Version 0.4-0

Guide to the remoter Package

Just the Basics

Drew Schmidt

Guide to the remoter Package

January 4, 2018

Drew Schmidt
wrathematics@gmail.com

Version 0.4-0

Acknowledgements and Disclaimer

Work for the **remoter** package is supported in part by the project *Harnessing Scalable Libraries for Statistical Computing on Modern Architectures and Bringing Statistics to Large Scale Computing* funded by the National Science Foundation Division of Mathematical Sciences under Grant No. 1418195.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. The findings and conclusions in this article have not been formally disseminated by the U.S. Department of Health & Human Services nor by the U.S. Department of Energy, and should not be construed to represent any determination or policy of University, Agency, Administration and National Laboratory.

The **remoter** logo comes from the image "Tradtelefon-illustration". Licensed under Public Domain via Commons.

This manual may be incorrect or out-of-date. The author(s) assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

This publication was typeset using LaTeX.

Contents

1 Introduction

The **remoter** package [11] allows you to control a remote R session from a local one. The local R session can be in a terminal, GUI, or IDE such as RStudio. The remote R session should be run in the background as, well, a server.

The package uses **ZeroMQ** [4] by way of the R package **pbdZMQ** [3] to handle communication. Our use of pbdZMQ is specialized to client/server communications, but the package is very general. For more details about **pbdZMQ** see the **pbdZMQ** package vignette [2].

The work for remoter was born out of the **pbdCS** package [10], which is part of the Programming with Big Data in R (pbdR) project[6]. pbdR is a series of R packages that enable the usage of the R language on large distributed machines, like clusters and supercomputers. See r-pbd.org/) for details.

1.1 Installation

You can install the stable version from CRAN using the usual `install.packages()`:

```
install.packages("remoter")
```

In order to be able to create and connect to secure servers, you need to also install the **sodium** package. The use of **sodium** is optional because it is a non-trivial systems dependency, but it is highly recommended. You can install it manually with a call to `install.packages("sodium")` or by installing **remoter** via:

```
install.packages("remoter", dependencies=TRUE)
```

For more information about the security features of **remoter**, see Section 6 below.

The development version is maintained on GitHub, and can easily be installed by any of the packages that offer installations from GitHub:

```
### Pick your preference
devtools::install_github("RBigData/remoter")
ghit::install_github("RBigData/remoter")
remotes::install_github("RBigData/remoter")
```

To simplify installations on cloud systems, we also have a Docker container available.

1.2 Package Functions

The package contains numerous functions. Some should be called from regular R, and others only from inside a running client.

The functions to call only from regular R (outside the client):

Function	Description
`server()`	Create a server
`client()`	Interactively connect to a server
`batch()`	Send batch commands to
`relay()`	Launch an intermediary to relay commands between client/server

Several of the functions to call only from inside the running client are:

Function	Description
`c2s()`	Transport an object from the client to the server.
`s2c()`	Transport an object from the server to the client.
`exit()`	Disconnect the client from the server.
`shutdown()`	Disconnect from and shut down the server.
`showlog()`	View the server log.
`evalc()`, `lsc()`, `rmc()`	Client versions of `eval()`, `ls()`, and `rm()`

We will discuss many of these functions throughout the remainder of this vignette.

2 Clients and Servers: Just the Basics

If you simply want to understand how **remoter** works, we do not need to involve remote computers right out the gate. Instead, we will create a local server and connect to it from another local R session.

So the first thing to do is to start up 2 separate R sessions. One will be the *server*, receiving commands, and the other will be the *client*, sending them.

2.1 The Server

In the R process designated to be the server, we will use the `server()` command to, well, start the server. Running this with no additional arguments will create a server. Optionally, one can specify a password via the `password` argument. Another useful feature is setting `showmsg=TRUE`, which will show in the server R process what messages are coming in. For now, let's run it with `showmsg=TRUE`:

```
remoter::server(showmsg=TRUE)
```

That's it! That R session is now listening for commands. We can shut the server down in a few ways. Probably the best (particularly when dealing with remote machines) is from the client itself. More on this later. The other way is to kill the hosting R process. Finally, you can terminate the server with `ctrl+c`, but the other methods are preferred.

2.2 The Client

Once the server is set up, we can connect to it with the `client()` command. Since we are connecting to a local server, the address we want to connect to is `"localhost"` (the default) or `"127.0.0.1"`. We will have to make sure that the `port` argument matches the listening port of our server, or we'll never connect. Finally, we can set the way the R prompt looks while the client is running by the `prompt` argument. You can set it to whatever you like, but disambiguating between your regular, local R session and the **remoter** client is very useful. Things can get confusing in a hurry if you aren't careful.

So to connect, in our R session designated to be the client (the only one left), we would enter:

```
remoter::client()
```

And you should be good to go. You can now enter R commands in the client and have them executed on the server. The following section will go into more detail about specifics on using the client/server setup.

3 Using remoter Interactively

Before proceeding, make sure you understand how to set up a client and a server. See the previous section for details.

3.1 Philosophy

By default, all code entered to the client is executed on the remote server. There are several utility functions to help execute code on the local R session (see section below). But you should assume that anything entered into the client session, *unless you explicitly specify to the contrary*, is executed only on the server.

3.2 Utility Functions

There are a few utility functions available that have to do with handling execution of things locally or moving data between client and server.

By default, all commands entered inside of the client are executed on the server. If you need to do some things in the local R session, you can kill the client and just reconnect when you're ready. Alternatively, you can use the `lsc()`, `rmc()`, and `evalc()` functions. These are client versions of `ls()`, `rm()`, and `eval()`.

For moving data between client and server, there are the `s2c()` and `c2s()` commands which transfer from server to client and from client to server, respectively. These functions transfer data that resides in the memory of the client/server. To transfer a file in chunks (without reading all of it into memory), see `?pbdZMQ::zmq.sendfile` or `?pbdZMQ::zmq.recvfile`.

3.3 Shutting Things Down

To terminate the client, enter the command `exit()`. By default, this will terminate the local client only, and leave the server running. If you wish to also shut down the server with the client, you can run `exit(client.only=FALSE)`. For hopefully obvious reasons, you can not terminate the server and leave the client running.

From the client side, running `exit()` will not shut down the interactive R session that was hosting the client. You can also disconnect the client from the server without shutting down the server by killing the client R session or executing `Ctrl-c` in the client.

4 Using remoter in Batch

Not every workflow works best interactively. This is why we also offer the `batch()` function. This allows you to pipe a script (either in a separate file, or typed out in the R session — examples below) to a **remoter** server without having to interactively control things.

Perhaps the simplest example is using the `script=` argument of `batch()`.

```
remoter::batch(script="1+1")
```

Then, assuming a **remoter** session is running on the local machine[1] the scintillating result of "2" will be returned.

You can also easily pipe off longer scripts stored in separate files. Say you have a script "myscript.r" like so:

myscript.r

```
x <- 1
y <- 2
x + y
```

Then you can send it for evaluation to the **remoter** server by running:

```
remoter::batch(file="myscript.r")
```

We conclude with a note of caution. If you have a "master script" that calls `source()` (or similar) on other scripts in different files, this will not work without modification. You should change your `source()` calls to the appropriate `batch()` call. In fact, you may think of `batch()` as `source()` for remote servers.

5 Security

Security in **remoter** comes in two forms currently:

1. password credentialing
2. public key encryption

The password is declared when the server is spawned as a launch option in `remoter::server()`. Without the use of encryption, it will be transmitted from client to server unsecurely.

Encryption is optional, and disabled by default. This is because encryption is handled by the **sodium** package [5], which uses the **libsodium** [1] library, which can be difficult to build on some platforms.

If you have the **sodium** package installed *on both the client and the server*, start the server with the option `secure=TRUE`, and your client will automatically connect securely. If the server was launched (by necessity or optionally) with `secure=FALSE`, then the client can not connect securely, even if the client machine has the **sodium** package installed.

If ever in any doubt, use the `is.secure()` command from the client to see if communications are encrypted.

6 Problems, Bugs, and Other Maladies

The package should basically be useable, but there are some issues you might want to be aware of.

[1] for using `batch()` with remote servers, the same caveats and rules apply as for `client()` — see the *Using remoter with Remote Machines* [9] guide for details

Problem: I lost my internet connection and the client is no longer sending messages.

Solution: Just `Ctrl+c` the client and re-run the `remoter::client()` call and you should be good to go. The server should still be running. You can therefore also have multiple clients connect to the same server, and they will share the same data (though they will not see each other's commands). I actually consider this a feature, but I'm not married to it and I could probably be convinced to change it.

Problem: The up/down arrow keys don't work right in the R terminal when using the client.

Explanation: That's because the client is just some R code sitting on top of the R REPL. This shouldn't be a problem if you're using an IDE like RStudio or the like, where you pass commands from an editor window to the R session. But as far as I am aware, this can not be fixed.

Problem: There's no security!

Explanation: Communications are optionally encrypted, if the **sodium** package is available. The reason it is optional is that **libsodium** is actually a fairly weighty systems dependency. This is a big problem for managed machines like clusters and supercomputers. You must have **sodium** installed on both the client and server machine, and start the server with the option `secure=TRUE` to use this, however.

There is also a password system. Passwords are read in from the user/client (and optionally at server creation) with the **getPass** package [12]. This will read with masking (i.e., without printing the password as it is typed). See the **getPass** package vignette [8] for more details.

Passwords are always hashed, whether or not the **sodium** package is available, as the hashing is done with the **argon2** package [7]. This is a reasonably new algorithm, and is believed to be very secure.

All that said, I am not a security person, so it is entirely possible that I have messed something up. Just know that I'm trying my best, and that if you believe something to be in error, I'd really like to know about it.

Problem: Something else is wrong!

Explanation: Please be so kind as to file an issue describing the problem. Be as descriptive as possible.

References

[1] libsodium, 2015.

[2] Wei-Chen Chen and Drew Schmidt. *A Quick Guide for the pbdZMQ Package (Ver. 0.1-0)*, 2015. R Vignette, URL http://cran.r-project.org/package=pbdZMQ.

[3] Wei-Chen Chen and Drew Schmidt. pbdZMQ: Programming with Big Data – Interface to ZeroMQ, 2015. R Package, URL http://cran.r-project.org/package=pbdZMQ.

[4] P. Hintjens. The zeromq guide – for c developers, 2013.

[5] Jeroen Ooms. *sodium: A Modern and Easy-to-Use Crypto Library*, 2015. R package version 0.2.

[6] G. Ostrouchov, W.-C. Chen, D. Schmidt, and P. Patel. Programming with Big Data in R, 2012.

[7] Drew Schmidt. argon2: Secure password hashing, 2017. R package version 0.2-0.

[8] Drew Schmidt. *Guide to the getPass Package*, 2017. R Vignette.

[9] Drew Schmidt. *Using remoter with Remote Machines*, 2017. R Vignette.

[10] Drew Schmidt and Wei-Chen Chen. *pbdCS: 'pbdR' Client/Server Utilities*, 2015. R package version 0.1-0.

[11] Drew Schmidt and Wei-Chen Chen. *remoter: Remote R: Control a Remote R Session from a Local One*, 2015. R package version 0.1-1.

[12] Drew Schmidt and Wei-Chen Chen. getPass: Masked user input, 2017. R package version 0.2-1.