Programming Exercise: Distributed Key-Value Store

Goal

The goal of this exercise is to take a single-instance key-value server and build a simple distributed key-value service on top of it. Specifically, we would like you to build a distributed key-value service that shards key/value pairs across multiple Redis servers. The service should be scalable and reliable.

The solution can be as simple as you like or as complicated as you can make it.

Please reply-all with your solution to the problem tar'ed up, along with a description of the data structure and algorithms you created, as well as instructions for us to build/run it.

Building Blocks

In order to facilitate re-use, please use Redis servers as the local key-value store. Your service can communicate with these redis services through Redis clients, which are offered in many languages (e.g., C, C++, Java, Python, Scala, Go, Node.js, etc.), enabling you to build your service in the language of your choice. You can find the list of clients at:

https://redis.io/clients

Redis Server

In order to build a single instance of redis, download the source, untar and build it:

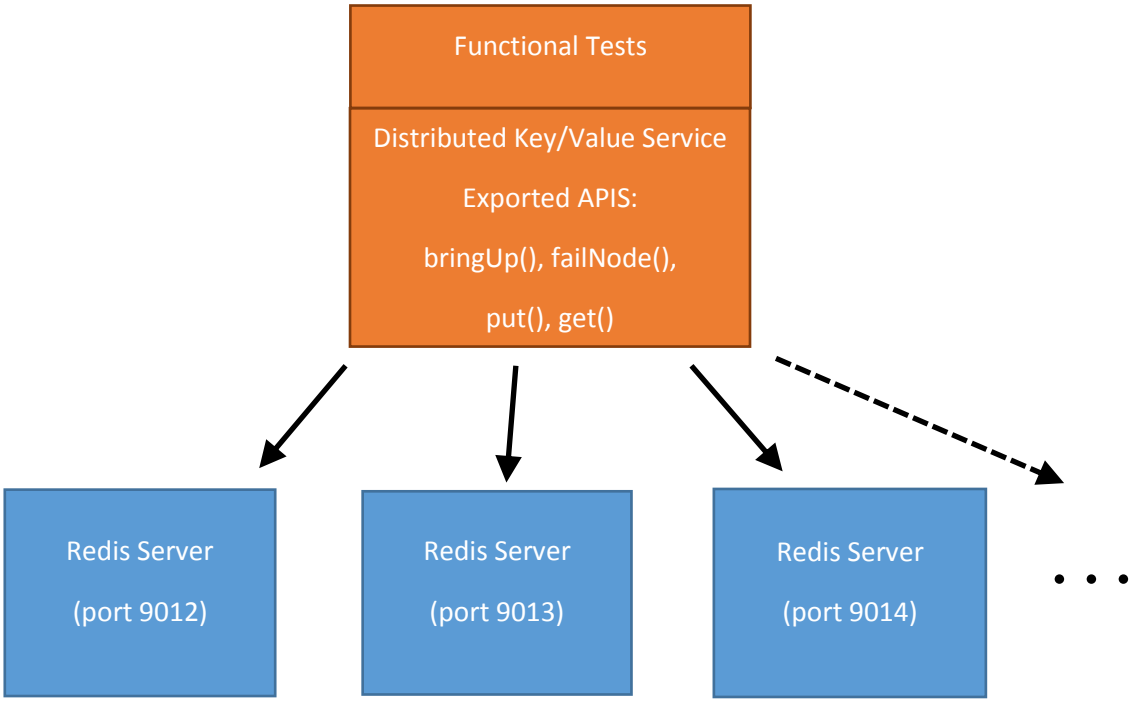
1. Download redis from https://redis.io/download
2. > tar xzf redis-4.0.6.tar.gz
3. > cd redis-4.0.6
4. > make

Starting a single instance of redis requires starting a redis server that listens on a specific port:

1. > ./src/redis-server --port <PORT_NUMBER>

Exercise

The diagram below shows the architecture we propose for you to use in this exercise. However, if you wish, feel free to use a different architecture that meets the requirements below. You should write the logic for the distributed key-value service and implement functional tests to demonstrate your working key/value service that exercises the exported APIs you created. For your convenience and ours, please run your service and all redis servers on localhost.



Requirements

We appreciate the time you have taken to show off your skillset to us and realize your time is valuable. To that end, the service you build need only support the following operations. Feel free to extend the internal implementation as you see fit.

Control APIs:

- `bringUp(int N, int capacity, int reliabilityLevel)`:
  - Starts a set *N* of Redis servers, each with a specific *capacity*, where capacity is the number of key-value pairs each server can store.
  - The cluster should support a given reliability level, such that it can tolerate at least *reliabilityLevel – 1* failures.
  - Note that this command should be executed before any other command is executed.
- `failNode(int port)`:
  - Kill a redis server at a given port
  - This can be called before and after I/O path APIs

I/O Path APIs:

- `put(int key, int value)`: store a key-value pair on one or more of redis servers
- `get(int key)`: retrieve the value for a given value

Demonstrating Functionality

Please write the functional tests you believe are necessary to demonstrate your working application. Given that you've built a reliable and distributed key-value service, we would also like you demonstrate

that your service is scalable and reliable in the face of failures. Finally, in order to show the limits of your implementation, please include negative functional tests.

Questions to consider

- What happens to existing keys that no longer have predefined *reliabilityLevel* after `failNode`() has been called?
- What should the *reliabilityLevel* be for `puts`() after `failNode`()
- Can your solution support a large number of nodes and a large load of requests?