Implementation of User Settings Control with Lua C API

Milan Berić, Vladan Bjelić, Đorđe Simić ,Nemanja Fimić

Abstract—**This paper explains how C++ functions for controlling user data can be connected to Lua by using Lua C API. Solution presented in this paper can be used as a guideline for applications that have the need to prevent direct file manipulation, but instead to allow file access only in controlled environment. Implementation considers completely disabling I/O module in Lua and instead using set of custom functions for file access.**

Keywords—**binding, Lua, Lua C API, Lua stack, value storage**

I. Introduction

Lua is a powerful, efficient, lightweight scripting language. It supports procedural programming, object-oriented programming, functional, data-driven programming, and data description. It combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics. Lua is dynamically typed and has automatic memory management with incremental garbage collection, making it ideal for configuration and scripting.

It can be used in combination with other programming languages when we want to take advantage of rapid prototyping.

Lua gives us great possibilities if there is need for creating secure user scripts. To create Lua sandbox, some of standard libraries that might be unsafe must be disabled. Objective of this paper is to show one possible solution of creating secure Lua sandbox by disabling I/O module in Lua scripts and creating custom interface that will be used for user settings management.

II. Environment

Lua is an embedded language, which means that it is not a stand-alone package, but a library that can be linked with other applications to incorporate Lua facilities into these applications [1]. Lua can be observed as extension language and as extensible language [2]. First one refers to ability of being used as a library to extend application's possibilities. The second one refers to adding new features which can't be implemented using only Lua. An application that uses Lua is able to register new functions inside the Lua environment (e.g. function written in C++ or another language can extend basic Lua functionalities). These two views of Lua, correspond to two kinds of interaction between C++ and Lua. If Lua is observed as an extension language, C++ code has control in communication and therefore it is called application code. Otherwise, Lua has control and C++ code behaves like library. In order to achieve communication with Lua, both application code and library code are using the same API. The so called Lua C API is the set of functions that allow interaction between C/C++ and Lua (Fig. 1). It consists of functions that access to Lua global variables, run pieces of Lua code, call Lua functions, register C/C++ functions so that they can later be called from Lua code etc. Because Lua can be compiled with both, C and C++, when using it in C++, functions from Lua header must have C linkage.

Milan Berić, RT-RK Institute for Computer Based Systems, Jovana Dučića 23a, 78000 Banja Luka, Bosnia and Herzegovina (e-mail: milan.beric@rt-rk.com)

Vladan Bjelić, Đorđe Simić and Nemanja Fimić, RT-RK Institute for Computer Based Systems, Narodnog fronta 23a, 21000 Novi Sad, Serbia (e-mails: vladan.bjelic@rt-rk.com, djordje.simic@rt-rk.com, nemanja.fimic@rt-rk.com)

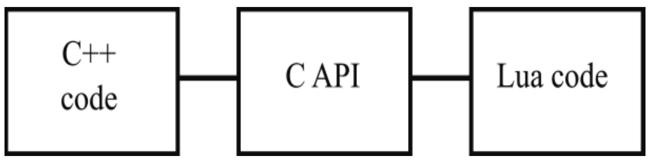

Fig. 1. Connecting C++ and Lua code

Lua cannot call any C/C++ function, it follows certain protocol to send the parameters and to get the results. It is only able to "see" C/C++ functions that passed registration process. Once function is registered and pointer to a C/C++ function is present in Lua, Lua application can call it through direct reference to its address. Any function registered in Lua has the same prototype defined in Lua header file as luaC_function (Expression 1).

*typedef int (*lua_Cfunction) (lua_State *L);* (1)

When observing from side of C/C++ code, a C/C++ function always gets Lua state as the only argument and returning value is the number of results we are sending to Lua. *lua_State* is structure that keeps the whole state of Lua interpreter. All information about a state is kept in this dynamic structure.

There are two problems when trying to exchange values between Lua and C: the type mismatch between a dynamic and a static type system and the mismatch between automatic and manual memory management. That is the reason why abstract stack is used when transferring values from Lua to C and vice versa. Stack is designed in a way that every slot is able to temporary keep Lua value of any type. While Lua manipulates this stack in a strict LIFO discipline (Last In, First Out), C/C++ can inspect any element inside the stack and even delete or insert element in any arbitrary position. Lua C API uses indices to refer to elements in stack. The first element that is pushed to the stack has index 1, the second one has index 2 etc. It's also possible to use top of the stack as reference and in that case negative indices are used instead (the last pushed element has index -1).

As each function has its own virtual stack, there is no need to clear stack before pushing return values. After it returns, Lua takes results from the top of the stack and automatically removes whatever is in the stack bellow them. When creating API functions, we must take care about type checking and type errors, error recovery, memory-allocation errors, as most functions in the Lua C API do not check the correctness of their arguments. To overpass lack of static type system in Lua, every value exchanged between Lua and C/C++ will be at some point translated to *lua_Object* type, which behaves as an abstract type in C/C++ that can hold any Lua value. Values of type *lua_Object* don't have meaning outside Lua (e.g. comparing two lua_Objects has no significance). Lua has automatic memory management and garbage collection. Therefore, *lua_Object* has a limited scope and is only valid inside the block where it was created (until the end of a C function called from Lua). A good programming practice is to convert Lua object to C values as soon as they are available and never store *lua_Object* in C global variables. Although current development is moving toward creating a better garbage collector, right now, Lua GC can be unpredictable sometimes. It uses simple mark and sweep algorithm, where we mark all reachable objects and sweep away the ones remaining. Garbage collection can cause performance problems if a large number of objects have been created. Since the garbage collector in Lua is tuned for average programs, collector may become too heavy when we can't avoid the creation of large amounts of garbage. In that situation, we can improve our application performances with custom collector adjustment for that particular case.

To check whether an element has a specific type, the API offers a family of macros and functions that return 1 if the object is compatible with the given type, and return 0 otherwise. All of them have *lua_is* prefix. Converting a temporary saved value in form of Lua object type to a appropriate C type can be done by using some of the *lua_get* functions. If conversion was unsuccessful, 0 is returned. The inverse operation, translating C values to *lua_Object* type recognizable by Lua, is performed by calling some of the functions from *lua_push* family. The Lua C API has one push function for each Lua type that can be represented in C. Every push function takes a C value that needs to be passed to Lua and translates it in background to an appropriate Lua object. Result of that conversion is placed at the top of the Lua stack. From that point, that value can be easily taken in scripts and assigned to a Lua variable, used as parameter of another Lua function etc.

Lua doesn't have debugger that could help during debugging scripts. Programmer has all necessary tools to build his own, but lack of built-in debugger makes writing scripts a lot harder. That is especially noticeable since Lua doesn't have type definition at all (each value carries its own type) and no warnings for typing errors. On the other side, clean and simple syntax is suitable for non-programmers. Lua's low learning curve, achieved by reducing API, gives designers and other non-technical personnel possibility to join in on some issues that in other situations would be typically programmer's job. Lua also supports several powerful concepts, such as returning multiple results from functions, first-class functions (language supports passing functions as arguments to other function, returning them as the values and assigning them to variables), lexical scoping (function enclosed in another function has full access to local variables of enclosing function), anonymous functions, coroutines etc. We must pay attention when creating secure user scripts, since not all functions are safe from aspect of security. List of safe functions and potentially hazard functions that could break outside sandbox is very well documented.

III. Implementation

For security reasons, certain applications have requirement for completely disabling some of Lua libraries (e.g. entire I/O module, OS module, debug module etc.) despite the fact that Lua users may try to include some of them. Even if Lua I/O library is disabled in order to prevent Lua scripts from direct file manipulation, there still may be need for some mechanism of permanently storing user values.

That can be achieved by creating separate C++ library and extending Lua with some of its functionalities. In order to easily manipulate with key/value pairs from C++ code, value storage class is implemented. This class takes care of writing value storage related data to files, reading them, deleting

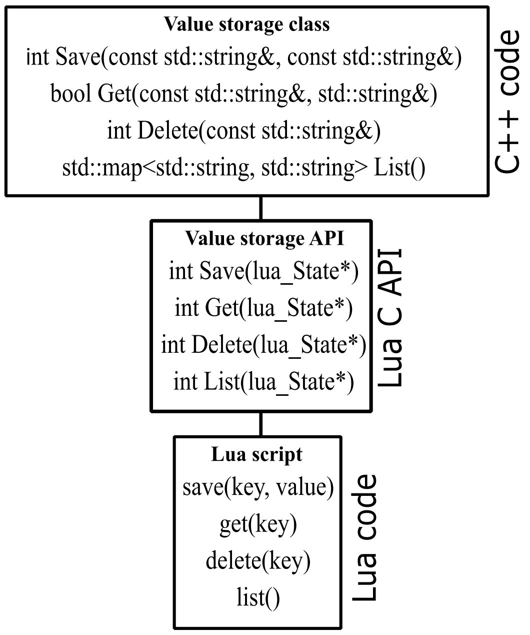

Fig. 2. Connecting Lua with the functions for controlling user settings.

them, loading saved settings between two consecutive run of the application, iterating through settings to find specific one, generating hash values for file names etc.

Lua, on the other hand, doesn't need to "worry" how things are implemented. It only wants to use certain subset of these functionalities [3]. To cover basic functionalities of manipulation with storing settings, *Save*, *Get*, *List* and *Delete* functions have been created (Fig. 2). Inside each of them, we need to take parameters from stack, call corresponding C++ function, push returning value to the stack and return number of values we are leaving on stack.

Next logical step after Lua C functions were created is to register them so that they can be used inside Lua scripts. One way to achieve that is by declaring an array of functions and their respective names. Elements of this array have type *luaL_Reg*, which is a structure with two fields: a string with function name and a function pointer [4]. The last pair in the array must be "{NULL, NULL}" to indicate its end (Fig. 3). After that, we create metatable at the top of the stack, register all functions from array into it and return that metatable. This metatable can be part of some other metatable and act as a library.

```
lua_Reg keyv_indextable[] = {
    {"save", Save},
    {"get", Get},
    {"delete", Delete},
    {"list", List},
    {NULL, NULL}
}
```

Fig. 3. Array with lua_Reg elements.

To properly call *save* function from Lua script, two string arguments need to be sent, where first argument represents key name and the second one carries value of the pair. First argument is pushed to the Lua stack at the index 1. Second argument can be found on Lua stack at index 2. Lua C *Save* function takes both of them from the stack and checks whether their types correspond to the expected ones. Function returns error code in case when arguments are not convertible to the strings. Otherwise, *Save* function from C++ value storage class is called, key/value is stored inside existing map member and saved in the file.

To read value of previously saved pair, *get* function in Lua script is called with one string argument (key of wanted pair). If Lua C *Get* function successfully converts argument to string, *Get* function from C++ value storage class is called. Map that keeps track of all saved key/value pairs is searched in order to find pair with given key. If key is found, string with value is returned. That value is pushed to the Lua stack and it can be easily used in Lua script.

In similar way, to delete some of previously saved pairs, we call *delete* function with key string as argument (that argument is the key of the pair we want to delete). All saved keys can be listed by calling *list* function that has no parameters.

To fully provide user settings manipulation, we can optionally send other arguments to *delete* and *list* functions, and in that way define theirs behavior slightly different. Second optional parameter in *delete* function is a bool data that carries information do we want to delete record by a key or by a value. Only when it's true, all key/value pairs that have value equal to value given in the first parameter will be deleted. This mode of deleting pairs is more time consuming, since we always have to iterate through and check all saved pairs, although we could have only one pair with given value, so it must be used carefully.

In *list* function, we can filter pairs by passing up to 100 keys as parameters (limitation is determined with MAXPARAMS constant). Only records with given keys will be listed. This mode of listing records is faster than the one without filtering, because we don't have to spend time to push all records to the Lua stack.

Besides simple data type, we are allowed to save associative arrays which are implemented with tables. Those arrays can be indexed with numbers or strings. In our *save* function, if second parameter represents table, we proceed with serialization of the table in order to transform it to the string. After the table has been transformed to simple data type, it can be stored, and later transformed back to the table in reverse procedure.

IV. Testing

To check performances of given solution, series of measurements are conducted and execution times have been compared. Firstly, functions from value storage class are called directly from C++ code. After that, same functions are called indirectly from Lua script.

TABLE I
Environment for tests

CPU	Intel(R) Core(TM) i7-4790K CPU @ 4.00GHz
OS type	Ubuntu 14.04 LTS 64 bit
RAM memory	2x8GB Kingston DDR3 (1600 MHz)
Lua version	Lua 5.3

For testing purposes, all pairs have simple form where key is formed by concatenating "a" character and serial number of observed pair, and value is simple a string that holds serial number (eg. {"a483","483"}). Each test is conducted on set of 100.000 key/value pairs, and results are presented bellow.

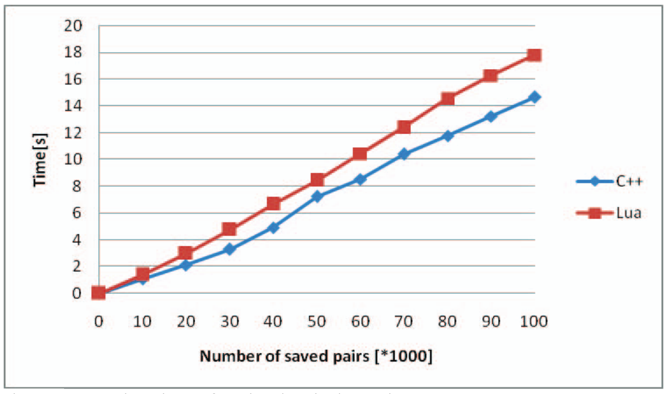

Fig. 4. Execution time of saving key/value pairs.

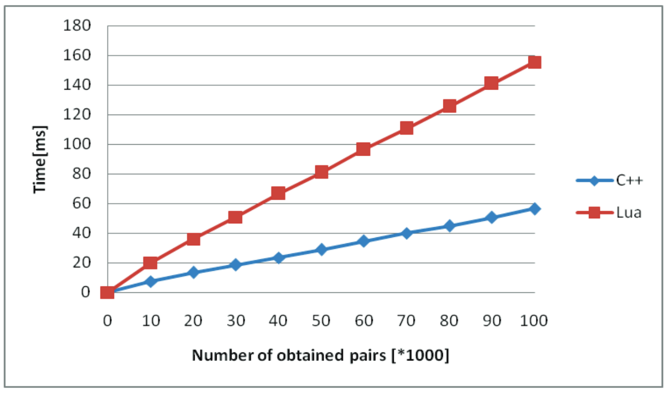

Fig. 5. Execution time of getting key/value pairs.

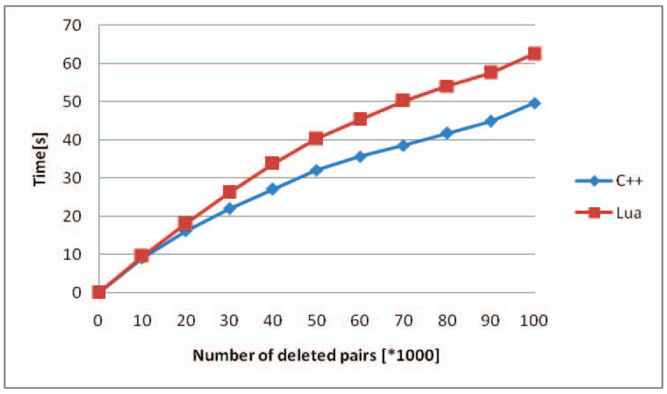

Fig. 6. Execution time of deleting key/value pairs.

In implementation of value storage class, starting with the assumption that reading value is more frequent operation than other functions, all saved pairs are also kept locally in the map, so that they can be fast retrieved when necessary. That approach makes *Get* function by far the cheapest from perspective of time-consuming (Fig 5.). On the other side, *Save* (Fig 4.) and *Delete* (Fig 3.) functions need to modify files, and that is the reason why their execution last longer.

As seen on figures 3-5, execution of value storage's functions is slightly slower when using indirect calls from Lua scripts, which is caused by additional time needed for communication between Lua and C++.

V. Conclusion

Deficiency related to longer execution time of functions from extended Lua interface can be fully compensated by significantly reducing effort required to build complex GUIs. Since Lua scripts don't need recompilation to change behavior, prototyping is very fast.

When compared to the other fast, professional key/value storages available on market, such as LevelDB and KingDB, this solution has much longer time of saving key/value pairs, but getting values is several times faster, since we already keep all records in the cache. With that in mind we can conclude that solution is especially suitable for applications where reading values is more often operation than saving.

Extensibility of Lua library gives us powerful tool to utilize advantages of different programming languages. Although *io* module has been completely disabled in Lua, limited file access is implemented by using binding library to connect C++ value storage functions and Lua scripts. In that way, user settings can be easily controlled from Lua scripts, without affecting the one of the application's primary requirements - to forbid direct file manipulation.

Acknowledgment

This work was partially supported by the Ministry of Education, Science and Technological Development of the Republic of Serbia, under grant number: TR32041.

References

[1] Roberto Ierusalimschy, "Programming in Lua", *2nd ed., Rio de Janeiro,* 2006.

[2] Roberto Ierusalimschy, Luiz Henrique de Fidueiredo, Waldemar Celes, " Lua, an extensible extension language", *PUC-Rio*, 1995.

[3] Kurt Jung, Aaron Brown, "Beginning Lua Programming", *Wiley Publishing Inc. , Indianapolis, Indiana,* 2007.

[4] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, Waldemar Celes Filho, "Lua 5.2 Reference manual", *PUC-Rio*, 2011.