Version 0.4-0

Using remoter with Remote Machines

Taking R to the Cloud and Beyond

Drew Schmidt

Using remoter with Remote Machines

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This manual may be incorrect or out-of-date.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

This guide is for those who understand the basics of using **remoter**, and wish to learn how to interact with a remotely hosted server. For the basics, please first see the *Guide to the remoter Package* [4] guide.

Before we begin, a quick word about addresses and ports.

An address should not include any protocols, like `tcp://` or `http://`. The address should also not contain any ports (denoted by a `:`), as this value goes in a separate argument.

A port is a non-negative integer. The minimum value for a valid port is 1024 (values 1-1023 are privileged), and the maximum value is 65535. That said, you are strongly encouraged to use port values between 49152 and 65535. The documentation for **pbdZMQ** [3] discusses this in detail. Specifically, see `?pbdZMQ::random_port`.

Of course, all the usual issues apply. The server should be able to accept communications on the desired port. One way to handle this is by opening the desired port. Opening ports is very standard stuff, but dependent on the system you are using, so consult relevant documentation if you aren't sure what to do. Another way is by tunneling over ssh, which we discuss in a later section.

2 Creating a Remote Server

Before beginning, you need to spawn your server. This is something you only need to do once, and you need to do it in such a way that it is allowed to run persistently, even after you log off.

The easiest setup is if your server is available via ssh, and probably running headless (without a monitor/GUI desktop environment). This is what your typical linux cloud instance looks like. In this case, we suggest you use a tool like tmux or screen. This way, you can re-attach your server session and easily read the logs live, which is very useful for debugging. However, this is not strictly necessary.

If you are using something like tmux or screen, then your workflow would look something like:

1. ssh to your remote (you only need to do this once!)
2. Start a **tmux** or **screen** session
3. Start R and run `remoter::server()` (see `?server` for additional options). Or even better, run `Rscript -e remoter::server()` so the server dies if something goes wrong.
4. Detach your tmux/screen session and log out.

Alternatively, if you wish to avoid tmux/screen, you still need to ssh to your machine. Then you can run the R session in the background by a fork via something like:

```
Rscript -e "remoter::server()" &
```

and disconnect. If for some reason the server becomes unreachable via **remoter**, you will have to manually connect again via ssh to kill the rogue process with the `kill` command.

If you are running a full desktop environment, which is typical with Windows servers, you should be able to launch an R process (say RGui) and start the server there via `remoter::server()`. Admittedly, we have no experience with this configuration, so if you have experience here, we would love to hear from you.

3 Connecting to a Remote Server

Because **remoter** is just and R package to connect to the remote, you need only fire up your favorite R interface. This can be the terminal, RStudio, RGui (Windows), R.app (Mac), or even Emacs. Whatever your choice, connect as with a local server, but specifying the correct remote address (and possibly port):

```
remoter::client("my.remote.address")
```

So for example, say you have set up a server (as described above) on EC2 with address `"ec2-1-2-3-4.compute-1.amazonaws` listening on port `56789`. Then you would run:

```
remoter::client("ec2-1-2-3-4.compute-1.amazonaws.com", port=56789)
```

That's it! Everything else should work just as when you were running the server locally. The only hiccup is opening up that port. If that is not possible for you for whatever reason, then you may need to set up an ssh tunnel, which we describe in the following section.

4 Tunneling Over ssh

If you can't or don't want to open up a port on a remote system, you can always tunnel over ssh (assuming of course you actually have legitimate access to the machine...).

The **pbdRPC** package [1] has offers some basic tunneling functionality. At the time of writing, it is somewhat new and experimental. But the reader is encouraged to check the package's vignette [2] for more details.

Even without **pbdRPC**, creating a tunnel is not terribly difficult. Say you have user account `user` on remote with address `my.remote.machine`. Suppose your remote machine is running a **remoter** server, listening on port 55555. Then you can run:

```
ssh user@my.remote.machine -L 55556:localhost:55555 -N
```

To be totally unambiguous:

- server port (running on remote): 55555
- client port (running on your laptop): 55556

This will allow you to connect to the remote machine as follows:

```
remoter::client("localhost", port=55556)
```

You can also spawn the server in the ssh tunnel call. For example, you might run:

```
ssh user@my.remote.machine -L 55556:localhost:55555 'Rscript -e "remoter
    ::server(port=55555)"'
```

This will automatically launch a **remoter** server listening on port 55555, tunneled over `localhost:55566`. If you are working on a managed system, like a cluster or supercomputer, you might need to run something like `module load R` first:

```
ssh user@my.remote.machine -L 55556:localhost:55555 'module load R &&
    Rscript -e "remoter::server(port=55555)"'
```

5 Working with Relays

When an intermediary between the client/server is necessary, it is generally preferable to work with ssh tunnels whenever possible. However, some use cases may require or suggest a different strategy. To that end, as of **remoter** version 0.3-1, we now offer a different kind of spawnable instalce: "relays". These serve as "middlemen" between the client and server, and are particularly useful for resources like clusters and supercomputers where the login and compute nodes are separate. Internally, the relay is a server that does nothing but pass messages between the client and server. Figure 1 shows the basic conceptual idea about how relays work.

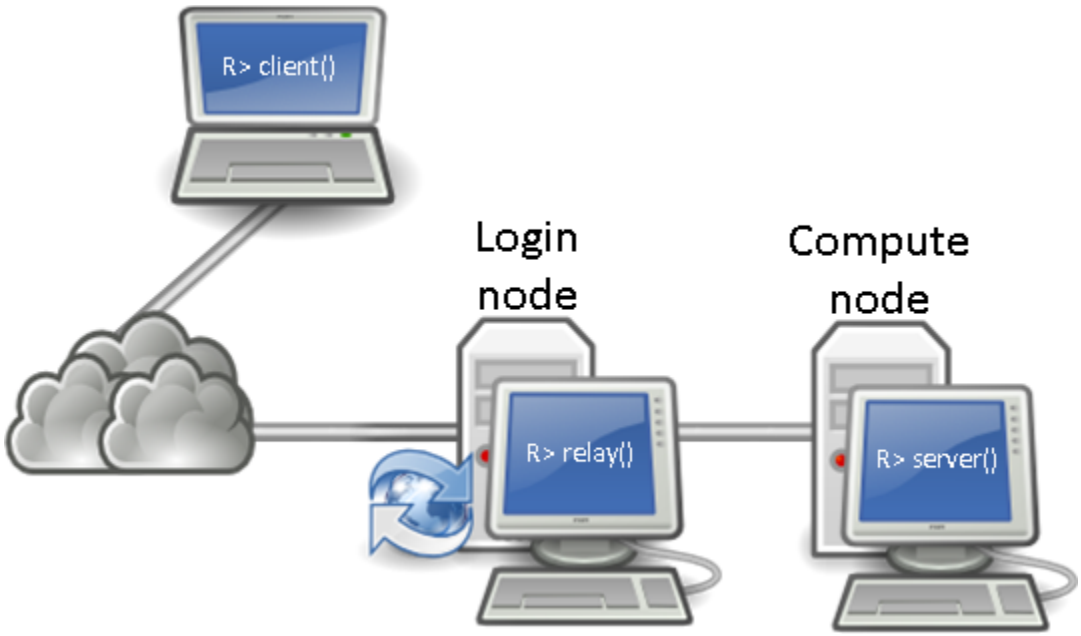

Figure 1: Remoter relays and their relationship to the client and server. The diagram is composed of icons from the OSA Icon Library

To spawn a relay, you can do:

```
remoter::relay(addr=my.server.addr, sendport=my.server.port)
```

As the name suggests in the above example, `my.server.addr` and `sendport` represent the address and port of the server (what you would use for `addr` in `remoter::client()` if you could connect directly). Then the client will connect to the relay, not the server (that's the whole point!) something like:

```
remoter::client(addr=my.relay.addr, port=my.relay.port)
```

Here `my.relay.addr` is the address of the relay, and `my.relay.port` should math the argument `recvport` used when creating the relay (default is `r as.integer(formals(remoter::relay)[["recvport"]])`).

References

[1] Wei-Chen Chen. pbdRPC: Programming with big data – remote procedure call, 2017. R Package, URL https://cran.r-project.org/package=pbdMPI.

[2] Wei-Chen Chen. *A Quick Guide for the pbdRPC Package*, 2017. R Vignette, URL https://cran.r-project.org/package=pbdRPC.

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