Smart contract C

Table des matières

Contenu

[Fonctionnement global d’un node C 2](#_Toc1)

[Modifications du main.c et callback 2](#_Toc2)

[Main.c 2](#_Toc3)

[Callback 3](#_Toc4)

[Les SmartContracts Lua 4](#_Toc5)

[Format des SmartContracts 4](#_Toc6)

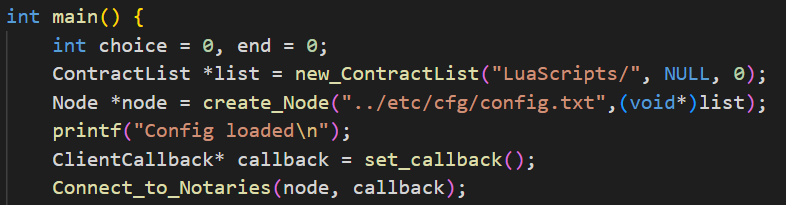
[Fonctions C utilisable actuellement 5](#_Toc7)

# Fonctionnement global d’un node C

Si vous souhaitez comprendre le fonctionnement global d’un node C, à savoir les fichiers config, la librairie, le makefile et le main.c, veuillez aller voir la documentation Cexample.

# Modifications du main.c et callback

## Main.c

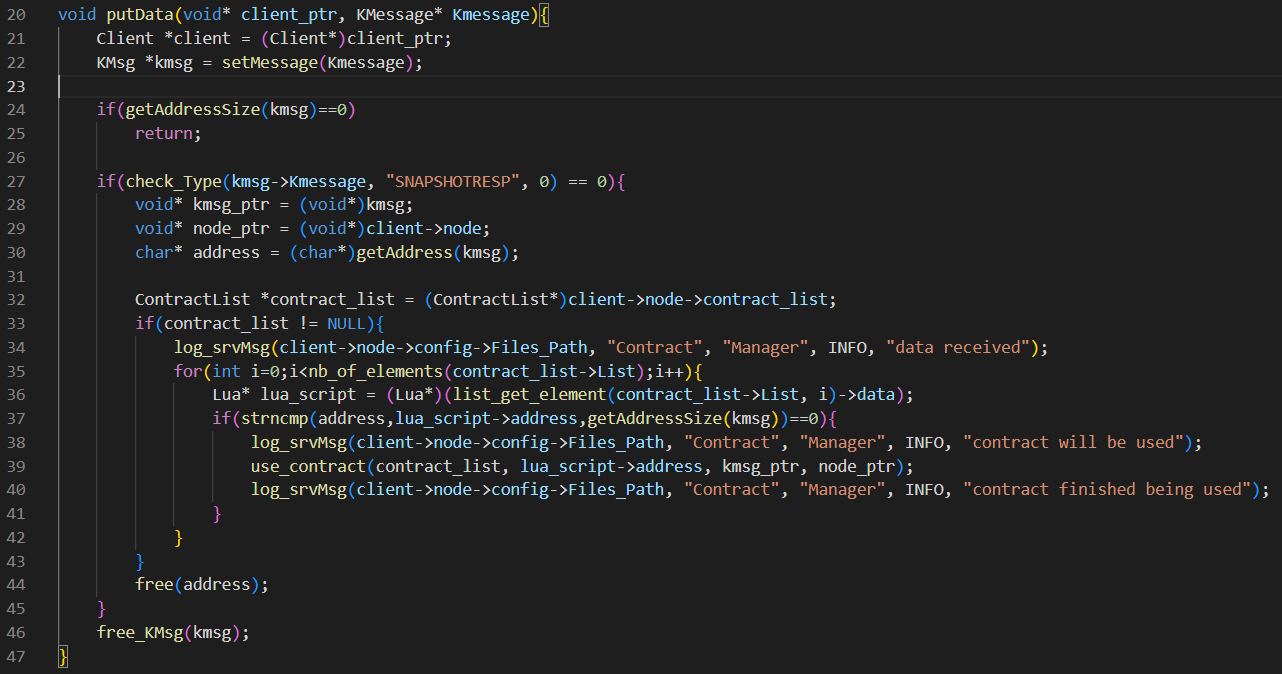


Contrairement à l’exemple KalimaCExample, ici nous créons un pointeur vers une structure ContractList que l’on va passer en paramètre du node. De plus nous créons un callback que nous passons en paramètre de la fonction de connexion aux notaries.

Le new\_ContractList va créer un filewatcher qui va surveiller le dossier spécifié en 1er paramètre. Ici le dossier surveillé sera LuaScripts/ . Dans ce dossier, tous les fichiers de type .lua seront enregistrés et pourront être utilisés par le programme. Avec le filewatcher, si des fichiers sont modifiés, supprimés ou ajoutés dans le dossier, ils seront automatiquement ajoutés dans le programme même si ce dernier est déjà en cours d’exécution. Le 2ème paramètre de la fonction correspond à un ajout au path possible au niveau de la BlockChain. Nous préciserons plus tard ce que cela veut dire en détaillant le callback. Enfin le dernier paramètre est un booléen qui va spécifier sur les contrats Lua seront cryptés ou pas. En effet, la librairie Kalima permet l’utilisation de contrats cryptés et non cryptés. Dans un cas d’utilisation réel de SmartContracts, il est nécessaire d’utiliser des contrats vérifiés par la BlockChain et cryptés. Cependant pour réaliser des tests comme ici, nous pouvons nous permettre d’utiliser des contrats non-cryptés stockés en local.

Le callback que nous passons en paramètre du connect\_to\_Notaries correspond à différent cas de figures qui peuvent arriver au moment de la réception de données au niveau du client. En passant par un callback, l’utilisateur peut alors faire ce qu’il veut en fonction des cas de figures. Nous verrons plus en précision plus tard.

## Callback



Dans cet exemple, la seule fonction du callback utilisée est le putData. Cette fonction est appelée au niveau du client pour chaque message reçu de la BlockChain (à noter que quand un message est envoyé depuis le node, une réponse sera reçue de la Blockchain. Cette réponse contient les mêmes informations que le message envoyé. Si une réponse est reçue, cela veut dire que le message a été accepté par la BlockChain). En paramètre de la fonction nous avons un pointeur vers le client qui a reçu le message, ainsi qu’un pointeur vers le message reçu.

Un KMessage Kalima est le format utilisé pour les messages échangés entre les nodes. Un Kmessage contient les informations clés telles que l’adresse de la BlockChain où nous souhaitons envoyer une donnée, la clé du message, le body ...

Un KMsg est une surcouche plus structurée du Kmessage qui contient toutes les informations utiles.

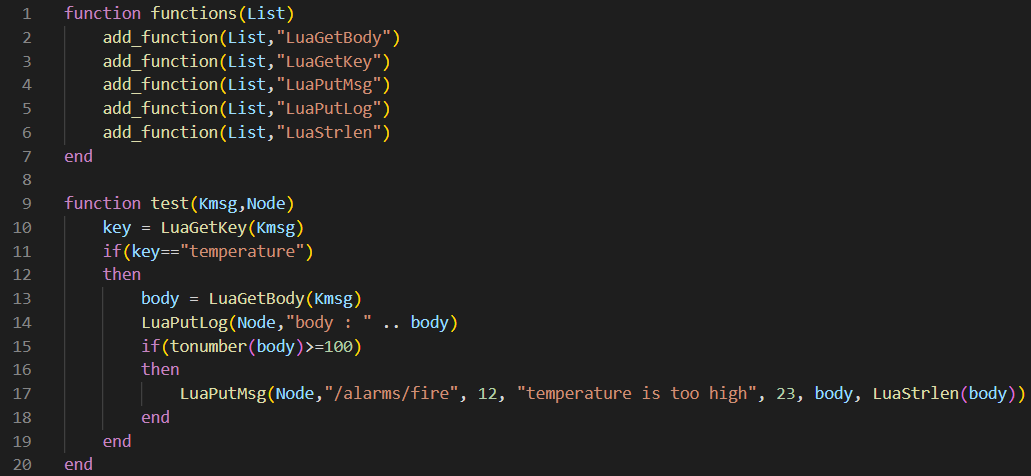
Dans cette fonction, nous commençons tout d’abord par vérifier qu’il n’y a pas de problème au niveau de l’adresse. S’il y a un problème, nous terminons la fonction. Ensuite nous vérifions que le message traité n’est pas une réponse du snapshot. Pour information, lorsqu’un node se connecte à la BlockChain, il envoie une demande de snapshot. Si cette demande est acceptée, il recevra en réponse toutes les données de la BlockChain sur les adresses sur lesquels le node est autorisé. Nous souhaitons donc ici que le traitement du callback ne s’applique pas au snapshot.

Enfin nous regardons dans la liste des SmartContracts Lua enregistrés depuis le dossier spécifié dans le main si l’adresse du message réçu correspond à l’adresse du contrats. Pour mieux comprendre, voici un exemple : nous voulons un contrat qui regarde les données de l’adresse /test sur la BlockChain. Il faudra alors appeler le contrat Lua test.lua. A ce niveau là, le code va donc vérifier que l’adresse “test” correspond au nom du contrat (”test.lua”). Il est également possible avec le deuxième paramètre du new\_ContractList du main de rajouter un préfixe à l’adresse de la Blockchain. Par exemple, si nous voulons surveiller l’adresse /demo/test, nous ne pouvons pas créer un fichier demo/test.lua. Nous pourvons donc spéficier en paramètre que les scripts auront pour préfixe par défaut au niveau de la BlockChain “/demo/”.

Enfin si nous trouvons un SmartContract qui est associé à l’adresse, il ne reste qu’a exécuter ce contrat (à noter que le contrat prendra toujours en paramètre un pointeur vers un Kmsg et un pointeur vers le node. Ceci permet de faire beaucoup d’actions).

# Les SmartContracts Lua

## Format des SmartContracts



Voici ici le SmartContract Lua utilisé dans le tuto KalimaCSmartContract. Pour l’ajout et l’utilisation automatique des SmartContracts Lua, il est important de respécter ce format.

La fonction functions est nécessaire pour l’automatisation. En effet, quand vous voulez que votre contrat utilise des fonctions C prédéfinies, il est important qu’il les comprenne. Pour automatiser cela, nous avons fait cette fonction qui est appelée quand un contrat va être utilisé par le programme, ce qui va permettre à Lua de comprendre toutes les fonctions utilisées dans la vrai fonction. Dans cette fonction il faut donc ajouter toutes les fonctions qui seront utilisées plus tard avec add\_function(List, <nom de la fonction>), List étant une liste chaînée.

La fonction test est la vraie fonction qui représente le SmartContract. C’est ici que vous pourrez écrire votre propre SmartContract. Dans cet exemple, nous allons récupérer la clé depuis le kmsg passé en paramètre. Si cette clé est “temperature” alors on récupère le body, on l’affiche dans le fichier log des SmartContracts, puis si ce body est supérieur à 100, on envoie un message à la BlockChain à l’adresse /alarms/fire, avec pour key “temperature is too high” et comme body la température. A noter que l’on peut bien voir que toutes les fonctions utilisées ici ont été données dans la fonctions functions.

## Fonctions C utilisable actuellement

Comme précisé précédemment, si l’on souhaite utiliser des fonctions C dans les SmartContracts automatiquement, il est nécessaire de les spécifier au préalable. Toutes ces fonctions sont prédéfinies à l’avance dans le code. Si vous souhaitez ajouter vos propres fonctions, il est pour l’instant nécessaire de nous contacter.

Voici une liste des fonctions disponibles pour l’instant :

**int** **LuaGetBody**(void\* kmsg) : *Function that will be used in Lua contract to get body from Pointer to Kmsg.*

**Int LuaGetKey**(void\* kmsg) : *Function that will be used in Lua contract to get body from Pointer to Kmsg.*

**Int LuaPutMsg**(void\* node, string address, int address\_size, string key, int key\_size, string body, int body\_size) : *Function that will be used in Lua contract to send message to BlockChain.*

**Int LuaGetTemperature**(string payload) : *Function that will be used in Lua contract to get temperature from Kerlink's payload.*

**Int LuaGetHumidity**(string payload) : *Function that will be used in Lua contract to get humidity from Kerlink's payload.*

**Int LuaGetVOC**(string payload) : *Function that will be used in Lua contract to get VOC from Kerlink's payload.*

**Int LuaGetCO2**(string payload) : *Function that will be used in Lua contract to get CO2 from Kerlink's payload.*

**Int LuaStrlen**(string str) : *Function that will be used in Lua contract to get string length*

**Int LuaPutLog**(void\* node, string log\_message) : *Function that will be used in Lua contract to print string to Logs.*

*.*