OPAM: a Package Management Systems for OCaml
Version 1.0.0 Roadmap

THIS DOCUMENT IS A DRAFT

Thomas GAZAGNAIRE
thomas.gazagnaire@ocamlpro.com

April 23, 2012

Contents

Overview

This document specifies the design of a package management system for OCaml (OPAM). For the first version of OPAM, we have tried to consider the simplest design choices, even if these choices restrict user possibilities (but we hope not too much). Our goal is to propose a system that we can build in a few months. Some of the design choices might evolve to more complex tasks later, if needed.

A package management system has typically two kinds of users: *end-users* who install and use packages for their own projects; and *packagers*, who create and upload packages. End-users want to install on their machine a consistent collection of *packages* – a package being a collection of OCaml libraries and/or programs. Packagers want to take a collection of their own libraries and programs and make them available to other developpers.

This document describes the fonctional requirements for both kinds of users.

Conventions

In this document, `$home`, `$opam`, `$lib`, `$bin`, `$build`, `$opamserver` and `$package` are assumed to be set as follows:

- `$home` refers to the end-user home path, typically `/home/thomas/` on linux, `/Users/thomas/` on OSX `C:\Documents and Settings\thomas\` on Windows.
- `$opam` refers to the filesystem subtree containing the client state. Default directory is `$home/.opam`.
- `$lib` refers to where the end-user wants the libraries to be installed. Default directory is `$opam/OVERSION/lib` (`OVERSION` is the OCaml compiler version).

- `$bin` refers to where the end-user wants the binaries to be installed. Default directory is `$opam/OVERSION/bin` (`OVERSION` is the OCaml compiler version).
- `$build` refers to where packages are built before being installed. Default directory is `$opam/OVERSION/build` (`OVERSION` is the OCaml compiler version).
- `$doc` refers to where package documentation is installed. Default directory is `$opam/OVERSION/doc/` (`OVERSION` is the OCaml compiler version).
- `$opamserver` refers to the filesystem subtree containing the server state. Default directory is `$home/.opam-server`.
- `$package` refers to a path in the packager filesystem, where lives the collection of libraries and programs he wants to package.

Variable are written in capital letters: for instance `NAME`, `VERSION`, `OVERSION`, ...

1 Milestone 1: Foundations

The first milestone of OPAM focuses on providing a limited set of features, dedicated to package management only: configuration, build and install steps are out-of-scope. Moreover, we limit OPAM to support the installation of one version per packages only; moreover, this first version of OPAM supports only one compiler version.

1.1 Client state

The client state is stored on the filesystem, under `$opam`:

- `$opam/config` is the main configuration file. It defines the OPAM version, the repository addresses and the current compiler version. The file format is described in §1.2.1.
- `$opam/index/NAME.VERSION.opam` are OPAM specification files for all available versions of all available packages. The format of OPAM files is described in §1.2.3.
- `$opam/descr/NAME.VERSION.txt` are textual files, containing the description for each available packages. The first line of this file is the package synopsis.
- `$opam/OVERSION/installed` is the list of installed packages with their version for a given compiler version. The format of installed packages file is described in §1.2.2.
- `$opam/OVERSION/config/NAME.config` are package configuration files, containing environment variables for each installed packages. The file format is described in §1.3.
- `$opam/OVERSION/install/NAME.install` are package installation files, containing the installed files for each installed packages. The file format is described in §1.6.3.
- `$opam/archives/NAME.VERSION.tar.gz` are source archives for all available versions of all available packages.
- `$build/NAME.VERSION/` are tempory folders used to decompress the corresponding archives, for all the previously and currently installed package versions.
- `$bin/` contains the installed binaries.
- `$lib/NAME/` contains the installed libraries for the package `NAME`.

1.2 File Syntax

Base types The base types are:

- `STRING` a doubly-quoted OCaml string, for instance: `"foo"`
- `SYMBOL` a symbol contains only non-letter and non-digit characters, for instance: `<=`
- `IDENT` an ident starts by a letter and is followed by any number of letters, digit and symbols, for instance: `foo-bar`

Compound types Values of base types can be composed together to build complex types:

- `[ X ]` a space-separated list of values of type `X`
- `( X )` a space-separated optional list of values of type `X`
- `{ X }` a space-separated collection of values of types `X` (whose order is thus not meaningful).

Files All structured OPAM files share the same syntax:

- A file is a space-separated list of `items`
- An `item` is either:
  - `IDENT = value`
  - `IDENT STRING { item }`
- a `value` is either:
  - `STRING`
  - `SYMBOL`
  - `[ VALUE ]`
  - `VALUE ( VALUE )`

1.2.1 Configuration files

`$opam/config` has the following format:

```
opam-version = 1.0
sources = [ STRING ]
ocaml-version = STRING
```

The field `sources` contains the list of OPAM repositories (default is `"opam.ocamlpro.com"`). Initially, the field `ocaml-version` corresponds to the output of `'ocamlc -version'`.

There are two kinds of repository sources:

- READ-ONLY repositories with `https://`, `http://` and `ftp://` as prefix. They are synchronized using `rsync`.
- READ-WRITE repositories: they have `opam://` as prefix. They are synchronized using a custom protocol. The server should run `opam-server` on port 9999 to accept client connections.

1.2.2 Installed packages

`$opam/OVERSION/installed` has the following format:

```
STRING = STRING
STRING = STRING
...
```

Each line `NAME = VERSION` in this file means that the version `VERSION` of package `NAME` has been compiled with OCaml version `OVERSION` and has been installed on the system in `$opam/OVERSION/lib/NAME`.

1.2.3 OPAM files

`$opam/index/NAME.VERSION.opam` has the following format:

```
opam-version = 1.0

package NAME {
  version     = STRING
  description = STRING
  maintainer  = STRING
  depends     = VALUE
  conflicts   = VALUE
  libraries  = [ STRING ]
  syntax     = [ STRING ]
}
```

The first line specifies the OPAM version.

The contents of `version` is `VERSION`. The contents of `description` is the name of the file, among the package files, containing the package textual description. The first line of this file is interpreted as the package synopsis. `maintainer` contains the contact address of the package maintainer.

The `depends` and `conflicts` fields contain expressions over package names, optionally parametrized by version constrains. An expression is either:

- A package name: `"foo"`;
- A package name with version constraints: `"foo" (>= "1.2" & <= "3.4")`
- A disjunction of expressions: `E | F`
- A conjunction of expressions: `E & F`
- An expression with parenthesis: `( E )`

For instance `"foo" (<= "1.2") & ("bar" | "gna" (= "3.14"))` is a valid formula whose semantic is: *a version of package* `"foo"` *lesser or equal to* 1.2 *and either any version of package* `"bar"` *or the version* 3.14 *of package* `"gna"`.

The `libraries` and `syntax` fields contain the libraries and syntax extensions defined by the package.

1.3 Configuration files

`$opam/OVERSION/config/NAME.config` has the following syntax:

```
library STRING {
  include  = [ STRING ]
  asmlink  = [ STRING ]
  bytelink = [ STRING ]
  requires = [ STRING ( STRING ) ]
  pp       = [ STRING ( STRING ) ]
}

syntax STRING {
  include  = [ STRING ]
  asmlink  = [ STRING ]
  bytelink = [ STRING ]
  requires = [ STRING ( STRING ) ]
  pp       = [ STRING ( STRING ) ]
}

IDENT = STRING
IDENT = [ STRING ]
...
```

Each `library` and `syntax` block defines full compile-time options to use when linking with this library (not including the dependencies options, which will be built dynamically using the `requires` and `pp` fields).

- `include` is the list of directory to open when compiling a project using the library (or the syntax extension). It should at least contain `[ "-I" "/full/path/to/NAME" ]`.
- `asmlink` is either the list of libraries to use when linking a project in native code with the library. It should at least contain `[ "-I" "/full/path/to/NAME" "NAME.cmxa" ]`
- `asmlink` is either the list of libraries to use when linking a project in byte code with the library. It should at least contain `[ "-I" "/full/path/to/NAME" "NAME.cma" ]`
- `requires` is the list of libraries which needs to be linked with the current one. The syntax `"foo" ("bar" "gna")` means only libraries `"bar"` and `"gna"` in package ”foo” will be considered. The syntax `"foo"` means *all* libraries in package `"foo"` will be considered.
- `pp` is the list of syntax extension to use when compiling a program using the library. The syntax is similar to `requires`. Once expended, the list of arguments is used with the `-pp` command-line option of the chosen compiler.

The remaining fields `IDENT = STRING` or `IDENT = [ STRING ]` are used to defined global variables associated to this package, and are used to substitute variables in template files (using the syntax `%{PACKAGE}:VAR%`, see §**??**.

1.3.1 Install files

`$opam/OVERSION/install/NAME.install` has the following format:

```
lib  = [ "name.cmi" "name.cmo" "name.cmx" "name.o" ]
bin  = [ "foo.byte" ("foo") ]
doc  = [ "doc" ]
misc = [
  [ "foo.el" "/usr/share/emacs/site-lib"         ]
]
```

Files listed under `lib` should be copied to `$lib/NAME/`. File listed under `bin` should be copied to `$bin/`. Files listed under `doc` should be copied to `$doc/NAME/`. Files listed under `misc` should be processed as follows: for each line `FILE DST`, the tool should ask the user if he wants to install `FILE` to the absolute path `DST`.

1.4 Server state

The filesystem of OPAM repositories are mirrored on the client filesystem under `$opamserver/HOSTNAME` for each remote `HOSTNAME`. This filesystem contains:

- `$opamserver/HOSTNAME/index/NAME.VERSION.opam`, which are OPAM files for all available versions of all available packages. The format of specification files is described in §1.2.3.
- `$opamserver/HOSTNAME/archives/NAME.VERSION.tar.gz` are the source archives for all available versions of all available packages.

Depending on the kind of OPAM repository, the most adapted synchronization tools will be run between the server and client filesystems. Moreover, the server files are installed at the right place in the client state using symbolic links when it is possible (ie. not on windows ...).

1.5 Server API

In this section all function are defined for a given OPAM repository.

1.5.1 Basic types

```
type repo    = string
type name    = string
type version = string
type opam
type archive
```

Names and versions are strings. Archives and specification files are either binary strings or filenames.

1.5.2 Getting the list of packages

```
val getList: repo -> (name * version) list
```

`getList repo` updates the given repository and returns the list of available versions for all packages.

1.5.3 Getting specification files

```
val getOPAM: repo -> (name * version) -> opam
```

`getOPAM repo (name,version)` returns the corresponding OPAM file.

1.5.4 Getting package archive

```
val getArchive: repo -> (name * version) -> archive
```

`getArchive repo (name,version)` returns the corresponding package archive.

1.5.5 Uploading new archives

```
val newArchive: repo -> (opam * archive) -> unit
```

`newArchive(opam,archive)` takes as input an OPAM file and the corresponding package archive, and upload the server state. This function works only for READ-WRITE repository. In case of a READ-ONLY one, a suitable error message is returned to the user.

1.5.6 Binary Protocol

In case of READ-WRITE repositories, the server state can be queried and modified by any OPAM clients, using the following binary protocol

- Communication between clients and servers always start by an hand-shake to agree on the protocol version.
- All the basic values (names, versions and binary data) are represented as OCaml strings.
- More complex values are marshaled using a simple binary protocol: the first byte represents the message number, and then each message argument is stacked in the message with its size as prefix. The list of messages *from the client to server* is:

Client-to-Server Message	Arguments	Description
`GetList`	–	Ask for the list of all OPAM files

`GetOPAM`	`name    : string`
	`version: string`
		Ask for the binary representation of a given OPAM file

`GetArchive`	`name    : string`
	`version: string`
		Ask for the binary representation of a given archive file

`NewArchive`	`name    : string`
	`version: string`
	`opam    : string`
	`archive: string`
		Create a new package on the server. The client should provide the OPAM file and the source archive.

`UpdateArchive`	`name    : string`
	`version: string`
	`opam    : string`
	`archive: string`
	`key     : string`
		Update a new version of a given package on the server. The client should also provide a security key

- Answers from the server are encoded in the same way (ie, a byte for the message number, followed by optional arguments prefixed by their size). List arguments are encoded by stacking first the lenght, and then all the elements of the list in sequential order. The list of messages *from servers to clients* is:

Server-to-Client Message	Arguments	Description
`GetList`	`list    : (string*string) list`	Return the list of available package names and versions
`GetOPAM`	`opam    : string`	Return an OPAM file
`GetArchivwe`	`archive: string`	Return an archive file
`NewArchive`	`key     : string`	Return a security key
`UpdateArchive`	–	The update went OK
`Error`	`error   : string`	An error occurred

Note that when an error is raised by an arbitrary function at server side, the client receives `Error _`.

1.6 Client commands

1.6.1 Creating a fresh client state

When an end-user starts OPAM for the first time, he needs to initialize `$opam/` in a consistent state. In order to do so, he should run:

```
$ opam init [HOSTNAME]*
```

Where `HOSTNAME` are OPAM repositories. If no OPAM repository is specified, default is `opam://opam.ocamlpro.com`.

This command will:

1. create the file `$opam/config` containing:

   ```
   version: 1.0
   sources: [HOSTNAME]+
   ocaml-version: OVERSION
   ```

   where `OVERSION` is obtained by calling `'ocamlc -version` (ie. we assume the user have already installed the OCaml compiler).
2. create an empty `$opam/OVERSION/installed` file.
3. ask the server for all available packages using `getList` (§1.5.2) and get all the corresponding spec files using `getOpam` (§1.5.3).
4. dump all the spec files into `$opam/index/NAME.VERSION.opam`.
5. create empty directories `$opam/archives`; and create `$lib` and `$bin` if they do not exist.

1.6.2 Listing packages

When an end-user wants to have information on all available packages, he should run:

```
$ opam list
```

This command will parse `$opam/OVERSION/installed` to know the installed packages, and `$opam/index/*.opam` to get all the available packages. It will then build a summary of each packages. For instance, if `batteries` version `1.1.3` is installed, `ounit` version `2.3+dev` is installed and `camomille` is not installed, then running the previous command should display:

```
batteries   1.1.3  Batteries is a standard library replacement
ounit      2.3+dev  Test framework
camomille      --  Unicode support
```

In case the end-user wants a more details view of a specific package, he should run:

```
$ opam info NAME
```

This command will parse `$opam/OVERSION/installed` to get the installed version of `NAME` and will look for `$opam/index/NAME.*.opam` to get available versions of `NAME`. It can then display:

```
package:  NAME
version:  VERSION                  # '--' if not installed
versions: VERSION1, VERSION2, ...
description:
  LINE1
  LINE2
  LINE3
```

1.6.3 Installing a package

When an end-user wants to install a new package, he should run:

```
$ opam install NAME
```

This command will:

1. look into `$opam/index/NAME.*.opam` to find the latest version of the package.
2. compute the transitive closure of dependencies and conflicts of packages using the dependency solver (see §1.7). If the dependency solver returns more than one answer, the tool will ask the user to pick one, otherwise it will proceed directly.
3. the dependency solver should have sorted the collections of packages in topological order. Them, for each of them do:
   (a) check whether the package archive is installed by looking for the line `NAME VERSION` in `$opam/OVERSION/installed`. If not, then:
      i. look into the archive cache to see whether it has already been downloaded. The cache location is: `$opam/archives/NAME.VERSION.tar.gz`.
      ii. if not, then download the archive and store it in the cache.
      iii. decompress the archive into `$build/`. By convention, we assume that this should create `$build/NAME.VERSION/`.
      iv. run `$build/NAME.VERSION/build.sh`. By convention, package archives should contains such a file.
      v. process `$build/NAME.VERSION/NAME.install`. The file format is described in §1.6.3.

Remark This installation scheme is not always correct, as installing a new package should uninstall all packages depending on that one. For instance, let us consider 3 packages `A`, `B` and `C`; `B` and `C` depend on `A`; `C` depends on `B`. `A` and `B` are installed, and the user request `C` to be installed. If the version of `A` is not correct one but the version of `B` is, the tool should: install the latest version of `A`, recompile `B`, compile `C`. It is understood that, with this first milestone, `B` will not be recompiled. This issue will be fixed in next milestones of OPAM.

1.6.4 Updating index files

When an end-user wants to know what are the latest packages available, he will write:

```
$ opam update
```

This command will ask the server the list of available packages using `getList` (see §1.5.2); then ask for the missing OPAM files using `getOpam` (see §1.5.3). Finally it will dump the missing OPAM files into `$opam/index/NAME.VERSION.opam`.

1.6.5 Upgrading installed packages

When an end-user wants to upgrade the packages installed on his host, he will write:

```
$ opam upgrade
```

This command will call the dependency solver (see §1.7) to find a consistent state where *most* of the installed packages are upgraded to their latest version. It will install each non-installed packages in topological order, similar to what it is done during the install step, See §1.6.3.

1.6.6 Getting package configuration

The first version of OPAM contains the minimal information to be able to use installed libraries. In order to do so, the end-user (or the packager) should run:

```
$ opam config [-list|-var NAME:VAR|-subst FILENAME]
```

This command will return:

- the list of all variables defined in installed packages
- the content of variable `VAR` defined in installed package `NAME`
- the file `FILENAME` where every occurrence of `%{NAME:VAR%}` in `FILENAME.in` is replaced by its content

XXXX

1.6.7 Uploading packages

When a packager wants to create a package, he should:

1. create `$package/NAME.VERSION.opam` containing in the format specified in §1.2.3.
2. create `$package/NAME.install` containing the list of files to install. File format is described in 3(a)v); filnames should be relative to `$package`.
3. create the script `./build.sh` which will be called by the end-user installer. This script should configure and build the package on the end-user host.
4. create an archive `NAME.VERSION.tar.gz` of the sources he wants to distribute, including `$NAME.install`, `build.sh` and optionaly `$NAME.opam`.
5. run the following command:

   ```
   $ opam-upload NAME
   ```

   This command looks into the current directory for a file named `NAME.opam`, and it will parse it to get the version number. Then it looks in the current directory for the archive `NAME.VERSION.tar.gz`. It will then use the server API 1.5 to upload the package on the server.

1.7 Dependency solver

Dependency solving is a hard problem and we do not plan to start from scratch implementing a new SAT solver. Thus our plan to integrate (as a library) the Debian depency solver for CUDF files, which is written in OCaml.

- the dependency solver should run on the client;
- the dependency solver should take as input a list of packages (with some optional version information) the user wants to install and it should return a consistent list of packages (with version numbers) to install;
- version information should be translated from arbitrary strings (used in OPAM files, see §1.2.3) to integers (used by CUDF). We assume that version numbers are always incremented.
- part of the input can be cached in `$opam/index.cudf` if necessary.

2 Milestone 2: Correctness of Installation

This milestone focus on correctness of installation and upgrade.

2.1 Upgrading & installing are always correct

When the user wants to upgrade, he gets a list of packages in topological order to install. When a package version is different from the installed package version, the package should be built and should replace the previous one. Then, all the packages depending on this package should be recursively reinstalled (even if they have correct version numbers).

2.2 Removing packages

When the user wants to remove a package, he should write:

```
$ opam-remove NAME
```

This command will check whether the package `NAME` is installed, and if yes, it will display to the user the list packages that will be uninstalled (ie. the transitive closure of all forward-dependencies). If the user accepts the list, all the packages should be uninstalled, and the client state should be let in a consistent state.

3 Milestone 3: Link Information

This milestone focuses on adding the right level of linking information, in order to be able to use packages more easily.

3.1 Getting package link options

The user should be able to run:

```
$ opam-config -bytelink NAME
$ opam-config -asmlink NAME
```

This command will return the list of link options to pass to `ocamlc` when linking with libraries exported by `NAME`.

In order to be able to do so, packagers should provide a file `NAME.descr` which gives link information such as:

```
library foo {
  requires: bar, gni
  link: -linkall
  asmlink: -cclib -lfoo
 }
```

3.2 Getting package recursive configuration

The user should be able to run:

```
$ opam-config -r -dir NAME
$ opam-config -r -bytelink NAME
$ opam-config -r -asmlink NAME
```

This command will return the good options to use for package `NAME` and all its dependencies, in a form suitable to be used by OCaml compilers.

4 Milestone 4: Server Authentication

This version focuses on server authentication.

4.1 RPC protocol

The protocol should be specified (using either a binary format or a JSON format).

4.2 Server authentication

The server should be able to ask for basic credential proofs. The protocol can be sketched as follows:

- packagers store keys in `$opam/keys/NAME`. These keys are random strings of size 128.
- the server stores key hashes in `$opamserver/hashes/NAME`.
- when a packager wants to upload a fresh package, he still uses `newArchive`. However, the return type of this function is changed in order to return a random key. OPAM clients then stores that key in `$opam/keys/NAME`.
- when a packager wants to uplaod a new version of an existing package, he uses the function `val updateArchive: (opam * string * string) -> bool`. `updateArchive` takes as argument an OCaml value representing the OPAM file contents, the archive file as a binary string and the key as a string. The server then checks whether the hash of the key is equal to the one stored in `$opamserver/hashes/NAME`; if yes, it updates the package and return `true`, if no if it returns `false`.
- packager email should be specified in `NAME.opam`:

5 Milestones 6: Pre-Processors Information

This milestone focus on the support of pre-processors.

5.1 Getting package preprocessor options

The user should be able to run:

```
$ opam-config -bytepp NAME
$ opam-config -asmpp NAME
```

This command will return the command line option to build the preprocessor exported by package `NAME`.

In order to do so, packagers should describe exported preprocessors in the corresponding `NAME`.descr:

```
syntax foo {
  requires: bar, gni         // list of syntax dependencies
  pp: -parser o -printer p   // common options to asmpp and bytepp
  bytepp: ...
}
```

6 Milestones 7: Support of Multiple Compiler Versions

This milestone focus on the support of multiple compiler versions.

6.1 Compiler Description Files

For each compiler version `OVERSION`, the client and server states will be extended with the following files:

- `$opam/compilers/OVERSION.comp`
- `$opamserver/compilers/OVERSION.comp`

Each `.comp` file contains:

- the location where this version can be downloaded. It can be an archive available via `http` or using CVS such as `svn` or `git`.
- eventual options to pass to the configure script. `-prefix=$opam/OVERSION/` will be automatically added to these options.
- options to pass to `make`.
- eventual patch address, available via `http` or locally on the filesystem

For instance, `3.12.1+memprof.comp` (OCaml version 3.12.1 with the memory profiling patch) looks like:

```
src:       http://caml.inria.fr/pub/distrib/ocaml-3.12/ocaml-3.12.1.tar.gz
build:     world world.opt
patches:   http://bozman.cagdas.free.fr/documents/ocamlmemprof-3.12.0.patch
```

And `trunk-tk-byte.comp` (OCaml from SVN trunk, with no *tk* support and only in bytecode) looks like:

```
src:       http://caml.inria.fr/pub/distrib/ocaml-3.12/ocaml-3.12.1.tar.gz
configure: -no-tk
build:     world
```

6.2 Milestone 8: Version Pinning

6.3 Milestones 9: Parallel Build

6.4 Milestone 10: Version Comparison Scheme

6.5 Milestone 11: Database of Installed Files