The `rredis` Package

Bryan W. Lewis
blewis@illposed.net

April 30, 2010

1 Introduction

The `rredis` package provides a native R interface to Redis. Redis is an in-memory key/value database with many innovative features written by Salvatore Sanfilippo. It supports data persistence, networked client/server operation, structured value types, server replication, data expiration, and it's very fast.

The following simple example illustrates a typical use of the rredis package:

```
> library('rredis')
> redisConnect()
> redisSet('x',rnorm(5))
[1] TRUE
> redisGet('x')
[1]  0.808448325 0.341482747 -0.728739322 -0.105507214 -0.002349064
```

The key name "x" is associated with the R vector produced by `rnorm(5)` and stored in Redis. Note that the R object associated with "x" is available to other rredis clients, and indeed to any Redis client that can de-serialize R objects. Neither the Redis server nor the rredis clients need reside on the machine on which the result was generated. Depending on the Redis server settings, "x" can be persistent–that is the value and its association with "x" will persist even if the Redis server is terminated and re-started.

Values in Redis are classified by type. Value types are perhaps the most distinguishing feature of Redis.

- The canonical *string* type holds general-purpose objects, for example any serializable R object, text, or arbitrary binary data.

- The *list* type represents lists of Redis *string* objects, ordered by insertion order. Data can be accessed from lists with stack-like PUSH and POP operations, or by directly indexing ranges of elements. Importantly, redis lists support atomic blocking and asynchronous operation.
- Redis *sets* are unordered collections of unique Redis *strings* equipped with typical set operations like unions and intersections. Uniqueness is enforced by Redis at insertion-time. Redis also supports ordered sets, but the rredis package does not yet include them (the next version will!).

Expiration intervals or absolute expiration times may be set on any Redis value, even when the database is persistent. The Redis server can handle lots of small transactions with aplomb, easily exceeding 50,000 transactions/second even on very limited hardware[1]. Although Redis is an in-memory database, it uses a custom virtual memory system to support large objects and databases larger than available RAM.

2 Supported Platforms

The Redis server is written in ANSI C and supported on most POSIX systems including GNU/Linux, Solaris, *BSD, and Mac OS X. The server is not officially supported on Windows systems at the time of this writing (March, 2010).

The rredis package for R is supported on all supported R platforms, including Microsoft Windows, and can connect to a Redis server running on a supported platform.

Redis clients are available for lots of languages other than R, including Java, C, C#, Ruby, Python, PHP, Tcl, Perl, Erlang, Clojure, Javascript, Scala, and more...

2.1 Obtaining and Installing the Redis server

Redis is an open-source project available from http://code.google.com/p/redis, with source code available from Github at http://github.com/antirez/redis.

It is not necessary to "install" Redis to use it. One may download the code, compile it, and run it in place. We include an example command-line procedure applicable to most POSIX operating systems for completeness.

```
wget http://redis.googlecode.com/files/redis-1.2.5.tar.gz
tar xf redis-1.2.5.tar.gz
cd redis-1.2.5
make
# <<Some output from your C compiler>>
```

[1] Redis easily exceeds 100,000 transactions/second on typical high-end workstations

At this point, unless an error occurred, you have a working copy of Redis. The Redis server is completely configured by the file `redis.conf`. In order to run the Redis server as a background process, edit this file and change the line:

```
daemonize no
```

to:

```
daemonize yes
```

You may wish to peruse the rest of the configuration file and experiment with the other server settings as well. Finally, start up the Redis server with

```
./redis-server ./redis.conf
```

3 The rredis Package by Example

We explore operation of many of the Redis features available to R through a few examples. Seek out the rredis package documentation and the excellent Redis Wiki referenced therein for additional help and examples.

3.1 Basic Operation and Redis Strings

Redis *strings* represent the canonical value type. They are used to store any R object that can be serialized to a bit-stream. Most R objects are serializeable. Notable exceptions include objects with open connections and external reference pointers.

We assume from now on that the rredis package is loaded in the running R session using either

```
require('rredis')
```

or

```
library('rredis')
```

prior to running any example.

Open a connection to a Redis server with `redisConnect`. By default, `redisConnect()` attempts to connect to a Redis server locally on a default port (6379). Explicitly specify a host and/or port to connect to a server running on a computer different from the computer on which the R session is running, for example,

```
redisConnect(host='illposed.net', port=5555)
```

to connect to a Redis server running on host 'illposed.net' at port 5555.

Once connected we can easily store and retrieve values in the Redis database with `redisSet` and `redisGet`:

```
> x <- rnorm(5)
> print(x)
[1] -0.3297596 1.0417431 -1.3216719 -0.8186305 -0.2705817
> redisSet('x',x)
[1] TRUE
> y <- redisGet('x')
> print(y)
[1] -0.3297596 1.0417431 -1.3216719 -0.8186305 -0.2705817
> all.equal(x,y)
[1] TRUE
> redisGet('z')
NULL
```

Note that one must explicitly specify a key name ("x" in the above example) and that Redis key names need not correspond to R variable names. Unlike R variable names, Redis key names may not contain spaces.

The SET/GET operations are atomic–that is, multiple SET and or GET operations are guaranteed not to simultaneously occur. And `redisGet` always returns immediately, even if a value is not available in which case it returns NULL (see the example).

The true power of Redis becomes apparent when we share values across multiple clients. For example, start up a new R session and try:

```
> library('rredis')
> redisConnect()
> y <- redisGet('x')
> print(y)
[1] -0.3297596 1.0417431 -1.3216719 -0.8186305 -0.2705817
```

The default behavior of Redis is to make the database persistent, so the value associated with "x" in the above examples will last until it is overwritten or explicitly removed, even if the Redis server is re-started. One may immediately purge Redis of all key/value pairs with the (dangerous) `redisFlushAll` command.

Redis supports multiple distinct key workspaces, indexed by number. Access may be switched between workspaces with the `redisSelect` function as illustrated below. We also use `redisKeys` to list all key names in the current workspace.

```
> redisKeys()
[[1]]
[1] "x"
```

```
> redisSelect(1)
[1] "OK"
> redisKeys()
NULL

redisSelect(0)
> redisKeys()
[[1]]
[1] "x"
```

The number of available workspaces is user-configurable in the `redis.conf` file (the default is 16). Note also that index values in Redis begin with 0.

One may easily store and retrieve multiple objects in one operation with `redisMSet` and `redisMGet`. The example also illustrates how values may be expired (in this case, after one second) with `redisExpire`.

```
> redisMSet(list(x=pi,y=runif(5),z=sqrt(2)))
[1] TRUE
> redisMGet(c('x','y','z'))
$x
[1] 3.141593
$y
[1] 0.85396951 0.80191589 0.21750311 0.02535608 0.11929247
$z
[1] 1.414214

> redisExpire('z',1)
[1] TRUE
> Sys.sleep(1)
> redisGet('z')
NULL
```

3.2 Sharing Data with Clients other than R

Redis provides a particularly convenient system for sharing data between diverse applications. We illustrate cross-application communication with simple examples using R and the `redis-cli` command-line program that is included with the Redis server.

Open a terminal window and navigate to the directory in which the Redis server was compiled (see Section 2.1). You will find a command-line application named `redis-cli`. Store a sample value in the Redis database with:

```
./redis-cli set shell "Greetings, R client!"
OK
```

Now, leaving the terminal window open, from an R session, try:

```
> redisGet('shell')
[1] "Greetings, R client!\n"
```

And, voilà, R and shell communicate text through Redis.

The reverse direction requires more scrutiny. From the R session, run:

```
> redisSet('R', 'Greetings, shell client!')
```

And now, switch over to the shell client and run:

```
./redis-cli get R
<<Partially decipherable garbage>>
```

This example produces undesirable results because the default behavior of the R `redisSet` command is to store data as R objects, which the shell client cannot decipher. Instead, we must encode the R object (in this case, a character string) in a format that shell can understand:

```
> redisSet('R', charToRaw('Greetings, shell client!'))
[1] TRUE
```

And now, switch over to the shell client and run:

```
./redis-cli get R
Greetings, shell client!
```

It can be tricky to share arbitrary R objects with other languages, but raw character strings usually provide a reasonable, if sometimes inefficient, common tongue.

3.3 Redis Lists

Redis list value types provide us with a remarkably powerful and rich set of operations. Redis lists may be used to set up data queues and they may be accessed either synchronously or asynchronously.

We walk through basic Redis list operation in the first example below. The example shows how `redisLPush` pushes values onto a list from the left, and `redisRPush` pushes values from the right.

```
> redisLPush('a',1)
[1] 1
> redisLPush('a',2)
[1] 2
> redisLPush('a',3)
```

```
[1] 3
> redisLRange('a',0,2)
[[1]]
[1] 3
[[2]]
[1] 2
[[3]]
[1] 1

> redisLPop('a')
[1] 3
> redisLRange('a',0,-1)
[[1]]
[1] 2
[[2]]
[1] 1

> redisRPush('a','A')
[1] 3
> redisRPush('a','B')
[1] 4
> redisLRange('a',0,-1)
[[1]]
[1] 2
[[2]]
[1] 1
[[3]]
[1] "A"
[[4]]
[1] "B"

> redisRPop('a')
[1] "B"
```

Like the `redisGet` function, `redisLPop` and `redisRPop` always return immediately, even when no value is available in which case they return NULL. Redis includes a blocking variant of the list "Pop" commands that is illustrated in the next example.

```
> redisBLPop('b',timeout=1)
NULL

> redisLPush('b',runif(5))
[1] 1
```

```
> redisBLPop('b',timeout=1)
$b
[1] 0.3423658 0.4188430 0.2494071 0.9960606 0.5643137
```

In the first case above, the NULL value is returned after a one-second timeout because no value was immediately available in the list. Once populated with data, the second attempt consumes the list value and returns immediately.

We can also block on multiple lists, returning when data is available on at least one of the lists:

```
> redisFlushAll()
[1] "OK"
> redisLPush('b',5)
[1] 1
> redisBLPop(c('a','b','c'))
$b
[1] 5
```

Although blocking list operations seem simple, they provide an extraordinarily powerful environment for coordinating events between multiple R (and other client) processes. The following example illustrates a simple event stream in which data is emitted periodically by a shell script, and consumed and processed as events arrive by an R process.

First, open an R window and block on the "a" and "b" lists:

```
> redisFlushAll()
> for (j in 1:5) {
+ x <- redisBLPop(c('a','b'))
+ print (x)
+ }
```

Your R session should freeze, waiting for events to process.

Now, open a terminal window and navigate to the directory that contains the `redis-cli` program. Run (the following may all be typed on one line):

```
for x in 1 2 3 4 5;do sleep $x;
  if test $x == "2";
    then ./redis-cli lpush a $x;
    else ./redis-cli lpush b $x;
  fi;
done
```

And now you will see your R session processing the events as they are generated by the shell script:

```
$b
```

```
[1] "1"

$a
[1] "2"

$b
[1] "3"

$b
[1] "4"

$b
[1] "5"
```

Now, imagine that events may be processed independently, and that they occur at an extraordinary rate–a rate too fast for R to keep up. The solution in this case is simple, start up another R process and it will handle events as they come in, relieving the first R process of about half the event load. Still not enough, start up another, etc.

Keeping in mind that the R clients can run on different computers, we realize that this simple example can easily lead to a very scalable parallel event processing system that requires very little programming effort!

3.4 Redis Sets

The Redis set value type operates somewhat like Redis lists, but only allowing unique values within a set. Sets also come equipped with the expected set operations, as illustrated in the following example.

```
> redisSAdd('A',runif(2))
[1] TRUE
> redisSAdd('A',55)
[1] TRUE
> redisSAdd('B',55)
[1] TRUE
> redisSAdd('B',rnorm(3))
[1] TRUE
> redisSCard('A')
[1] 2
> redisSDiff(c('A','B'))
[[1]]
[1] 0.5449955 0.7848509
```

```
> redisSInter(c('A','B'))
[[1]]
[1] 55

> redisSUnion(c('A','B'))
[[1]]
[1] 55

[[2]]
[1] 0.5449955 0.7848509

[[3]]
[1] -1.3153612 0.9943198 -0.3725513
```

Redis sets do not include blocking operations.

4 A Few Remarks

Redis is a simple to use, powerful key/value database. Remember that Redis key names may not contain spaces.

The Redis server is not yet supported on Windows operating systems. The rredis R client package will work on Windows (as well as most other) systems and communicate with a Redis server running, for example, on Linux.

Redis does many things exceedingly well, but does not do everything well. Redis is not optimized for slicing list and set data, for example in OLAP-style queries or to extract subsets of long column-oriented data. Use an appropriate database for those types of operations.

Although Redis includes a basic authentication mechanism, **don't use it**. If you require secure access to the server, or encrypted communication with the server, we recommend this alternative: Set up Redis on a (perhaps virtual) machine that only authenticated users may access. Then block all but local access to the Redis server port (by default, 6379). Finally, tunnel access to the port through an authenticated SSH session or other secure port forwarding mechanism.