Introducing a New Client/Server Framework for Big Data Analytics with the R Language

Drew Schmidt
University of Tennessee
Knoxville, TN
wrathematics@gmail.com

Wei-Chen Chen
pbdR Core Team
Silver Spring, MD
wccsnow@gmail.com

George Ostrouchov
Oak Ridge National Lab
Oak Ridge, TN
ostrouchovg@ornl.gov

ABSTRACT

Historically, large scale computing and interactivity have been at odds. This is a particularly sore spot for data analytics applications, which are typically interactive in nature. To help address this problem, we introduce a new client/server framework for the R language. This framework allows the R programmer to remotely control anywhere from one to thousands of batch servers running as cooperating instances of R. And all of this is done from the user's local R session. Additionally, no specialized software environment is needed; the framework is a series of R packages, available from CRAN. The communication between client and server(s) is handled by the well-known **ZeroMQ** library. To handle server side computations, we use our established pbdR packages for large scale distributed computing. These packages utilize HPC standards like MPI and ScaLAPACK to handle complex, tightly-coupled computations on large datasets. In this paper, we outline the new client/server architecture components, discuss the pros and cons to this approach, and provide several example workflows that bring interactivity to potentially terabyte size computations.

CCS Concepts

•**Mathematics of computing** → **Statistical software;** •**Human-centered computing** → *Command line interfaces;* •**Computing methodologies** → Distributed programming languages;

Keywords

R, Analytics, Remote Computing, Big Data

1. INTRODUCTION

Batch computing was the norm in data analysis before initial versions of the S language were introduced. This data analysis "workflow" language was developed for interactive use of mathematical subroutines, possibly inspired by notions of Tukey's exploratory data analysis [31]. The interactive aspect together with eventual free, open source, in memory, and extensible R implementation [21] of the S language is largely responsible for R's current immense popularity [7].

In contrast, driven by the need for efficient use of expensive platforms, parallel high performance computing and its software artifacts in the form of scalable libraries are almost exclusively batch. Specifically, they are written as single program, multiple data (SPMD) batch codes. We created the pbdR project [19] to harness these excellent scalable libraries for use from R in SPMD form. While this enables the R language to scale to previously impossibly large problems and distributed machines, it removes the popular interactive aspect of the language. Since the project started in 2012, we have released over 15 R packages for distributed parallel computing in R providing I/O, matrix computation, profiling, and various machine learning algorithms (see pbdr.org). In this paper, we describe a new client/server framework that brings back the missing interactivity while retaining the SPMD server side capabilities of our earlier pbdR packages to handle large scale data analysis.

Our client/server approach is in part inspired by HPC visualization software [12, 2], however our implementation is radically different and significantly more modular in design. We are also aware of another client/server [32] recently reported in a conference proceedings. It relies on our pbdR packages for its server capability to drive scalable libraries. However, few implementation details are given and the software does not appear yet to be released.

From our experience, an important aspect of interacting with Big Data is avoiding the introduction of single processor bottlenecks. This means that the data resides on a parallel file system, it is read in parallel by several processors and it is processed in parallel. In a client/server setup, only the parallel server touches the Big Data and the client sees only some reductions and results of the computations on the server. While this is a use recommendation behind our client/server, the user is in complete control of data movement and is not prevented from using it in unproductive ways.

Interactive computing on large cluster resources is still an unsolved problem. It is handled as a single user allocation of a fraction of nodes. This clearly runs counter to the need for efficient use of the expensive resource while it sits idle as the user is concocting or typing the next interactive action. As a result, it is difficult to envision long interactive sessions with large numbers of nodes to be supported by supercomputing center policies. However, SPMD code development on a few nodes is certainly an immediatelly compelling use case. We envision that the availability of this interactive

XSEDE16, July 17 - 21, 2016, ,

ACM ISBN 978-1-4503-4755-6/16/07…$15.00

DOI: http://dx.doi.org/10.1145/2949550.2949651

capability will spur other new uses. For example, with the recent addition of solid state disks (SSD) to compute nodes, checkpoints to SSD may provide an ability to share an interactive allocation between several users, possibly a step toward solving interactive use of large clusters. Further research is needed in this direction.

1.1 Background

It is common for R programmers to stumble in the transition to working on remote machines. Indeed, many R programmers are scientists and statisticians with otherwise limited to nonexistent computing skills beyond a laptop. In fact, the ease of entry to R programming is in part responsible for this phenomenon. Several tools have already been developed to aid in this transition of moving R computations to remote resources, and so before proceeding, it is worthwhile to take a moment to better understand this landscape.

RStudio [22] is a popular integrated development environment (IDE) for R. In addition to functioning as a local application, **RStudio** can be installed and run as a web server in a remote environment. So instead of the user opening a local application, they merely point their web browser to the hosted address. Another tool for moving R computations to the web created by the same developers behind **RStudio** is **shiny** [8]. This is an R package whose purpose is to allow developers to create custom web applications and dashboards powered by R. However, the server components of these utilities were designed for cloud environments, so they generally do not "play nice" with traditional HPC resources. Indeed, it is rare even in XSEDE machines to have **RStudio** available to users, and if so, they typically leave it as an application requiring forwarding X over secure shell (ssh), which is less than ideal. Furthermore, these server components are licensed under the strict Affero GPL [13], which is not always compatible with even other open source software.

Another R package, **servr** [35], is a simple http server made directly available to R. Utilizing this package is the **rmote** package [14], which allows for a remote resource to render R help and some supported plot code and then to have the results displayed in a web browser. This is useful to be sure, but it does not handle general computation, requiring users to control the remote R session over ssh without the use of their preferred editor, or to install one of the aforementioned servers. Again we note that the latter may not even be possible in managed environments, such as those offered by XSEDE.

While R does not natively offer the ability to open or manage sockets, there are several lower level package extensions which allow the user to directly manage socket connections. Among these, the oldest is **snow** [30], whose name is an acronym expanding to "simple network of workstations". There is also the **Rmpi** package [36], which does socket communication over MPI. However, the use of these two packages is somewhat awkward. Each uses a manager/worker paradigm and requires commands to be manually, explicitly transmitted to worker processes from the manager. And neither makes any attempt to move the manager off of a remote resource and onto the user's workstation or laptop.

Finally, there is the **rzmq** package [3], which makes available to R a set of bindings for the **ZeroMQ** package [16]. However, this package is non-trivial to install, and lacks some desirable support features.

For the remainder of this paper, we will describe three new R packages we have developed, namely **pbdZMQ**, **remoter**, and **pbdCS**. These packages constitute a new system for moving R computations to remote systems, supporting everything from cloud computing to supercomputing resources, such as those provided by XSEDE.

2. DESCRIPTION OF THE CLIENT/SERVER FRAMEWORK

2.1 Design and Architecture

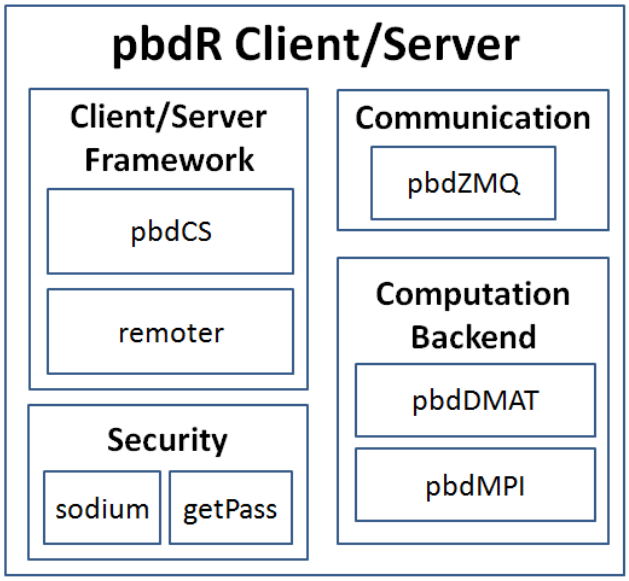

Figure 1: The pbdR client/server software stack. The new client/server framework we describe here is encompassed by the communication layer pbdZMQ and the interface packages remoter and pbdCS. The security layer includes packages for password input and encryption. Finally, the computation backend uses the established pbdR packages for large scale, distributed, statistical computing.

The pbdR client/server framework consists of a series of R extensions. Each package is permissively licensed, and supports all major platforms (Windows, Mac OS X, Linux, and Solaris) for both client and server use. Figure 1 offers a sketch of the structure of the framework in. The packages support three new, separate constructs: client, server, and relay. For the moment, we focus exclusively on the client and server constructs. We discuss all of these packages in the remainder of this section.

The **remoter** package [25] provides the interface for the client/server framework. It features fully interactive communication between R sessions driven by user responses at the command prompt. There is no disconnect between the user's input and the return, unlike the **snow** and **Rmpi** packages, which require function calls for communication. The inputs from the client are transmitted as strings to the server, where they are parsed and executed. Any outputs or returns from the server are automatically passed back to the client and handled accordingly.

Excluding communication code, the client and server are custom REPL's written entirely in R. In this way, no specialized environments need be set up or maintained. If a user can install R packages, they can connect to and launch pbdR servers. This drastically reduces the mental costs for users who may only be familiar with R, such as statisticians and domain scientists. To them, the difference between R and pbdR sessions will be very difficult to detect. And since the client and server are pure R sessions, this makes it very simple to transmit objects directly between client and server, even across different platforms. Additionally, this makes it possible for each of the client and server to run inside of GUI's, such as the popular **RStudio**.

The communication layer is handled by a custom set of bindings to the well-known **ZeroMQ** library [16]. These too are distributed as an R package, **pbdZMQ** [10]. The package brings low-level communication functions to R which are faithful recreations of the **ZeroMQ** source [16]. It also provides detailed examples of messaging patterns originally available in the **ZeroMQ** guide [15], reconstituted for R programmers. The installation of **pbdZMQ** is straightforward. If no system installation of **ZeroMQ** is available (either because one could not be found automatically or because one was not manually provided), then a stable source of **ZeroMQ** shipped with **pbdZMQ** will be automatically compiled and linked. This makes it so that even users with very limited experience compiling and installing system software can easily utilize the framework. Finally, the package also has some new useful constructs. In addition to backwards compatibility with **rzmq**, there is a new interface that mimics the **ZeroMQ** bindings in python. Additionally, there are several new utility functions to simplify file transport between clients and servers.

Some security is available for users who wish to directly connect client and server, as opposed to using ssh tunnels. In this case, we utilize public key cryptography for messages between client and server using the **sodium** package [18]. This package wraps and offers a simple interface to the **libsodium** library [1], a platform independent version of **NaCL** [29]. The package uses the Curve25519 algorithm [4], a popular elliptic-curve-Diffie-Hellman function. We note for the sake of clarity that among all of the packages constituting various components of the client/server framework pictured in Figure 1, this is the only one not developed by the authors. Finally, we also developed a portable, cross-platform package for user input masking in the **getPass** package [23]. Through this, we support password protected servers.

By default, the client will willingly connect to unsecure servers (though this option can be changed); but a server can only accept secure/unsecure connections, based on how it was set up. The perspective from the server during the initial handshake with the client is:

1. Confer about security. Exchange public keys if secure.
2. Credential password if one is set.
3. Do basic package version and safety checks.

Regardless of whether or not the server is secure, messages transmitted between client and server must first be serialized. This is performed via the `serialize()` function of base R. In doing so, we can transmit R objects themselves instead of different, customized representations of those objects. This is important, because R has several object oriented programming systems and a very diverse user-contributed package ecosystem. At the time of writing, this ecosystem has over 8300 packages available from the main package repository, CRAN. It would be impossible to support even just the more popular components of this ecosystem by constructing specialized client/server representations. Our approach does incur some overhead, discussed in Section 3, but it is the most general and supports the largest collection of packages possible.

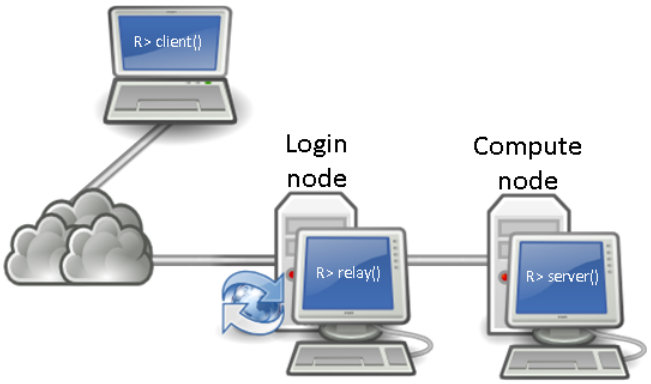

Figure 2: The pbdR client/server as executed on a typical supercomputing resource. The client connects to a relay running on the login node via an ssh tunnel. The relay then sends commands to the server(s) running on the compute node(s) and transmits responses back from server to client.

Moving beyond just the client and server, we finally return to the third aforementioned construct, that of the relay. A relay is a "middleman" between the client and server, and is primarily intended for use in supercomputing environments, where the login and compute nodes tend to be separate. It allows the R programmer to connect from their laptop or workstation indirectly to the compute node of a large HPC resource, by first connecting a relay running on the login node to the server(s) running on the compute node, and finally the client to the relay. Relays are part client and part server. They have an incoming port for clients to connect to, and an outgoing port and address of the server it connects to. Figure 2 provides a visual explanation for this setup, and Section 2.3 elaborates on some of the particulars of utilizing this framework in such environments.

2.2 Using the Client/Server

Because the client/server is entirely R code from the user's perspective, using the system is exceedingly simple. To create a server, one need only call:

```
R> remoter::server()
```

From an R session running on the desired machine. This can be run interactively or in batch, and so this can easily be executed via `Rscript` after creating an ssh tunnel. There are of course numerous options, including changing the communication port from the default value, and various security and logging modes. Once the server is created, the user can connect to it from an R session running on their laptop or workstation by issuing the command:

```
R> remoter::connect(addr="my.remote.
    address")
```

Here of course, "my.remote.address" is the desired address. This address must be given as a string, and can be a valid ip address or a domain such as "ec2-1-2-3-4.compute-1.amazonaws.com". The address can not contain other information such as protocols or ports, the latter of which should be specified as a separate argument. All connections take place over tcp. Once connected, any command entered into the user's R session will be executed on the remoter server. Special handlers are written to deal with warnings, errors, and user events like entering `ctrl+c`. These help maintain the illusion that code is executing normally. All one need do to disconnect from the server is issue the command `exit()`.

Additionally, there are several useful helper utilities for managing data between client and server. Most notably are the `s2c()` and `c2s()` commands, which stand for server-to-client and client-to-server, respectively. The former is a function that takes an object currently stored on the server's R session and transmits it to the client's R session. The object will by default even have the same name, although this can easily be altered. The `c2s()` function is the reverse, moving an object from client to server.

There are several other utilities which allow for the direct management of the client workspace without needing to disconnect from the server. However, unless explicitly specified when disconnecting, the server is persistent. So the client session can happily drop in and drop out from remote computation as needed or desired. So if the client loses internet connectivity, the user need only reissue the `remoter::client()` command to reconnect.

2.3 Use in HPC Environments

The client makes only a few assumptions about what constitutes a "server". Namely:

- The server can evaluate R code
- The client and server communicate via **pbdZMQ**

As such, we can readily extend a **remoter** server into a collection of servers for high performance computing. The **pbdCS** package [24] does just that, by bridging **pbdZMQ** and the MPI and **ScaLAPCK** [5] layer of the pbdR ecosystem [9, 11, 26, 27]. When utilized on large resources such as those provided by XSEDE, this ecosystem easily supports analyses of massive datasets at interactive speeds [20, 28]. Here the, batch, SPMD servers are commanded by the interactive **remoter** client. But instead of having commands executed on in one R process, they are handled in a distributed, massively parallel fashion on a few to thousands of coordinated SPMD servers via the pbdR computational backend.

The **pbdCS** package offers a new **remoter**-like command, `pbdserver()`, which will launch one of the many batch SPMD servers. This can be used from the command line by combining the `mpirun` and `Rscript` utilities. For example, from the shell, we can launch 2 pbdR servers via:

```
$ mpirun -np 2 Rscript -e \
    "pbdCS::pbdserver()"
```

The package also includes several additional utilities for achieving the same result directly from R, including `pbdSpawn()`:

```
R> pbdCS::pbdSpawn(nranks=2)
```

Finally, there is the command `pbdclient()`, which is an alias for `remoter::client()`. The latter is included only for aesthetic purposes.

Figure 3 shows the messaging patterns for the user issuing a command to the client and having it execute on multiple batch, SPMD servers. In this case, rank 0 of the SPMD servers communicates directly with the client just as with the basic client/server pattern described above. However, rank 0 is not held out specifically for communication; i.e., it is involved in batch computations. Once a command is sent from client to rank 0, it must be scattered to the other MPI ranks. This communication is handled by **ZeroMQ** by default. This may incur some slight performance penalty; however, the communication does not operate via "busy waiting", which is more energy efficient and generally more appropriate to this kind of design. However, we note that it is possible to use MPI to disperse client commands from rank 0 to the other ranks, and this is performed via an MPI `bcast()` operation.

Currently, there is no infrastructure supporting interaction with batch schedulers; we discuss this issue further in Section 4. However, there is some automation in managing the distribution of the ip address of rank 0 among the other batch servers when using **ZeroMQ** for command broadcasting.

2.4 Command Execution and Object Storage

For both **remoter** and **pbdCS** servers, there are many similarities. For instance, the way commands are passed from client to server (rank 0 of the multiple MPI servers in the **pbdCS** case) is the same, consisting of a simple **ZeroMQ** socket send. This command is then executed on the server(s), and then some information is returned from server(s) back to the client. However, moving data can be very expensive, and so this last element is worthy of some discussion.

Whenever a server in communication with a client completes a command, successfully or otherwise, it returns a "status" object. This is a list containing information such as the number of warnings the command produced, the information (if any) that should be printed, the "visibility" of the return, whether or not the command executed successfully, and so on. These are all recreations of how the basic R REPL operates. So for example, say someone stores the output of a command into a variable `x`, such as:

```
x <- complicated_function(some_data)
```

In this case, the execution occurs silently, or "invisibly". Within the pbdR context, this means that the information returned from server to client is very small, only a few kilobytes. So even if the resulting computation is very large, it will only live on the server until deliberately moved by the client via one of the previously mentioned helper utilities like `s2c()` or `c2s()`.

On the other hand, if the return is "visible", such as by the client simply executing `x` or `print(x)`, then in R this would print the object. So too in the pbdR system, this will print the object in the client's R session, so this requires transferring some data. However, only the printed information, not necessarily the full object is returned. Many objects in

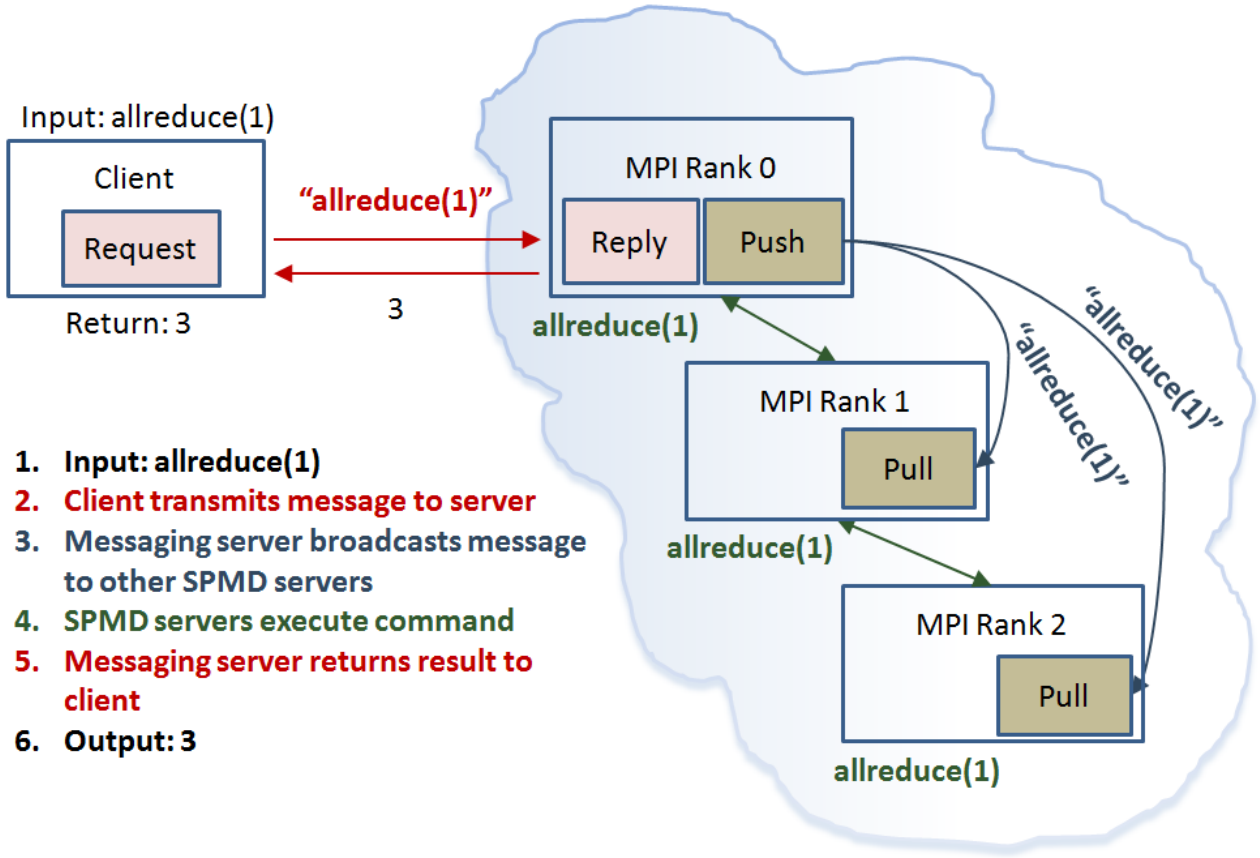

Figure 3: An example with a single input and return of the new client/server framework. Here, the user enters `allreduce(1)`, which will use communication between the SPMD servers to return the total number of servers running. The client captures this input and passes it along to rank 0 of the SPMD servers. Then, using the Push/Pull sockets, the client's message is broadcast from rank 0 to all other SPMD servers. Finally, the command is evaluated, and the result from rank 0 is returned to the client.

R have custom print methods, such as those in the **pbdDMAT** package. These provide some small summary information about potentially very large objects. So the client can potentially inspect large objects without needing to actually move all of the data between client and server. So for pbdR computations, there is no risk of the server accidentally sending a very large object to a small client session. For large objects with "large" prints, then potentially a large amount of data must be transferred. How best to deal with the more general problem is not immediately apparent, and we will return to this problem in Section 4.

3. PERFORMANCE IMPACT AND BENCHMARKS

Usage of the client/server framework is not entirely free from a performance standpoint. This materializes in two key ways. First, the sizes of messages inflates somewhat, depending on the circumstances. And second, there is a generally small, but measurable wall-clock time penalty incurred in using the system. In the remainder of this section, we demonstrate these issues with several benchmarks, and discuss their impact. For each of the benchmarks, we use only a single pbdR server for simplicity.

3.1 Message Size Inflation

We begin with a look at the "message inflation" issue. Recall that in the pbdR client/server framework, native R objects, not abstracted representations, are transmitted between the two R sessions. Whenever an object is sent between client and server, it must first be serialized to a "raw" vector. This is performed by calling the R function `serialize()`. For very small objects, the memory overhead is, proportionally, very high; although in absolute terms, the cost is minimal. Additionally, the number of entries of the vector grows quite large, but proportional to the size of the input. This is a serious problem with some versions of R, where vectors must be indexed by a 32-bit integer. However, this problem is mostly solved in the latest versions of R. Though we note that on managed systems, such as supercomputing resources within XSEDE, often only much older versions of R are available, and so the issue is still relevant.

Table 1 shows measurements of several message sizes and their various costs in terms of both vector length and memory size. In each case, the lengths are given by the `length()` command, and the sizes by the `object.size()` command from base R. The first two columns show the length and size of the input vector, consisting of the specified number of values randomly sampled from a standard normal distribution,

Length	Bytes	Serialized Length	Serialized Bytes	Encrypted Length	Encrypted Bytes
1	48	30	72	46	272
10	168	102	168	118	352
100	840	822	864	838	1064
1000	8040	8022	8064	8038	8264
10000	80040	80022	80064	80038	80264
100000	800040	800022	800064	800038	800264
1000000	8000040	8000022	8000064	8000038	8000264

Table 1: The overhead of serializing and additionally encrypting a vector of double precision numbers of given length.

stored as double precision values. The second two columns show the length and size of the serialized version of the input. This is the data that is transmitted between client and server when encryption is not used. The final column shows the length and size of the data when the serialized data is encrypted. This is the data that is transmitted between client and server whenever encryption is used.

This shows that while there is some additional overhead when using encryption, it is not significant. However, the overhead required to serialize the data for transmission in the first place is significant for very small amounts of data, and insignificant for very large amounts of data. Although we note that the serialized data constitutes a copy, so moving very large data in this way quickly becomes prohibitive. The degree to which this impacts performance will be made clear in the next set of benchmarks. However, note that the length of the message vector is generally 8 times the size of its source. For newer versions of R, this is a non-issue

3.2 Wall-clock Time Overhead

To get some sense for the overhead of the client/server system, we perform several different benchmarks, each under several environments. We begin with a discussion of the compute environments.

First, we consider a typical workstation, a desktop with an Intel SandyBridge processor. In this environment, we examine the performance of three different workflows:

1. vanilla R
2. R with a pbdR client/server
3. R with a pbdR client/server communicating over a relay

Here, each of the client, server, and relay is running locally on the same workstation.

Next, we perform the benchmarks on a t2.micro instance of Amazon's Elastic Compute Cloud (EC2), on a system running in northern Virginia. At the time of writing, this instance consists of a single virtual (non-dedicated) CPU and 1 GiB of ram [17]. Finally, each of the benchmarks is performed on the former XSEDE resource Beacon [6]. Beacon is a 48 node cluster computer connected with an FDR InfiniBand interconnect, and each compute node has 2 8-core Intel Xeon E5-2670 processors with 256 GB of memory. In each of the final two tests, the machine running the client is the same as the machine running client, server, and/or relay in the first three sets of tests.

As for the benchmarks themselves, they consist of:

1. A "no-op": the command `NULL`
2. A simple expression: the command `1+1`
3. A small "compute" piece: `x = rnorm(1e6)`
4. Transferring $\approx$ 7.63 MiB of data: `s2c(x)`
5. Exiting the client/server: `exit()`

Recall that `s2c()` is a function which will take an object from the server's R session and reconstruct it in the client's. Note that the purpose of these benchmarks is not to measure compute overhead of R itself, which is well-studied and documented. Nor is it the goal to focus on the performance of the batch pbdR backend. Our goal here is to determine the impact that the client/server framework has. Recall that data analysis is an interactive discipline. If any system, such as our client server, had such large overhead as to effectively become batch computing, then it is unlikely to be adopted by practitioners of the field.

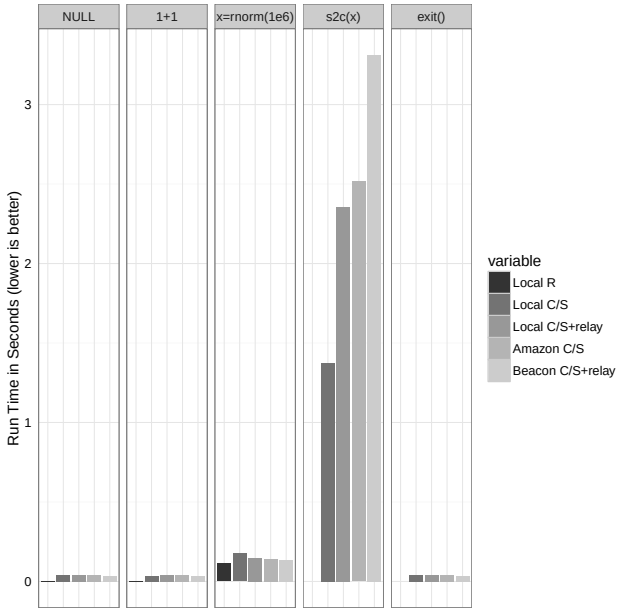

Figure 4: Wall-clock time measurement of some simple operations, demonstrating the costs. Notice that for all operations except for large data transfers, the client/server easily achieves interactive speeds.

Figure 4 shows the results of the benchmarks. This figure is less meant for the purposes of comparison, and more to evaluate the suitability of the client/server in its goals

to achieve interactive speeds. From the picture, it is immediately apparent that overall, there is minimal, acceptable performance loss to the user for typical operations. In the case of the no-op and the simple expression, the run times for the client/server system are drastically higher than those for R, but in absolute terms, effectively unnoticeable. For the data generation benchmark, each of the different environments performed roughly the same, with just the slight overhead from passing messages from client to server tacked on. For large computations, this cost is very quickly amortized, and so these benchmarks strongly suggest that for workflows where significant data is not moving between client, server, and relay, that the performance cost of the pbdR client/server framework is undetectable to the user.

However, we note that the cost for moving data, as with `s2c()` is non-trivial. Indeed, it is the most costly of our tests by far. No such analogue exists in a local R session, so there the cost is effectively free. In the client/server setting however, we see that moving data between client, server, and relay can be quite costly. Interestingly, the cost of adding a relay locally is nearly the same as the cost from going from a purely local environment to running the server remotely on EC2. It may be possible to enhance the performance of the relay system by experimenting with different **ZeroMQ** messaging patterns, or possibly abandoning R for the relay and using a custom application for such purposes.

Note also that these benchmarks plainly show what we argued in Section 2.4 to be true. That is, objects are not passed from server to client upon execution. The execution time for the data generation was well under one quarter of one second in each case. However, the movement from server to client via deliberate `s2c()` call took upwards of 3.5 seconds in the worst case.

Finally, we note that both the EC2 instance and Beacon were relatively geographically close to the machine running the benchmarks. Were that not the case, we would expect the performance to degrade slightly.

3.3 Overhead Discussion and Conclusions

The benchmarks make it very clear that moving large objects directly between the client and server is expensive both in terms of memory usage and wall-clock time. As both previously discussed in Section 2.4 and then proven in the benchmarks, the results of large computations are not necessarily passed from server to client; and if using the pbdR system for large data processing, then they never are unless explicitly requested. All of this suggests a general workflow:

1. Move data to the remote resource using GridFTP, ftp, sftp, etc.
2. Perform computations with R and pbdR.
3. To export data:
   (a) If the data is large, then save to disk and transfer back via ftp, et al. as before.
   (b) If the data comfortably fits on a laptop, then it can be directly transferred to the client via `s2c()`.

4. CHALLENGES AND FUTURE WORK

One immediate drawback to **remoter** is that it only works interactively. Ironically after years of resistance, R users are finally moving past strictly interactive computing and embracing batch in droves, by way of the popular **knitr** system [34]. With **knitr**, R programmers create "reproducible documents", which are specialized markdown or LaTeX documents containing source code elements. These elements are automatically evaluated in R (or other supported languages/frameworks) at the time the document is processed into a "human readable" format, such as pdf or html. Having large computations executed on a much more powerful remote server is an attractive feature for this workflow. We note that supporting batch computing with **remoter** does not immediately make it usable in such a workflow, but it is a necessary first step.

Additionally, at the moment the **remoter** package does not handle visualizations very well. Some R plotting systems such as **ggplot2** [33] can be made to work in the current setup, but it is not completely natural or intuitive. Going forward, it may be reasonable to integrate the **rmote** package, or a similar solution using a web server for displaying graphics.

As mentioned in a previous section, a user could print the entirety of a very large object's data. This combined with the benchmarks above suggest that one could potentially lock up the client/server system for a very long time. It is simple to examine an object's memory profile, and so it would be possible to only print objects that are within some size tolerance. However, as noted above, some objects have custom reduced print methods, and it is impossible a priori to know which objects this applies to. One solution would be to create an object "whitelist" of safe objects to print. Another would be to simply stick to the size cutoff, but offer some sort of "full print" function to the user.

Finally, there is the issue of working with the batch schedulers on managed machines, such as those within the XSEDE ecosystem. As noted in a previous section, there is currently no collection of utilities for interacting with batch schedulers. We have made some early developments in solving this problem, but the work is still immature. Eventually, this work will allow the user to launch and control a job running on a large HPC resources from his/her laptop with a single command from an R terminal. This will include connecting to the login node of the machine, creating a relay for the client/server, launching a job on the compute nodes, and spinning up the SPMD servers. One difficulty is balancing the variance among configurations on such machines with the desire to support as many of them as possible for users. Though we note that the problem is not insurmountable, as VisIt [12] has a similar approach.

5. ACKNOWLEDGMENTS

We thank the three anonymous reviewers whose comments and suggestions helped improve and clarify this manuscript.

This work 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. This work used resources supported by the National Science Foundation under Grant Number 1137097 and by the University of Tennessee through the Beacon Project. Any opinions, findings, conclusions, or recommendations expressed in this work are those of the authors and do not necessarily reflect the views of the National Science Foundation or the

University of Tennessee.

The submitted manuscript has been authored by a contractor of the U.S. Government under Contract No. DE-AC05-00OR22725. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

Figure 2 is composed of elements from the OSA Icon Library http://www.opensecurityarchitecture.org/cms/library/icon-library which is licensed CC-BY-SA. Thus Figure 2 is also licensed CC-BY-SA.

6. REFERENCES

[1] Libsodium encryption library. https://download.libsodium.org/doc/, 2015.

[2] J. Ahrens, B. Geveci, and C. Law. Paraview: An end-user tool for large data visualization. 01 2005.

[3] W. Armstrong. rzmq: R bindings for ZeroMQ, 2014. R Package.

[4] D. J. Bernstein. Curve25519: new diffie-hellman speed records. In *Public Key Cryptography-PKC 2006*, pages 207–228. Springer, 2006.

[5] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[6] R. G. Brook, A. Heinecke, A. B. Costa, P. Peltz, V. C. Betro, T. Baer, M. Bader, and P. Dubey. Beacon: Deployment and application of intel xeon phi coprocessors for scientific computing. *Computing in Science & Engineering*, 17(2):65–72, 2015.

[7] S. Cass. The 2015 top ten programming languages: New languages enter the scene, and big data makes its mark. *IEEE Spectrum*, 20 July 2015.

[8] W. Chang, J. Cheng, J. Allaire, Y. Xie, and J. McPherson. *shiny: Web Application Framework for R*, 2015. R package version 0.12.1.9000.

[9] W.-C. Chen, G. Ostrouchov, D. Schmidt, P. Patel, and H. Yu. pbdMPI: Programming with big data – interface to MPI, 2012. R Package.

[10] W.-C. Chen, D. Schmidt, C. Heckendorf, and G. Ostrouchov. pbdZMQ: Programming with big data – interface to ZeroMQ, 2015. R Package.

[11] W.-C. Chen, D. Schmidt, G. Ostrouchov, and P. Patel. pbdSLAP: Programming with big data – scalable linear algebra packages, 2012. R Package.

[12] H. Childs, E. Brugger, B. Whitlock, J. Meredith, S. Ahern, D. Pugmire, K. Biagas, M. Miller, C. Harrison, G. H. Weber, H. Krishnan, T. Fogal, A. Sanderson, C. Garth, E. W. Bethel, D. Camp, O. Rübel, M. Durant, J. M. Favre, and P. Navrátil. VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data. In *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*, pages 357–372. Oct 2012.

[13] F. S. Foundation. Gnu affero general public license, November 2007.

[14] R. Hafen. *Serve Graphics over HTTP a from Remote Server*, 2016. R package version 0.3.4.

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

[16] P. Hintjens. Zeromq: The guide, 2013.

[17] M. LLC. Introducing t2, the new low-cost, general purpose instance type for amazon ec2, July 2014.

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

[19] G. Ostrouchov, W.-C. Chen, D. Schmidt, and P. Patel. Programming with big data in r, 2012.

[20] P. Patel, D. Schmidt, W.-C. Chen, and G. Ostrouchov. High-level analytics with r and pbdr on cray systems. Technical report, Oak Ridge National Laboratory (ORNL); Oak Ridge Leadership Computing Facility (OLCF); Joint Institute for Computational Sciences (JICS), 2014.

[21] R Core Team. R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. 2013, 2014.

[22] RStudio Team. RStudio: integrated development for R. *RStudio, Inc., Boston, MA. URL http://www. RStudio. com/ide*, 2014.

[23] D. Schmidt and W.-C. Chen. *getPass: Masked User Input*, 2016. R package version 0.1-0.

[24] D. Schmidt and W.-C. Chen. *pbdCS: pbdR Client/Server Utilities*, 2016. R package version 0.1-0.

[25] D. Schmidt and W.-C. Chen. remoter: Remote R: Control a remote R session from a local one, 2016. R Package.

[26] D. Schmidt, W.-C. Chen, G. Ostrouchov, and P. Patel. pbdBASE: Programming with big data – core pbd classes and methods, 2012. R Package.

[27] D. Schmidt, W.-C. Chen, G. Ostrouchov, and P. Patel. pbdDMAT: Programming with big data – distributed matrix algebra computation, 2012. R Package.

[28] D. Schmidt, G. Ostrouchov, W.-C. Chen, and P. Patel. Tight coupling of R and distributed linear algebra for high-level programming with big data. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 811–815, 2012.

[29] P. Schwabe and R. Nijmegen. Nacl: a new crypto library dj bernstein, u. illinois chicago & tu eindhoven tanja lange, tu eindhoven joint work with.

[30] L. Tierney, A. J. Rossini, N. Li, and H. Sevcikova. snow: Simple network of workstations, 2012. R package (v:0.3-9).

[31] J. Tukey. *Exploratory data analysis*. Addison-Wesley Pub. Co, Reading, Mass, 1977.

[32] Z. Wang, S. Fan, R. Gu, C. Yuan, and Y. Huang. iPLAR: Towards Interactive Programming with Parallel Linear Algebra in R. In G. Wang, A. Zomaya, G. Perez, and K. Li, editors, *Algorithms and Architectures for Parallel Processing*, volume 9531 of *Lecture Notes in Computer Science*, pages 104–117. Springer International Publishing, 2015.

[33] H. Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York, 2009.

[34] Y. Xie. *Dynamic Documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition, 2015. ISBN 978-1498716963.

[35] Y. Xie. *servr: A Simple HTTP Server to Serve Static Files or Dynamic Documents*, 2016. R package version 0.3.

[36] H. Yu. Rmpi: Parallel statistical computing in R. *R News*, 2(2):10–14, 2002.