Redis Proxy - Candidate Brief

Introduction

In this coding assignment, you will be building a transparent Redis proxy service. This proxy is implemented as an HTTP web service which allows the ability to add additional features on top of Redis (e.g. caching and sharding). In the following text, the term “the proxy” refers to this proxy you will create. You have to implement all the requirements below which define the minimally viable deliverable.

This problem is designed to test your abilities to do the kind of systems programming we do at Segment, with a focus on **concurrency**, **networking**, **integration** and some **algorithmic optimizations** as well as to test your ability to write software that would be maintained and extended by others.

How to approach this

In general, approach this the way you would if this was a project you undertook as part of your regular job. Specifically, if during the normal course of business you would use a standard library or external library, then do so. If not, then implement the functionality from scratch. We will evaluate this in the same way we evaluate code that we would deploy to production and maintain in a team. Where the specification is unclear or falls short, you should make reasonable assumptions and design choices. When doing this, is it important to thoroughly document your assumptions and design in the README or other relevant documentation artifacts you choose to produce (e.g. code comments or user manual).

To help you make these decisions, keep in mind that we ask candidates to complete these coding exercises in order to:

1. See how they would implement a software solution based on a problem statement which reflects some aspects of the problems we solve on a daily basis.
2. Gauge their technical strengths, which we can use for follow-up conversations.

Where a candidate already has a relevant public code portfolio, we often skip this step. We are, therefore, less interested in seeing how well you can stick to every single detail of a detailed specification (although that definitely helps, especially if

the specification is clear and unambiguous) as much as we are interested in seeing what you are capable of.

Evaluation

NOTE: We will really follow the evaluation procedure outlined below and will not proceed with evaluating the submission if the steps below do no work as expected. Please also pay attention to the `Single-click build and test` requirement which appears further down in the document. It is not an optional requirement and explicitly outlines the platform requirements (i.e. what you can expect from the environment in which the code would be evaluated) and constraints (i.e. what changes you are allowed to make, or not make, to the environment in which you are executing the build and test).

When we receive your submission, the first thing we'll do is to unpack the code archive (or `git clone` it, if appropriate), enter the directory and run `make test`. The expectation is that, by following the steps above, the code would build itself and run all relevant tests. We expect it to, at least, contain an end-to-end test for each requirement you claim to implement. After successfully running the tests, we'll review the code and design.

For example, this should "just work":

```
tar -xzvf assignment.tar.gz
cd assignment
make test
```

Background

Redis describes itself as an "in-memory data structure store" and is deployed as a server process which responds to various text commands. It has an impressive array of client libraries, making it easy to integrate to it from almost any programming language. It stores a variety of data types under string-valued keys, allowing values to be subsequently retrieved and manipulated, using the same key that was used to initially store them.

Requirements

The table below defines requirements that the proxy has to meet and against which the implementation would be measured. It allows the proxy to be used as a simple read-through cache. When deployed in this fashion, it is assumed that all writes are directed to the backing Redis instance, bypassing the proxy.

Name	Description
HTTP web service	Clients interface to the Redis proxy through HTTP, with the Redis “GET” command mapped to the HTTP “GET” method. Note that the proxy still uses the Redis protocol to communicate with the backend Redis server.
Single backing instance	Each instance of the proxy service is associated with a single Redis service instance (called the “backing Redis”). The address of the backing Redis is configured at proxy startup.
Cached GET	A GET request, directed at the proxy, returns the value of the specified key from the proxy’s local cache if the local cache contains a value for that key. If the local cache does not contain a value for the specified key, it fetches the value from the backing Redis instance, using the Redis GET command, and stores it in the local cache, associated with the specified key.
Global expiry	Entries added to the proxy cache are expired after being in the cache for a time duration that is globally configured (per instance). After an entry is expired, a GET request will act as if the value associated with the key was never stored in the cache.
LRU eviction	Once the cache fills to capacity, the least recently used (i.e. read) key is evicted each time a new key needs to be added to the cache.
Fixed key size	The cache capacity is configured in terms of number of keys it retains.
Sequential concurrent	Multiple clients are able to concurrently connect to the

processing	proxy (up to some configurable maximum limit) without adversely impacting the functional behaviour of the proxy. When multiple clients make concurrent requests to the proxy, it is acceptable for them to be processed sequentially (i.e. a request from the second only starts processing after the first request has completed and a response has been returned to the first client).
Configuration	The following parameters are configurable at the proxy startup: • Address of the backing Redis • Cache expiry time • Capacity (number of keys) • TCP/IP port number the proxy listens on
System tests	Automated systems tests confirm that the end-to-end system functions as specified. These tests should treat the proxy as a black box to which an HTTP client connects and makes requests. The proxy itself should connect to a running Redis instance. The test should test the Redis proxy in its running state (i.e. by starting the artifact that would be started in production). It is also expected for the test interact with the backing Redis instance in order to get it into a known good state (e.g. to set keys that would be read back through the proxy).
Platform	The software build and tests pass on a modern Linux distribution or Mac OS installation, with the only assumptions being as follows: 1. The system has the following software installed: • `make` • `docker` • `docker-compose` • `Bash` 2. The system can access DockerHub over the internet.
Single-click build and test	After extracting the source code archive, or cloning it from a Git repo, entering the top-level project directory and executing `make test` will build the code and run all the relevant tests. Apart from the downloading and

	manipulation of docker images and containers, no changes are made to the host system outside the top-level directory of the project. The build and test should be fully repeatable and not requires any of software installed on the host system, with the exception of anything specified explicitly in the `Platform` requirement.
Documentation	The software includes a README file with: • High-level architecture overview. • What the code does. • Algorithmic complexity of the cache operations. • Instructions for how to run the proxy and tests. • How long you spent on each part of the project. • A list of the requirements that you did not implement and the reasons for omitting them.

Bonus Requirements

The requirements below add some additional complexity to the design and can be implemented as a bonus. However, we strongly encourage candidates who are applying for a role which has a strong backend systems focus to implement these as well. To be clear, implement these requirements **in addition** to the ones stated above if you want to impress us even more 🙂

Parallel concurrent processing	Multiple clients are able to concurrently connect to the proxy (up to some configurable maximum limit) without adversely impacting the functional behaviour of the proxy. When multiple clients make concurrent requests to the proxy, it would execute a number of these requests (up to some configurable limit) in parallel (i.e. in a way so that one request does not have to wait for another one to complete before it starts processing).
Redis client protocol	Clients interface to the Redis proxy through a subset of the Redis protocol (as opposed to using the HTTP protocol). The proxy should implement the parts of the Redis protocol that is required to meet this specification.