The OpenCPU Server PDF Manual

Version 2.0

The latest version of this document is available at https://github.com/opencpu/server-manual. Typos, comments, suggestions about this manual go in the issues page for the repo.

Contents

1. What is OpenCPU

`OpenCPU` is a system for embedded scientific computing and reproducible research. The server exposes an `HTTP API` to develop and execute scripts, functions and reports. The system addresses many of the domain specific problems inherent to scientific computing, and abstracts away technicalities behind a well defined intuitive `HTTP` interface. This provides a foundation for scalable applications with embedded statistical analysis, visualization and reporting.

`OpenCPU` uses `R` only for what it is good at: analysis and graphics. The system separates the statistical computing from other parts of your application. `OpenCPU` runs on a remote server, interfaced only through the `HTTP API`. Clients need no knowledge of `R`: the `OpenCPU API` defines a mapping between `HTTP` requests and `R` function calls which results in a natural `RPC` protocol. Any software program that speaks `HTTP` can call `R` functions and scripts, without the need to understand, generate or parse `R` code.

```
curl http://localhost/ocpu/library/stats/R/rnorm/json --data n=3
[
  3.05644,
  0.38511,
  1.11983
]
```

From there it is up to the client on how to process or present the output. `OpenCPU` deliberately does not include, suggest or enforce the use of any specific web development language, `GUI`, etc. You are not restricted to a limited set of available widgets or panels that fits the `R` paradigm. `OpenCPU` manages incoming requests, security, resource allocation, data `I/O` and other technicalities. Nothing more, nothing less. This is the main difference with frameworks that include built-in templates to generate out-of-the-box widgets from `R` code. Emphasis in the design of `OpenCPU` is on interoperability, reliability and simplicity in order to develop scalable, production quality software. Because `OpenCPU` layers on `HTTP`, we can leverage existing technology (cache-control, `SSL`, load balancing, etc). Finally, the `API` definition makes it possible to gracefully extend, re-implement or replace parts of the system in the future.

1.1. Using OpenCPU in a team

`OpenCPU` decouples the roles of analyst and web developer. The analyst implements and documents `R` functions or scripts, just as he is used to. The web developers can use their favorite language, tools and web frameworks to call such analyses over `HTTP`. `OpenCPU` provides the bridge between these two systems, without imposing additional constraints on either side. There is no need for the web developer to learn `R`, nor does the analyst have to worry about `GUI` or web related technicalities. This separation is supposed to keep applications well organized and easy to maintain, while making collaboration within a team of statisticians and developers easier than when the two parts are tightly intertwined.

1.2. Security model

Rather than traditional user-role based access control, `OpenCPU` uses mandatory access control (`MAC`) based security. On Debian/Ubuntu systems, this is implemented using `AppArmor`: a

security module in the `Linux` kernel. Thereby we can define and enforce security policies on a by-process level. This protects against system abuse or excessive use of hardware resources without limiting the `R` functionality that clients can use. The security policies are completely customizable, and section 3.5 talks a bit more on security and `AppArmor`.

Because Redhat systems do not support `AppArmor`, `OpenCPU` runs without the advanced security policies on these platforms. Instead, it runs in the standard SElinux `httpd_modules_t` context. This is fine for internal use, but it is not recommended to expose your Fedora/EL OpenCPU server to the web without further configuring SELinux for your application.

The `MAC` based security profiles make it possible to open up an `OpenCPU` server directly to the web. However it is also perfectly fine to employ `OpenCPU` solely as a computational back-end inside some larger system or application, similar to e.g. your database server. In such a design, users do not interact directly with the `OpenCPU` server, and can only use functionality exposed in your application layer.

1.3. OpenCPU Apps

`OpenCPU` defines a standard way to build and share *apps*. An `OpenCPU` app is an `R` package which, in addition to the regular contents, ships with some web page(s). These pages interact with the `R` functions in this package through the `OpenCPU API`. By convention, these web pages (`html`/`css`/`js` files) are included in the `/inst/www/` directory of the `R` source package. This way, `OpenCPU` apps provide a convenient way to package and ship standalone `R` web applications.

Because `OpenCPU` apps are simply `R` packages, they are developed, distributed and installed the same way as any other `R` package. Several example apps are available from the `rwebapps` github organization at https://github.com/rwebapps. For example to run the markdownapp app:

```
library(opencpu)
ocpu_start_app("rwebapps/markdownapp")
```

If the application was installed on an `OpenCPU` cloud server, the app is directly available. Point your browser to http://your.server.com/ocpu/apps/rwebapps/markdownapp.

To make development of `OpenCPU` apps easier, a `JavaScript` client library is available called opencpu.js. This library depends on `jQuery` and uses `$.ajax` to provide `JavaScript` wrappers to the `OpenCPU API`. It is not required to use this `JavaScript` library but it provides a convenient basis for building `R` web applications.

1.4. The OpenCPU single-user server

Two implementations of `OpenCPU` are available: a single-user server that runs inside an interactive `R` session, and a cloud server that builds on `apache` and `nginx`. The single-user server is intended for development and local use only. The latest version can easily be installed in `R` from `CRAN`:

```
install.packages("opencpu")
```

Load the `OpenCPU` package to start the local server:

```
library(opencpu)
ocpu_start_server()
```

Because `R` is single-threaded, the single-user server does not support concurrent requests (but httpuv does a good job in queuing them). Also it does not enforce any security. The single-user server is great for developing apps, that can later be published on the `OpenCPU` cloud server. When using the single-user server, we can easily load the apps for local use:

```
ocpu_start_app("rwebapps/markdownapp")
```

The `ocpu_start_app` function will automatically open the app in the default web browser.

2. Installing the OpenCPU cloud server

The `OpenCPU` cloud server version 2.0 runs on **Ubuntu 16.04 (Xenial) or higher**. The `OpenCPU` system consists of a number of standard Ubuntu installation packages. These are:

- `opencpu` – Alias for `opencpu-server`.
- `opencpu-server` – The main `OpenCPU` API server. Depends on `R` and `apache2`.
- `opencpu-full` – Installs `opencpu`, `texlive`, `git`, `rstudio-server` and more.
- `opencpu-cache` – Caching server. Not recommended for users. For internal use only.

2.1. Getting an Ubuntu server

The recommended installation of `OpenCPU` 2.0 requires Ubuntu version 16.04. Any flavour of Ubuntu will do, e.g. Ubuntu Desktop, Ubuntu Server, Kubuntu, Edubuntu, etc. The preferred way of running `OpenCPU` is on a clean Ubuntu Server edition. A copy of the Ubuntu Server installation disc ISO can be obtained from the Ubuntu download pages at http://www.ubuntu.com/download/server.

Another way to get started is by spinning up a pay-by-hour server instance on Amazon EC2. When using `OpenCPU` on EC2, it is highly recommended to pick one of the **compute optimized** instance types, for example `c3.large`. The Ubuntu team provides very nice ready-to-go daily updated preinstalled Ubuntu-Server images for EC2 at http://cloud-images.ubuntu.com/trusty/current/.

Yet another possibility is to install Ubuntu Server on a virtual machine within another OS. On Linux we can use KVM, for Windows the free VMware Player or Oracle Virtualbox are available, and on OSX we can use Virtualbox or Parallels to run virtual machines. Regardless of the platform, it is good practice to run the `OpenCPU` cloud server on a separate virtual machine rather than installing it directly in your application server. Isolating the `OpenCPU` server within a virtual machine is nice from a security point of view and prevents `OpenCPU` or `R` from interfering with other software on your server. Furthermore hypervisors such as KVM provide nice control over hardware resources for guest machines. The `OpenCPU` public demo servers run as virtual machines on a KVM hypervisor which has resulted in excellent performance, smooth upgrades and no major incidents for several years.

2.2. Basic OpenCPU installation

Before installing `OpenCPU`, make sure the system is up to date:

```
sudo apt-get update
sudo apt-get upgrade
```

To install `OpenCPU` start by adding the repository to the system:

```
sudo add-apt-repository ppa:opencpu/opencpu-2.0 -y
sudo apt-get update
```

We can now go ahead and install the server:

```
sudo apt-get install opencpu-server
```

Installation on a clean server might take a while because `R` and Latex both have many dependencies. After installation is done, we should be able to open a browser and point it to the `/ocpu` path at server address e.g: http(s)://your.server.com/ocpu. If the welcome page shows up, the installation has succeeded.

Finally, to get install `OpenCPU` together with a lot of other potentially useful things:

```
sudo apt-get install opencpu-full
```

This package will install `opencpu-server`, `texlive`, `rstudio-server`, `git` and some more. Note that this takes is at least several GB of disk space on a fresh system.

2.3. Uninstall OpenCPU

To uninstall either of the `OpenCPU` packages:

```
sudo apt-get purge opencpu-server
```

Removing the `opencpu` *repository* from the system is done by deleting the repository file from the `/etc/apt/sources.list.d/` directory.

3. Managing the OpenCPU cloud server

To enable `OpenCPU` in Apache2 on Ubuntu:

```
sudo a2ensite opencpu
sudo apachectl restart
```

To disable `OpenCPU` site in Apache2 on Ubuntu:

```
sudo a2dissite opencpu
sudo apachectl restart
```

3.1. Relevant log files

When things are not working as expected, in most cases the problem is reported by `apache` or `Linux`. If the `OpenCPU` server is not coming online at all, we might find out whats wrong from the web server error logs:

```
sudo tail /var/log/apache2/error.log
sudo tail /var/log/opencpu/apache_error.log
```

If you are seeing permission denied errors, either in the `OpenCPU API` or in the `apache` logs, this is likely a problem with the `AppArmor` security profile. `AppArmor` security violations are logged to `kern.log`:

```
sudo tail /var/log/kern.log
```

Section 3.5 has more information on customizing `OpenCPU` security policies in the cloud server.

3.2. Installing R packages on the server

In order for packages to be accessible through `/ocpu/library/`, they need to be installed in the global library. Note that `R` needs to be started as root to install packages in the global library. To install a single source package:

```
wget http://cran.r-project.org/src/contrib/glmnet_1.9-5.tar.gz
sudo R CMD INSTALL glmnet_1.9-5.tar.gz --library=/usr/local/lib/R/site-library
```

Alternatively we can use an interactive session to install packages from `CRAN` or `Github`:

```
sudo -i
R
```

From here it is business as usual:

```
install.packages("ggplot2")
install.packages("glmnet")

library(devtools)
install_github("dplyr", "hadley")
```

After restarting the `OpenCPU` service, the packages are accessible through the `API`:

```
http://your.server.com/ocpu/library/ggplot2
http://your.server.com/ocpu/library/glmnet
http://your.server.com/ocpu/library/dplyr
```

Alternatively, some precompiled `r-cran-xxx` packages are available from Ubuntu repositories:

```
sudo apt-get install r-cran-xml
```

Appendix A explains how to use third party repositories like `c2d4u` to install packages that are not included with Ubuntu by default.

3.3. Configuring the OpenCPU cloud server

The `OpenCPU` configuration file is written in `JSON` and located at `/etc/opencpu/server.conf`. It has a hand full of server-wide settings and options to enable/disable or fine-tune certain parts of the system. A few important options are:

- **httpcache.post** – Value of `cache-control` header: time in seconds that `HTTP POST` requests are cached if cache server is installed. Defaults to 5 minutes.

- **preload** – List of packages to be preloaded when the server starts. This can severely increase performance of frequently used packages.

- **timelimit.post** – Time in seconds after which `HTTP POST` requests get terminated. Defaults to 90 seconds.

- **rlimit.as** – Maximum memory that can be allocated by a single request. Default 2 GB.

- **rlimit.fsize** – Maximum size of files that a request can create. Default 100 MB.

- **rlimit.nproc** – Maximum number of concurrent processes that the server is allowed to use. Defaults to 50 processes.

To modify configurations, the administrator can edit this file or alternatively create a new file in the directory `server.conf.d`. Files in `/etc/opencpu/server.conf.d/` with a filename ending in `.conf` are automatically loaded by opencpu-server. These settings override settings in `/etc/opencpu/server.conf`. The server needs a restart for configurations to take effect:

```
sudo apachectl restart
```

Make sure that all configuration files are formatted using valid `JSON` at all times. Finally there is an additional optional configuration file that is only used for github authentication information at `/etc/opencpu/secret.conf`. This file is readable by the `OpenCPU` system, but not by `OpenCPU` users. It has three fields: `auth_token`, `client_id` and `client_secret`. Setting a valid `Github` "application access token" makes it possible to use continuous integration for private `Github` repositories. It also raises the hourly github `API` limits from 100 to 5000 hits per hour. Currently this is only relevant for listing repositories through the `/ocpu/gist` and `/ocpu/github` API.

3.4. Setting up SSL certificates for HTTPS

Installation of `OpenCPU` will automatically enable `HTTPS` in your web server. Hence you should be able to access `OpenCPU` via `https://your.server.com/ocpu`. By default, the server uses self-signed "snake-oil" `SSL` certificates that are included with Ubuntu. These are convenient for development, but don't provide real security and will trigger the "untrusted host" warning in most modern browsers. Once you go in production, it is recommended to replace these certificates with your own `SSL` certificates, signed by a certificate authority.

To set up `HTTPS` for `opencpu-server` the certificates need to be configured in `apache`. This is done in `/etc/apache2/sites-available/default-ssl`. The file contains comments with further instructions. After configuring the certificates, restart `apache` for things to take effect.

3.5. Customizing the security profile

The `OpenCPU` cloud server uses `AppArmor` profiles and the `RAppArmor` package to enforce security policies. The profiles used by `OpenCPU` are stored in `/etc/apparmor.d/opencpu.d/`. For an introduction on the topic see the RAppArmor Github page and RAppArmor article in JSS.

The default profiles included with `OpenCPU` are quite liberal but prevent most types of malicious behavior. However, if you plan to use `OpenCPU` in production, it is recommended to review these policies and revise them according to your needs. To keep your custom rules separated from the general profiles, add them to the `/etc/apparmor.d/opencpu.d/custom` file. After modifying a security profile restart `AppArmor`:

```
sudo service apparmor restart
```

One nice way to debug a security profile is by monitoring the `kern.log` file while using `OpenCPU`:

```
sudo tail -f /var/log/kern.log | grep opencpu
```

If any `R` package or process called from `OpenCPU` attempts to do something that is not allowed in the current security profile, a line containing `DENIED` is printed to `kern.log`.

4. Testing the OpenCPU API

A complete overview of the `OpenCPU API` is available on the website: www.opencpu.org. Below some examples that illustrate basic functionality and verify that things are working. Also note that `OpenCPU` ships with a little testing page that can be used to do `HTTP` requests. The testing page can be found at http://your.server.com/ocpu/test.

4.1. Using Curl to test

Basic `HTTP GET` requests can be performed either by simply opening a URL in a web browser, or with a curl command:

```
curl -L http://your.server.com/ocpu/library/
abc
abind
abn
...
```

The `-L` flag makes `curl` follow redirects which is sometimes needed. In addition, the `-v` flag prints more verbose output which includes `HTTP` request/response headers:

```
curl -L -v http://your.server.com/ocpu/library/
```

To perform a `HTTP POST` in curl, either use the `-X POST` or `-d [args]` flag. For example:

```
curl http://your.server.com/ocpu/library/utils/R/sessionInfo -X POST
curl http://your.server.com/ocpu/library/stats/R/rnorm -d "n=10&mean=100"
```

4.2. Reading package objects, manuals and files

The following URLs are examples of resources in `OpenCPU` which are retrieved using `HTTP GET`:

```
http://your.server.com/ocpu/test
http://your.server.com/ocpu/library
http://your.server.com/ocpu/library/stats/R
http://your.server.com/ocpu/library/stats/R/rnorm
http://your.server.com/ocpu/library/stats/man
http://your.server.com/ocpu/library/stats/man/rnorm/text
http://your.server.com/ocpu/library/stats/man/rnorm/html
http://your.server.com/ocpu/library/stats/man/rnorm/pdf
http://your.server.com/ocpu/library/MASS
http://your.server.com/ocpu/library/MASS/data/Boston
http://your.server.com/ocpu/library/MASS/data/Boston/json
http://your.server.com/ocpu/library/MASS/data/Boston/csv
http://your.server.com/ocpu/library/MASS/data/Boston/rda
```

Apart from `R` objects and manuals, `OpenCPU` also hosts any static files in package:

```
http://your.server.com/ocpu/library/knitr/examples/
http://your.server.com/ocpu/library/knitr/examples/knitr-minimal.Rnw
http://your.server.com/ocpu/library/knitr/doc/knitr-intro.R
http://your.server.com/ocpu/library/brew/example1.brew
http://your.server.com/ocpu/library/devtools/NEWS
http://your.server.com/ocpu/library/devtools/DESCRIPTION
```

4.3. Calling a function

Performing a `HTTP POST` on a function results in a function call where the `HTTP` request arguments are mapped to the function call. In `OpenCPU`, a successful `POST` requests usually returns a `HTTP 201` status, and the response body contains the locations of the output data:

```
curl http://your.server.com/ocpu/library/stats/R/rnorm -d "n=10&mean=100"
/ocpu/tmp/x032a8fee/R/.val
/ocpu/tmp/x032a8fee/stdout
/ocpu/tmp/x032a8fee/source
/ocpu/tmp/x032a8fee/console
/ocpu/tmp/x032a8fee/info
```

The output can then be retrieved using `HTTP GET`. When calling an `R` function, the output object is always called `.val`. However, calling scripts might result in other `R` objects. In this case we could `GET`:

```
http://your.server.com/ocpu/tmp/x032a8fee/R/.val
http://your.server.com/ocpu/tmp/x032a8fee/R/.val/json
http://your.server.com/ocpu/tmp/x032a8fee/R/.val/ascii
http://your.server.com/ocpu/tmp/x032a8fee/R/.val/rda
http://your.server.com/ocpu/tmp/x032a8fee/console
```

In a very similar fashion we can produce a plot:

```
curl http://your.server.com/ocpu/library/ggplot2/R/qplot \
  -d "x=[1,2,3,4,5]&y=[8,9,7,8,7]&geom='line'"
/ocpu/tmp/x07a773be/R/.val
/ocpu/tmp/x07a773be/graphics/1
/ocpu/tmp/x07a773be/source
/ocpu/tmp/x07a773be/console
/ocpu/tmp/x07a773be/info
```

Which we can then retrieve in PNG, PDF or SVG format using `HTTP GET`:

```
http://server.com/ocpu/tmp/x07a773be/graphics/1/png?width=800&height=500
http://server.com/ocpu/tmp/x07a773be/graphics/1/pdf?width=12&height=7
http://server.com/ocpu/tmp/x07a773be/graphics/1/svg?width=12&height=7
```

There is one exception to this rule: in the common special case where the client is only interested in the `JSON` representation of the object returned by the function, the `HTTP POST` request path can be post-fixed with `/json`:

```
curl http://your.server.com/ocpu/library/stats/R/rnorm/json -d "n=2"
[
  -1.2804,
  -0.75013
]
```

In this case, a successful call will return 200 (instead of 201), and the response body contains the output from the function in `JSON`; no need to do an additional `GET` request. However in most cases, the two-step procedure is preferred.

Finally It is also possible to specify *input* arguments to a function call using `JSON`:

```
curl http://your.server.com/ocpu/library/stats/R/rnorm \
-H "Content-Type: application/json" -d '{"n":10, "mean": 10, "sd":10}'
```

4.4. Executing a script

Besides calling a function, the `HTTP POST` method can also be used to execute a script. The script is interpreted according to it's file extension. Currently the following extensions are supported: `R` (Rscript), `Rnw`, `Rmd` (knitr), `brew` (brew), `tex` (latex), `md` (pandoc):

```
curl -X POST http://server.com/ocpu/library/knitr/examples/knitr-minimal.Rnw
curl -X POST http://server.com/ocpu/library/knitr/examples/knitr-minimal.Rmd
curl -X POST http://server.com/ocpu/library/knitr/doc/knitr-intro.R
curl -X POST http://server.com/ocpu/library/brew/example1.brew
```

In contrast to a function, a script has no formal arguments. Instead, the `HTTP POST` arguments can be passed to the script interpreter. For example, when calling a `brew` script, we can pass a value for the `output` argument in brew (e.g. to send the output stream to a specific file), and `Rmd`/`md` files have a `format` argument to specify the pandoc output format:

```
curl server.com/ocpu/library/knitr/examples/knitr-minimal.Rmd -d format=docx
curl server.com/ocpu/library/knitr/examples/knitr-minimal.Rmd -d format=odt
curl server.com/ocpu/library/knitr/examples/knitr-minimal.Rmd -d format=html
curl server.com/ocpu/library/brew/example1.brew -d output=out.txt
```

4.5. Online package repositories

If the `OpenCPU` cloud server has direct access to the internet, it can also provide access to packages and/or scripts from online repositories, such as `CRAN` or `Github`. This is convenient for development: package authors and users can test and use `OpenCPU` apps without the need for an administrator to install packages on the server. However, note that the administrator can easily disable these features so they might not be available on every `OpenCPU` server.

Github

The `/ocpu/apps/:user/:package/` API interfaces to **packages** which are on the `master` branch in a `Github` user repository. For this to work, the github repository must contain the source `R` package, and the repository name must be identical to the package name:

http://your.server.com/ocpu/apps/jeroen/
http://your.server.com/ocpu/apps/jeroen/curl
http://your.server.com/ocpu/apps/jeroen/curl/R
http://your.server.com/ocpu/apps/jeroen/curl/R/curl_fetch_memory
http://your.server.com/ocpu/apps/jeroen/curl/man/curl_fetch_memory

Appendices

A. Installing `r-cran` packages from `c2d4u`

R packages can be installed on the server in the usual ways, as was described earlier. However, on `Linux`, `R` packages need to be compiled from source which requires more time and additional build dependencies. As an alternative, Dirk Eddelbuettel and Michael Rutter have "debianized" many of the popular `R` packages on `CRAN` and Bioconductor. Thereby, precompiled binaries of these `R` packages can be installed using `apt-get`. For Ubuntu, many of these packages are available from Launchpad:

```
sudo add-apt-repository ppa:marutter/rrutter -y
sudo add-apt-repository ppa:marutter/c2d4u -y
sudo apt-get update
```

This adds the required repositories to the system and retrieves the latest package list. To see which packages are currently available:

```
apt-cache search r-cran
apt-cache search r-bioc
```

And to install a package:

```
sudo apt-get install r-cran-ggplot2
```

And then navigate to http://your.server.com/ocpu/library/ggplot2/

B. Using OpenCPU with RStudio

The OpenCPU cloud server works great together with RStudio server. Both systems run best on Ubuntu and will happily share the same R installation. You can use RStudio server to develop and install R packages/apps and make them available through the OpenCPU API. Starting version 1.3, the OpenCPU repository includes a copy of rstudio-server for easy installation.

```
sudo apt-get install rstudio-server
```

By default both apache and nginx are configured to proxy the /rstudio/ path. Hence after installing both opencpu and rstudio-server they can be accessed directly through:

https://your.server.com/ocpu/
https://your.server.com/rstudio/

Consult https://www.rstudio.com/ide/docs/server/ for information on managing and configuring rstudio-server. Note that when a non-root user installs a package in R or RStudio, the package will not be installed in the global library, but in the user personal package library. Personal package libraries are available through the API at the /user/:name/library, e.g: https://your.server.com/ocpu/user/jeroen/library.

Note that user libraries are only be available to OpenCPU if the respective user has set the file mode permissions (chmod) to public readable (which is the default on Ubuntu).