

C Smart Contract Example

Library (lib)

This library contains all the headers of the library, as well as the static archive lib_KalimaNodeLib.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.

Config file (etc/cfg)

```
1 LedgerName=KalimaLedger
2 NODE_NAME=Node Example
3 privachain=org.kalima.tuto
4 FILES_PATH=Files/
5 BLOCKCHAIN_PUBLIC_KEY_FILE=RSA/public.pem
6 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.
- Privachain -> This is the privachain we will be connected into. If you want to connect to another Kalima Blockchain, you will have to change the privachain.
- FILES_PATH -> This is the directory in which all Kalima related files will be created. When you launch your node for the first time, this directory will be created if it doesn't already exist, and you will find inside the files for RSA, DevID and logs.
- BLOCKCHAIN_PUBLIC_KEY -> This is the path for the blockchain public key if you need for some reasons to have it locally. It isn't used for now.
- SerialID -> This ID allow your node to be authorized on the Kalima Blockchain which you can obtain here: <https://inscription.tuto.kalimadb.com/airdrop> (you have 10 serialId received by email)

Makefile

```
HOST_CC=gcc
CC=$(HOST_CC) -pthread
CFLAGS=-W -Wall
LDFLAGS=
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
LIBRARY=libKalimaNodeLib.a

all: $(EXEC)

Exec:
$(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 Files/log ]; then rm -rf Files/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 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 must be in the project path in your terminal).

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

Main project (src + inc)

Main.c

```
int choice = 0, end = 0;
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);
ClientCallback* clientCallback = set_clientCallback();
add_ClientCallback(node, clientCallback);
sleep(2);
printf("GO!\n");
Menu();
while(end == 0){
    scanf("%d", &choice);
    if ( choice == 1) {
        send_10_messages(node);
    } else if (choice == 2) {
        send_modulable_message(node);
    } else if (choice == 3) {
        end = 1;
    }
}
```

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 (by default "etc/cfg/config.txt", you can also use a different one by putting it as a parameter when launching the executable). 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.
- ClientCallBack: Will be explained in the callback explanations.

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 a default message

```
void send_default_message(Node *node){
    char str[100];
    char temp;
    printf("Message to send to Blockchain :\n");
    scanf("%c", &temp); // Clear buffer
    fgets(str, sizeof(str), stdin); // Get input
    List *kProps = new_proplist();
    set_prop(kProps, "ttl", 3, "-1", 2); // will give ttl=-1, meaning the message will stay in blockchain (ttl = time to live)
    KMsg *kmsg = getMessage(node->deviceid, UUID_SIZE, get_encoded_Type("PUB"), "/sensors", 0, "Default message", 15, 1, str, strlen(str), kProps);
    send_KMessage(node, kmsg->Kmessage);
    printf("Message sent\n");
    sleep(2); // Waiting for response (in logs)
}
```

This choice sends the message put by the user to the Blockchain address `"/sensors"` with the key `"Default message"`. The time to live (ttl) is put as `"-1"` which means that the message will stay in the Blockchain memcache.

- 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.

Callback.c

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:** 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.
 - **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.
 - **onReject:** This callback is called when your node tries to connect to the blockchain, but it's not authorized. So, you can display a message to inform the user 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

To better understand, here is what we are doing in the CExample tutorial:

We have 2 callbacks:

```
ClientCallback* set_clientCallback(){
    return new_ClientCallback(c_send, onNewAddress, onAddressSynchronized, onReject);
}

MemCacheCallback* set_memcacheCallback(){
    return new_MemCacheCallback(putData, removeData, "/sensors");
}
```

The ClientCallback will be set in the main function seen earlier. It will use the functions that we will see later.

The MemcacheCallback will be set in the onAddressSynchronized of the ClientCallback, we will see precisions later. This callback is set for the "/sensors" memcache.

For the ClientCallback, only the onAddressSynchronized will be really used for this example. Here is what it looks like:

```

void onAddressSynchronized(void* node_ptr, char* address, uint8_t address_size){
    Node* node = (Node*)node_ptr;
    if(node!=NULL && address != NULL && address_size != 0){
        if(strncmp(address, "/Kalima Contracts", 17)==0){
            ContractList* contract_list = (ContractList*)node->contract_list;
            MemCache* memcache = getMemCache(node->clone, address, address_size);
            SkipListNode t *kvmmap = SkipListGetFirst( memcache->kvmmap );
            for ( ; kvmmap; kvmmap=kvmmap->forward[0] ) {
                KMessage* msg = build((unsigned char*)((KeyValueObj *)kvmmap->key)->value+4);
                KMsg *kmsg = setMessage(msg);
                char* key = (char*)((KeyValueObj *)kvmmap->key)->key;
                int key_size = ((KeyValueObj *)kvmmap->key)->key_size;
                char* file;
                set_String(key, key_size, (void**)&file);
                strtok(file, ".");
                char* filetype = strtok(NULL, "0");
                if(filetype != NULL && strcmp(filetype, "lua", 3) == 0){
                    char* url = (char*)getProp(kmsg, "downloadURL");
                    log_srvMsg(node->config->log_Path, "Contract", "Manager", ERR, "Curl request ...");
                    curl_req(url, key, contract_list->Contract_path);
                    log_srvMsg(node->config->log_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];
                        sprintf(contract_log, contract_log_size, "%s%s%s", "remote filename : ", key, " / local filename : ", lua_script->filename);
                        log_srvMsg(node->config->log_Path, "Contract", "Manager", DEBUG, contract_log);
                        if(strncmp(lua_script->filename, key, key_size)==0){
                            log_srvMsg(node->config->log_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(node->config->log_Path, "Contract", "Manager", ERR, "Error getting aes infos from props");
                                free(key);
                                return;
                            }
                            log_srvMsg(node->config->log_Path, "Contract", "Manager", DEBUG, "Decrypt SmartContract");
                            decrypt_lua_script(lua_script, lua_script->filepath, signature, aesKey, aesIV);
                            free(signature), free(aesKey), free(aesIV);
                        }
                    }
                    free(file);
                }
            }
        }
        if(strncmp(address, "/sensors", address_size)==0){
            MemCacheCallback* memcacheCallback = set_memcacheCallback();
            add_MemCacheCallback(node->clone, memcacheCallback);
        }
    }
}

```

When you follow the instructions received by mail after registering in the [airdrop](#), you will see how to add your own smart contracts to the Blockchain. You will also see in which address the information necessary to use your contracts will be in the Blockchain. In this example, the information is in the "/Kalima_Contracts" path but it will need to be changed with your own path. So what we are doing here is when the path for your contracts is synchronized with the Blockchain, you will download each contract on a local directory (in your FILES_PATH) using the "curl_req" function and then each contract will be decrypted to be used later. The only thing that will need to change is the address. The rest can stay the same.

In this function we will also add a MemcacheCallback for the "/sensors" address.

Now here are the functions for this MemcacheCallback :

```

void putData(void* clone_ptr, KMessage* Kmessage){
    Clone* clone = (Clone*)clone_ptr;
    ContractList *contract_list = (ContractList*)clone->node->contract_list;
    KMsg *kmsg = setMessage(Kmessage);
    if(clone==NULL || kmsg==NULL || getAddressSize(kmsg)==0) return;
    char* address = (char*)getAddress(kmsg);

    char* key = (char*)getKey(kmsg);
    char* body = (char*)getBody(kmsg);
    printf("Message added on address : %s / Key : %s / Body : %s\n", address, key, body);

    if(contract_list != NULL){
        log_srvMsg(clone->node->config->log_Path, "Contract", "Manager", INFO, "Using contract sensors.lua");
        Lua* Lua_contract = load_Contract(contract_list, "sensors.lua");
        if(Lua_contract == NULL){free(address); return;}
        lua_getglobal(Lua_contract->L,"main");
        lua_pushcfunction(Lua_contract->L,LuaGetBody);
        lua_setglobal(Lua_contract->L, "LuaGetBody");
        lua_pushcfunction(Lua_contract->L,LuaGetKey);
        lua_setglobal(Lua_contract->L, "LuaGetKey");
        lua_pushcfunction(Lua_contract->L,LuaPutMsg);
        lua_setglobal(Lua_contract->L, "LuaPutMsg");
        lua_pushcfunction(Lua_contract->L,LuaPutLog);
        lua_setglobal(Lua_contract->L, "LuaPutLog");
        lua_pushcfunction(Lua_contract->L,LuaStrLen);
        lua_setglobal(Lua_contract->L, "LuaStrLen");
        lua_pushlightuserdata(Lua_contract->L,(void*)kmsg);
        lua_pushlightuserdata(Lua_contract->L,(void*)clone->node);
        lua_pcall(Lua_contract->L, 2, 0, 0);
        log_srvMsg(clone->node->config->log_Path, "Contract", "Manager", INFO, "Finished using contract");
    }
    free(key), free(body), free(address);
}

void removeData(void* clone_ptr, KMessage* Kmessage){
    Clone* clone = (Clone*)clone_ptr;
    KMsg *kmsg = setMessage(Kmessage);
    if(clone==NULL || kmsg==NULL || getAddressSize(kmsg)==0) return;
    char* address = (char*)getAddress(kmsg);

    char* key = (char*)getKey(kmsg);
    char* body = (char*)getBody(kmsg);
    printf("Message removed on address : %s / Key : %s / Body : %s\n", address, key, body);
    free(key), free(body), free(address);
}

```

In this example, when a message is removed from the blockchain (from the memcache), we print it in the removeData function.

When a message is added the putData function will be called. It will be printed and then a smartcontract will be use with this message as a parameter. Here is a description on what each function does and how we use a Lua contract:

- 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 as we will see in the LuaFunctions description.
- 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 thecontract.
- 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 thefunction. "0" is the number of paramaters returned from the contract and the last "0" is forthe error handling.

This example creates a callback for the data on the "/sensors" address. But you are free to add callback for multiple addresses and you can do whatever you want with those callbacks. You can for example use Lua contract with those data as you can see in the C SmartContract example.

LuaFunctions.c

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. If this function uses multiple parameters, to get them you will have to use lua_touserdata(L,2) ... 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->log_Path, "Main", "Log", INFO, log);  
    return 0;  
}
```

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

Running the project

Authorization

Before launching the project, it is necessary that you do the correct procedure to be authorized on the Blockchain.

For this, it is simple, you just register here: <https://inscription.tuto.kalimadb.com/airdrop>. Once it is done you will receive an email explaining every step to be successfully authorized on the blockchain. Basically, you just need to take one of the 10 serialID received by email on the config file and launch the program, it will create RSA keys and a

encrypted devID in the directory specified as "FILES_PATH" in the config file. Those files are very important and mustn't be deleted.

The next time you launch the project, if the directories created stay the same and the serialID is also the same, everything should still work.

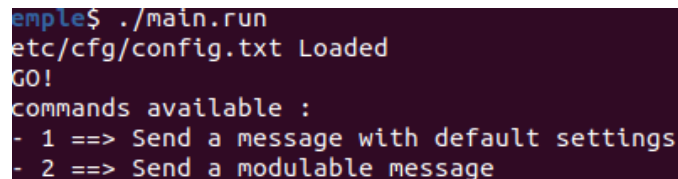
If for some reason your directories got deleted, you changed your system or you changed your serialID, the authorization process must be done again.

Launch

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.

A terminal window with a dark background and light-colored text. The text shows the command './main.run' being executed, followed by 'etc/cfg/config.txt Loaded', 'GO!', and a list of available commands: '1 ==> Send a message with default settings' and '2 ==> Send a moduable message'.

```
example$ ./main.run
etc/cfg/config.txt Loaded
GO!
commands available :
- 1 ==> Send a message with default settings
- 2 ==> Send a moduable message
```

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.

You are now free to send data to the Blockchain using either of the options seen previously.

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 other problems, you can look at the logs when the program has problems and contact us.