lua* lua_script = (Lua*)(list_get_element(contract_list->List, i)->data);
                int contract_log_size = 19+strlen(key)+21+strlen(lua_script->filename);
                char contract_log[contract_log_size];
                snprintf(contract_log, contract_log_size, "%s%s%s", "remote filename : ", key, " / local filename : ", lua_script->filename);
                log_srvMsg(client->node->config->Files_Path, "Contract", "Manager", DEBUG, contract_log);
                if(strncmp(lua_script->filename, key, strlen(key)) == 0){
                    log_srvMsg(client->node->config->Files_Path, "Contract", "Manager", DEBUG, "SmartContract is on local system");
                    char* signature = (char*)getProp(kmsg, "signature");
                    char* aesKey = (char*)getProp(kmsg, "aesKey");
                    char* aesIV = (char*)getProp(kmsg, "aesIV");
                    if(signature == NULL || aesKey == NULL || aesIV == NULL){
                        log_srvMsg(client->node->config->Files_Path, "Contract", "Manager", ERR, "Error getting aes infos from props");
                        free(key);
                        return;
                    }
                    log_srvMsg(client->node->config->Files_Path, "Contract", "Manager", DEBUG, "Decrypt SmartContract");
                    decrypt_lua_script(lua_script, lua_script->filepath, signature, aesKey, aesIV);
                    free(signature), free(aesKey), free(aesIV);
                }
            }
            free(key), free(file);
        }
    }
}

```

If we're dealing with encrypted contracts from the Blockchain, it is necessary to decrypt them locally to use them.

First, it is necessary to understand that when you successfully connect to the Blockchain, you receive locally every data in the Blockchain memcache where you are authorized. You also receive the data necessary to decrypt the contracts.

This data will be received with address `"/Kalima_scripts"`. We then check if the contracts concerned are of the `".lua"` type and if yes, we decrypt them.

We first get the contracts from curl requests using data received by the Blockchain, then check if the file safely arrived in the local contract directory. If yes, we get the data needed to decrypt the contract and decrypt it (the decrypted content will be saved as a string and loaded later using the string).

```

if(check_Type(kmsg->Kmessage, "SNAPSHOTRESP", 0) == 0){
    void* kmsg_ptr = (void*)kmsg;
    void* node_ptr = (void*)client->node;

    if(contract_list != NULL){
        log_srvMsg(client->node->config->Files_Path, "Contract", "Manager", INFO, "data received");
        if(strncmp(address, "/sensors", getAddressSize(kmsg)) == 0){
            log_srvMsg(client->node->config->Files_Path, "Contract", "Manager", INFO, "Using contract sensors.lua");
            Lua* lua_contract = load_Contract(contract_list, "sensors.lua");
            if(lua_contract == NULL){
                log_srvMsg(client->node->config->Files_Path, "Contract", "Manager", ERR, "Error loading contract");
                free(address);
                return;
            }
            lua_getglobal(lua_contract->L, "main");
            lua_pushcfunction(lua_contract->L, lua_GetBody);
            lua_setglobal(lua_contract->L, "lua_GetBody");
            lua_pushcfunction(lua_contract->L, lua_GetKey);
            lua_setglobal(lua_contract->L, "lua_GetKey");
            lua_pushcfunction(lua_contract->L, lua_PutMsg);
            lua_setglobal(lua_contract->L, "lua_PutMsg");
            lua_pushcfunction(lua_contract->L, lua_PutLog);
            lua_setglobal(lua_contract->L, "lua_PutLog");
            lua_pushcfunction(lua_contract->L, lua_Strlen);
            lua_setglobal(lua_contract->L, "lua_Strlen");
            lua_pushlightuserdata(lua_contract->L, kmsg_ptr);
            lua_pushlightuserdata(lua_contract->L, node_ptr);
            lua_pcall(lua_contract->L, 2, 0, 0);
            log_srvMsg(client->node->config->Files_Path, "Contract", "Manager", INFO, "Finished using contract");
        }
    }
}

```

We only want to use the contract on new data, so we must check that the data is not from the data received at the time of connection.

Once it is done, here we check that the address of the message received is `"/sensors"` (which means a new data has been added to this address in the Blockchain) and if yes, we run the `"sensors.lua"` contract with the following method:

- `load_Contract` : Will load a contract. If the contract is found in the specified folder, The contract structure will be returned. If not, NULL will be returned.
- `lua_getglobal` : Will specifies which function inside the contract we are calling (here main).
- `lua_pushcfunction` : Will push a C function inside the contract. It is necessary if you want to use your own C functions inside a contract.
- `lua_setglobal` : Will specifies how the function pushed right before is called inside the contract. We could for example say that a C function named `"x"` will be called using `"y"` in the contract.
- `lua_pushlightuserdata` : Will push a pointer inside the main function (as entry parameter).
- `lua_pcall` : Will run the contract. The `"2"` specifies the number of paramaters sent to the function. `"0"` is the number of paramaters returned from the contract and the last `"0"` is for the error handling.

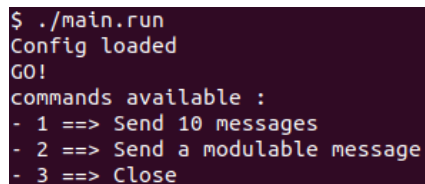
So here we're basically calling the main function of the `sensors.lua` contract. We're then sending to the contract the functions we'll be using inside, then we send the entry paramaters to finally run it.

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 Smartcontract 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.