sproxy

This is a simple reverse proxy implementation, written in Go by *Nikolaos Stavros Gavalas*.

Go was chosen for three main reasons: to its

1. great capabilities in multi-threaded programming
2. very powerful `http` library, that allows very complex routing/proxying logic to be developed easily
3. Personal preference, as I was looking for a good opportunity to develop in Go again.

Usage

Building

To simplify and speed up the various steps required to build/test, a Makefile is provided that supports a number of different functions. More details can be found when running `make help`:

```
$ make help

-----------------------------------------------------
Sproxy: a simple reverse proxy implementation in Go.

Requirements:
- make
- golang 1.17
- docker
- docker-compose
- helm (tested with version 3.8.0)
- a working kubernetes cluster
-----------------------------------------------------

Targets:
image: build an image using Docker (Go app is also built inside the image)
push: push the image to Docker Hub (Warning: requires Nick's credentials for this particular image)
build: build a go binary locally.
dockerun: run the container locally. Address is hardcoded to localhost:8080
helmon: run a helm install
helmoff: run a helm delete
test: Spin-up a test env with httpbin and sproxy
notest: bringdown the test environment
help: print this help file
```

Important: The whole environment is made on the assumption of existing credentials for my personal account on Dockerhub. Target “build” won’t run and the images produced will be by default tagged for my own Dockerhub account.
Helm installation will work though, as the image I created is public and pushed on Dockerhub.

Testing

Testing mechanism is a two step process:

- a docker-compose file that spins up 5 httpbin backends and `sproxy`
- a parallel curl that picks up hosts from `testhosts/curl.txt`

To run a basic test:

```
make image
make test
# wait for tests to complete

# get the metrics
curl localhost:8080/admin/metrics

# bring down docker-compose
make notest
```

Deploying

The provided helm chart will deploy sproxy to a kubernetes cluster.

Application configuration is being done via a ConfigMap which is mounted in the running pod. The data source of this ConfigMap is the original `config.yml` file, which is symlinked in the helm chart.

Obviously, the setup config is for a test environment and there is no other config provided since there are no actual required targets. Nevertheless, switching the service type to a ClusterIP would make the cluster normally available inside a kubernetes cluster. From that point on, a proper config.yml would allow for usage of the service in a kubernetes cluster.

```
# to deploy
make helm
# to delete
make nohelm
```

Service Instrumentation

Instrumenting the service enables measurement of key metrics in order to ensure service availability, performance and scalability.

The main pillars to measure for would be:

- **Availability.**
  This one is also a requirement for a kubernetes deployment. The simplest form of this is also implemented in this assignment, in `health.go`. The approach might be rather simplistic, but it actually relies on Golang's *http* mechanism which will collapse altogether if some fatal uncaught error happens (e.g. too many open files will make the /admin/health endpoint unreachable too).
- **Total Processing time**
  This will provide us with a valuable metric that can show us the total time to serve each request, in percentiles. It is a great metric to understand what is the min/max/avg turnaround time for every request. In this piece of software, `metrics.go` defines the **handlerDuration** metric (which is labeled as "randomproxy" in the main function). The following is a result after 1000 curl parallel requests:

```
$ curl -s localhost:8080/admin/metrics |grep sproxy_total_request_duration_seconds
# HELP sproxy_total_request_duration_seconds Total turnaround time for a request.
# TYPE sproxy_total_request_duration_seconds histogram
sproxy_total_request_duration_seconds_bucket{path="randomproxy",le="0.005"} 4
sproxy_total_request_duration_seconds_bucket{path="randomproxy",le="0.01"} 9
sproxy_total_request_duration_seconds_bucket{path="randomproxy",le="0.025"} 14
sproxy_total_request_duration_seconds_bucket{path="randomproxy",le="0.05"} 18
sproxy_total_request_duration_seconds_bucket{path="randomproxy",le="0.1"} 19
sproxy_total_request_duration_seconds_bucket{path="randomproxy",le="0.25"} 52
sproxy_total_request_duration_seconds_bucket{path="randomproxy",le="0.5"} 184
sproxy_total_request_duration_seconds_bucket{path="randomproxy",le="1"} 529
sproxy_total_request_duration_seconds_bucket{path="randomproxy",le="2.5"} 920
sproxy_total_request_duration_seconds_bucket{path="randomproxy",le="5"} 999
sproxy_total_request_duration_seconds_bucket{path="randomproxy",le="10"} 1000
sproxy_total_request_duration_seconds_bucket{path="randomproxy",le="+Inf"} 1000
sproxy_total_request_duration_seconds_sum{path="randomproxy"} 1111.4201623000006
sproxy_total_request_duration_seconds_count{path="randomproxy"} 1000
```

- **Backend processing times**
  This is a metric that would provide more granular information on the request processing times: It would allow us to know how much time is consumed in the actual backend, while serving the request.
  This should be measured inside `proxy.go`, by creating a wrapper function for `proxy.ServeHTTP()` that can instrument/time it separately. This can be correlated to the previous total processing time metric in order to correlate if/how the proxy responds to slower backend response times.

- **Error rate**
  Measuring errors in a reverse proxy is a long story, because it could be either internal issues (handled/unhandled errors) or missing backends. This is not exactly a metric, rather a set of different metrics that need to be measured in order to have correct visibility on what's going on (and also create error budgets).
  Examples:

  - *total number of errors over total number of requests:* This would need a new Prometheus counter that would increase on erroneous replies (what makes an HTTP error is proxy specific, so I won't deal with it here) and a calculation of the total HTTP replies (this can be measured directly on `http.Server()` in Go).
  - *Unreachable backends:* Right now the application doesn't handle unreachable backends, it just returns an empty reply and throws an error in the log. Normally this should be handled and a proper response should be returned to the requesting client. In addition, we should know how often this particular event happens, versus our (nonexisting for now) logic of health-checking the backends. This Metric should provide insights on how frequently the backend health-checking should occur. This will also provide insights on how resilient the backends are.