An informal history of LuaRocks

Hisham Muhammad

April 16, 2013

1 Pre-history: the Kepler build system

We can trace the origins of LuaRocks back to Kepler, a research project that aimed to build a web development platform using Lua. Kepler brought together the development of various existing modules such as CGILua and LuaFileSystem, and developed the missing pieces that were needed to build a complete web stack. The goal was to ultimately produce packages that would allow a user to install all necessary modules for web development easily and use them in an integrated manner. Further, all this had to be done portably, supporting at least Windows, Linux and Mac OS X. Kepler, therefore, had two main technical challenges: the development of new modules and the creation of a model for integrated, portable deployment.

The latter, in particular, had always been an issue in the Lua world (or a non-issue, depending on who you asked to). Following the Lua traditions of being small, simple and avoiding setting policies, module authors have often neglected integration matters, usually dealing with them with responses such as "the module is just one C file; just copy it to your project" or "just rename the file to whatever you want to call it", and so on. This has led to a very fragmented ecosystem, with very little reuse of functionality between modules. It also made it hard for developers to keep track of which third-party modules they included in their projects, what were their versions and whether they had any recent updates with bugfixes, not to mention that in the event of updates the same manual tweaks would have to be done all over again.

The starting point for Kepler was a set of existing Lua modules that extended the language with various functionalities, such as networking (LuaSocket), file and directory manipilation (LuaFileSystem) and manipulating HTML form data (CGILua). These were separate projects, with different maintainers and their own build systems. Most of those Lua modules, at the time, shipped with Unix makefiles visibly inspired by the Lua makefiles: many variables allowing the user to modify parameters such as compiler flags and filenames, often through a separate file called `config` which contained a series of variable assignments. The documentation would instruct the user to edit the `config` file manually to suit their needs. Often, those makefiles did not include the usual "install" target, or installed files to locations that didn't match the standard Lua paths.

On Windows, Kepler sidestepped theses issues by shipping a graphical installer with pre-compiled modules, so that most users didn't have to deal with build issues. On Unix platforms, that approach is not feasible due to ABI and path incompatibilities. Unix users expected to be able to compile the framework themselves, and to have the build process configured according to their particular systems. Ideally, such configuration should be automatic; realistically, it should be at least automatable.

I joined the Kepler project in May 2006, hired by André Carregal to work on integration issues for Unix. My first task in Kepler was to tackle this problem, and produce one package for Unix systems that users could unpack and build without having to dive into the configuration files for each module. This translated to writing one big makefile that controlled the build process for about a dozen sub-projects, launching their makefiles, setting variables and occasionally editing configuration files with `sed`. This work resulted in the Unix package for Kepler 1.0, in November 2006[1].

From that big makefile, clear patterns emerged on how Lua modules were built and installed. Also, it was clear that it was not practical to maintain the all-in-one package in parallel to the development of the sub-projects, as each followed their own release cycle[2]. For end-users, upgrading individual modules after the full package was installed was also a concern. Other languages already dealt with this problem by having their own portable package managers: some examples are CPAN for Perl, RubyGems for Ruby, Eggs for Python, Cabal for Haskell. Soon, we decided that this was the direction we should follow after Kepler 1.0 was released: a high-level build and package manager tailored for Lua modules, which offered abstractions for those common patterns we identified in Kepler. Indeed, the first mention of LuaRocks in the Lua mailing list by André Carregal (then describing it as a "concept") was as early as March 2006, predating my own involvement with the Kepler project.

2 Road to 1.0

The initial development of LuaRocks ran in parallel with the development of what was to become Kepler 1.1, an update of the collection of modules, this time targeting Lua 5.1. The use in Kepler shaped the list of requirements for LuaRocks, such as the initial set of modules to be supported and portability goals. Another major influence in the design of LuaRocks was my previous experience implementing the GoboLinux distribution, for which I also had written a package management tool called Compile. These influences are quite apparent in the main features of the original design of LuaRocks.

2.1 Parallel installation of module versions

Compared to other existing package managers, the most peculiar requirement was support for keeping multiple simultaneous versions of the same package in an installed tree, so that you could, for example, install two modules A and B, where A depends on C

[1]While Lua 5.1 had been released in February 2006, Kepler 1.0 still targeted Lua 5.0, because many modules had not been ported yet and the majority of the Lua userbase still used that version at the time. Still, Kepler sub-projects promoted the use of the Lua 5.1 module system, through the Compat-5.1 module, which implemented the new module system for Lua 5.0.

[2]Some modules, such as Orbit, were written by developers hired by Kepler; others, such as LuaSocket, were third-party code integrated as part of the core of the package.

version < 2 and B depends on C version ≥ 2. The desire was that LuaRocks should allow all four modules to remain installed in the tree simultaneously, and provide runtime support so that when A or B is loaded, the correct version of C is picked. This was motivated by having to deal with API changes in dependencies while developing modules for Kepler, especially when those dependencies were also in unstable versions[3]. One major problem with Kepler 1.0 was that, because of its monolithic build system, the integration of modules had to happen in lockstep, which affected testing and slowed down development. Supporting parallel versions in LuaRocks meant that developers would not have to maintain multiple environments in order to test different modules and ensure that they were properly isolated from each other.

The approach I took for supporting multiple versions was based on the design of the GoboLinux filesystem hierarchy. GoboLinux is a Linux distribution which has the distinction of using plain directories as its package database: the subtree of files belonging to a given package are installed under a private subdirectory, such as `/Programs/Lua/5.2.0/`, and those files are integrated in the system through symbolic links, so that different kinds of files can be found at the expected places, such as `/usr/lib` and `/usr/include`. This allows multiple versions of packages to be kept on disk simultaneously, and those can be "enabled" and "disabled" through scripts which adjust the symbolic links. In the original design of LuaRocks, I adopted a similar approach, installing the files of each package under its own subtree of a versioned hierarchy. Instead of symbolic links, LuaRocks provided its own `require` function, which checked which package version should be used for a module, based on previously loaded modules, and dynamically adjusted lookup paths and then called Lua's original `require` function to load the module.

2.2 Rockspecs: declarative package specifications

Package management tools usually specify a file format through which packages are defined. Those files can be as simple as a makefile, as is the case with FreeBSD ports, or may contain various metadata and embedded build scripts, such as .spec files for the RPM package manager. For specifying LuaRocks packages, which we call "rocks", we devised a file format called "rockspec", which is actually a Lua file containing a series of assignments to predefined variable names such as `dependencies` and `description`, defining metadata and build rules for the package.

Rockspecs are loaded by LuaRocks as Lua scripts using an empty Lua environment, and therefore usual Lua constructs work, but no facilities from the Lua standard libraries can be used from within rockspec files. However, while rockspecs are imperatively executed, they do not function as a "build script". A rockspec does not list the sequence of build operations in order as a makefile or an RPM .spec would (Figure 1a), but rather contains table definitions which describe the build method declaratively (Figure 1b). This expands on the design originally developed for the Compile build manager in which specification files (called "recipes" in Compile) have a `recipe_type` field that instructs which build tool should be used: `autoconf`, `makefile`, `cmake`, `scons`, etc. Based on the recipe type, Compile is then able to drive the appropriate tool and launch specific callback functions written in the recipe file. In LuaRocks, I followed the same approach of

[3]Such was the case, in particular, when LuaSocket 2.0 was in beta.

```
%define luaver 5.1
%define lualibdir %{_libdir}/lua/%{luaver}
%define luapkgdir %{_datadir}/lua/%{luaver}
Name:  luasocket
Version:  2.0.2
Release:  8%{?dist}
Summary:  Network socket extension for Lua
# ...
Source0:  http://.../luasocket-2.0.2.tar.gz
Patch0:  lua-socket-unix-sockets.patch
# ...
%prep
%setup -q -n luasocket-%{version}
%patch0 -p1 -b .unix
%build
make %{?_smp_mflags} CFLAGS="%{optflags}
↪ -fPIC"
%install
rm -rf $RPM_BUILD_ROOT
make install
↪ INSTALL_TOP_LIB=$RPM_BUILD_ROOT%{lualibdir}
↪ INSTALL_TOP_SHARE=$RPM_BUILD_ROOT%{luapkgdir}
%clean
rm -rf $RPM_BUILD_ROOT
# ...
```

(a) RPM .spec file[4]

```
package = "LuaSocket"
version = "2.0.2-5"
source = {
   url = "http://.../luasocket-2.0.2.tar.gz",
}
description = {
   summary = "Network support for the Lua
language",
   -- ...
}
build = {
   type = "make",
   build_variables = {
      CFLAGS = "$(CFLAGS) -I$(LUA_INCDIR)",
      LDFLAGS = "$(LIBFLAG) -O -fpic",
      LD = "$(CC)"
   },
   install_variables = {
      INSTALL_TOP_SHARE = "$(LUADIR)",
      INSTALL_TOP_LIB = "$(LIBDIR)"
   },
   },
   -- ...
}
```

(b) LuaRocks rockspec[5]

Figure 1: Excerpts from specification files for LuaSocket 2.0.2 using RPM and LuaRocks, including basic package identification, download URL and build instructions

defining "build types" (set with the `build.type` table field), but there are no imperative callback functions: each build type is entirely parameterized declaratively through table fields.

Adding the concept of "build types" to LuaRocks was another unusual design decision, compared to language-specific package managers in general. Often, language-specific repositories favor one specific build tool: easy_install and later pip for Python, Rake for Ruby, ExtUtils::MakeMaker and later Module::Build for Perl. The Lua world, however, did not have a standard build system. Still, we wanted to appeal to Lua module developers in general, beyond the Kepler project. After all, most of the value of a package management tool is in the size and diversity of its repository of available packages. This meant we've had to make sure that LuaRocks adapted to the variety of practices of Lua module authors, and not the other way around. This was also a way to future-proof LuaRocks in case any of the many proposed Lua-based build tools gained traction in the community.

The first beta of LuaRocks, version 0.1, shipped with support for three build types: `make`, `cmake` and `command`. The `make` type was expected to be the most used, since all Kepler packages used makefiles and we thought this to be the most common practice among Lua module authors, at least on Unix. The `cmake` build type was the first code contribution by a developer from outside the Kepler team, very early in the project's lifetime: Peter Drahoš contributed it with an eye on using LuaRocks as a base for the LuaDist project, which aimed to build a Lua distribution entirely built using CMake. The `command` type is a catch-all backend for unsupported build tools: it allows writing a pair of commands in the rockspec (build.build_command and build.install_command) which are launched using os.execute.

The contribution of the cmake type seemed to confirm our initial suspicions that developers would contribute various build backends to the tools and we would have to deal with a varied ecosystem of build tools for Lua. However, ever since, no other build backends were submitted for inclusion in LuaRocks by developers. This was perhaps influenced by the fact that as early as 0.3, we introduced a new build type, then called simply module, but since renamed to builtin. While there were a few Lua-based build tools around by then (Premake, Prime Mover, LuaIBS, Meique, and more were released since), we realized that the actual most common method of distribution of Lua modules was to use no build tool at all: many modules are pure Lua, or a single .c file, and their authors often just put them online and write a README advising the user to download the files and add them to their project. The builtin build type was designed for those cases: pure Lua modules need only to be listed in the modules table; it also launches the C compiler passing proper flags and supports both building from a single C file as well as linking multiple objects into a module. We specifically implemented internal support for the GCC and Visual Studio toolchains, covering the platforms supported by Kepler. This also provided an easy way for developers who often shipped Unix-only makefiles to support Windows builds.

The builtin backend proved to be quite popular. As of this writing, of the 258 projects in the LuaRocks repositories, 195 of them use the builtin build type, and only 26 use make. In particular, from those 195 rocks, 29 of them originally used the make build type and later switched to builtin, suggesting that it was a good strategy to allow developers to warm up to the idea of using LuaRocks by letting them start to use it along with their existing build systems. The make build type often exposed shortcomings in the developers' makefiles, such as poor support for specifying custom install paths and linker flags. This was often noticed when Mac users attempted to install rocks written by Linux developers and vice versa, and also as developers transitioned from x86 to x86-64. The builtin type handles those issues transparently.

2.3 Portability

The initial portability requirements were set by the Kepler project: Windows, Linux and Mac OS X. I started by targeting Unix platforms, as we already had a build system in place for those platforms and LuaRocks was built from that. The build differences between Mac and Linux that affected building Lua modules were already understood and implemented in the Kepler makefiles, so we only had to make sure that those differences were properly abstracted.

The remaining challenge was supporting Windows. Before LuaRocks, Kepler developers built Windows packages using Visual Studio projects or custom makefiles written for Microsoft nmake. We managed to reuse those nmake makefiles through the make build type in LuaRocks, but the Kepler conventions set as defaults for that build type were not useful for other developers, so supporting nmake on Windows and GNU Make on Unix

[3]Full file at http://pkgs.fedoraproject.org/cgit/lua-socket.git/tree/lua-socket.spec?h=f18

[4]Full file at http://luarocks.org/repositories/rocks/luasocket-2.0.2-5.rockspec

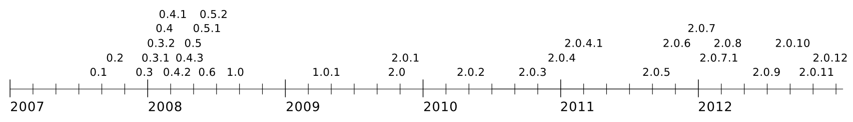

Figure 2: Timeline of LuaRocks releases

never saw widespread use. For most Kepler modules, we eventually dropped the use of makefiles, and instead opted to use the `builtin` build type, which invokes the Microsoft compiler directly.

Another portability issue was how to perform filesystem operations in the tool itself. For bootstrapping purposes, LuaRocks is written as a pure Lua application and does not assume the availability of any other Lua modules in the system. To perform operations not provided by stock Lua, such as manipulating directories or downloading remote files, it can either launch external programs (e.g. `wget`) or use additional modules such as LuaSocket, depending on what is available. The original idea was that on Unix we could ship a pure Lua program that would initially run making use of external programs and then would use itself to optionally install the extra modules; on Windows, where those external programs are not available by default, the bootstrap mode would not work but we would ship a "fatter" package including all needed module dependencies, akin to the Kepler installer. However, we ended up using the bootstrap mode on Windows as well: with very few modifications, Peter Drahoš managed to use LuaRocks without any additional Lua modules on Windows by using UnxUtils, a collection of programs that simulates a minimal Unix command-line environment (`cp`, `rm`, etc.). Ever since, we've been shipping those binaries along with the LuaRocks .zip distribution on Windows[6].

2.4 Release cycle and reception

The first release tarball, labeled 0.1, vas published in August 2007. A 0.2 release followed two months later. In late 2007 we released 0.3, the first to contain the `builtin` build type (then called `module`) and the first to be packed as a Windows-friendly .zip file, bundling UnxUtils and containing an installer batch file. Over the next few months, development picked up speed with a series of 0.x releases in the first half of 2008, in the typical "release early, release often" tradition of open source software. After thirteen 0.x releases, LuaRocks 1.0 was released in September 2008 (See Figure 2).

During that period, the collection of available rocks grew slowly but steadily. Besides packaging the Kepler modules as rocks, I went looking for popular modules to package, taking a clue from the list of top downloads for LuaForge, which had become at the time the standard catalog of Lua projects. The success of LuaForge was a powerful indicator of the existance of a considerable demographic of module authors and perhaps even their willingness to coalesce into a community, if given the resources. By writing lots of rockspecs myself, I wanted to build momentum to the tool, iron out its feature set ac-

[6]In fact, "bootstrap mode" proved to be less problematic on Windows than "Lua-module-based mode": due to the semantics of Windows filesystems, attempting to remove rocks that are in use by LuaRocks itself causes problems. There has been discussion on how to work around this problem, but it is still an open issue.

cording to the needs of existing modules, and provide the community with examples of best practices to follow when writing rockspecs. Eventually, we started receiving contributions of rockspecs written by the module authors themselves. I also started compiling a list of new rockspecs and added them to the release notes of each beta, to stress on the Lua mailing list the fact that the repository was growing and the tool was getting more and more useful. I saw this certain amount of "good publicity" as necessary, as there was clearly a resistance in parts of the Lua community against any kind of policy.

I knew quite well where I was going into, and that shipping a portable tool that would play nicely with all the different environments out there would be thorny. Every single mistake or omission echoed for a long time. For example, out of the desire of playing nicely with module authors' existing infrastructures, we favored the `make` build type early on, whenever a module included a makefile. Many of those makefiles, however, did not properly use the users' `CFLAGS`. When the emergence of x86-64 platforms made handling the `-fpic` flag mandatory, many rocks that relied on those makefiles were broken on those systems. While the `builtin` type was soon fixed to handle this flag transparently (even though it allowed for users to configure it from day one), older rocks (or rather, upstream makefiles) would still need to be fixed. Still, for long, users would refer to those failures as the "LuaRocks `-fpic` bug". As we got reports of broken `make`-based rocks, it was often easier to just convert them to the `builtin` type and bypass the upstream makefile entirely[7].

LuaRocks was also met with resistance by users of package managers in Linux distributions, who wanted to have modules available as native packages, rather than installing yet another package manager. This feeling is understandable, but comparing the sheer numbers of packages provided by distributions versus the number of modules available in mature module repositories from scripting languages, it becomes clear that the approach of converting everything into native packages is untenable: for example, while Ubuntu features 37,000 packages, Perl's CPAN alone contains over 23,000. These numbers make a good case for having portable package managers for scripting languages. While the number of available Lua modules is still far from that, it doesn't mean the potential isn't there.

Having experience as a maintainer of a Linux distribution myself, I went out of my way to avoid clashes with files managed by the host operating system. At first, this meant going against end-user expectations: the default behavior of LuaRocks running as an unpriviledged user was to install modules in a private location of the users' home directory (typically `~/.luarocks` on Unix). As LuaRocks evolved as a build tool I hoped that some integration would be possible with package managers, such as adding LuaRocks metadata into `.rpm` or `.deb` files, but while there were some talks with distro maintainers, those never went very far[8].

In spite of those challenges, the LuaRocks 1.0 release cycle achieved its goals. Kepler 1.1 was released in June 2008 using LuaRocks in its installer. Later, all-in-one packages

[7]Out of this frustration with the state of upstream makefiles, I added a page called "Recommended practices for Makefiles" to the LuaRocks documentation: `http://luarocks.org/en/Recommended_practices_for_Makefiles`

[8]One positive metric, though, is that in all these years we have never received a single bug report from distribution users or maintainers that LuaRocks was breaking other things in their systems.

were also distributed for Unix and Windows which contained Lua, LuaRocks and the Kepler modules. Sputnik, a wiki/content management system which was developed by Yuri Takhteyev in close cooperation with Kepler, also integrated LuaRocks into its distribution package. The Sputnik package made a creative use of LuaRocks, bundling it as its installer tool and using it as a plugin manager.

3 The redesign for 2.0

After LuaRocks 1.0, we slowed down the pace of releases. Incidentally, my two-year contract with Kepler expired around that time, and the project was struggling to find funding to sustain itself. Many of the Kepler subprojects had built a userbase around then, so those of us who were invested in it decided to carry on with them as volunteer open source projects. I remained as the lead maintainer of LuaRocks, and we carried the project forward.

With version 1.0, the rockspec format was declared frozen, so new releases of the tool could not introduce features that broke compatibility. This was intended to give developers a stable target when writing rockspecs, to allow Linux distributions to pick up the package and to give the project an overall aura of stability. A minor release, 1.0.1, followed six months after 1.0, containing bugfixes and minor features for the command-line tool. Around that time, we had enough feedback from users and developers so that we could start thinking about changes for the next major revision.

3.1 The revised tree structure

The number one complaint about LuaRocks was that it required patching the standard `require` function from Lua. Modules that modify the standard globals of Lua, either redefining functions or adding entries to the standard module tables, always got some criticism at the Lua mailing list, but it was not consensually considered a bad practice: modules such as `stdlib` and `lua-ex` did it, to cite two examples[9]. In the original design of LuaRocks, we chose to wrap `require` so that version constraints could be passed as extra parameters. This would allow programmers to write down versioning requirements explicitly in their own scripts, doing, for instance, `require("socket", ">=2.0")`. The standard `require` function ignores additional parameters, so the script would remain compatible with standard Lua, while users loading the `luarocks.require` module would benefit from versioning support. This, however, also meant that every LuaRocks user had to load `luarocks.require` in order to use modules installed with LuaRocks, since the module files were not installed in standard Lua locations. We did not consider that a strong limitation at first since RubyGems also required its users to issue a similar `require 'rubygems'` command[10].

In actual practice, most users didn't need any of the features provided by our custom `require` function, namely, specifying versioning in scripts and handling multiple

[9]Over time, however, that practice got more and more frowned upon, to the point that, as of Lua 5.2, the expected behavior is that modules shouldn't even register their own table as a global, but only return itself through the output of `require`.

[10]That was true up to Ruby 1.8; after that, RubyGems support is preloaded by default in Ruby

versions of the same module simultaneously. The latter was the main motivator for the custom directory structure, which was what prevented one feature that many users did want: the ability to load installed modules from vanilla Lua, without any additional runtime support.

Being compatible with vanilla Lua meant changing the directory structure so that it matched the default lookup paths used by the standard loaders. The immediate consequence was that modules from different rocks are moved into a common directory, so manifest files became indispensible for keeping track of which files belong to each rock. This was not a major problem, because we already used a manifest file to cache this mapping for performance reasons, but it required adding consistency checks for orphaned files. Those manifest files are essentially a plain-text database for package management, implemented as Lua tables. Each rock has its own `rock_manifest` file (containing also the MD5 sum for each deployed file), and a global `manifest` file caches dependency information from all installed rockspecs, as well as mappings from modules to rocks and from rocks to modules.

This change broke compatibility with existing installations, so users of LuaRocks 1.0.x had to purge their installations and start over. The rockspec format, however, remained the same. Our decision of keeping the installation structure properly abstracted in variables proved to be correct. In fact, when building a rock LuaRocks 2.0 creates a temporary directory and installs the rock there using a LuaRocks-1.0-style tree, and then creates the manifest and deploys the files to their final destinations. This has the side effect that the installation directories passed through variables to the build systems of modules (such as Makefiles) are not their actual, final locations: this is good because it prevents developers from successfully hardcoding directories to their installations, and therefore modules built with LuaRocks are, by necessity, relocatable. Relocatable binaries are not the norm in the Unix world, but they are expected on Windows. This is still an occasional source of complaints from developers who want to hardcode paths to resource files or to generate documentation at build time that refers to absolute paths, but at this point supporting relocatable binaries seems a more useful feature.

Once we moved to the new format, a question was how to retain the features that the custom `require` used to provide. We still wanted to support the installation of multiple versions of a rock (and, in all honesty, we didn't want to have to figure out how to deal with removing older versions, in light of possible cascading dependency conflicts). In the LuaRocks 1.0 design, two versions of, say, `socket.so` were simply installed to different directories. With a single directory for installing modules, we had to figure out what to do.

The solution was to rename old modules so they can coexist in the same directory (adding rock name and version as a prefix), and provide a custom package loader that translates the names based on the versions of previously loaded modules, in a similar mechanism to the `require` function of LuaRocks 1.0. The idea is that plain Lua will always find the latest installed version of each module, as that file will have a standard pathname such as `/usr/local/lib/lua/5.1/socket.so`, and users who need support for loading different versions can load `luarocks.loader`, a module that installs a custom package loader, which is the proper extensibility mechanism for the `require` function. This loader keeps in memory a "context", which is the list of previously required mod-

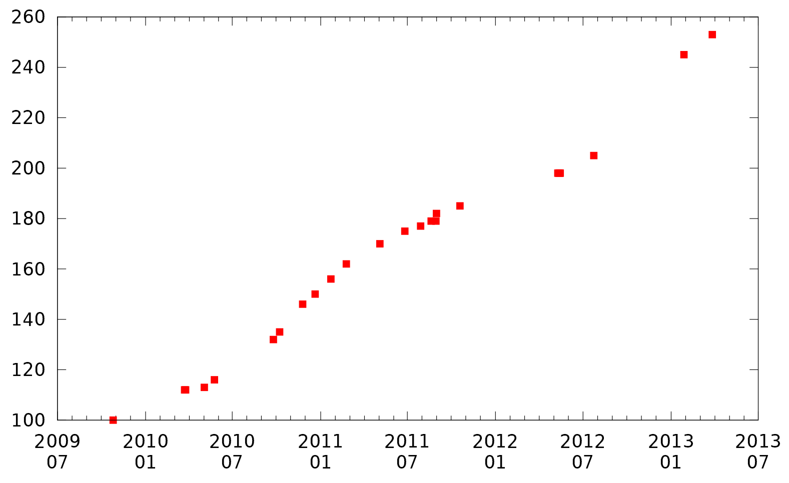

Figure 3: Number of modules in the LuaRocks repository during the 2.0 series, from October 2009 to March 2013

ules, the rocks they belong to and their dependencies, so that when a new module is required, versions are picked based on dependencies from the current context. Once a module is selected, LuaRocks keeps track if the file has a plain name or if it was prefixed, and then that name is forwarded to the standard Lua loaders.

This approach is straightforward for Lua scripts because the package loader for Lua files does not mandate that modules must match their filenames. C modules, however, cannot be renamed to any name, because the standard loader for C files expects to find an exported symbol whose name matches that of the module being required, such as `luaopen_socket` for the `socket.so` module. Fortunately, Lua provides support for embedding versioning to a C module filename through a prefix: as the Lua manual demostrates, "if the module name has a hyphen, its prefix up to (and including) the first hyphen is removed. For instance, if the module name is `a.v1-b.c`, the function name will be `luaopen_b_c`". In the above example, an older version of `socket.so` will be renamed to `luasocket_2_0_1_3-socket.so`, and the LuaRocks package loader decides which file to load based on the context.

Explicit versioning can no longer be done passing additional arguments to `require`, but users who wish to configure versioning programmatically can use a function from the `luarocks.loader` module called `add_context`, which pre-configures the loader to add a given rock version to its operating context.

3.2 Release cycle and reception

LuaRocks 2.0 was released in October 2009. Apart from a couple of release candidates, there were no betas this time. The change in the format of the local trees was substantial, and we wanted the transition to be a one-time event. Development happened in the open[11] and was discussed in the mailing list, so interested parties could follow the

[11]We originally used CVS hosted by LuaForge, then switched to Subversion in April 2009 and finally moved to Git hosted by GitHub in August 2010.

direction we were going. At the time of release, the changes were not controversial — my feeling is that retaining full compatibility with both the .rockspec and .rock formats played a big role in keeping a sense of continuity. Release 2.0 was well-received, and it was largely seen as a step forward and a declaration of commitment to the project, especially in the light of the uncertainties around the original Kepler project.

In the early days of LuaRocks, I scanned the LuaForge entries and routinely looked for module announcements at the Lua mailing list, looking for modules that I could package as rocks. I did this to populate the repositories, and to test the resilience of the tool in dealing with different projects and Makefiles. By the time LuaRocks 2.0 was released, I mostly stopped doing that (except for a few modules whose rockspecs I remain as the de facto maintainer, such as LPeg), and the repository grew based solely on contributions by developers. Figure 3, generated from archive snapshots of the repository index, shows the growth of the collection, from October 2009 when the repository contained exactly 100 projects, up to March 2013, when we just surpassed 250 projects. Another indicative of a healthy community is that many rockspecs contain dependencies on other rocks, showing that the culture of module reuse is strengthening.

During this time, the 2.0 series had a number of point releases. No further changes to the local tree format were made, and save for bugfixes, these releases are essentially compatible. They were mainly driven by feedback and contributions from users, and were focused on improving portability, adding new commands to the `luarocks` and `luarocks-admin` command-line tools, and improving user experience with better environment detection. Some of the features were in fact developed as contract work, sponsored by companies using the tool, and shared back into the LuaRocks code base.

4 Current status and recent developments

LuaRocks is used in production systems around the world and is included in repositories of various Linux distributions. As of this writing, the rocks repository features 750 rockspecs for 258 different projects. The directions of development nowadays are essentially dictated by the needs of the community, while trying to balance concerns of compatibility, portability and the social aspect of trying to be a welcoming platform: we need to cater not only to the needs of those who already use LuaRocks, but try to appeal to Lua module developers who still don't use it. For many developers, especially those used to other languages that already have similar ecosystems in place, LuaRocks is their introduction to writing and sharing Lua modules, so the least hoops they have to jump to be able to contribute, the better.

4.1 Compatibility

The LuaRocks codebase is compatible with both Lua 5.1 and 5.2. The version of Lua to be used can be either detected by the `configure` script at installation time, or explicitly selected by the user. Lua 5.2 support was added by removing hardcoded references to Lua 5.1 and detecting features at runtime — the same installation can run with Lua 5.1 or 5.2, given proper configuration files. When Lua 5.2 was first released, it was not clear how developers (which are, after all, the LuaRocks userbase) would manage the transition.

It was expected that some would remain using Lua 5.1, so dropping support for that version was out of the question. What we didn't know was if users would be interested in running LuaRocks with Lua 5.1 and 5.2 at the same time, and if they would prefer to do so with a single LuaRocks installation or would install two copies in parallel. We ended up supporting both approaches, but they require crafting configuration files carefully. There is still room for improvement, to make it easier for developers to test their modules in both Lua 5.1 and 5.2.

Another issue during this transition was the compatibility of the various modules in the repository with Lua 5.2. LuaRocks always supported specifying a version of Lua in the dependencies table of rockspecs. Most rockspecs, however, optimistically specified their Lua compatibility as `"lua >= 5.1"`, only to learn that those modules were not fully compatible with Lua 5.2 upon its release. We've been dealing with this on a case-by-case basis: as we get reports of compatibility problems in the LuaRocks mailing list, we fix the rockspecs accordingly. Another question was whether to maintain separate module repositories for Lua 5.1 and 5.2. So far, we've been keeping all modules in a single repository, but we may have to revisit this issue in the future if problems arise for Lua 5.1 users, either with separate repositories or, at least, separate manifest files.

LuaJIT compatibility is another point of concern, as it implements a Lua dialect between Lua 5.1 and 5.2, plus extensions of its own. LuaJIT is fully compatible with Lua 5.1 and LuaRocks works fine with it in Lua 5.1 mode, but some Lua 5.2 modules may be turn out to compatible with LuaJIT but not with Lua 5.1 and there's currently no way to specify this. Currently, LuaRocks does not implement explicit detection for LuaJIT, which could be useful, for example, for handling modules written using the LuaJIT FFI. Still, we haven't heard much from the LuaJIT user community, and in any case it would be easy to have a separate repository for FFI-based modules.

4.2 The rocks repository

The mailing list and the default rocks repository are the epicenters of the LuaRocks community. The repository has a simple generated index webpage which lists all rocks, along with their descriptions and versions. I receive rockspecs via the mailing list and update the repository using luarocks-admin, with occasional help from Fabio Mascarenhas, another Kepler alumnus. Since this is a manual process, there is usually a delay; developers can't publish rocks automatically. This has allowed me to review rockspecs and guide authors with regard to best practices, and ensure some level of consistency in the nascent repository, but it's far from an optimal process. An important aspect of the management of the repository is that I don't curate it in any way. All valid submissions are included, even if for instance there is already another module that implements the same functionality. We avoid naming clashes in rocks as a rule for operational reasons, but try to discourage naming clashes in modules, when those are detected. It is not the task of the LuaRocks repository to choose between competing modules in the ecosystem; that is up to the community.

As a future direction, I would like to have a more automated system in place in which I wouldn't be a "bottleneck" for developers wishing to make their code available through

LuaRocks. Recently, Leaf Corcoran released MoonRocks[12], a website which provides a repository where users can register their own rockspecs directly. We've discussed the possibility of eventually turning it (or a variation of it) into the default repository for rocks, but there are still open questions.

The demise of LuaForge left a void in the Lua community, as there is no definite catalog of projects anymore. The LuaRocks repository serves that role partially, but there is more to Lua than modules: in 2010 LuaForge already hosted over 500 projects. There has been an initiative to create a new version of the site around 2009, but nothing materialized. It would be very nice to see a successor of LuaForge in the future; if that ever happens, integrating the rocks repository into it would be a logical step.

4.3 The rockspec format

The rockspec format remains largely frozen since LuaRocks 1.0. The only additions were support for new VCSs as virtual protocols in the `source.url` field (such as "`git://`"). The addition of new protocols was foreseen, so the use of those in earlier versions causes graceful errors. We also chose to keep feature-for-feature compatibility, but not bug-for-bug: we fixed the code in cases of incorrect behavior, so there are in fact rockspecs in the repository which only work with later LuaRocks versions, but the published specification of the rockspec format remains the same since 1.0. We felt this was important so that developers felt confidence in expending the effort of learning the format.

Since then, we have identified some shortcomings in the specification and possibilities for improvement. The next major release of LuaRocks will probably be a good time for a major revision of the specification, while certainly keeping backward compatibility. LuaRocks is prepared to recognize incompatibilities through the `rockspec_format` field since version 1.0, so the transition shouldn't be very traumatic, especially if we implement support for LuaRocks to upgrade itself, which is another frequent wishlist item.

[12]`http://rocks.moonscript.org/`