Lua: Architecture and first steps

version 1.0

author: Thierry FOURNIER
contact: tfournier at arpalert dot org

HAProxy is a powerful load balancer. It embeds many options and many configuration styles in order to give a solution to many load balancing problems. However, HAProxy is not universal and some special or specific problems doesn't have solution with the native software.

This text is not a full explanation of the Lua syntax.

This text is not a replacement of the HAProxy Lua API documentation. The API documentation can be found at the project root, in the documentation directory. The goal of this text is to discover how Lua is implemented in HAProxy and using it efficiently.

However, this can be read by Lua beginners. Some examples are detailed.

Why a scripting language in HAProxy

HAProxy 1.5 makes at possible to do many things using samples, but some people wants to more combining results of samples fetches, programming conditions and loops which is not possible. Sometimes people implement these functionnalities in patches which have no meaning outside their network. These people must maintain these patches, or worse we must integrate them in the HAProxy mainstream.

Their need is to have an embedded programming language in order to no longer modify the HAProxy source code, but to write their own control code. Lua is encountered very often in the software industry, and in some open source projects. It is easy to understand, efficient, light without external dependancies, and leaves the resource control to the implementation. Its design is close to the HAProxy philosophy which uses components for what they do perfectly.

The HAProxy control block allows one to take a decision based on the comparison between samples and patterns. The samples are extracted using fetch functions easily extensible, and are used by actions which are also extensible. It seems natural to allow Lua to give samples, modify them, and to be an action target. So, Lua uses the same entities as the configuration language. This is the most natural and reliable way fir the Lua integration. So, the Lua engine allow one to add new sample fetch functions, new converter functions and new actions. These new entities can access the existing samples fetches and converters allowing to extend them without rewriting them.

The writing of the first Lua functions shows that implementing complex concepts like protocol analysers is easy and can be extended to full services. It appears that these services are not easy to implement with the HAProxy configuration model which is base on four steps: fetch, convert, compare and action. HAProxy is extended with a notion of services which are a formalisation of the existing services like stats, cli and peers. The service is an autonomous entity with a behaviour pattern close to that of an external client or server. The Lua engine inherits from this new service and offers new possibilities for writing services.

This scripting language is useful for testing new features as proof of concept. Later, if there is general interest, the proof of concept could be integrated with C language in the HAProxy core.

The HAProxy Lua integration also provides also a simple way for distributing Lua packages. The final user needs only to install the Lua file, load it in HAProxy and follow the attached documentation.

Design and technical things

Lua is integrated into the HAProxy event driven core. We want to preserve the fast processing of HAProxy. To ensure this, we implement some technical concepts between HAProxy and the Lua library.

The following paragraph also describes the interactions between Lua and HAProxy from a technical point of view.

Prerequisite

Reading the following documentation links is required to understand the current paragraph:

```
   HAProxy doc: http://cbonte.github.io/haproxy-dconv/configuration-1.6.html
   Lua API:     http://www.lua.org/manual/5.3/
   HAProxy API: http://www.arpalert.org/src/haproxy-lua-api/1.6/index.html
   Lua guide:   http://www.lua.org/pil/
```

more about Lua choice

Lua language is very simple to extend. It is easy to add new functions written in C in the core language. It not require to embed very intrusive libraries, and we do not change compilation processes.

The amount of memory consumed can be controlled, and the issues due to lack of memory are perfectly caught. The maximum amount of memory allowed for the Lua processes is configurable. If some memory is missing, the current Lua action fails, and the HAProxy processing flow continues.

Lua provides a way for implementing event driven design. When the Lua code wants to do a blocking action, the action is started, it executes non blocking operations, and returns control to the HAProxy scheduler when it needs to wait for some external event.

The Lua process can be interrupted after a number of instructions executed. The Lua execution will resume later. This is a useful way for controlling the execution time. This system also keeps HAProxy responsive. When the Lua execution is interrupted, HAProxy accepts some connections or transfers pending data. The Lua execution does not block the main HAProxy processing, except in some cases which we will see later.

Lua function integration

The Lua actions, sample fetches, converters and services are integrated in HAProxy with "register_*" functions. The register system is a choice for providing HAProxy Lua packages easily. The register system adds new sample fetches, converters, actions or services usable in the HAProxy configuration file.

The register system is defined in the "core" functions collection. This collection is provided by HAProxy and is always available. Below, the list of these functions:

 - core.register_action()
 - core.register_converters()
 - core.register_fetches()

- core.register_init()
- core.register_service()
- core.register_task()

These functions are the execution entry points.

HTTP action must be used for manipulating HTTP request headers. This action can not manipulates HTTP content. It is dangerous to use the channel manipulation object with an HTTP request in an HTTP action. The channel manipulation can transform a valid request in an invalid request. In this case, the action will never resume and the processing will be frozen. HAProxy discards the request after the reception timeout.

Non blocking design

HAProxy is an event driven software, so blocking system calls are absolutely forbidden. However, the Lua allows to do blocking actions. When an action blocks, HAProxy is waiting and do nothing, so the basic functionalities like accepting connections or forwarding data are blocked while the end of the system call. In this case HAProxy will be less responsive.

This is very insidious because when the developer tries to execute its Lua code with only one stream, HAProxy seems to run fine. When the code is used with production stream, HAProxy encounters some slow processing, and it cannot hold the load.

However, during the initialisation state, you can obviously using blocking functions. There are typically used for loading files.

The list of prohibited standard Lua functions during the runtime contains all that do filesystem access:

- os.remove()
- os.rename()
- os.tmpname()
- package.*()
- io.*()
- file.*()

Some other functions are prohibited:

- os.execute(), waits for the end of the required execution blocking HAProxy.

- os.exit(), is not really dangerous for the process, but its not the good way for exiting the HAProxy process.

- print(), writes data on stdout. In some cases these writes are blocking, the best practice is reserving this call for debugging. We must prefer to use core.log() or TXN.log() for sending messages.

Some HAProxy functions have a blocking behaviour pattern in the Lua code, but there are compatible with the non blocking design. These functions are:

- All the socket class
- core.sleep()

Responsive design

HAProxy must process connexions accept, forwarding data and processing timeouts as soon as possible. The first thing is to believe that a Lua script with a long execution time should impact the expected responsive behaviour.

It is not the case, the Lua script execution are regularly interrupted, and

HAProxy can process other things. These interruptions are exprimed in number of Lua instructions. The number of interruptions between two interrupts is configured with the following "tune" option:

    tune.lua.forced-yield <nb>

The default value is 10 000. For determining it, I ran benchmark on my laptop. I executed a Lua loop between 10 seconds with differents values for the "tune.lua.forced-yield" option, and I noted the results:

configured instructions between two forced yields	Number of loops executed in milions
10	160
500	670
1000	680
5000	700
7000	700
8000	700
9000	710 <- ceil
10000	710
100000	710
1000000	710

The result showed that from 9000 instructions between two interrupt, we reached a ceil, so the default parameter is 10 000.

When HAProxy interrupts the Lua processing, we have two states possible:

- Lua is resumable, and it returns control to the HAProxy scheduler,
- Lua is not resumable, and we just check the execution timeout.

The second case occurs if it is required by the HAProxy core. This state is forced if the Lua is processed in a non resumable HAProxy part, like sample fetches or converters.

It occurs also if the Lua is non resumable. For example, if some code is executed through the Lua pcall() function, the execution is not resumable. This is explained later.

So, the Lua code must be fast and simple when is executed as sample fetches and converters, it could be slow and complex when is executed as actions and services.

Execution time

The Lua execution time is measured and limited. Each group of functions have its own timeout configured. The time measured is the real Lua execution time, and not the difference between the end time and the start time. The groups are:

- main code and init are not submitted to the timeout,
- fetches, converters and action have a default timeout of 4s,
- task, by default does not have timeout,
- service have a default timeout of 4s.

The corresponding tune option are:

- tune.lua.session-timeout (fetches, converters and action)
- tune.lua.task-timeout   (task)
- tune.lua.service-timeout (services)

The tasks does not have a timeout because it runs in background along the

HAProxy process life.

For example, if an Lua script is executed during 1,1s and the script executes a sleep of 1 second, the effective measured running time is 0,1s.

This timeout is useful for preventing infinite loops. During the runtime, it should never triggered.

The stack and the coprocess

The Lua execution is organized around a stack. Each Lua action, even out of the effective execution, affects the stack. HAProxy integration uses one main stack, which is common for all the process, and a secondary one used as coprocess. After the initialization, the main stack is no longer used by HAProxy, except for global storage. The second type of stack is used by all the Lua functions called from different Lua actions declared in HAProxy. The main stack permits to store coroutines pointers, and some global variables.

Do you want to see an example of how seems Lua C development around a stack ? Some examples follows. This first one, is a simple addition:

```
lua_pushnumber(L, 1)
lua_pushnumber(L, 2)
lua_arith(L, LUA_OPADD)
```

Its easy, we push 1 on the stack, after, we push 2, and finally, we perform an addition. The two top entries of the stack are added, poped, and the result is pushed. It is a classic way with a stack.

Now an example for constructing array and objects. Its little bit more complicated. The difficult consist to keep in mind the state of the stack while we write the code. The goal is to create the entity described below. Note that the notation "*1" is a metatable reference. The metatable will be explained later.

```
name*1 = {
   [0] = <userdata>,
}

*1 = {
   "__index" = {
      "method1" = <function>,
      "method2" = <function>
   }
   "__gc" = <function>
}
```

Let's go:

```
lua_newtable()               // The "name" table
lua_newtable()               // The metatable *1
lua_pushstring("__index")
lua_newtable()               // The "__index" table
lua_pushstring("method1")
lua_pushfunction(function)
lua_settable(-3)             // -3 is an index in the stack. insert method1
lua_pushstring("method2")
lua_pushfunction(function)
lua_settable(-3)             // insert method2
lua_settable(-3)             // insert "__index"
lua_pushstring("__gc")
lua_pushfunction(function)
lua_settable()               // insert "__gc"
lua_setmetatable(-1)         // attach metatable to "name"
```



```
lua_pushnumber(0)
lua_pushuserdata(userdata)
lua_settable(-3)
lua_setglobal("name")
```

So, coding for Lua in C, is not complex, but it needs some mental gymnastic.

The object concept and the HAProxy format

The objects seems to not be a native concept. An Lua object is a table. We can note that the table notation accept three forms:

```
1. mytable["entry"](mytable, "param")
2. mytable.entry(mytable, "param")
3. mytable:entry("param")
```

These three notation have the same behaviour pattern: a function is executed with the itself table as first parameter and string "param" as second parameter The notation with [] is commonly used for storing data in a hash table, and the dotted notation is used for objects. The notation with ":" indicates that the first parameter is the element at the left of the symbol ":".

So, an object is a table and each entry of the table is a variable. A variable can be a function. These are the first concepts of the object notation in the Lua, but it is not the end.

With the objects, we usually expect classes and inheritance. This is the role of the metable. A metable is a table with predefined entries. These entries modify the default behaviour of the table. The simplest example is the "__index" entry. If this entry exists, it is called when a value is requested in the table. The behaviour is the following:

1 - looks in the table if the entry exists, and if it the case, return it

2 - looks if a metatable exists, and if the "__index" entry exists

3 - if "__index" is a function, execute it with the key as parameter, and returns the result of the function.

4 - if "__index" is a table, looks if the requested entry exists, and if exists, return it.

5 - if not exists, return to step 2

The behaviour of the point 5 represents the inheritance.

In HAProxy all the provided objects are tables, the entry "[0]" contains private data, there are often userdata or lightuserdata. The matatable is registered in the global part of the main Lua stack, and it is called with the case sensitive class name. A great part of these class must not be used directly because it requires an initialisation using the HAProxy internal structs.

The HAProxy objects uses unified conventions. An Lua object is always a table. In most cases, an HAProxy Lua object need some private data. These are always set in the index [0] of the array. The metatable entry "__tostring" returns the object name.

The Lua developer can add entries to the HAProxy object. He just works carefully and prevent to modify the index [0].

Common HAproxy objects are:

- TXN   : manipulates the transaction between the client and the server
- Channel   : manipulates proxified data between the client and the server

```
 - HTTP       : manipulates HTTP between the client and the server
 - Map        : manipulates HAProxy maps.
 - Fetches    : access to all HAProxy sample fetches
 - Converters : access to all HAProxy sample converters
 - AppletTCP  : process client request like a TCP server
 - AppletHTTP : process client request like an HTTP server
 - Socket     : establish tcp connection to a server (ipv4/ipv6/socket/ssl/...)
```

The garbage collector and the memory allocation

Lua doesn't really have a global memory limit, but HAProxy implements it. This permits to control the amount of memory dedicated to the Lua processes. It is specially useful with embedded environments.

When the memory limit is reached, HAProxy refuses to give more memory to the Lua scripts. The current Lua execution is terminated with an error and HAProxy continue its processing.

The max amount of memory is configured with the option:

```
   tune.lua.maxmem
```

As many other script languages, Lua uses a garbage collector for reusing its memory. The Lua developper can work without memory preoccupation. Usually, the garbage collector is controlled by the Lua core, but sometimes it will be useful to run when the user/developer requires. So the garbage collector can be called from C part or Lua part.

Sometimes, objects using lightuserdata or userdata requires to free some memory block or close filedescriptor not controlled by the Lua. A dedicated garbage collection function is providedthrought the metatable. It is referenced with the special entry "__gc".

Generally, in HAProxy, the garbage collector does this job without any intervention. However some object uses a great amount of memory, and we want to release as quick as possible. The problem is that only the GC knows if the object is in use or not. The reason is simple variable containing objects can be shared between coroutines and the main thread, so an object can used everywhere in HAProxy.

The only one example is the HAProxy sockets. These are explained later, just for understanding the GC issues, a quick overview of the socket follows. The HAProxy socket uses an internal session and stream, these sessions uses resources like memory and file descriptor and in some cases keeps a socket open while it is no loner used by Lua.

If the HAProxy socket is used, we forcing a garbage collector cycle after the end of each function using HAProxy socket. The reason is simple: if the socket is no longer used, we want to close the connection quickly.

A special flag is used in HAProxy indicating that a HAProxy socket is created. If this flag is set, a full GC cycle is started after each Lua action. This is not free, we loose about 10% of performances, but it is the only way for closing sockets quickly.

The yield concept / longjmp issues

The "yield" is an action which does some Lua processing in pause and give back the hand to the HAProxy core. This action is do when the Lua needs to wait about data or other things. The most basically example is the sleep() function. In a event driven software the code must not process blocking systems call, so the sleep blocks the software between a lot of time. In HAProxy, an Lua sleep does a yield, and ask to the scheduler to be waked up in a required sleep time.

Meanwhile, the HAProxy scheduler dos other things, like accepting new connection or forwarding data.

A yield is also executed regularly, after a lot of Lua instruction processed. This yield permits to control the effective execution time, and also give back the hand to the haproxy core. When HAProxy finish to process the pending jobs, the Lua execution continue.

This special "yield" uses the Lua "debug" functions. Lua provides a debug method called "lua_sethook()" which permits to interrupt the execution after some configured condition and call a function. This condition used in HAProxy is a number of instruction processed and when a function returns. The function called controls the effective execution time, and if it is possible send a "yield".

The yield system is based on a couple setjmp/longjmp. In brief, the setjmp() stores a stack state, and the longjmp restores the stack in its state which had before the last Lua execution.

Lua can immediately stop is execution if an error occurs. This system uses also the longjmp system. In HAProxy, we try to use this sytem only for unrecoverable errors. Maybe some trivial errors targets an exception, but we try to remove it.

It seems that Lua uses the longjmp system for having a behaviour like the java try / catch. We can use the function pcall() to executes some code. The function pcall() run a setjmp(). So, if any error occurs while the Lua code execution, the flow immediately return from the pcall() with an error.

The big issue of this behaviour is that we cannot do a yield. So if some Lua code executes a library using pcall for catching errors, HAProxy must be wait for the end of execution without processing any accept or any stream. The cause is the yield must be jump to the root of execution. The intermediate setjmp() avoid this behaviour.

```
   HAproxy start Lua execution
     + Lua puts a setjmp()
        + Lua executes code
        + Some code is executed in a pcall()
           + pcall() puts a setjmp()
              + Lua executes code
                 + A yield is require for a sleep function
                   it cannot be jumps to the Lua root execution.
```

Another issue with the processing of strong errors is the manipulation of the Lua stack outside of an Lua processing. If one of the functions called occurs a strong error, the default behaviour is an abort(). It is not acceptable when HAProxy is in runtime mode. The Lua documentation propose to use another setjmp/longjmp to avoid the abort(). The goal is to puts a setjmp between manipulating the Lua stack and using an alternative "panic" function which jumps to the setjmp() in error case.

All of these behaviours are very dangerous for the stability, and the internal HAProxy code must be modified with many precautions.

For preserving a good behaviour of HAProxy, the yield is mandatory. Unfortunately, some HAProxy part are not adapted for resuming an execution after a yield. These part are the sample fetches and the sample converters. So, the Lua code written in these parts of HAProxy must be quickly executed, and can not do actions which require yield like TCP connection or simple sleep.

HAproxy socket object

The HAProxy design is optimized for the data transfers between a client and a server, and processing the many errors which can occurs during these exchanges. HAProxy is not designed for having a third connection established to a third party server.

The solution consist to puts the main stream in pause waiting for the end of the exchanges with the third connection. This is completed by a signal between internal tasks. The following graph shows the HAProxy Lua socket:

```
                      +--------------------+
                      | Lua processing     |
 ------------------\  | creates socket     | ------------------\
  incoming request  > | and puts the       |  Outgoing request  >
 ------------------/  | current processing | ------------------/
                      | in pause waiting   |
                      | for TCP applet     |
                      +-----------------+--+
                        ^               |
                        |               |
                        | signal        | read / write
                        |               | data
                        |               |
          +-------------+---------+     v
          | HAProxy internal      +----------------+
          | applet send signals   |                |
          | when data is received |                | -------------------\
          | or some room is       |  Attached I/O  |  Client TCP stream  >
          | available             |  Buffers       | -------------------/
          +--------------------+--+                |
                               |                   |
                               +-------------------+
```

A more detailed graph is available in the "doc/internals" directory.

The HAProxy Lua socket uses a full HAProxy session / stream for establishing the connection. This mechanism provides all the facilities and HAProxy features, like the SSL stack, many socket type, and support for namespaces. Technically it support the proxy protocol, but there are no way to enable it.

How compiling HAProxy with Lua
==============================

HAProxy 1.6 requires Lua 5.3. Lua 5.3 offers some features which makes easy the integration. Lua 5.3 is young, and some distros do not distribute it. Luckily, Lua is a great product because it does not require exotic dependencies, and its build process is really easy.

The compilation process for linux is easy:

```
 - download the source tarball
   wget http://www.lua.org/ftp/lua-5.3.1.tar.gz

 - untar it
   tar xf lua-5.3.1.tar.gz

 - enter the directory
   cd lua-5.3.1

 - build the library for linux
   make linux

 - install it:
   sudo make INSTALL_TOP=/opt/lua-5.3.1
```


HAProxy builds with your favourite options, plus the following options for embedding the Lua script language:

```
 - download the source tarball
   wget http://www.haproxy.org/download/1.6/src/haproxy-1.6.2.tar.gz

 - untar it
   tar xf haproxy-1.6.2.tar.gz

 - enter the directory
   cd haproxy-1.6.2

 - build HAProxy:
   make TARGET=linux \
        USE_DL=1 \
        USE_LUA=1 \
        LUA_LIB=/opt/lua-5.3.1/lib \
        LUA_INC=/opt/lua-5.3.1/include

 - install it:
   sudo make PREFIX=/opt/haproxy-1.6.2 install
```

First steps with Lua
====================

Now, its time to using Lua in HAProxy.

Start point

The HAProxy global directive "lua-load <file>" allow to load an lua file. This is the entry point. This load become during the configuration parsing, and the Lua file is immediately executed.

All the register_*() function must be called at this time because there are used just after the processing of the global section, in the frontend/backend/listen sections.

The most simple "Hello world !" is the following line a loaded Lua file:

```
   core.Alert("Hello World !");
```

It display a log during the HAProxy startup:

```
   [alert] 285/083533 (14465) : Hello World !
```

Default path and libraries

Lua can embed some libraries. These libraries can be included from different paths. It seems that Lua doesn't like subdirectories. In the following example, I try to load a compiled library, so the first line is Lua code, the second line is an 'strace' extract proving that the library was opened. The next lines are the associated error.

```
   require("luac/concat")

   open("./luac/concat.so", O_RDONLY|O_CLOEXEC) = 4

        [ALERT] 293/175822 (22806) : parsing [commonstats.conf:15] : lua runtime
        error: error loading module 'luac/concat' from file './luac/concat.so':
                ./luac/concat.so: undefined symbol: luaopen_luac/concat
```

Lua tries to load the C symbol 'luaopen_luac/concat'. When Lua tries to open a library, it tries to execute the function associated to the symbol "luaopen_<libname>".

The variable "<libname>" is defined using the content of the variable "package.cpath" and/or "package.path". The default definition of the "package.cpath" (on my computer is) variable is:

```
/usr/local/lib/lua/5.3/?.so;/usr/local/lib/lua/5.3/loadall.so;./?.so
```

The "<libname>" is the content which replaces the symbol "<?>". In th previous example, its "luac/concat", and obviously the Lua core try to load the function associated with the symbol "luaopen_luac/concat".

My conclusion is that Lua doesn't support subdirectories. So, for loading libraries in subdirectory, it must fill the variable with the name of this subdirectory. The extension .so must disappear, otherwise Lua try to execute the function associated with the symbol "luaopen_concat.so". The following syntax is correct:

```
package.cpath = package.cpath .. ";./luac/?.so"
     require("concat")
```

First useful example

```
core.register_fetches("my-hash", function(txn, salt)
   return txn.sc:sdbm(salt .. txn.sf:req_fhdr("host") .. txn.sf:path() .. txn.sf:sr…
end)
```

You will see that these 3 line can generate a lot of explanations :)

Core.register_fetches() is executed during the processing of the global section by the HAProxy configuration parser. A new sample fetch is declared with name "my-hash", this name is always prefixed by "lua.". So this new declared sample fetch will be used calling "lua.my-hash" in the HAProxy configuration file.

The second parameter is an inline declared anonymous function. Note the closed parenthesis after the keyword "end" which end the function. The first parameter of these anonymous function is "txn". It an object of class TXN. It provides access functions. The second parameter is an arbitrary value provided by the HAProxy configuration file. This parameter is optional, the developer must check if its present.

The anonymous function registration is executed when the HAProxy backend or frontend configuration references the sample fetch "lua.my-hash".

This example can writed with an other style, like below:

```
function my_hash(txn, salt)
   return txn.sc:sdbm(salt .. txn.sf:req_fhdr("host") .. txn.sf:path() .. txn.sf:sr…
end

core.register_fetches("my-hash", my_hash)
```

This second form is clearer, but the first one is compact.

The operator ".." is a string concatenation. If one of the two operands are not a string, an error occurs and the execution is immediately stopped. This is important to keep in mind for the following things.

Now I write the example on more than one line. Its an easiest way for commenting the code:


```
 1. function my_hash(txn, salt)
 2.    local str = ""
 3.    str = str .. salt
 4.    str = str .. txn.sf:req_fhdr("host")
 5.    str = str .. txn.sf:path()
 6.    str = str .. txn.sf:src()
 7.    local result = txn.sc:sdbm(str, 1)
 8.    return result
 9. end
10.
11. core.register_fetches("my-hash", my_hash)
```

local

The first keyword is "local". This is a really important keyword. You must understand that the function "my_hash" will be called for each HAProxy request using the declared sample fetch. So, this function can be executed many times in parallel.

By default, Lua uses global variables. so in this example, il the variable "str" is declared without the keyword "local", it will be shared by all the parallel executions of the function and obviously, the content of the requests will be shared.

This warning is very important. I tried to write useful Lua code like a rewrite of the statistics page, and its very hard to thing to declare each variable as "local".

I guess than this behaviour will be the cause of many trouble on the mailing list.

str = str ..

Now a parenthesis about the form "str = str ..". This form allow to do string concatenations. Remember that Lua uses a garbage collector, so what happens when we do "str = str .. 'another string'" ?

```
str = str .. "another string"
^     ^   ^  ^
1     2   3  4
```

Lua execute first the concatenation operator (3), it allocates memory for the resulting string and fill this memory with the concatenation of the operands 2 and 4. Next, it free the variable 1, now the old content of 1 can be garbage collected. and finally, the new content of 1 is the concatenation.

what the matter ? when we do this operation many times, we consume a lot of memory, and the string data is duplicated and move many times. So, this practice is expensive in execution time and memory consumption.

There are easy ways to prevent this behaviour. I guess that a C binding for concatenation with chunks will be available ASAP (it is already written). I do some benchmarks. I compare the execution time of 1 000 times, 1 000 concatenation of 10 bytes written in pure Lua and with a C library. The result is 10 times faster in C (1s in Lua, and 0.1s in C).

txn

txn is an HAProxy object of class TXN. The documentation is available in the HAProxy Lua API reference. This class allow the access to the native HAProxy sample fetches and converters. The object txn contains 2 members dedicated to the sample fetches and 2 members dedicated to the converters.

The sample fetches members are "f" (as sample-Fetch) and "sf" (as String sample-Fetch). These two members contains exactly the same functions. All the HAProxy native sample fetches are available, obviously, the Lua registered sample fetches are not available. Unfortunately, HAProxy sample fetches names are not compatible with the Lua function names, and they are renames. The rename convention is simple, we replace all the '.', '+' and '-' by '_'. The '.' is the object member separator, and the "-" and "+" is math operator.

Now, that I'm writing this article, I known the Lua better than I wrote the sample-fetches wrapper. The original HAProxy sample-fetches name should be used using alternative manner to call an object member, so the sample-fetch "req.fhdr" (actually renamed req_fhdr") is should be used like this:

```
txn.f["req.fhdr"](txn.f, ...)
```

However, I think that this form is not elegant.

The "s" collection return a data with a type near to the original returned type. A string return an Lua string, an integer returns an Lua integer and an IP address returns an Lua string. Sometime the data is not or not yet available, in this case it returns the Lua nil value.

The "sf" collection guarantee that a string will be always returned. If the data is not available, an empty string is returned. The main usage of these collection is to concatenate the returned sample-fetches without testing each function.

The parameters of the sample-fetches are according with the haproxy documentation.

The converters runs exactly with the same manner as the sample fetches. The only one difference is that the fist parameter is the converter entry element. The "c" collection returns a precise result, and the "sc" collection returns always a string.

The sample-fetches used in the example function are "txn.sf:req_fhdr()", "txn.sf:path()" and "txn.sf:src()". The converter are "txn.sc:sdbm()". The same function with the "s" collection of sample-fetches and the "c" collection of converter should be written like this:

```
 1. function my_hash(txn, salt)
 2.    local str = ""
 3.    str = str .. salt
 4.    str = str .. tostring(txn.f:req_fhdr("host"))
 5.    str = str .. tostring(txn.f:path())
 6.    str = str .. tostring(txn.f:src())
 7.    local result = tostring(txn.c:sdbm(str, 1))
 8.    return result
 9. end
10.
11. core.register_fetches("my-hash", my_hash)
```

tostring
~~~~~~~~

The function tostring ensure that its parameter is returned as a string. If the parameter is a table or a thread or anything that will not have any sense as a string, a form like the typename followed by a pointer is returned. For example:

```
t = {}
print(tostring(t))
```

returns:

```
table: 0x15facc0
```


For objects, if the special function __tostring() is registered in the attached metatable, it will be called with the table itself as first argument. The HAProxy objects returns its own type.

About the converters entry point

In HAProxy, a converter is a stateless function that takes a data as entry and returns a transformation of this data as output. In Lua it is exactly the same behaviour.

So, the registered Lua function doesn't have any special parameters, juste a variable as input which contains the value to convert, and it must return data.

The data required as input by the Lua converter is a string. So HAProxy will always provide a string as input. If the native sample fetch is not a string it will ne converted in best effort.

The returned value will have anything type, it will be converted as sample of the near HAProxy type. The conversion rules from Lua variables to HAProxy samples are:

```
Lua        | HAProxy sample types
-----------+---------------------
"number"   | "sint"
"boolean"  | "bool"
"string"   | "str"
"userdata" | "bool" (false)
"nil"      | "bool" (false)
"table"    | "bool" (false)
"function" | "bool" (false)
"thread"   | "bool" (false)
```

The function used for registering a converter is:

```
core.register_converters()
```

The task entry point

The function "core.register_task(fcn)" executes once the function "fcn" when the scheduler starts. This way is used for executing background task. For example, you can use this functionnality for periodically checking the health of an other service, and giving the result to each proxy needing it.

The task is started once, if you want periodic actions, you can use the "core.sleep()" or "core.msleep()" for waiting the next runtime.

Storing Lua variable between function in the same session

All the functions registered as action or sample fetch can share an Lua context. This context is a memory zone in the stack. sample fetch and action uses the same stack, so both can access to the context.

The context is accessible via the function get_priv and set_priv provided by an object of class TXN. The value given to set_priv replaces the current stored value. This value can be a table, it is useful if a lot of data can be shared.

If the value stored is a table, you can add or remove entries from the table without storing again the new table. Maybe an example will be clearer:

```
local t = {}
txn:set_priv(t)
```
~~~~~~~~

```
t["entry1"] = "foo"
t["entry2"] = "bar"

-- this will display "foo"
print(txn:get_priv()["entry1"])
```

HTTP actions

... comming soon ...

Lua is fast, but my service require more execution speed

We can wrote C modules for Lua. These modules must run with HAProxy while they are compliant with the HAProxy Lua version. A simple example is the "concat" module.

It is very easy to write and compile a C Lua library, however, I don't see documentation about this process. So the current chapter is a quick howto.

The entry point

The entry point is called "luaopen_<name>", where <name> is the name of the ".so" file. An hello world is like this:

```
#include <stdio.h>
#include <lua.h>
#include <lauxlib.h>

int luaopen_mymod(lua_State *L)
{
   printf("Hello world\n");
   return 0;
}
```

The build

The compilation of the source file requires the Lua "include" directory. The compilation and the link of the object file requires the -fPIC option. Thats all.

```
cc -I/opt/lua/include -fPIC -shared  -o mymod.so mymod.c
```

Usage

You can load this module with the following Lua syntax:

```
require("mymod")
```

When you start HAProxy, this module just print "Hello world" when its loaded. Please, remember that HAProxy doesn't allow blocking method, so if you write a function doing filesystem access or synchronous network access, all the HAProxy process will fail.