# A Quick Guide for the pbdRPC Package

Wei-Chen Chen  

pbdR Core Team  
Silver Spring, MD, USA

## Contents

1. Introduction  
   1.1. Basic ssh and srpc() ................................. 2  
   1.2. Basic ssh() ........................................ 2  
   1.3. Basic plink.exe and plink() ....................... 3  

2. Handling Login Information  

3. Handling Machine Information  

4. An Application with remoter  

5. An Application with pbdCS  

6. Local Port Forwarding  
   6.1. Arguments for Local Port Forwarding .................. 12  
   6.2. Arguments for Tunneling .......................... 12  

7. An Advance Application with pbdMPI  

8. FAQs  
   8.1. General ........................................... 15  

References  

© 2017 Wei-Chen Chen.  
Permission is granted to make and distribute verbatim copies of this vignette and its source provided the copyright notice and this permission notice are preserved on all copies.  
This publication was typeset using \LaTeX.
Disclaimer:
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.

Warning:
This document is written to explain the main functions of pbdRPC (Chen and Schmidt 2017), version 0.1-1. Every effort will be made to ensure that future versions are consistent with these instructions, but features in later versions may not be explained in this document.
Information about the functionality of this package, and any changes in future versions can be found on website: “Programming with Big Data in R” (pbdR) at http://r-pbd.org/ (Ostrouchov et al. 2012).

1. Introduction

This package, pbdRPC (Chen and Schmidt 2017), provides one high-level function, srpc(), that can securely send commands to remote servers via ssh (OpenSSH) or plink/plink.exe (PuTTY). The high-level function is a very light yet secure implementation because the communications are encrypted, by default SSH version 2 using RSA key pairs, between a local client and remote servers.

The high-level function srpc() is a unified interface via system call to either ssh or plink across most popular OSs including Linux, Mac OSX, MS Windows, and Solaris. Simply speaking, the function srpc() is a wrapper of two low-level functions, ssh() and plink(). However, these functions can ask remote servers to execute commands without logging in the servers provided that an authentication is setup properly.

Four RPC controls are provided by the package to simply the functions:

1. .pbd_env$RPC.CT is main RPC controls taking care several basic functionalities of three functions, srpc(), ssh(), and plink().

2. .pbd_env$RPC.LI has information of login account for logging in the remote server include authentication using private keys. See Section 2 for details.

3. .pbd_env$RPC.RR has examples of executing multiple commands on a remote server which is an application related to an R package, remoter (Schmidt and Chen 2016b). See Section 4 for details.

4. .pbd_env$RPC.CS has examples of executing multiple commands on a pbdCS cluster which is an application related to an R package, pbdCS (Schmidt and Chen 2016a). See Section 5 for details.

Note that .pbd_env will be first generated when the library pbdRPC is loaded, then default objects RPC.CT, RPC.LI, RPC.RR, and RPC.CS will be generated.

In general, only RPC.CT and RPC.LI are required to be set by users according to their environment configurations. The RPC.RR and RPC.CS are simple templates to show two pbdR applications. Users are welcomed to adopt those templates and to develop personal applications wherever automations can be benefit by remote procedure calls.
Most OSs (Linux, Solaris, Mac OSX) have the system command ssh (OpenSSH) installed, so the ssh() is a wrapper function to the system ssh command. For Windows, the plink.exe (from PuTTY) will be compiled with pbdRPC, so the plink() is a wrapper function to the executable file, plink.exe. Note that for non-Windows system, the plink can be compiled as well.

1.1. Basic ssh and srpc()
Suppose a sshd is set and running correctly on a server running a Linux system at an ip address “192.168.56.101” and a port “22”. Further, suppose an account called “snoweye” is created and a password for the account is set on the server. From a terminal or a shell of non-Windows systems, one may use

Basic ssh in shell

```
$ ssh snoweye@192.168.56.101 'whoami'
```

to access the server and to ask the server to execute a command whoami. Typing the password for the login account may be needed. The command whoami is available on most Linux systems, it should return the command result, “snoweye”, on the screen/stdout without logging in a shell environment on the server. In the same setup, the command whoami can be replaced by any other proper programs, shell scripts, or procedures. For Windows system, one may use plink.exe instead of ssh from a terminal cmd.exe. See Section 1.3 for details.

Within R, the example below will have the same results as the above shell command.

Basic srpc() in pbdRPC and R

```r
> library(pbdRPC, quietly = TRUE)
> ### Alter login information as needed
> rpcopt_set(user = "snoweye", hostname = "192.168.56.101")
> srpc("whoami")
```

The command results may be captured by R as well.
Regardless the system, the high level function srpc() can unify the calls to either ssh() or plink() functions. One may use ssh() in non-Windows system, but use plink() in Windows system. The srpc() automatically detects the system first, then calls the corresponding function. Currently, no external plink.exe or plink is implemented even though it is possible. The details of ssh() and plink() are given in examples below.

1.2. Basic ssh()
Inside R and via pbdRPC, this can be done by

Basic ssh() in pbdRPC and R

```r
> library(pbdRPC, quietly = TRUE)
> ssh("snoweye@192.168.56.101 'whoami'")
```

provided all other options (port, forwarding, etc) are set correctly. Note that the password for the account may be required when an authentication file (id_rda) is not available.
Note that multiple commands can be automatically given at once as shell commands under an shell prompt, such as ";", "&&", ";>", "<", "|" or "&" etc. For example, the code below will tell current id, date/time, and files.

```
Multiple commands to ssh in shell
$ ssh snoweye@192.168.56.101 'whoami;date;ls -a'
```

The multiple commands can be applied to `ssh()` and `plink()` as

```
Multipel commands to ssh() in pbdRPC and R
> library(pbdRPC, quietly = TRUE)
> ssh("snoweye@192.168.56.101 'whoami;date;ls -a'"
```

See Section 4 and `.pbd_env$RPC.RR` for more details.

### 1.3. Basic `plink.exe` and `plink()`

In Windows system and inside the `cmd.exe`, one may similarly use the code below

```
Basic plink.exe in cmd.exe
C:\> plink.exe snoweye@192.168.56.101 'whoami'
```

to access the server provided `plink.exe` is in the PATH.

Inside `RGui` and via `pbdRPC`, this can be done by

```
Basic plink() in pbdRPC and R
> library(pbdRPC, quietly = TRUE)
> plink("snoweye@192.168.56.101 'whoami'"
```

provided all other options (port, forwarding, etc) are set correctly. The multiple commands can be applied to `plink()` as well.

By default, the `plink()` will open an `cmd.exe` to execute the command `whoami` because the password input is not allowed inside `RGui`. When the authentication file (`id_rsa.ppk`) is available, one may want to disable the opening `cmd.exe` below.

```
Advance plink() in pbdRPC and R
> .pbd_env$RPC.CT$use.shell.exec <- FALSE
> ret <- plink("snoweye@192.168.56.101 'whoami'"
> print(ret)
```

Because the `shell.exec()` is disable, the `plink()` call may accept returns of the remote server and capture/save the returns in an `R` object, `ret`.

The `use.shell.exec` is for Windows system only and required to be `TRUE` when `RGui` is mainly used. The `plink()` in `RGui` may hang forever or crash when input/typing of a password or a passphrase is needed for logging in the server. `RGui` has different stdin and stdout than a usual terminal. The `use.shell.exec` can be switched to `FALSE` when the authentication is correct and no passphrase is needed, i.e. no stdin input/typing. However, `Rcmd` running within a `cmd.exe` may be OK with stdin input/typing when `use.shell.exec = FALSE`.

Other solutions to replace internal `plink.exe` of `pbdRPC` include:
• The `plink.exe` can be installed from the PuTTY as well.

• Windows PowerShel and git also provide `ssh.exe` but additional installation/configuration is unavoidable.

2. Handling Login Information

Suppose an Oracle VM VirtualBox runs Xubuntu 15.10 as the guest OS within a Windows 8 system as the host OS. The VM has a virtual network adapter (host-only) with IP address 192.168.56.101, so that one can login to the VM using either `telnet`, `plink`, or `ssh` from the Windows 8 system. Note that `telnet` and `ssh` uses ports 23 and 22 as default, respectively. Suppose further the login id is called “snoweye”, then one may use the function `rpcopt_set()` to assign/overwrite the login information to `.pbd_env$RPC.LI` below.

```
> ### Alter login information as needed
> rpcopt_set(user = "snoweye", hostname = "192.168.56.101", pport = 22)
```

The basic login information `RPC.LI` below describes that `srpc()` will

• use `ssh` (exec.type) to execute a command (given by `srpc()`, `ssh()`, or `plink()`)

• with `args` (additional arguments to `ssh` or `plink.exe`)

• and a `user` account (snoweye)

• login into a `hostname` (server ip = 192.168.56.101 or host name)

• from a `pport` (server port = 22), and

• may use authentication keys in `priv.key` or `priv.key.ppk`.

```
> .pbd_env$RPC.LI
$exec.type
[1] "ssh"

$args
[1] ""

$pport
[1] 22

$user
[1] "snoweye"

$hostname
[1] "192.168.56.101"
```
Currently, the `exec.type` is only for non-Windows systems, and it will be ignored on Windows systems ("plink" will be used). Also, `ssh` uses "-p" (lower case) to input the server port argument. `plink.exe` uses "-P" (upper case) to input the server port argument. Therefore, the `args` should not include "-p" nor "-P" to avoid confusion in the unified function `srpc()`. Similarly, the "-i" may not be include in the `args` as well because additional authentication may be required.

The account may have the private key for authentication to avoid typing the login password for the user account. The private keys may be stored in files indicated by `prive.key` for `ssh()` or `prive.key.ppk` for `plink()`. When all setups are correct, command calls can be executed at the hostname (192.168.56.101) remotely. By default, the `prive.key.ppk` will be read from the current working directory (from `getwd()`) in Windows systems. In this case ("./id_rsa.ppk"), the file C:/Users/login_account/Documents/id_rsa.ppk is probably read for authentication.

To generate private and public keys is pretty standard for most Linux systems via the `ssh-keygen` command which will generate keys in OpenSSH format. One may use `puttygen` in Linux to convert OpenSSH format to PuTTY format for Windows. See Section 8.1 for the conversion from `id_rsa` to `id_rsa.ppk`. For Windows systems, one may also use `puttygen.exe` to obtain both keys.

### 3. Handling Machine Information

In Section 2, we have seen a very tedious way to handle login information which also includes some information for a single machine. In this section, we introduce a better way to handle both login information and multiple machines. The function `machine()` will generate a constructor-like object containing all required information. It is as simple as the example below.

```r
> library(pbdRPC, quietly = TRUE)
>
### Multiple machine information as needed
> m1 <- machine(user = "snoweye", hostname = "192.168.56.101", pport = 22)
> m2 <- machine(user = "snoweye", hostname = "192.168.56.102", pport = 22)
> m3 <- machine(user = "snoweye", hostname = "192.168.56.103", pport = 22)
```

With the above objects `m1`, `m2`, and `m3`, the function `rpc()` can assess freely to three machines with simpler interface then the function `srpc()`. For example, one may quickly check the access of three machines as the example below.
Basic `rpc()` in `pbdRPC` and R

```r
> rpc(m1, "uname")
> rpc(m2, "uname")
> rpc(m3, "uname")
```

4. An Application with `remoter`

The `remoter` (Schmidt and Chen 2016b) and `pbdZMQ` (Chen et al. 2015) provide client/server interface to control a remote R (e.g. running on a single server, Xubuntu, ip=192.168.56.101) from a local R (e.g. running on a single laptop, Windows 8). Combining with `pbdMPI` (Chen et al. 2012) and `pbdCS` (Schmidt and Chen 2016a), one may extent the remote R to the R clusters by running R’s in a distributed/SPMD environment.

- See Schmidt et al. (2016) for an introduction of `remoter` and `pbdCS`.
- See http://github.com/snoweye/user2016.demo for a demo of both packages.
- See pbdR-Tech (http://snoweye.github.io/pbdr/) and HPSC (http://snoweye.github.io/hpsc/) websites for more applications of SPMD and how to utilize R in clusters (Chen and Ostrouchov 2012).

In a simplified scenario such as the setting in Section 2, one may use the following commands to “start”, “check”, and “kill” a remote R server under a shell environment provided Rscript is in PATH of the login server (pre-load or set by the `00_set_devel_R`).

```
remoter server at 192.168.56.101

$ source ~/work-my/00_set_devel_R
$ nohup Rscript -e 'remoter::server()' > .rrlog 2>&1 < /dev/null &
$ ps ax|grep '[r]emoter::server'
$ kill -9 $(ps ax|grep '[r]emoter::server'|awk '{ print $1}')
```

In an well established server, one can use `ssh` or `plink.exe` to send those commands from a local laptop. Furthermore, one may also use `pbdRPC` directly within an R environment to send those commands. The example is in the code below.

Using `pbdRPC` to control `remoter`

```r
> library(pbdRPC, quietly = TRUE)
> # Alter login information as needed
> # rpcopt_set(user = "snoweye", hostname = "192.168.56.101")
> m <- machine(user = "snoweye", hostname = "192.168.56.101")
> .pbd_env$RPC.RCT$use.shell.exec <- FALSE
> preload <- "source ~/work-my/00_set_devel_R; "
> start_rr(m, preload = preload)
> character(0)
> library(remoter)
Loading required package: pbdZMQ
```
Attaching package: 'remoter'

The following object is masked from 'package:grDevices':

 dev.off

The following objects are masked from 'package:utils':

 ?, help

> client(addr = "192.168.56.101")
WARNING: server not secure; communications are not encrypted.

remoter> 1+1
[1] 2
remoter> q()
>
> check_rr(m)
[1] " 2014 ? S1 0:00
/home/snoweye/work-my/local/R-devel/lib64/R/bin/exec/R --slave
--no-restore -e remoter::server()"
> kill_rr(m)
character(0)

where client() is for connect to the remote R server started by start_rr(). Note that all
commands in the above example were typed inside a local R in the local laptop. However,
the computation 1+1 was done by a remote R on the server (192.168.56.101).

The start_rr(), check_rr(), and kill_rr() are all wrapper functions of srpc()/rpc() to
submit different commands stored in .pbd_env$RPC.RR$start, .pbd_env$RPC.RR$check,
and .pbd_env$RPC.RR$kill, respectively. The tedious details of RPC.RR are in the code below
which all can be simply sent by srpc() to execute on the server.

RPC.RR for controlling remoter

```r
> .pbd_env$RPC.RR
$check
[1] "ps ax|grep 'remoter::server'"

$killed
[1] "kill -9 $(ps ax|grep 'remoter::server'|awk '{print $1}')"

$start
[1] "nohup Rscript -e 'remoter::server()' > .rrlog 2>&1 < /dev/null &"

$preload
[1] "source ~/work-my/00_set_devel_R; "
```

5. An Application with pbdCS
Similar to the remoter, the pbdCS (Schmidt and Chen 2016a) provides interactivity for clusters running R’s via the pbdMPI (Chen et al. 2012) in SPMD computing framework (Ostrouchov et al. 2012; Chen and Ostrouchov 2012). See Schmidt et al. (2016) for an introduction of remoter and pbdCS, and see https://github.com/snoweye/user2016.demo for a demo of both packages.

In a simplified scenario such as the setting in Section 2, several pbdCS R’s can run 4 instances on the server, Xubuntu, ip=192.168.56.101 as the example below.

### pbdCS cluster with 4 R instances

```
$ source ~/work-my/00_set_devel_R
$ nohup mpiexec -np 4 Rscript -e 'pbdCS::pbdserver()' > .cclog 2>&1 < /dev/null &
$ ps ax|grep '[p]bdCS::pbdserver'
$ kill -9 $(ps ax|grep '[p]bdCS::pbdserver'|awk '{ print $1}')
```

The example above is very similar to the one in Section 4, but further demonstrates how to “start”, “check”, and “kill” a pbdCS cluster with 4 R launched by/within the MPI program mpiexec.

In an well established server, one can use ssh or plink.exe to send those commands from the local laptop. Furthermore, one may also use pbdRPC directly within an R environment to send those commands. The code below shows the example.

#### Using pbdRPC to control pbdCS

```
> library(pbdRPC, quietly = TRUE)
> ### Alter login information as needed
> # rpcopt_set(user = "snoweye", hostname = "192.168.56.101")
> m <- machine(user = "snoweye", hostname = "192.168.56.101")
> .pbd_env$RPC.CT$use.shell.exec <- FALSE
> preload <- "source ~/work-my/00_set_devel_R; "
> start_cs(m, preload = preload)
> character(0)
> library(pbdCS)
> pbdCS::pbdclient(addr = "192.168.56.101")

pbdR> library(pbdMPI)
> allreduce(1)
[1] 4
pbdR> q()
> check_cs(m)
[1] "12578 ? S1 0:00 mpiexec -np 4 Rscript -e pbdCS::pbdserver()"
[2] "12580 ? S1 0:00 /home/snoweye/work-my/local/R-devel/lib64/R/bin/exec/R --slave --no-restore -e pbdCS::pbdserver()"
[3] "12581 ? S1 0:00 /home/snoweye/work-my/local/R-devel/lib64/R/bin/exec/R --slave --no-restore -e pbdCS::pbdserver()"
```
where `pbdclient()` is for connect to the `pbdCS` cluster started by `start_cs()`.

The `start_cs()`, `check_cs()`, and `kill_cs()` are all wrapper functions of `srpc()` to submit different commands stored in `.pbd_env$RPC.CS$start`, `.pbd_env$RPC.CS$check`, and `.pbd_env$RPC.CS$kill`, respectively. The details of RPC.CS are in the example below.

RPC.CS for controlling `pbdCS`

```r
> .pbd_env$RPC.CS
$check
[1] "ps ax|grep '([p]bdCS::pbdserver')"

$kill
[1] "kill -9 $(ps ax|grep '([p]bdCS::pbdserver')|awk '{print $1}')"

$start
[1] "nohup mpiexec -np 4 Rscript -e 'pbdCS::pbdserver()' > .cslog
  2>&1 < /dev/null &"

$preload
[1] "source ~/work-my/00_set_devel_R; "
```

6. Local Port Forwarding

**Warning:**
System security issues may arise when the materials of this section are implemented in open/public domains. Consulting with network security experts may be required.

The `remoter` command `client()` has a default setting to connect to the `remoter` server using `addr = "localhost"` and `port = 55555` which assumes the `remoter` server and client are both working at `localhost`. This may only be possible for convenience of development and debugging only. In general, the server can be anywhere and more powerful than a laptop. Again, we may consider the environment setup in Sections 4 and 5 to demonstrate local port forwarding, even though the setup is over simplified it is quite common for most general users. The server is running at `192.168.56.101:55555`, so the argument `addr = "192.168.56.101"` in the `remoter` command `client(addr = "192.168.56.101")` from the `localhost` is necessary.

Note that this above case may not be a good reason to show local port forwarding. However, it can avoid typing address or to be independent to the `addr`. One may consider to forward the `localhost` port 55555 to the server directly.
The following code serves the purpose of local port forwarding in `pbdRPC` using `srpc()`, then start a `remoter` server and launch a connection via `client()` without changing arguments.

```
Forward localhost:55555 to 192.168.56.101:55555

> library(pbdRPC, quietly = TRUE)
> ### Alter login information as needed
> rpcopt_set(user = "snoweye", hostname = "192.168.56.101")
> # .pbd_env$RPC.CT$use.shell.exec <- FALSE
> # srpc(args = "-N -T -L 55555:192.168.56.101:55555", wait = FALSE)
> tunnel(rhostname = "192.168.56.101")
cmd: Tasklist /FI "IMAGENAME eq plink.exe"
```

<table>
<thead>
<tr>
<th>Image Name</th>
<th>PID</th>
<th>Session Name</th>
<th>Session#</th>
</tr>
</thead>
<tbody>
<tr>
<td>plink.exe</td>
<td>9052</td>
<td>Console</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>64</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| cmd: Tasklist /FI "IMAGENAME eq cmd.exe"

```
```

Image Name PID Session Name Session# Mem Usage
------------------------- ------ ------------ -----
plink.exe 9052 Console 3 64 K

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```
```

<table>
<thead>
<tr>
<th>Image Name</th>
<th>PID</th>
<th>Session Name</th>
<th>Session#</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmd.exe</td>
<td>5060</td>
<td>Console</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2,624</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```
```

<table>
<thead>
<tr>
<th>Image Name</th>
<th>PID</th>
<th>Session Name</th>
<th>Session#</th>
</tr>
</thead>
<tbody>
<tr>
<td>plink.exe</td>
<td>9052</td>
<td>Console</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>4,316</td>
<td></td>
<td></td>
</tr>
<tr>
<td>plink.exe</td>
<td>3972</td>
<td>Console</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>3,992</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| cmd: Tasklist /FI "IMAGENAME eq cmd.exe"

```

<table>
<thead>
<tr>
<th>Image Name</th>
<th>PID</th>
<th>Session Name</th>
<th>Session#</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmd.exe</td>
<td>5060</td>
<td>Console</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2,624</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cmd.exe</td>
<td>4644</td>
<td>Console</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2,460</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```

```cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
```
Attaching package: "pbdZMQ"

The following object is masked from "package:base":

  ls

Attaching package: "remoter"

The following object is masked from "package:grDevices":

  dev.off

The following objects are masked from "package:utils":

  ?, help

> client()  # equivalent to client(addr = "192.168.56.101")
WARNING: server not secure; communications are not encrypted.

remoter> 1+1
[1] 2
remoter> shutdown()
>
> ### Check/kill the alive background processes for local port forwarding.
>
> > rpc_ps()
cmd: Tasklist /FI "IMAGENAME eq plink.exe"

<table>
<thead>
<tr>
<th>Image Name</th>
<th>PID</th>
<th>Session Name</th>
<th>Session#</th>
</tr>
</thead>
<tbody>
<tr>
<td>plink.exe</td>
<td>9052</td>
<td>Console</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>4,348K</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

cmd: Tasklist /FI "IMAGENAME eq cmd.exe"

<table>
<thead>
<tr>
<th>Image Name</th>
<th>PID</th>
<th>Session Name</th>
<th>Session#</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmd.exe</td>
<td>5060</td>
<td>Console</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2,624K</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

> > rpc_kill(c(9052, 5060))
>
> > rpc_ps()
cmd: Tasklist /FI "IMAGENAME eq plink.exe"
INFO: No tasks are running which match the specified criteria.
cmd: Tasklist /FI "IMAGENAME eq cmd.exe"
INFO: No tasks are running which match the specified criteria.
First, `srpc(args = "-N -T -L 55555:192.168.56.101:55555")` forwards the connection between 55555 of the local host and 192.168.56.101:55555, the remote host and port. This is also equivalent to the function `tunnel`. Note that this call (local process) is running in background and is not disconnected even after quitting R, because `intern = FALSE` (default) and `wait = FALSE` are set to `srpc()` and passed down to its callee (in a shell). The possible process id (pid) for the forwarding should be printed in R via the function `rpc_ps()`. For example, the code above shows that 9052 is running for `plink.exe` which is also a child process of 5060 running for `cmd.exe`. The additional command “kill -p [pid]” may be needed to manually kill the local process (pid) when the forwarding is not needed anymore. Or, use the function `rpc_kill(pid)` in R. See Section 6.1 or ssh’s man page for details of arguments `-N` and `-L (inside args)` to the ssh or plink.exe.

Second, `client()` tries to connect with `localhost:55555` by default because it is from the laptop. The connection is then redirected to `192.168.56.101:5555` as well because the local port is being forwarded.

### 6.1. Arguments for Local Port Forwarding

Note that the ssh and plink.exe has similar functionalities for local port forwarding.

The argument `-L in ssh or plink.exe is a typical option for local port forwarding. The usage from the man page of the ssh says

```
-L [bind_address:]port:host:hostport
   Specifies that the given port on the local (client) host is
to be forwarded to the given host and port on the remote side.
... skipped ...
```

Because the call of local port forwarding needs to be either alive or active during the access of other applications to the `[bind_address:]port`, two other useful arguments are `-N and -T` that can combine and use with local port forwarding. The usages of both arguments from the man page say

```
-N Do not execute a remote command. This is useful for just
    for warding ports (protocol version 2 only).
-T Disable pseudo-terminal allocation.
```

i.e. batch and background modes are preferable.

### 6.2. Arguments for Tunneling

Theoretically, this is possible to be used in `srpc()`. However, there is no appropriate example yet.

Note that the ssh and plink.exe has similar functionalities for local port forwarding.

The argument `-R in ssh or plink.exe is a typical option for tunneling. The usage from the man page of the ssh says
From ssh man page

```
-R [bind_address:]port:host:hostport
    Specifies that the given port on the remote (server) host is to be forwarded to the given host and port on the local side.
```

7. An Advance Application with pbdMPI

Examples of the pbdMPI (Chen et al. 2012) are introduced in this section first under a shell/terminal mode. Then, the examples will be combined with pbdRPC to show how srpc() sends requests from an interactive R session to a remote server and execute the examples in the shell/terminal model. Multiple commands can be manually combined in one srpc() call. The return values can also be captured by the interactive R session with an addition argument.

The pbdMPI is a general MPI interface running in SPMD by default. A typical example is to run the pbdMPI via mpiexec and Rscript from a shell/terminal as in below with outputs.

```
pbdMPI::allreduce(1) in 4 cores
```

```
$ mpiexec -np 4 Rscript -e
    'library(pbdMPI, quietly=T); allreduce(1); finalize()'
[1] 4
[1] 4
[1] 4
[1] 4
```

The outputs are printed from 4 cores. Each core has a allreduce(1) call synchronically to reduce three 1's from other peer cores. The default operation for allreduce() is a summation sum(), so the total is 4. Note that the 4 cores may not be in a single machine as long as MPI setup correctly.

The example below will give the total 8 in each core because the value to be reduced is a 2 from each core.

```
pbdMPI::allreduce(2) in 4 cores
```

```
$ mpiexec -np 4 Rscript -e
    'library(pbdMPI, quietly=T); allreduce(2); finalize()'
[1] 8
[1] 8
[1] 8
[1] 8
```

Similarly, the total will be 12 in each core when 3’s are reduced from 4 cores.

With the examples above, one may want to execute them from a local machine/laptop. i.e. pbdMPI is run on remote servers while pbdRPC is run in local. Note that two systems between server and local machines are generally different.

Assume environment setups are similar as Sections 4 and 5. The pbdRPC commands within an interactive R session are shown below.
In order to capture the return values (four character strings) from the remote server, the argument `intern = TRUE` set to the `srpc()` is required as shown in the example.

When `verbose = TRUE`, the `srpc()` shows the command being passed to the shell or terminal.

In this case, one may test the message shown above to obtain the same result from the interactive R session as below.

```
C:\Users\snoweye> cd Documents
C:\Users\snoweye>Documents>
C:\Users\snoweye\Documents\R\win-library/3.4\pbdRPC\libs\x64\plink.exe -P 22 -i ./id_rsa.ppk snoweye@192.168.56.101 "source ~/work-my/00_set_devel_R; mpiexec -np 4 Rscript -e 'library(pbdMPI, quietly=T); allreduce(3); finalize()'"
```

```
> library(pbdRPC)
> #### Alter login information as needed
> rpcopt_set(user = "snoweye", hostname = "192.168.56.101")
> .pbd_env$RPC.CT$use.shell.exec <- FALSE
> #### Set the RPC commands
> preload <- "source ~/work-my/00_set_devel_R; "
> cmd.mpi <- "mpiexec -np 4 Rscript -e "
> cmd.code <- "library(pbdMPI, quietly=T); allreduce(3); finalize()"
> #### Put the RPC commands together
> cmd <- paste(preload, cmd.mpi, cmd.code, sep = "")
> cmd #### Similar to the shell example above
> #### Send the command to remote server, snoweye@192.168.56.101
> srpc(cmd = cmd)
> [1] [1] 12
> [1] 12
> [1] 12
> [1] 12
> #### Turn on verbose
> .pbd_env$RPC.CT$verbose <- TRUE
> #### Capture the return values
> ret <- srpc(cmd = cmd, intern = TRUE)
> C:\Users\snoweye\Documents\R\win-library/3.4\pbdRPC\libs\x64\plink.exe -P 22 -i ./id_rsa.ppk snoweye@192.168.56.101 "source ~/work-my/00_set_devel_R; mpiexec -np 4 Rscript -e 'library(pbdMPI, quietly=T); allreduce(3); finalize()'"
> str(ret)
> ret
```

From a `cmd.exe` command prompt windows

```
C:\Users\snoweye> cd Documents
C:\Users\snoweye>Documents>
C:\Users\snoweye\Documents\R\win-library/3.4\pbdRPC\libs\x64\plink.exe -P 22 -i ./id_rsa.ppk snoweye@192.168.56.101 "source ~/work-my/00_set_devel_R; mpiexec -np 4 Rscript -e 'library(pbdMPI, quietly=T); allreduce(3); finalize()'"
```

```
[1] 12
```
Note that the slash symbols / may be replaced by anti-slash or back slash symbols \ in MS Windows when Rtools is not loaded correctly. Double quote ” may be needed as well when the command path to the plink.exe contains any spaces or any non-ascii characters.

8. FAQs

8.1. General

1. **Q:** Does pbdRPC support Windows system?  
   **A:** Yes, the plink.exe from PuTTY will be the program to send commands to remote servers. An internal built plink.exe will be provided and wrapped by the pbdRPC command plink().

2. **Q:** Is an authentication used in pbdRPC? How does it work?  
   **A:** Yes, the authentication is the same way to ssh and plink.exe provided public and private keys are setup correctly. For example, when an RSA key is used, the ssh will by default search ~/.ssh/id_rsa or via the option “-i ./id_rsa” for a local private key. Similarly, the plink.exe uses the option “-i ./id_rsa.ppk” for a local private key. Inside pbdRPC, one can use the options of the control .pbd_env$RPC.LI$priv.key and .pbd_env$RPC.LI$pri.key.ppk to indicate the file of the private key. Then, ssh(), plink(), and srpc() commands will automatically access those files, accordingly.

3. **Q:** Can a ssh private key be converted to plink’s private key? i.e. convert OpenSSH format to PuTTY format.  
   **A:** Yes, the puttygen on linux can convert the id_rsa (OpenSSH format) to id_rsa.ppk (PuTTY format) as the commands below.

   ```shell
   $ sudo apt-get install putty
   $ puttygen id_rsa -O private -o id_rsa.ppk
   ```

4. **Q:** Is it possible to capture the returns from the RPC calls by srpc(), ssh(), or plink()? How?  
   **A:** Yes, set the arguments intern = TRUE and wait = TRUE to the RPC calls can obtain the outputs as used by system(). The plink() used in RGui may not be able to capture the outputs unless the authentication is set because shell.exec() is used instead of system().

5. **Q:** Does srpc() support SSH (reverse/remote) tunneling or port forwarding?  
   **A:** Yes, theoretically there is no problem. However, there is no appropriate example to show that in R. See Sections 6.1 and 6.2 for the command arguments.
6. **Q:** Error messages from an R session or a shell/terminal are shown as the error below.

---

**Error Message**

*FATAL ERROR: Network error: Connection refused*

or

**Error Message**

*ssh: connect to host 192.168.56.101 port 22: Connection refused*

---

**A:** The messages are mainly because of incorrect setups of SSH service. Check SSH service on remote servers.

- Install SSH server, such as the command below.

  **Shell Command**

  `$ sudo apt-get install openssh-server`

- Check network, port, SSH service, and firewall configurations.
- Turn on connection permissions for SSH ports.
- Make sure password or authentication are correct.
References


