Elastic computing with R and Redis

Bryan W. Lewis
blewis@illposed.net

January 12, 2023

1 Introduction

The future package[1] defines a simple and uniform way of evaluating R expressions asynchronously. Future "backend" packages implement asynchronous processing over various distributed and shared-memory systems.

Redis[2] is a fast networked key/value database that includes a stack-like data structure (Redis "lists"). This feature makes Redis useful as a lightweight task queue manager.

The future.redis package defines a simple elastic distributed computing backend for future using the redux package[3] to communicate with Redis. This style of distributed computing is well-suited to modern cloud environments, and especially cloud spot markets. Key features of future.redis are:

- future.redis allows for dynamic pools of workers. New workers may be added at any time, even in the middle of running computations. This feature is geared for modern cloud computing environments, including spot compute markets.
- future.redis computations are partially fault tolerant. Failure of worker R processes, for instance due to spot instance termination, are detected and affected tasks are automatically re-submitted.
- future.redis makes it particularly easy to run parallel jobs across different operating systems. It works equally well on GNU/Linux, Mac OS X, and Windows systems. Back end parallel R worker processes are effectively anonymous–they may run anywhere as long as all the R package dependencies required by the task at hand are available.

This vignette refers the following application processes: a Redis server; a *manager* R process – this is the R process that submits futures to be processed; zero or more *worker* R processes – these are the R processes that run the futures and return results.

Each of the above processes may run on different computers communicating through Redis over a network. Or, they may all run on the same computer and communicate through Redis over a local network.

Note that the number of backend workers can be *zero*. In such cases, the manager R process can submit tasks, but if it requests results it will block until workers become available to complete the work (or until a user cancels the operation with CTRL + C or similar signal).

2 Obtaining and Configuring a Redis server

Redis is an extremely popular open-source networked key/value database, and operating system-specific packages are available for all major operating systems, including Windows. For more information see: `https://redis.io/download`.

The Redis server is completely configured by the file `redis.conf`. It's important to make sure that the `timeout` setting is set to `0` in the `redis.conf` file when using future.redis. You may wish to peruse the rest of the configuration file and experiment with the other server settings. In particular if you plan to communicate with Redis over more than one computer make sure that it's configured to listen on all appropriate network interfaces–and if those computers are on an untrusted network work carefully to secure their communication and to prohibit outside access to the Redis server!

3 Getting started

These notes refer to your interactive R session as the *manager* R process. Install the future.redis package along with those packages it depends on in R the usual way with `install.packages("future.redis")`.

Once installed, register the future.redis backend in your interactive R session with:

```
library("future.redis")
plan(redis)
```

The "redis" plan is a function that takes a number of optional arguments, including the name of a task work queue to set up in Redis. The default task queue name is "RJOBS". Additional Redis server communication parameters and other options can be set there, see `?plan` for help.

Starting local workers

In order to actually process computation, you need to start at least one *worker* R process to pull tasks from the work queue, run the tasks, and return results. The following example illustrates a convenient way to start R workers on the local computer:

```
startLocalWorkers(n = 2, linger = 1, log = "/dev/null")
```

The `linger` option controls how quickly the workers quit once the task queue is removed, in this case after approximately one second. A short linger interval is useful for interactive examples and quick clean-up. The `log` option controls where the workers print R messages during their operation. This can be a file (local to the workers) for logging and debugging purposes. If unspecified, workers display messages on their stderr and stdout streams, which will show up interleaved with the manager R process streams (also useful for simple debugging). We'll see an example of that later.

Run an example!

At last, let's run a really simple example. It simply returns the process ID of each worker R process:

```
Map(value, replicate(2, future(Sys.getpid())))

# [[1]]
# [1] 27859
#
# [[2]]
# [1] 27857
```

If all goes well, you should see distinct numbers (illustrated in comment section above) indicating that the workers are indeed running in separate system processes.

Finally, to signal to the workers to quit and clean up the work queue completely from Redis, use:

```
removeQ()
```

After the `linger` interval, each worker will discover that the work queue has been removed and exit.

Again, with messages

Let's run the previous example again but this time without suppressing stderr and stdout message streams in the workers. This produces messy, but useful for quick debugging, output:

```
startLocalWorkers(n = 2, linger = 1)
Map(value, replicate(2, future(Sys.getpid())))

# Retrieved task a3e078e331e83a5f78b2a3a7a067
# Retrieved task c85df7199ce2bcfcb4fc7d1090e2
# Obtained future RJOBS.a3e078e331e83a5f78b2a3a7a067...
# Obtained future RJOBS.c85df7199ce2bcfcb4fc7d1090e2...
# Submitting result to RJOBS.c85df7199ce2bcfcb4fc7d1090e2.out...
# Submitting result to RJOBS.a3e078e331e83a5f78b2a3a7a067.out...
# [[1]]
# [1] 28470
#
```

```
# [[2]]
# [1] 28472

removeQ()

# Normal worker shutdown
# Normal worker shutdown
```

The above messages are truncated for clarity. Your results will look a little different, but the idea is the same.

You can, of course, also start R workers on other computers, as long as they have R and the future.redis package installed and a way to communicate with Redis over a (secured) network. See `?worker` and the vignettes for running future.redis on AWS for examples.

4 Technical Details

This section outlines a few technical details. Generally, this section assumes that Redis is configured and available and that the task queue is named “RJOBS”.

Sequences of messages between R and Redis are usually coalesced into transactions. Blocking communication with Redis, whenever it occurs, is rate-limited to allow for R session signal interrupt processing since the redux Redis communication package functions are uninterruptible. (The upshot is that the package will perform poorly with lots of very short parallel tasks–use a different approach for that kind of computation.)

Each future defined by a manager R process is serialized and associated with a “task” in a Redis-maintained task queue. The gist of operation is outlined in Figure 1. Futures are dispatched via a shared task queue (“RJOBS” for

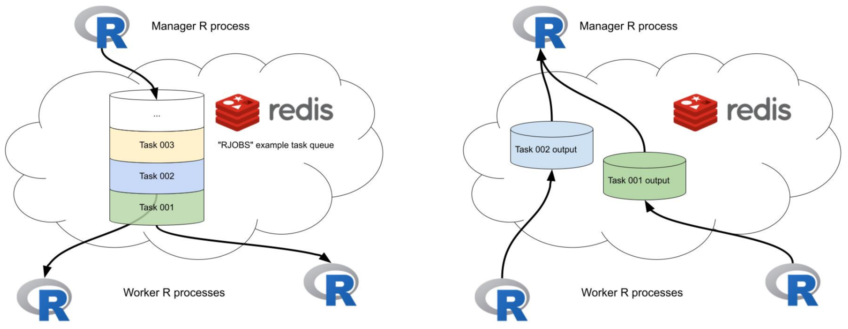

Figure 1: Adding tasks to a task queue named RJOBS (left) and retrieving results (right).

instance). The result of each future appears in Redis in a distinct result output queue. Tasks associated with futures are internally named by the package with unique IDs.

Redis Key Organization

The future.redis package uses a number of keys to coordinate computation and fault detection. All keys are prefixed by the task queue name defined by the manager R process. A brief description of typical keys follows for a task queue named “RJOBS”, using numbers as unique task IDs for clarity.

Queue-global keys

RJOBS: The task first-in-first-out queue, a Redis "list" value type. The manager R process places task IDs to run in this queue. Worker R processes pull tasks IDs to run from this queue.

RJOBS.live: Existence of this key is a hint to the system that the RJOBS queue is in use. Removal of this key using 'removeQ()' or otherwise is a signal to worker R processes listening for work on the RJOBS queue to exit.

Per-task keys

RJOBS.001: The value of this redis key is the serialized future associated with task 001. This key is removed once the manager R process retrieves the completed future.

RJOBS.001.status: A character-valued key with information about the current state of this task, including terms “submitted”, “running”, and “finished”. This key is removed once the manager R process retrieves the completed future.

RJOBS.001.live: Existence of this key indicates that a worker process has pulled the task from the queue and is currently processing it. This is an ephemeral Redis key with a short timeout. Its existence is assured by a thread on the worker R process. If the worker R process crashes, or loses connection with Redis, then RJOBS.001.live will expire and be deleted. If the manager R process observes the inconsistent state of RJOBS.001.status == “running” but non-existence of RJOBS.001.live, then the task is assumed failed and possibly re-submitted.

RJOBS.001.out: The task output queue (a Redis list). The manager R process blocks on this queue waiting for work output, with periodic timeouts to check for problems.

Use the `removeQ()` function to delete all keys in Redis associated with a specified task queue, for instance after manually cancelling an operation with CTRL+C or similar as a cleanup step. Note that `removeQ()` requests that R worker processes listening to the queue exit.

Worker Fault Detection and Recovery

Elastic computation goes two ways: we can expand the worker pool at any time, even during running computations. But we can also *contract* the worker pool at any time, even in the middle of running computations. The latter case requires a way to re-schedule tasks interrupted by workers that stop for any reason.

The future.redis package uses a few simple mechanisms baked into Redis to detect and handle such faults.

Task liveness and status key symmetry

When a worker pulls a task for processing from the queue, it sets up two keys in Redis:

1. a task status key: `QUEUE_ID.TASK_ID.status` that is set to "running",
2. an ephemeral task liveness key: `QUEUE_ID.TASK_ID.live`.

Setting of these keys happens in a single Redis transaction (from the point of view of the manager R process, these two keys are set up in one atomic operation). "Ephemeral" means that the liveness key is set to expire after a short interval.

Then the worker R process starts a thread running to maintain the state of the liveness key by periodically extending its expiration interval a short while. Task liveness is maintained in a (very limited) thread so that the worker R process can independently go about its business of processing the task.

If the worker R process crashes, or if it is externally terminated by a scheduling system (like the AWS spot market), or if it somehow loses network communication with Redis for an interval longer than the task liveness key expiration interval, then the liveness key disappears in Redis (handled for us by the Redis server).

Now, back on the manager R process, when either the `result` or `resolved` methods of the future associated with the task are invoked, the manager R process checks for symmetry between these keys. In particular, if the task status key says "running" and the task liveness key does not exist (because it has expired), then the manager assumes that the worker is gone and re-submits the task...

Task re-submission

Task re-submission is nearly identical to a repeat of task creation with one important exception: the number of re-submissions is bounded by a parameter in the `redis` future plan function. See `?redis` for information on the `max_retries` parameter. If the number of task re-submissions exceeds that entry, then the task is declared failed, set to an error condition, removed from the queue and marked finished. See the `fault.r` file in the tests section for examples of worker fault-tolerance testing.

Lazarus workers

It's possible that a task liveness key expires due to a network interruption between the worker and the Redis server. Network problems are common in distributed systems. In such cases the worker continues to process its task. When the worker R process re-establishes communication with Redis it first checks the `QUEUE_ID.TASK_ID.status` key to see if the task ID is still valid. For instance, the manager R process may have detected its absence and rescheduled the task and it's already been completed by some other worker. In that case the (now surplus) computed task result is discarded by the worker. Otherwise, the worker simply registers the result normally and marks `QUEUE_ID.TASK_ID.status = "finished"`.

Manager Fault Detection and Recovery

This feature is not yet implemented. Currently, an interruption in communication between the Redis server and the manager R process causes the manager to stop with an error. Future versions of the package will attempt to recover in this case.

References

[1] `https://cran.r-project.org/package=future`

[2] `https://redis.io`

[3] `https://cran.r-project.org/package=redux`