Scripting the Linux Routing Table with Lua

Lourival Vieira Neto, Marcel Moura, Ana Lúcia de Moura and Roberto Ierusalimschy
Ring-0 Networks and Departamento de Informática, PUC-Rio
Rio de Janeiro, Brazil
lourival.neto@gmail.com, marcel.stanley@gmail.com, analuciadm@gmail.com, roberto@inf.puc-rio.br

Abstract

Scriptable Operating System is a design concept based on the idea that operating systems should allow users to write scripts to tailor the system to their needs. This paper presents how Lunatik, a kernel-scripting framework for Linux, has evolved over time in terms of flexibility and extensibility, from being an environment for scripting kernel subsystems, to become a much richer scripting environment, suitable for extending the Linux operating system itself. It then details how Lunatik's modularized architecture, along with its enriched set of extension libraries may be used for implementing a kernel-level adaptive routing service, which allows the system to monitor the reliability of a network route and adjust its routing table dynamically in case of failure by using an alternative route, previously chosen by an alternative gateway election protocol.

Keywords

Lua, Kernel Scripting, Network Routing

1 Introduction

Lunatik is a framework for scripting the Linux kernel with the Lua programming language [7, 8], based on the design concept of Scriptable Operating Systems [13]. This framework was previously used for scripting the Netfilter and XDP subsystems [11], allowing users to create complex network filters using Lua. For instance, it has been used for filtering L7 traffic such as HTTP and HTTPS.

In this paper, we present the evolution of this framework and its usage on implementing a kernel-level adaptive routing service. This service allows the system to monitor the reliability of a network route and adjust its routing table dynamically in case of failure. For instance, a home router might use such service to fall back to the cellular network using a mobile phone as an alternative gateway.

For implementing this adaptive routing service, we relied on Lunatik libraries for binding kernel facilities, such as netdevice notification chain, kthread, socket, RCU (read-copy-update) and FIB (Forwarding Information Base). This service also implements a protocol that allows nodes to advertise themselves as alternative gateways and the router to notify a node that it has been elected as the new gateway (or that the network has been reestablished and using that node as gateway is no longer necessary).

The rest of this paper is organized as follows. Section 2 discusses the evolution and current state of the Lunatik framework. Section 3 presents our kernel-level adaptive routing service. Finally, Section 4 presents our final remarks.

2 The Lunatik Framework

When first introduced, Lunatik provided an environment for scripting the Linux kernel that consisted of the Lua interpreter properly embedded in the kernel, a common user interface — a device driver — for dynamically loading scripts, and an API for using the scripting environment from kernel subsystems [13]. To make subsystems scriptable, kernel developers created *bindings* that allowed functions and data structures to be shared between the kernel and Lua scripts. Figure 1 illustrates Lunatik's original architecture.

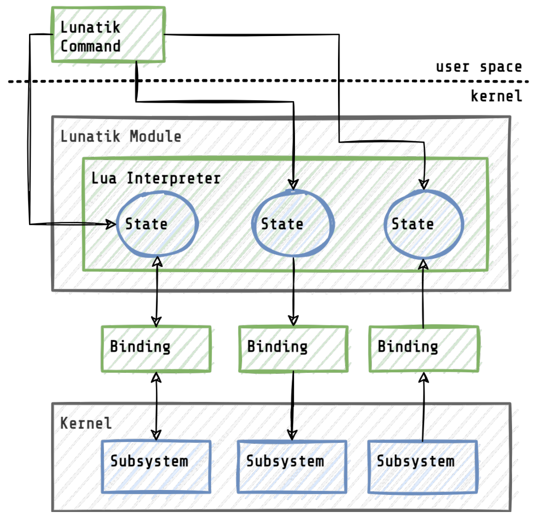

Figure 1: Lunatik's original architecture

Lunatik's initial implementation provided a suitable environment for scripting kernel subsystems that could execute in process context, such as CPUfreq [13]. However, it was not adequate for subsystems that involve stricter execution requirements, such as those running in software interrupt context, as is typically the case with networking subsystems. To meet those requirements, the following version of Lunatik dropped its common user interface and APIs. Developers then had to use the plain Lua C API for embedding the Lua interpreter in their kernel subsystems, and needed to provide their own mechanism for loading scripts from user space. Based on this Lunatik version, NFLua [4, 11] and XDPLua [12, 11], scripting environments for Netfilter [10] and XDP (eXpress Data Path) [6] respectively, used dedicated Netlink sockets [1] for loading scripts from user space. Moreover, NFLua allowed users to create new Lua execution states dynamically using its Netlink socket and XPDLua created one Lua state per CPU core in advance. In both cases, the specific scripting environments were responsible for synchronizing concurrent accesses to their Lua states.

The current version of the Lunatik framework provides a richer scripting environment, with improved APIs and libraries for both extending the kernel and the framework itself. Its new implementation, presented in Figure 2, is now modular and partially developed in Lua.

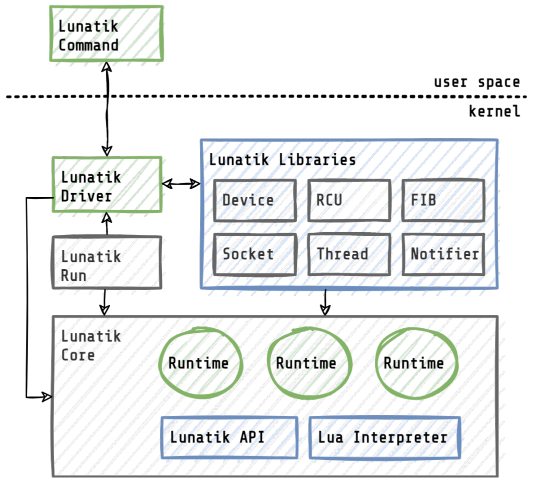

Figure 2: Modular implementation of the Lunatik framework

We have split the original Lunatik kernel module into several kernel modules and kernel-level Lua scripts. The Lunatik core module contains the Lua interpreter modified to run in the kernel and APIs for creating and managing *Lunatik runtimes*. This new data structure wraps a Lua execution state, a lock type (e.g., *mutex* or *spinlock*), and a flag to indicate if the control execution flow can sleep. Thus, the framework can now use appropriate synchronization and memory allocation primitives for scripts running both in process and software interrupt context.

The Lunatik run module implements a base scripting environment for the kernel as a whole, instead of providing support for creating specific environments for each target subsystem. It creates a special runtime for running a kernel-level Lua script that is responsible for bootstrapping the framework. This kernel-level script implements a device driver, the Lunatik driver, that provides the framework interface with the user space, allowing the Lunatik command-line tool to create new runtimes to load and execute Lua scripts for extending the kernel. The new Lunatik APIs permit loading files synchronously and directly from the file system, enabling us to simplify the management of Lua states. This feature also allows user-defined Lua scripts to be modularized and implemented using multiple files.

Lunatik's newest version also introduces a new set of Lua extension libraries that enriches the scripting environment. The Device library permits implementing character device drivers in Lua. The Socket library provides support for kernel networking handling by offering APIs to send and receive messages using the underlying kernel socket API. The Thread library gives access to kernel thread facilities [2], and allows spawning new kernel threads with Lunatik runtimes. Through the Notifier library, a script can register callback functions on keyboard and netdevice notifier chains [3]. The RCU library provides an interface for the kernel Read-copy update (RCU) synchronization mechanism [9]. It allows creating and sharing hash tables among Lunatik runtimes. The FIB library gives access to the kernel Forwarding Information Base (FIB), allowing scripts to add and remove routing rules.

3 Adaptive Routing Service

Using the current version of the Lunatik framework, we have developed a kernel-level service to monitor network interfaces and adjust the system routing table dynamically in case of link failure. This adaptive routing service can be used to fall back to an alternative gateway when the default gateway interface is down. For instance, a home router may rely on this service to fall back to the cellular network using a mobile phone as the alternative gateway.

Our adaptive routing service consists of two components, implemented by Lua scripts: a service daemon and a notification handler. The service daemon is responsible for listening to alternative gateway candidates. The notification handler is used for detecting when the network interface of the default gateway is down.

We have developed a simplified network protocol to support this adaptive routing service. This protocol allows nodes to advertise themselves as alternative gateways and the router to notify a node to enable or disable itself as the elected alternative gateway.

This adaptive routing protocol works as follows. Nodes periodically send UDP datagrams to the router indicating the latency of their alternative route in microseconds. Whenever the router receives a datagram from a node, it checks if the current elected gateway has expired or if the node has a better latency than the current elected gateway. If one of these conditions is true, the router elects the node as the new alternative gateway. Whenever the router detects a link failure, it updates its routing table, setting a predefined IP address as the default gateway, and sends a TCP message to the alternative gateway notifying it to enable its alternative route. Whenever the router detects that the link has been reestablished, it restores its routing table and sends a TCP message to the alternative gateway notifying it to disable its alternative route.

To run the adaptive routing service, first we need to create a custom routing table for the alternative route. This table must indicate the predefined IP address as the default gateway. We can do this using the regular iproute2 utilities. Then, we need to run the service using the Lunatik driver. We can do this by calling the Lunatik command-line tool, indicating the Lua script file that implements such service, as follows:

```
lunatik run failover/service
```

The Lunatik command writes the following Lua chunk in the Lunatik device to create a new runtime to run the Lua script file (installed on `/lib/modules/lua/failover/service.lua`).

```
local script = "failover/service"
lunatik.__runtimes[script] = lunatik.runtime(script)
```

Figure 3 shows the Lua script loaded to execute the service.

```
1  local lunatik  = require("lunatik")
2  local thread   = require("thread")
3  local notifier = require("notifier")
4  local inet     = require("socket.inet")
5  local fib      = require("fib")
6  local rcu      = require("rcu")
7  local conf     = require("failover.conf")
8
9  local netdev = notifier.netdev
10 local notify = notifier.notify
11
12 local elected = rcu.table(8)
13 rcu.publish("elected", elected)
14
15 local function alt_gateway(state)
16   local client <close> = inet.tcp()
17   client:connect(elected.ip, conf.port)
18   client:send(state)
19 end
20
21 local state = "down"
22 local function failover(event, ifname)
23   if ifname ~= conf.wan then return notify.OK end
24
25   if event == netdev.UP then
26     state = "up"
27   elseif event == netdev.CHANGE then
28     if state == "up" then
29       state = "down"
30       fib.newrule(conf.table_id, conf.priority)
31       alt_gateway("up")
32     else
33       state = "up"
34       fib.delrule(conf.table_id, conf.priority)
35       alt_gateway("down")
36     end
37   end
38 end
39
40 notifier.netdevice(failover)
41
42 local daemon = "failover/daemon"
43 thread.run(lunatik.runtime(daemon), daemon)
```

Figure 3: Adaptive Routing Service

When the service starts, it first requires the Lua extension libraries, described in the previous sections, and a Lua script (`failover/conf.lua`), used for configuration (lines 1 – 9). Using the RCU library, the service also creates and shares a hash table that will be used to store the elected alternative gateway (lines 11 – 12).

In line 39, the service uses the Notifier library to register a Lua callback that handles network device events. Thus, the service can detect when the state of the network interface of the default gateway has changed and take proper action on enabling the alternative route. To enable the alternative route, the service uses the FIB and the Socket libraries. In lines 29 and 33, the callback function, `failover`, uses the Notifier library, respectively, to add and remove a routing rule. Function `alt_gateway` (lines 14 – 18) uses the Socket library to notify the elected alternative gateway.

Finally, after registering the notifier callback, the service starts a new kernel thread to implement a daemon for listening to alternative gateway candidates and electing a new gateway (lines 41 – 42).

The service daemon listens to the local network for receiving advertisements of the alternative gateways. The daemon firstly retrieves the RCU hash table created beforehand by its parent runtime to store the elected gateway. After that, it creates a socket to bind a UDP port for listening to the alternative gateways. Then, whenever it receives an advertisement from an alternative gateway, it checks if the latency indicated is better than the one from the current elected gateway. If that is true, it elects the gateway as the new alternative route.

4 Final Remarks

In this paper we have discussed how the Lunatik framework has evolved in terms of flexibility and extensibility, providing now a much richer environment for scripting the Linux kernel. Its new modular implementation permits creating kernel extensions in Lua directly on top of the framework, instead of depending only on adding bindings for calling scripts from kernel subsystems. It also provides facilities for creating new libraries for extending the framework itself, allowing the creation of a new sort of kernel extensions with Lua.

We have also presented a kernel-level adaptive routing service that allows the system to fall back to an alternative gateway when a network interface is down. This service was developed as two small Lua scripts with around 40 lines of code each, using the libraries provided by the Lunatik framework to extend the kernel with a new feature.

The new version of Lunatik can also interoperate with other previous kernel scripting environments that use the Lua interpreter embedded in the kernel. For instance, NFLua and XDPLua can still use the plain Lua C API and also share libraries with the new environment. In the adaptive routing service case, we could use NFLua and XDPLua to implement specific filters when the alternative gateway is enabled. Moreover, ZFS [5], which uses its own Lua interpreter version, could share a single Lua interpreter with Lunatik without embracing the framework as a whole.

Acknowledgments

We thank Pedro Tammela, Victor Nogueira, Chengzhi Tan, Caio Messias, Iruatã Souza and Jorge Pereira for all their

valuable contribution to the development of the Lunatik framework.

References

[1] Ayuso, P. N.; Gasca, R. M.; and Lefevre, L. 2010. Communicating between the kernel and user-space in Linux using Netlink sockets. *Software: Practice and Experience* 40(9). https://perso.ens-lyon.fr/laurent.lefevre/pdf/JS2010_Neira_Gasca_Lefevre.pdf.

[2] Corbet, J. 2004. Kernel Threads Made Easy. https://lwn.net/Articles/65178/.

[3] Corbet, J. 2005. Making Notifiers Safe. https://lwn.net/Articles/160953/.

[4] CUJO LLC. NFLua. https://github.com/cujoai/nflua.

[5] FreeBSD. 2021. ZFS-PROGRAM(8) System Manager's Manual. https://man.freebsd.org/cgi/man.cgi?query=zfs-program&manpath=FreeBSD+13.2-RELEASE+and+Ports.

[6] Hoiland-Jorgensen, T.; Brouer, J. D.; Borkmann, D.; Fastabend, J.; Herbert, T.; Ahern, D.; and Miller, D. 2018. The eXpress data path: fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies (CoNEXT'18)*, 54–66. Association for Computing Machinery (ACM), New York, USA. https://dl.acm.org/doi/abs/10.1145/3281411.3281443.

[7] Ierusalimschy, R. 2016. *Programming in Lua*. Lua.org, Fourth edition.

[8] Lua.org. The Programming Language Lua. http://www.lua.org.

[9] McKenney, P. 2007. What is RCU, Fundamentally? https://lwn.net/Articles/262464/.

[10] netfilter.org. Netfilter: firewalling, NAT, and packet mangling for Linux. http://www.netfilter.org.

[11] Neto, L. V.; Nogueira, V.; de Moura, A. L.; and Ierusalimschy, R. 2020. Linux Network Scripting with Lua. In *Netdev 0x14, THE Technical Conference on Linux Networking*. https://netdevconf.info//0x14/pub/papers/22/0x14-paper22-talk-paper.pdf.

[12] Nogueira, V. B. XDPLua. https://victornogueirario.github.io/xdplua.

[13] Vieira Neto, L.; Ierusalimschy, R.; de Moura, A. L.; and Balmer, M. 2014. Scriptable Operating Systems with Lua. In *Proceedings of the 10th ACM Symposium on Dynamic Languages (DLS'14)*, 2–10. Association for Computing Machinery (ACM), New York, USA. https://dl.acm.org/doi/proceedings/10.1145/2661088.