A Quick Guide for the pbdZMQ Package

Wei-Chen Chen[1] and Drew Schmidt[2]

[1]pbdR Core Team

[2]Business Analytics and Statistics,
University of Tennessee,
Knoxville, TN, USA

Contents

This publication was typeset using LaTeX.

Acknowledgement

Chen was supported in part by the project "Bayesian Assessment of Safety Profiles for Pregnant Women From Animal Study to Human Clinical Trial" funded by U.S. Food and Drug Administration, Office of Women's Health. The project was supported in part by an appointment to the Research Participation Program at the Center For Biologics Evaluation and Research administered by the Oak Ridge Institute for Science and Education through an interagency agreement between the U.S. Department of Energy and the U.S. Food and Drug Administration.

Schmidt was 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.

Disclaimer

The findings and conclusions in this article have not been formally disseminated by the U.S. Department of Health & Human Services, U.S. Food and Drug Administration, nor Oak Ridge Institute for Science and Education. They should not be construed to represent any determination or policy of University, Agency, Administration and National Laboratory.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Warning: This document is written to explain the main functions of **pbdZMQ** (Chen *et al.* 2015), version 0.1-0. Every effort will be made to ensure 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" at `http://r-pbd.org/` (Ostrouchov *et al.* 2012).

1. Introduction

ZeroMQ (ØMQ) (Hintjens 2013b)[1] is a library for high-performance asynchronous messaging in scalable distributed applications. It provides APIs in several messaging patterns that, enabling developers a standardized way to form connections between different devices, including laptop computers, mobile devices, servers, clusters, and supercomputers. The APIs also simplify the complex calls to sockets and reduce the burden for developers of handling low-level network communications. Several popular programming languages provide bindings to these APIs.

In **pbdZMQ**, those ZeroMQ APIs are carefully wrapped in R via lower level C code and offers a few ZeroMQ patterns, including

- request-reply, in particular, one client and a server, and
- push-pull, in particular, one client and a set of servers.

These patterns are useful communication frameworks utilized in the **pbdCS** (Schmidt and Chen 2015) that combines two different messaging libraries, namely ZeroMQ and MPI, and utilizes their respective advantages in:

- user-to-server communication via **pbdZMQ**, and
- server-to-server computations for statistical programming via **pbdMPI** (Chen *et al.* 2012).

2. Installation

2.1. Installing

The **pbdZMQ** package requires an installation of the ZeroMQ library. So before we may discuss particulars of installing the R package, we take a moment here to describe the various ways in which you may install ZeroMQ itself. For convenience, we distribute with the package a distribution of ZeroMQ, although if you have access to a system installation, that may be preferable. We separate installation of ZeroMQ into 3 cases:

1. system package manager, such as the `libzmq` and `libzmq-dev` packages in Debian-derived systems,

[1] Available at `http://www.zeromq.org/`

2. **pbdZMQ**'s internal ZeroMQ library (4.1.0 rc1), or

3. an external ZeroMQ library (4.1.0 rc1 or later).

With System Package Manager This method is perhaps the easiest when a package managing system is available, such as on Linux, and returns the locations of the `libzmq` include and library paths via:

- `pkg-config -variable=includedir libzmq` and
- `pkg-config -variable=libsdir libzmq`.

In this setup, installation is very straightforward. From a shell, you can execute:

Shell Command

```
R CMD INSTALL pbdZMQ_0.1-0.tar.gz
```

Using Internal ZeroMQ This method uses the ZeroMQ library bundled with **pbdZMQ**, and should be fairly simple. This method has been successfully tested under Linux, Mac OSX, Windows, and FreeBSD. Solaris has been tested with no success.

Installation in this way can be simply done by adding the configure argument `-enable-internal-zmq`. In practice, this might look something like:

Shell Command

```
R CMD INSTALL pbdZMQ_0.1-0.tar.gz \
  --configure-args="--enable-internal-zmq"
```

Using External ZeroMQ This method assumes you have built your own ZeroMQ library somewhere, or perhaps one is offered to you by your system administrator. In any event, this method is only tested under Linux systems. As with the previous method, we were unsuccessful in our attempts to build on Solaris.

To build ZeroMQ yourself, you might do something like the following:

Shell Command

```
./configure \
   --prefix=/usr/local/zmq \
   --enable-shared=yes \
   --with-poller=select \
   --without-documentation \
   --without-libsodium
make -j 4
make install
```

which will install the library to `/usr/local/zmq/` where `/usr/local/zmq/include/` will have the header file `zmq.h` and `/usr/local/zmq/lib/` will have the shared library file `libzmq.so`.

With an external ZeroMQ available, we can install **pbdZMQ** via:

Shell Command

```
R CMD INSTALL pbdZMQ_0.1-0.tar.gz \
  --configure-vars="ZMQ_INCLUDE='-I/usr/local/zmq/include' \
                    ZMQ_LDFLAGS='-L/usr/local/zmq/lib -lzmq'"
```

2.2. Testing the Installation

To make sure that **pbdZMQ** is installed correctly, one may run a simple "hello world" test from *one* terminal to test the library as follows:

Shell Command

```
Rscript -e "demo(hwserver,'pbdZMQ',ask=F,echo=F)" &
Rscript -e "demo(hwclient,'pbdZMQ',ask=F,echo=F)"
```

This will run 5 iterations of sending and receiving 'Hello World' messages between two instances (simple server and client).

2.3. Polling System

Note that one may want to use different polling system provided by the ZeroMQ library. By default, the `select` method is used in **pbdZMQ** for Linux, Windows, and Mac OSX. However, users may want to use `autodetect` or try others for better polling. Currently, the options as given by ZeroMQ may be `kqueue`, `epoll`, `devpoll`, `poll`, or `select` depending on libraries and system. You may set the polling method at compile time via:

Shell Command

```
R CMD INSTALL pbdZMQ_0.1-0.tar.gz \
  --configure-vars="ZMQ_POLLER='autodetect'"
```

See the ZeroMQ manual for more details.

3. Examples

The package provides several simple examples based on *the ZeroMQ guide for C developers* by Pieter Hintjens (Hintjens 2013a). These are located in the `demo/` subdirectory of the **pbdZMQ** package source, and they include:

Examples	Descriptions
`hwclient.r`	hello world client
`hwserver.r`	hello world server
`tasksink.r`	task sink from two workers
`taskvent.r`	task ventilator send jobs to two workers
`taskwork.r`	task workers
`wuclient.r`	weather updating client
`wuserver.r`	weather updating server
`mspoller.r`	using poller to receive messages from multiple sockets
`msreader.r`	receive messages from multiple sockets
`mpclient.r`	hello world client in mpltiple part version
`mpserver.r`	hello world server in mpltiple part version

For instance, the task examples can be run by

Shell Command

```
Rscript taskwork.r &
Rscript taskvent.r
Rscript tasksink.r

### Remember to kill two worker processors at the end, such as
ps -x|grep "file=task.*\.r"|sed "s/\(.*\) pts.*/\1/"|xargs kill -9
```

Or, via `demo()` function as the hello world example in Section 2.

The weather updating examples can be run by

Shell Command

```
Rscript wuserver.r &
Rscript wuclient.r
rm weather.ipc
```

Or, via `demo()` function as the hello world example in Section 2.

4. Backwards Compatibility with rzmq

This package currently has a few wrapper functions to offer the same API as that of the **rzmq** package (Armstrong 2014). The intent is to offer backwards compatibility as much as possible, but possibly with a reduced functionality set. Users are encouraged to use native `zmq.*()` functions provided by **pbdZMQ**.

The wrapper functions are:

Functions
`send.socket()`
`receive.socket()`
`init.context()`
`init.socket()`
`bind.socket()`
`connect.socket()`

5. A Basic Client/Server

In this section, we will develop a more complicated and realistic example using **pbdZMQ**. The example will show the construction of a basic client and server. To do so, we will (eventually) use the Request/Reply pattern, where a message is passed from client to server, executed on the server, and then the result is passed back to the client as a message.

All server code is meant to be executed in batch; though it can be used from an interactive R session, we feel this somewhat misses the point. To do this, save the server code as, say, `server.r` and start the server by running

Shell Command

```
Rscript server.r
```

from a terminal. One could use `R CMD BATCH` in place of `Rscript`, though by default it will suppress some messages on the server that we will want to see. Finally, the client should be run inside of an interactive R session. This can be from RStudio, the Windows/Mac R guis, or by running the command `R` at the terminal — the way you are used to using R.

Throughout our examples here, we will be using the **rzmq**-like bindings available in **pbdZMQ**. The other interfaces are easy enough to figure out once you understand one of them; and the **rzmq** ones are arguably the most R-like of the three. The source code for the final example can be found in the `inst/examples/reqrep/` subtree of the **pbdZMQ** source, or under `examples/reqrep/` of the binary installation. The same directory contains versions of this example using the other **pbdZMQ** interfaces.

5.1. Our First Client/Server

The Server Our first server will be very humble. It will receive one command from the client, print that command, and send back a success message before terminating — nothing more. Save the following in a file, say `server.r`, and execute it by running `Rscript server.r` from a terminal:

Server

```
library(pbdZMQ)
ctxt <- init.context()
socket <- init.socket(ctxt, "ZMQ_REP")
bind.socket(socket, "tcp://*:55555")
```

```
cat("Client command: ")
msg <- receive.socket(socket)
cat(msg, "\n")

send.socket(socket, "Message received!")
```

Unfortunately, the first 4 lines are just boilerplate; see the package manual for an explanation. The good news is that this is about as complicated as it gets on the ZeroMQ side; everything beyond this is just R programming.

The Client From and interactive R session (*not* in batch!), enter the following:

Client

```
library(pbdZMQ)
ctxt <- init.context()
socket <- init.socket(ctxt, "ZMQ_REQ")
connect.socket(socket, "tcp://localhost:55555")

send.socket(socket, "1+1")
receive.socket(socket)
```

If all goes well, your message (namely, `"1+1"`) should be sent from the client to the server, and the response `"Message received!"` should be sent from server to client. Afterwards, the server will terminate and you are free to exit your interactive R session (i.e., the client).

This example is deliberately as basic as can be, and lacks 2 crucial features: server persistence, and remote execution of commands. We will develop examples with these features in the remainder of this section.

5.2. A Persistent Server

The Server Next, we make the server *persistent*, in the sense that it will not immediately die after receiving its first command. This is trivial, as all we need to do is encapsulate the receive/send piece inside a `while` loop. As before, save the following to a file and execute in batch via `Rscript`:

Server

```
library(pbdZMQ)
ctxt <- init.context()
socket <- init.socket(ctxt, "ZMQ_REP")
bind.socket(socket, "tcp://*:55555")


while(TRUE)
{
  cat("Client command: ")
  msg <- receive.socket(socket)
  cat(msg, "\n")
```

```
  send.socket(socket, "Message received!")
}
```

The `receive.socket()` command does not use *busy waiting*. You can verify this by starting up the server and then looking at a process monitor for your operating system; you should see no elevated activity.

The Client Set up the client as above (everything but the `send.socket()` line is necessary) in an interaction R session. Now that we have a persistent server, we can make a shorthand function that encapsulates sending a message (from client to server) and receiving a response (from server to client):

Client Send/Receive

```
sendrecv <- function(socket, data)
{
  send.socket(socket, data)
  receive.socket(socket)
}
```

This assumes that the various optional arguments in `send.socket()` and `receive.socket()` are acceptable; and for the purposes of this demonstration they are. But the reader is encouraged to consult the **pbdZMQ** manual for more details about these two functions.

Now, with the convenience function, we can simply execute:

Client Usage

```
sendrecv(socket, "1+1")
sendrecv(socket, "rnorm(10)")
```

or any other valid R command.

5.3. More Than Messaging

The final piece is to actually execute commands that are sent to the server, and to pass the result back to the client. This is very easy, and only requires a slight modification to the server code. Modify the server piece above to do the following just after receiving (and printing) the client's message:

Server Modification

```
result <- eval(parse(text=msg))

send.socket(socket, result)
```

Of course, you will also need to remove the original `send.socket()` line with the one here. A nasty source of bugs in client/server programming is sending when you should receive or vice-versa, leading to deadlocks.

One additional thing the observant reader may have already realized is that our client/server framework leaves the server running perpetually, with no reasonable way for the client to terminate it. This just requires basic filtering of incoming messages (from the client, on the server). So for example, we might want the message "EXIT" to terminate the server. Modifying the server to handle this is trivial, and we present the full server below:

Full Server

```
library(pbdZMQ)
ctxt <- init.context()
socket <- init.socket(ctxt, "ZMQ_REP")
bind.socket(socket, "tcp://*:55555")


while(TRUE)
{
  cat("Client command:  ")
  msg <- receive.socket(socket)
  cat(msg, "\n")

  if (msg == "EXIT")
    break

  result <- eval(parse(text=msg))

  send.socket(socket, result)
}

send.socket(socket, "shutting down!")
```

Notice that we essentially added just a few lines. The first and more obvious is the check on `msg` for the magic word "EXIT". The addition of the final `send.socket()` line at the end, which returns the string "shutting down!" to the client is necessary to prevent the client from hanging after the server shuts down. Recall, the client expects a response from the server for every message the client sends!

5.4. Other Issues

The above examples are all very basic, but should illustrate how one could proceed to a more complex client/server design using ZeroMQ from R. Of course, there are a host of issues that we have not gone into here that are very important. For example, we perform no scrubbing of inputs that are to be executed on the server. This could be more or less important depending on the application.

Another important issue we have not addressed is error and warning handling. The reader is encouraged to return to the example in Subsection 5.3 and try executing something like `sendrecv(socket, "object_does_not_exist")` or `sendrecv(socket, "warning('uh oh')")` to see what happens.

Additionally, each time the client wanted to send a message, the user had to manually pass it as an argument to the function `sendrecv()`. It is possible — though complicated — to create a custom REPL which will automatically handle the client/server send-evaluate-receive pattern as though the user were at a standard R terminal.

Finally, we have not addressed the important issue of logging user commands sent to the server. Although anyone comfortable with R should see the path forward.

For a more detailed example illustrating these points, see the **pbdCS** package.

6. Solaris 10

Solaris 10 with external ZeroMQ 4.0.7 library and GNU tools (OpenCSW) has be tested for **pbdZMQ** successfully. The `INSTALL.solaris` has the steps which basically are

1. install GNU tools from OpenCSW using `pkgutil`,
2. set path to GNU tools,
3. install external ZeroMQ 4.0.7 library using GNU tools, and
4. install **pbdZMQ** linked with external ZeroMQ.

The steps are tested without too much time effort for further changes and can quickly provide a compiled library and install the **pbdZMQ**. I write down some considerations about ZeroMQ library here as a note in next.

1. Current, the ZeroMQ library on OpenCSW is 2.2.0 which is too old to be compiled with **pbdZMQ**.
2. Newer ZeroMQ ($>$ 4.0.7) can not be compiled with OpenCSW, there are more bugs to fix for Solaris according github's pull request.
3. Using Sun CC is even more complicated because there are more compile flags needed to take care first.

6.1. Have a Solaris 10?

It is easy and has a lot of fun.

1. Install an Oracle VM VirtualBox which is available at `https://www.virtualbox.org/`.
2. Download a Solaris template from `http://www.oracle.com/technetwork/server-storage/vm/solaris-098101.html`.
3. Import the template to the VM.
4. Boot and setup root password, internet, share drives, etc.
5. Done.

The next to do is having compilers by installing OpenCSW first, then install libraries and software either from OpenCSW website or compile them manually. One can have most GNU tools and R installed from OpenCSW via `pkgutil` command, and compile ZeroMQ library externally using those tools, then install **pbdZMQ** by linking with the compiled ZeroMQ library. The details are given in the following Sections.

6.2. OpenCSW

The OpenCSW (www.opencsw.org) provides binary packages of GNU tools for Solaris 10. Ideally, gmake, gcc, g++, autoconf, automake, gtar are required which all available from the website. For example, the gcc is in "gcc4core" package which can be installed with a few `pkgutil` commands by

R Script

```
> pkgadd -d http://get.opencsw.org/now
> /opt/csw/bin/pkgutil -U
> /opt/csw/bin/pkgutil -y -i gcc4core
> /usr/sbin/pkgchk -L CSWgcc4core # list files
```

Typically, the binary packages are installed under `/opt/csw/`.

Make sure those GNU tools are also in the path and export it to the environment. For example, I have

R Script

```
PATH=/opt/csw/gnu:/opt/csw/bin:/opt/csw/i386-pc-solaris2.10/bin:
/usr/sbin:/usr/bin:/usr/openwin/bin:/usr/ucb
export PATH
```

in a file name `set_path.sh`. Then, I set it to the environment by

R Script

```
> . ./set_path.sh
```

I also have a symbolic link "make" to "gmake" to avoid configuration problems. If not, use `ln -s` to make one.

Further, the R may not be easy to compile in Solaris 10, but the binary package of R is also available in OpenCSW. It can be installed by

R Script

```
> pkgadd -d http://get.opencsw.org/now
> /opt/csw/bin/pkgutil -U
> /opt/csw/bin/pkgutil -y -i r_base
> /usr/sbin/pkgchk -L CSWr-base # list files
```

which has currently R version 3.1.1. It may be a little bit old, but it is good enough for testing ZeroMQ and **pbdZMQ**. The developing/daily version of R may not be easily compiled by the OpenCSW.

6.3. Compile ZeroMQ

The OpenCSW also has binary packages for ZeroMQ, named **libzmq1** and **libzmq1_dev**. The package is a little bit out of date (version 2.2.0 built in 2012). This old version is not comparable with current **pbdZMQ**. The solution is to compile and install an external ZeroMQ 4.0.7 which available from http://download.zeromq.org. This version 4.0.7 is the latest stable ZeroMQ version. The newer versions of ZeroMQ, such as >= 4.1.0, may have more bugs for Solaris 10.

I install the ZeroMQ library to `/work-my/local/zmq` with the following commands.

R Script

```
> gtar zxvf zeromq-4.0.7.tar.gz
> cd zeromq-4.0.7
> ./configure --prefix=/work-my/local/zmq
> make
> make install
```

There may have some warnings, but they are ok. The default installs the share library which can be linked by **pbdZMQ**.

6.4. Install pbdZMQ

With the OpenCSW and external ZeroMQ 4.0.7 installed, I can install **pbdZMQ** simply via

R Script

```
> R CMD INSTALL pbdZMQ_0.2-0.tar.gz \
  --configure-vars="ZMQ_INCLUDE='-I/work-my/local/zmq/include' \
                    ZMQ_LDFLAGS='-L/work-my/local/zmq/lib -lzmq'"
```

Then, the test can be done by

R Script

```
> Rscript -e "demo(hwserver,'pbdZMQ',ask=F,echo=F)" &
> Rscript -e "demo(hwclient,'pbdZMQ',ask=F,echo=F)"
```

when the **pbdZMQ** installation is done correctly.

7. FAQs

7.1. ZeroMQ Errors

1. **Q:** If the package installs successfully, but fails at initialization with

   Error Message

   ```
   error: unable to load shared object
   '/Users/your_username/Library/R/3.2/library/pbdZMQ/libs/pbdZMQ.so':

   dlopen(/.../R/3.2/library/pbdZMQ/libs/pbdZMQ.so, 6):
   Library not loaded: /.../src/zmq/lib/libzmq.4.dylib

   Referenced from: /.../R/3.2/library/pbdZMQ/libs/pbdZMQ.so
   ```

   or

   Error Message

   ```
   Library not loaded: /usr/lib/libzmq.4.dylib
   ```

   or

Error Message

```
Library not loaded: /usr/lib/libzmq.so.4.0.0
```

A: From the ZeroMQ installation messages:

R Script

```
If you ever happen to want to link against installed libraries
in a given directory, LIBDIR, you must either use libtool, and
specify the full pathname of the library, or use the `-LLIBDIR'
flag during linking and do at least one of the following:
   - add LIBDIR to the `LD_LIBRARY_PATH' environment variable
     during execution
   - add LIBDIR to the `LD_RUN_PATH' environment variable
     during linking
   - use the `-Wl,-rpath -Wl,LIBDIR' linker flag
   - have your system administrator add LIBDIR to
      `/etc/ld.so.conf'

See any operating system documentation about shared libraries for
more information, such as the ld(1) and ld.so(8) manual pages.
```

Further, for Mac OSX, the `otool` and `install_name_tool` commands are able to find and modify the linking path of `pbdZMQ.so` to the library `libzmq.4.dylib`. See the source of `pbdZMQ/src/install.libs.R` for how to change the linking path, accordingly. The basic steps are:

(a) to use `otool -L pbdZMQ.so` to figure which `libzmq.4.dylib` is linked. Note that `pbdZMQ.so` should be the one in the installed directory, e.g. `/your_R_library_root_path/library/pbdZMQ/libs/pbdZMQ.so`, and

(b) to use `install_name_tool -change org_dylib new_dylib pbdZMQ.so` to update to the right `new_dylib`, e.g. `/your_R_library_root_path/library/pbdZMQ/libs/libzmq.4.dylib`.

For linux/unix, the `readelf`, `chrpath`, `pathelf`, `ld.so`, and `/etc/ld.so.conf` (depending on systems and compilers) may serve the same roles as the `otool` and `install_name_tool` in Mac OSX.

References

Armstrong W (2014). "rzmq: R Bindings for ZeroMQ." R Package, URL http://cran.r-project.org/package=rzmq.

Chen WC, Ostrouchov G, Schmidt D, Patel P, Yu H (2012). "pbdMPI: Programming with Big Data – Interface to MPI." R Package, URL http://cran.r-project.org/package=pbdMPI.

Chen WC, Schmidt D, Ostrouchov G (2015). "pbdZMQ: Programming with Big Data – Interface to ZeroMQ." R Package, URL http://cran.r-project.org/package=pbdZMQ.

Hintjens P (2013a). "The ZeroMQ Guide – for C Developers." URL http://zguide.zeromq.org/page:all.

Hintjens P (2013b). "ZeroMQ: The Guide."

Ostrouchov G, Chen WC, Schmidt D, Patel P (2012). "Programming with Big Data in R." URL http://r-pbd.org/.

Schmidt D, Chen WC (2015). "pbdCS: pbdR Client/Server Utilities." R Package, URL http://github.com/wrathematics/pbdCS.