OPAM: a Package Management Systems for OCaml
Version 1.0.0 Roadmap

THIS DOCUMENT IS A DRAFT

Thomas GAZAGNAIRE
thomas.gazagnaire@ocamlpro.com

April 19, 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).
- `$opamserver` refers to the filesystem subtree containing the server state. Default directory is `$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`:

- `$opam/config` is the main configuration file. It defines the OPAM version, the repository address and the current compiler version. The file format is described in §1.1.1.
- `$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 §1.1.2.
- `$opam/index/NAME-VERSION.opam` are OPAM files for all available versions of all available packages. The format of OPAM files is described in §1.1.3.
- `$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`.

1.1.1 Configuration files

`$opam/config` has the following format:

```
version: 1.0
sources: HOSTNAME
ocaml-version: OVERSION
```

`HOSTNAME` is the name of the central OPAM repository (default is `opam.ocamlpro.com`). `OVERSION` corresponds to the output of `'ocamlc -version'`.

1.1.2 Installed packages

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

```
NAME1 VERSION1
NAME2 VERSION2
...
```

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`.

1.1.3 OPAM files

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

```
opam-version: 1.0

package:     NAME
version:     VERSION
description: TEXT
depends:     FORMULAE
conflicts:   FORMULAE
```

The first line specifies the OPAM version. The `package`, `description`, `depends` and `conflicts` should follow CUDF standards[1]: `FORMULAE` are conjonctions of disjonctions of constraints over (optionnaly versionned) packages, and `TEXT` can contain line breaks if the next lines starts by at least two spaces.

However, unlike CUDF specification, we allow `VERSION` to be an arbitrary string such as `1.0.1` or `2.3+dev`. We assume that version strings are lexicographic ordered[2].

1.2 Server state

The server state is stored on the filesystem (although the information can be cached in memory for the daemon version, in order to speed-up response time). The state is stored under `$opamserver/`:

- `$opamserver/index/NAME-VERSION.opam` are OPAM files for all available versions of all available packages. The format of OPAM files is described in §1.1.3.
- `$opamserver/archives/NAME-VERSION.tar.gz` are the source archives for all available versions of all available packages.

1.3 Server API

Server state can be queried and modified by any OPAM clients, using the following API. *Except the first message on a stream*, indicating the agreement of version used between the client and the server, all other messages are marshalled using standard OCaml `output_value` function. If a client sends a value of type `client_to_server_message`:

[1] http://www.mancoosi.org/reports/tr3.pdf
[2] This point should be addressed in a forthcoming milestone

```
open Namespace
open Path

type client_to_server_message =
  | C2S_GetList
  | C2S_GetOpam of name_version
  | C2S_GetArchive of name_version
  | C2S_NewArchive of name_version * binary_data * binary_data archive
  | C2S_UpdateArchive of
      name_version * binary_data * binary_data archive * security_key
```

The answer from the server can be decoded with `input_value`, it contains a constructor similar to the one sent (except that `C2S` becomes now `S2C`):

```
type server_to_client_message =
  | S2C_GetList of name_version list
  | S2C_GetOpam of binary_data
  | S2C_GetArchive of binary_data archive
  | S2C_NewArchive of security_key option
  | S2C_UpdateArchive of bool

  | S2C_error of string
```

Note that when an error is raised by an arbitrary function at server side, the client receives `S2C_error _`.

1.3.1 Basic types

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

Names and version are strings. Archive are binary strings. Type `opam` is an in-memory representation of OPAM files described in §1.1.3.

1.3.2 Getting the list of packages

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

`getList()` returns the list of the available versions for all packages. The collection of pairs can be computed when the server starts and be cached in memory.

1.3.3 Getting OPAM files

```
val getOpam: (name * version) -> opam
```

`getOpam(name,version)` returns the in-memory representation of the OPAM file for the corresponding package version.

1.3.4 Getting package archive

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

`getArchive(name,version)` returns the corresponding package archive as a binary string.

1.3.5 Uploading new archives

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

`newArchive(opam,archive)` takes as input an OPAM file and the corresponding package archive (stored as binary string), and upload the server state.

1.4 Client commands

1.4.1 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[:PORT]]
```

Where `HOSTNAME` is an optional machine name specifying the OPAM repository address and `PORT` is an optional port name on which to connect to the repository. If no hostname is specified, default is `opam.ocamlpro.com`; the default port is 9999.

This command will:

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

   ```
   version: 1.0
   sources: HOSTNAME[:PORT]
   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 the available packages using `getList` (§1.3.2) and get all the corresponding OPAM files using `getOpam` (§1.3.3).

4. dump all the OPAM files into `$opam/index/NAME-VERSION.opam`.

5. create empty directories `$opam/archives`; and create `$lib` and `$bin` if they do not exist.

1.4.2 Listing packages

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

```
$ opam-info
```

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 the available versions of `NAME`. It can then display:

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

1.4.3 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 `$opam/index/NAME-*.opam` to find the latest version of the package.
2. compute the transitive closure of dependencies and conflicts of packages using the dependency solver (see §1.5). If the dependency solver returns more than one answer, the tool will ask the user to pick one, otherwise it will proceed directly.
3. the dependency solver should have sorted the collections of packages in topological order. Them, for each of them do:
   (a) check whether the package archive is installed by looking for the line `NAME VERSION` in `$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`. This file has the following format:

```
lib: *.cmi, *.cmo, *.cmx, *.cma
bin: foo
misc:
  config     /usr/share/foo/
  doc/*.html /usr/shar/html/foo/
```

Files listed under `lib` should be copied to `$lib/NAME/`. File listed under `bin` should be copied to `$bin/`. 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`.

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.

1.4.4 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 §1.3.2); then ask for the missing OPAM files using `getOpam` (see §1.3.3). Finally it will dump the missing OPAM files into `$opam/index/NAME.opam`.

1.4.5 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 §1.5) 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 §1.4.3.

1.4.6 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 -dir NAME
```

This command will return the directory where the package is installed, in a form suitable to OCaml compilers, ie. `-I $lib/NAME`. For the first version of OPAM, no linking information (such as library names) is provided, and it is not possible to ask for recursive queries. It is understood that this can be painful; it will be fixed in next milestones of OPAM.

1.4.7 Uploading packages

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

1. create `$package/NAME.opam` containing in the format specified in §1.1.3.
2. create `$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 1.3 to upload the package on the server.

1.5 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 §1.1.3) 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 Milestone 5: Local Packages

This milestone focus on giving to the end-user the possibility to use local packages.

From the end-user perspective, local packages look similar to normal packages: their index files are stored in `$opam/index/NAME-VERSION.opam` and their archive files are stored in `$opam/archive/NAME-VERSION.opam`. However, local package only exist in the client state: they do not appear in the server state.

5.1 OPAM file

Local packages and normal packages are distinguished by a new field `local` (whose default value is `false`) (See §1.1.3 for a full description of the OPAM format). So for instance, a local package OPAM file will looks like:

```
opam-version: 1.0

package:     NAME
version:     VERSION
local:       true
description: TEXT
depends:     FORMULAE
conflicts:   FORMULAE
```

5.2 Creating local packages

When an end-user/packager wants to create a local package, he will follow the work-flow as described in §1.4.7 but he will finally run the following command:

```
$ opam-upload --local NAME
```

This command will use the server API(§1.3) *as-a-library*, with `$opamserver=$opam` (ie. the local server state is contained into the client state). In this case, we will not need a separate process to act as the server; thus it will not be necessary to use a binary protocol to exchange data between processes, and we will not need to use the server authentication protocols defined in previous milestones.

5.3 Conflicts

When the end-user updates the list of available packages, the local packages have priority (ie. `opam-update` will never overwrite a local package).

6 Milestones 6: Pre-Processors Information

This milestone focus on the support of pre-processors.

6.1 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: ...
}
```

7 Milestones 7: Support of Multiple Compiler Versions

This milestone focus on the support of multiple compiler versions.

7.1 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
```

7.2 Milestone 8: Version Pinning

7.3 Milestones 9: Parallel Build

7.4 Milestone 10: Version Comparison Scheme

7.5 Milestone 11: Database of Installed Files