The OpenCPU Server PDF Manual

Version 1.0

Jeroen Ooms

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

Contents

1. What is OpenCPU

Internet access, public cloud computing, live and open data and scientific super computers are transforming the landscape of data analysis. Researchers increasingly collaborate by sharing data, code and results online. This is a powerful way to learn and teach statistical analysis and at the same time it improves transparency and accessibility of the underlying methods. Moreover, by centralizing computation we can address some scalability challenges and facilitate direct integration of analyses in systems and applications. We expect that producing reproduciuble materials and analysis tools, in addition to a written report or article, will soon become an integral part of the scientific publication proces. The OpenCPU framework is a first attempt at building the foundations to support such integrated, collaborative, reproducible analysis systems.

The OpenCPU framework is based on R, Latex and Pandoc. The server exposes an HTTP API to share and execute scripts, functions and reproducible documents. 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, vizualization and reporting.

1.1. Seperating analysis from web development

When using OpenCPU, R is only used for what it is good at: analysis and graphics. The framework completely seperates the statistical computing from other parts of your system or application. OpenCPU runs on a remote server, interfaced only through the HTTP API. Clients need no knowledge of R or Latex; the OpenCPU API defines a mapping between HTTP requests and R function calls which results in a natural RPC-like 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 completely 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 computing technicalities. From there, it is completely up to the developer(s) and their imagination how to design their application. This is a major difference with frameworks that ship with built-in templates to generate parameratized out-of-the-box widgets from R code. Emphasis in the design of OpenCPU is on reliability and simplicity in order to develop scalable, production quality software. Because OpenCPU layers on HTTP, we can leverage existing technology (cache-control, https, load balancing, etc). Finally, the API definition makes it possible to gracefully extend, reimplement or replace parts of the system in the future.

1.2. 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 GUIs or web related technicalities. This seperation 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.

1.3. OpenCPU is really open

The design of OpenCPU has been driven by the paradigms of open-source, contributed code and reproducibile research which are central to the domain of scientific computing. This is where the design of OpenCPU deviates from a web development frameworks in other languages. A client is allowed to call any script or function from any package, and all source code is readable as well. In unix analogy: any user has `r-x` access on all functions and scripts on the system. OpenCPU does not restrict users to some predefined set of functionality: instead, all users are explicitly allowed to store and execute custom code on the server. OpenCPU invites users to play, share, run, produce and reproduce results with each other.

For these reasons, OpenCPU takes a different approach to security. OpenCPU uses AppArmor: a security module in the Linux kernel, to enforce security policies on a by-process level. These security restrictions make it possible to allow arbitrary code execution, while protecting against system abuse or excessive use of hardware resources. These policies are completely customisable, and section 3.5 talks a bit more on security and AppArmor. However, even though the system is designed to be open, it is not mandatory to use it that way. It is perfectly fine to employ OpenCPU solely as a reliable 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.

1.4. 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/etc 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 OpenCPU github organization at https://github.com/opencpu. For example to install the gitstats app, we can use the `devtools` package:

```
library(devtools)
install_github("gitstats", "opencpu")
```

If the application was installed on an OpenCPU cloud server, the app is directly available. Point your browser to http://your.server.com/ocpu/library/gitstats/www.

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 mandatory to use this javascript library, but it provides a convenient basis for building OpenCPU apps.

1.5. The OpenCPU single-user server

Two versions 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 the Github repository:

```
library(devtools)
install_github("opencpu", "jeroenooms")
```

When the opencpu package is loaded, the server is automatically started:

```
library(opencpu)
```

Because R is single-threaded, the single-user server does not support concurrent requests (but httpuv does a good job in queueing 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:

```
install_github("gitstats", "opencpu")
opencpu$browse("library/gitstats/www")
```

The `opencpu$browse` function will automatically open the app in the default web browser.

2. Installing the OpenCPU cloud server

The OpenCPU cloud server runs on Ubuntu 12.04 (precise) or higher. The OpenCPU system consists of a number of standard Ubuntu installation packages. These are:

- `opencpu` – Meta package which installs both `opencpu-server` and `opencpu-cache`.
- `opencpu-server` – The main OpenCPU API server. Depends on `R` and `apache2`.
- `opencpu-cache` – OpenCPU caching server. Depends on `nginx`.
- `opencpu-full` – Installs `opencpu` plus many `texlive` and `pandoc` packages.

2.1. Getting an Ubuntu server

OpenCPU requires Ubuntu version 12.04 or higher. Any version 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:

```
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 recommended to pick one of the *compute optimized* instance types, for example `c1.medium` or `c1.large`. The Ubuntu team provides very nice ready-to-go daily updated preinstalled Ubuntu-Server images for EC2:

```
http://cloud-images.ubuntu.com/raring/current/
```

Yet another possibility is to install Ubuntu on a virtual server inside another OS. For example, the free VMware Player is available for Windows, and on OSX we can use parallels to run virtual machines. This way Ubuntu and OpenCPU can safely be installed on top of a Windows or Mac system.

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, first add the OpenCPU repository to the system:

```
sudo add-apt-repository ppa:opencpu/opencpu-1.0 -y
sudo apt-get update
```

We can now go ahead and install the server:

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

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.

2.3. Custom OpenCPU installation

When installing the `opencpu` package as described above, both the OpenCPU API server (`opencpu-server`) as well as the OpenCPU cache server (`opencpu-cache`) are installed. However one can also install just one or the other. For example to install only the API server we could do:

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

Alternatively, to install only the cache server:

```
sudo apt-get install opencpu-cache
```

Note that installing just `opencpu-cache` is only useful in combination with another server running `opencpu-server`. This is how you would set up a load-balancer. Some additional configuration if needed to make the cache server proxy to OpenCPU servers other than your.server.com.

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`, `opencpu-cache`, `pandoc`, `texlive`, and much more. Note that this takes is at least several GB of disk space on a fresh system.

2.4. Uninstall OpenCPU

To uninstall either of the OpenCPU packages:

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

Alternatively, to remove all of OpenCPU at once:

```
sudo apt-get purge opencpu-*
```

Removing the `opencpu-0-8` *repository* from the system is done by deleting the file from the `/etc/apt/sources.list.d/` directory.

3. Managing the OpenCPU cloud server

To control the OpenCPU server:

```
sudo service opencpu start
sudo service opencpu stop
sudo service opencpu restart
```

This will automatically enable/disable OpenCPU in Apache2 and restart the server. If the `opencpu-cache` package has been installed this server can be controlled seperately:

```
sudo service opencpu-cache start
sudo service opencpu-cache stop
sudo service opencpu-cache restart
```

This will flush the cache and restart nginx. **Note that the cache server automatically sets up `iptables` to preroute incoming web traffic (80/443) through nginx.** This is one of the reasons why it is recommended to run OpenCPU on its own server.

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 `apache2` error log:

```
/var/log/apache2/error.log
```

If you are seeing permission denied errors, either in the OpenCPU API or in the `apache2` logs, this is likely a problem with the AppArmor security profile. AppArmor security violations are logged to:

```
/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 the `/ocpu/library` API, 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:

```
sudo R CMD INSTALL glmnet_1.9-5.tar.gz
```

Alternatively we can start an interactive session and install packages straight from CRAN or Github:

```
sudo 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 accassible 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 straight from the Ubuntu repositories:

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

However, note that these packages will only work if they were built using R version 3.0 or higher, which is **not the case** for the standard packages that ship with Ubuntu Raring 13.04 or lower. Appendix A explains how to use third party repositories like `c2d4u` instead.

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 finetune certain parts of the system. 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 that ends in `".conf"` will automatically be loaded by opencpu-server. These settings will override settings in `/etc/opencpu/server.conf`. The server needs a restart for changed configurations to take effect:

```
sudo service opencpu restart
```

Please make sure that any configuration files are formatted using valid JSON at all times. Finally their is an additional, not required configuration file that is only used for github authentication information:

```
/etc/opencpu/secret.conf
```

This file is readable by the OpenCPU system, but not by OpenCPU users. It has only two fields: `client_id` and `client_secret`. Setting a valid Github "application access token" will raise hourly github api limits from 100 to 5000 hits per hour. Currently this is only relevant for listing repositories from a user though 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 "snakeoil" SSL certificates that ship 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 when using `opencpu-server` *without* `opencpu-cache`, the certificates need to be configured in Apache2. This is done by editing the following file:

```
/etc/apache2/sites-available/default-ssl
```

The file contains comments with further instructions. On the other hand, on a server which does have `opencpu-cache` installed, the certifiates neet to be configured in the following `Nginx` configuration file:

```
/etc/nginx/sites-available/opencpu
```

After installing the certificates, restart apache2/nginx 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. For an introduction on the topic see the RAppArmor Github page and RAppArmor article in JSS. The profiles used by the OpenCPU cloud server are stored in:

```
/etc/apparmor.d/opencpu.d/
```

The default profiles included with OpenCPU are quite liberal but prevent most types of malicous 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 seperated from the general profiles, add them to the `/etc/apparmor.d/opencpu.d/custom` file. After modifying a security profile restart AppArmor and OpenCPU:

```
sudo service apparmor restart
sudo service opencpu 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 and demonstration 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/
```

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
```

Note that when using curl in the command line, shell characters such as `&` or `"` need to be escaped:

```
curl http://server.com/ocpu/library/graphics/R/plot -d x=cars\&main=\"Test\"
```

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/datasets
http://your.server.com/ocpu/library/datasets/R/cars
http://your.server.com/ocpu/library/datasets/R/cars/json
http://your.server.com/ocpu/library/datasets/R/cars/csv
http://your.server.com/ocpu/library/datasets/R/cars/tab
http://your.server.com/ocpu/library/datasets/R/cars/rda
```

Appart 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 fasion 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=600
http://server.com/ocpu/tmp/x07a773be/graphics/1/pdf?width=12&height=8
http://server.com/ocpu/tmp/x07a773be/graphics/1/svg?width=12&height=8
```

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 successfull call will not 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.

When a client requests a package from an online repository which is not already installed on the server, OpenCPU will attempt to install the package on the fly. For this reason, the first request might take quite a while. Also the request will fail if the package installation fails (for example due to missing dependencies). But once the package has been successfully installed, it will behave just as any other package on the server.

For performance reasons, the current implementation updates packages from external no more than once per day. Hence, changes that were recently pushed to Github might not immediately be refelected by the OpenCPU server.

CRAN

The `/ocpu/cran/:package/` API interfaces to packages which are *current* on CRAN:

```
http://your.server.com/ocpu/cran/
http://your.server.com/ocpu/cran/plyr
http://your.server.com/ocpu/cran/plyr/R
http://your.server.com/ocpu/cran/plyr/R/ldply
http://your.server.com/ocpu/cran/plyr/man
http://your.server.com/ocpu/cran/plyr/man/ldply
http://your.server.com/ocpu/cran/plyr/man/ldply/html
http://your.server.com/ocpu/cran/plyr/man/ldply/pdf
```

Github

The `/ocpu/github/: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/github/hadley/
http://your.server.com/ocpu/github/hadley/plyr
```

http://your.server.com/ocpu/github/hadley/plyr/R
http://your.server.com/ocpu/github/hadley/plyr/R/ldply
http://your.server.com/ocpu/github/hadley/plyr/man/ldply

Gist

The `/ocpu/gist/:user/:gistid/` API interfaces to **files** (scripts, documents) in a repository from a certain Gist user. The `/ocpu/gist` API does **not support R packages**. Just files:

http://your.server.com/ocpu/gist/hadley/
http://your.server.com/ocpu/gist/hadley/5721744
http://your.server.com/ocpu/gist/hadley/5721744/html.r

Execute a script or reproducible document straight from gist:

```
curl -X POST http://your.server.com/ocpu/gist/hadley/5721744/html.r
```

4.6. Test the caching server

An easy way to test the caching server is by calling the same function twice in a short time. If the cache server is working, output from both calls will be identical (and the second request returns much faster):

```
curl http://your.server.com/ocpu/library/stats/R/rnorm -d n=10
curl http://your.server.com/ocpu/library/stats/R/rnorm -d n=10
```

Try to disable the cache server and run the same experiment again:

```
sudo service opencpu-cache stop
```

You should now see different responses. Afterwards, re-enable the caching server:

```
sudo service opencpu-cache start
```

Another way to infer if the caching server is active is by looking at the `Server` response header:

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

If the cache server is online the header will be `Server:nginx/1.2.6 (Ubuntu)`. If the cache server is offline, it will be `Server:Apache/2.2.22 (Ubuntu)`.

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 Micheal's Launchpad repository:

```
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:

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

This is great but there is one caveat: because the packages are precompiled, they will only work if they were built using R 3.0 or higher. Fortunately this is the case for almost all packages on `c2d4u`. However, it is wise to double check this in R after installing, by inspecting the `Built` column from the `installed.packages()` output:

```
allpackages <- installed.packages()
which(allpackages[, "Built"] < 3)
```

If any packages show up which have been built with an older version of R, it is best to uninstall them because they won't work and will cause an error when loaded.

B. Using OpenCPU with RStudio

The OpenCPU single-user server has been tested to work on RStudio desktop on both Windows and Mac, as well as in RStudio server. If RStudio server is hosted behind a firewall, `httpuv` might not be accessible, but users can still use the `/custom/ocpu` path on the RStudio server port. For example http://rstudio.server.com/custom/ocpu.

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 install R packages (and OpenCPU apps) to make them available through the OpenCPU API.

When a non-root user installs a package in RStudio, the package will not be installed in the global library, but in the user's personal package library. User's personal package libraries are available through the API at the `/user/:name/library`, e.g:

```
http://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).