

# Expanding the Health Check Capabilities of Envoy

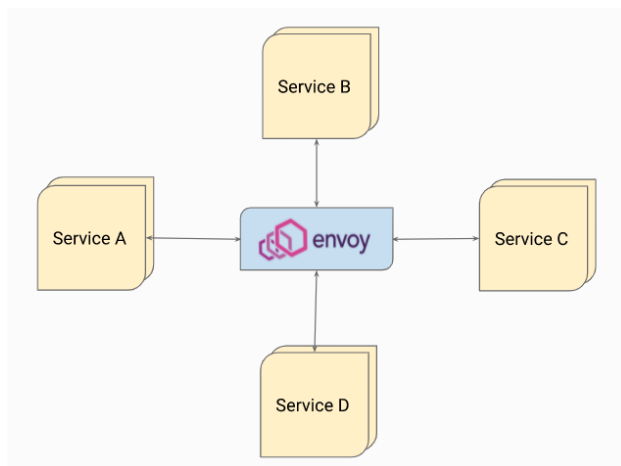
Brian Gregg  
Noah Picarelli-Kombert

## Project Description

Our project for this semester was to expand off of the existing health checking capabilities of Envoy to allow for user-defined degradation thresholds in health checks and load balancing.

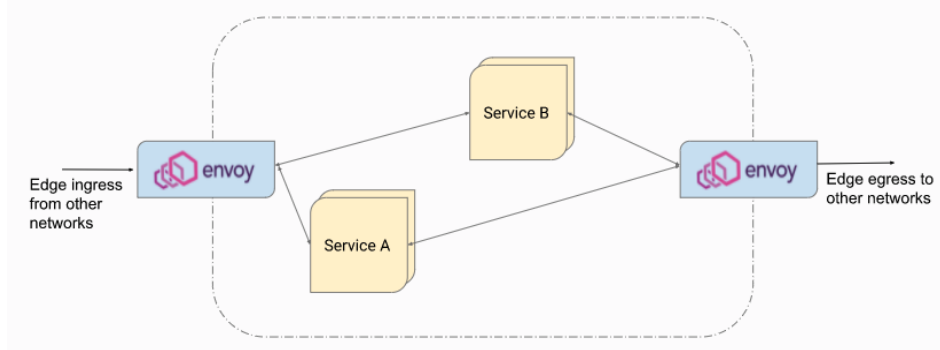
Envoy is an open-source communication proxy that runs with application servers. It works as a network of nodes that route requests and messages to and from a server. Envoy nodes are generic by default but can be configured and organized to provide a variety of services. Some examples include:

- An internal load balancer: the Envoy node receives traffic and determines which server to route it to



- An ingress/egress proxy for a network: two envoy nodes serve as the entry and exit points for communication within a network, facilitating communication with other

## networks and machines



These are the two examples that are illustrated in Envoy's documentation, but it is capable of far more. In particular, Envoy can be leveraged to perform health checks on a server. Envoy has three main types of health checks that it can perform: HTTP checks, L3/L4 checks, and Redis checks. Our project focuses on improving the Redis health checks.

Redis itself is an open-source data structure store. That is, a project can be built on a Redis server to give the developer access to a suite of data structures as well as caching capabilities. Envoy and Redis together can perform active health checking on an application: Envoy nodes handle sending a health check request and receiving a response while the Redis server provides an interface for health checking and runs the code to determine and send the response. The current version of Envoy supports two Redis health check types. The first is a PING check, which sends the string "PING" to the Redis server and expects "PONG" as a response for a healthy server. Anything other than "PONG" indicates a degraded server. The second is an EXISTS check, which sends a given key to the Redis server (as defined in the Envoy configuration file) and expects a null response for a healthy server and any other value for a degraded one. This allows for a server to manipulate its Redis data to mark itself as unhealthy by setting the key to any value, therefore telling Envoy to fail its health check.

We noticed a limitation in Envoy's Redis health check capabilities. Both existing health checks work off of a dichotomy, simply sending a "success" or "failure" signal depending on a hard-coded metric for server degradation. We wanted to allow for a spectrum of possible health

check responses determined by a user-given degradation threshold, thus making it possible for a single health check to evaluate a server by varying standards depending on what the user determines to be an unacceptable state at any given time. To accomplish this, we planned to incorporate a new type of health check that would evaluate server health based on its percentage of disk space used. The health check request would store a user-defined key value representing the minimum disk utilization percentage for a server to be considered degraded. The server would in turn respond with its disk utilization and Envoy would compare this to the key to find whether the check succeeded or failed. Our expectation is that this new health check could be configured to fit a given application and thus allow for more efficient load balancing.

## Project Code

The first step of the project was to modify Envoy's source code to add a new health check type. Each health check request is stored in the Envoy code as an object called `HealthCheckRequest`, and this object has a unique constructor for each type of health check. However, these constructors differ solely by the fact that `EXISTS` accepts a string called "key" as an argument while `PING` accepts no argument. `SPEC` would also need a string argument though, so its constructor declaration would essentially be a copy of the `EXISTS` one. The solution was to combine both constructors. As such, the code changed from:

```
RedisHealthChecker::HealthCheckRequest::HealthCheckRequest(const std::string& key) {
    std::vector<NetworkFilters::Common::Redis::RespValue> values(2);
    values[0].type(NetworkFilters::Common::Redis::RespType::BulkString);
    values[0].asString() = "EXISTS";
    values[1].type(NetworkFilters::Common::Redis::RespType::BulkString);
    values[1].asString() = key;
    request_.type(NetworkFilters::Common::Redis::RespType::Array);
    request_.asArray().swap(values);
}

RedisHealthChecker::HealthCheckRequest::HealthCheckRequest() {
    std::vector<NetworkFilters::Common::Redis::RespValue> values(1);
    values[0].type(NetworkFilters::Common::Redis::RespType::BulkString);
    values[0].asString() = "PING";
    request_.type(NetworkFilters::Common::Redis::RespType::Array);
    request_.asArray().swap(values);
}
```

To:

```

RedisHealthChecker::HealthCheckRequest::HealthCheckRequest(const std::string& key) {
    std::vector<NetworkFilters::Common::Redis::RespValue> values(2);
    values[0].type(NetworkFilters::Common::Redis::RespType::BulkString);
    if(isNumber(key)){
        values[0].asString() = "GET";
    }
    else {
        values[0].asString() = "EXISTS";
    }
    values[1].type(NetworkFilters::Common::Redis::RespType::BulkString);
    values[1].asString() = key;
    request_.type(NetworkFilters::Common::Redis::RespType::Array);
    request_.asArray().swap(values);
}

RedisHealthChecker::HealthCheckRequest::HealthCheckRequest() {
    std::vector<NetworkFilters::Common::Redis::RespValue> values(1);
    values[0].type(NetworkFilters::Common::Redis::RespType::BulkString);
    values[0].asString() = "PING";
    request_.type(NetworkFilters::Common::Redis::RespType::Array);
    request_.asArray().swap(values);
}

```

Note that an if/else statement has been added to assign the value associated with request type based on whether or not the given key represents a number or not (this will be elaborated on shortly). The assignment involving “GET” corresponds to our SPEC check while the one for “EXISTS”, of course, corresponds to the EXISTS check. The two can be put in the same constructor because SPEC is very similar to EXISTS. Both accept a key value and both request types are called and evaluated in a similar way. The only reason why “GET” is used here instead of “SPEC” is because these strings actually represent operations that Redis uses to retrieve the check responses. Naturally, “SPEC” does not exist as an operation and our work does not extend to modifying Redis, so we have used the existing generic “GET” operator to retrieve our new check’s responses.

Another example of the similarity between SPEC and EXISTS is in methods for the RedisHealthChecker class:

```

static const NetworkFilters::Common::Redis::RespValue& pingHealthCheckRequest() {
    static HealthCheckRequest* request = new HealthCheckRequest();
    return request->request_;
}

static const NetworkFilters::Common::Redis::RespValue&
existsHealthCheckRequest(const std::string& key) {
    static HealthCheckRequest* request = new HealthCheckRequest(key);
    return request->request_;
}

static const NetworkFilters::Common::Redis::RespValue&
specHealthCheckRequest(const std::string& key) {
    static HealthCheckRequest* request = new HealthCheckRequest(key);
    return request->request_;
}

```

These methods call the aforementioned HealthCheckRequest constructor to create a request of the type corresponding to the method name. The EXISTS and SPEC iterations are the same as they effectively share a constructor, and are given individual methods here solely to make clearer that SPEC actually exists.

Though they are similar, the main difference between SPEC and EXISTS is that the key that SPEC expects is a string representing a number between 0 and 100 while EXISTS expects a num-number string. Originally, EXISTS expected any string while PING simply expected no key at all. In fact, the RedisHealthChecker object (which accepts a health check request object as an argument in order to actually make the request) determined its health check type by evaluating the key in this constructor's if/else statement:

```

RedisHealthChecker::RedisHealthChecker(
    const Upstream::Cluster& cluster, const envoy::config::core::v3::HealthCheck& config,
    const envoy::extensions::health_checkers::redis::v3::Redis& redis_config,
    Event::Dispatcher& dispatcher, Runtime::Loader& runtime,
    Upstream::HealthCheckEventLoggerPtr& event_logger, Api::Api& api,
    Extensions::NetworkFilters::Common::Redis::Client::ClientFactory& client_factory)
    : HealthCheckerImplBase(cluster, config, dispatcher, runtime, api.randomGenerator(),
        std::move(event_logger)),
      client_factory_(client_factory), key_(redis_config.key()),
      auth_username_(
          NetworkFilters::RedisProxy::ProtocolOptionsConfigImpl::authUsername(cluster.info(), api)),
      auth_password_(NetworkFilters::RedisProxy::ProtocolOptionsConfigImpl::authPassword(
          cluster.info(), api)) {

    if (!key_.empty()) {
        type_ = Type::Exists;
    }
    else {
        type_ = Type::Ping;
    }
}

```

However, as our new health check also accepted a key, this method of distinguishing check types would not work; there would be no way to tell the difference between EXISTS and SPEC checks. Our solution was to make this change:

```
bool isNumber(const std::string& str)
{
    for (char const &c : str) {
        if (std::isdigit(c) == 0) return false;
    }
    return true;
}

RedisHealthChecker::RedisHealthChecker(
    const Upstream::Cluster& cluster, const envoy::config::core::v3::HealthCheck& config,
    const envoy::extensions::health_checkers::redis::v3::Redis& redis_config,
    Event::Dispatcher& dispatcher, Runtime::Loader& runtime,
    Upstream::HealthCheckEventLoggerPtr& event_logger, Api::Api& api,
    Extensions::NetworkFilters::Common::Redis::Client::ClientFactory& client_factory)
    : HealthCheckerImplBase(cluster, config, dispatcher, runtime, api.randomGenerator(),
        std::move(event_logger)),
      client_factory_(client_factory), key_(redis_config.key()),
      auth_username_(
          NetworkFilters::RedisProxy::ProtocolOptionsConfigImpl::authUsername(cluster.info(), api)),
      auth_password_(NetworkFilters::RedisProxy::ProtocolOptionsConfigImpl::authPassword(
          cluster.info(), api)) {}

if (!key_.empty() && !isNumber(key_)) {
    type_ = Type::Exists;
}
else if (!key_.empty() && isNumber(key_)){
    type_ = Type::Spec;
}
else {
    type_ = Type::Ping;
}
}
```

We implemented a new method for creating specific Redis health checks. To create a PING check, one gives no key as before. However, now one must provide a key that is not a numeric string to create an EXISTS check. If the key is a string representing a number, then the check will be of type SPEC. Note the new function `isNumber`, which was added in order to determine if a given string shows a number or not. This is also used in the previously-discussed `HealthCheckRequest` constructor to correctly set the type of the new request.

Several other minor changes needed to be made to accommodate the new health check, such as adding a new case to the `onInterval` method that creates health check request objects so that it could make our new health check. The last major change was to the `onResponse` method:

```

void RedisHealthChecker::RedisActiveHealthCheckSession::onResponse(
    NetworkFilters::Common::Redis::RespValuePtr&& value) {
    current_request_ = nullptr;

    switch (parent_.type_) {
    case Type::Exists:
        if (value->type() == NetworkFilters::Common::Redis::RespType::Integer &&
            value->asInteger() == 0) {
            handleSuccess();
        } else {
            handleFailure(envoy::data::core::v3::ACTIVE);
        }
        break;
    case Type::Ping:
        if (value->type() == NetworkFilters::Common::Redis::RespType::SimpleString &&
            value->asString() == "PONG") {
            handleSuccess();
        } else {
            handleFailure(envoy::data::core::v3::ACTIVE);
        }
        break;
    case Type::Spec:
        if (isNumber(value->asString()) &&
            std::stoi(value->asString()) < std::stoi(parent_.key_)) {
            handleSuccess();
        } else {
            handleFailure(envoy::data::core::v3::ACTIVE);
        }
        break;
    default:
        NOT_REACHED_GCOVR_EXCL_LINE;
    }
}

```

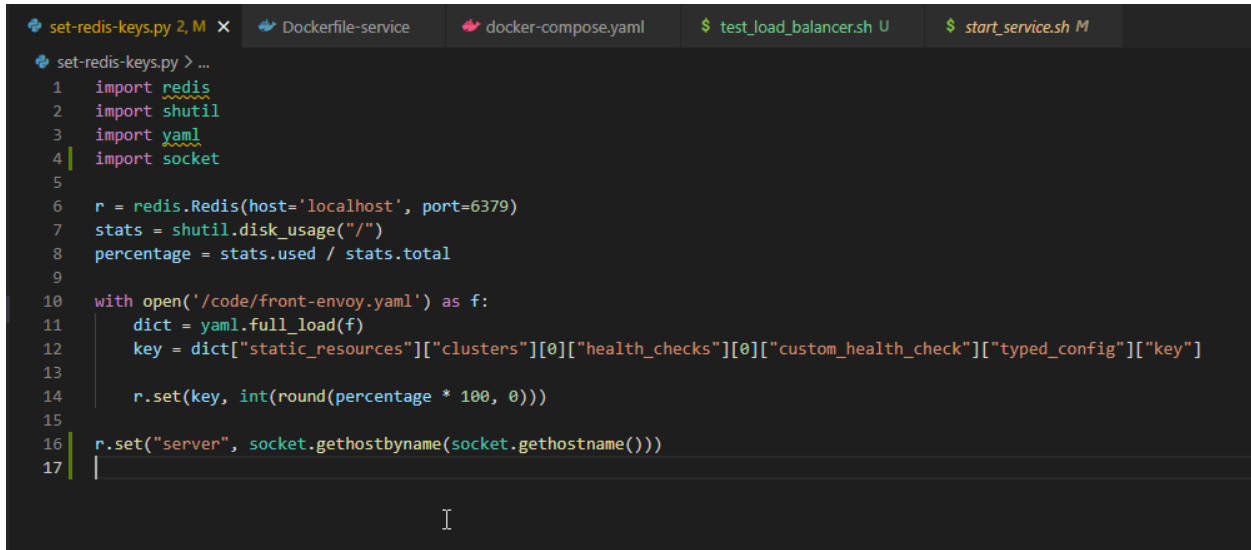
The final case of the switch statement was added so that responses to our new SPEC check could be properly received. The expectation is that the response will be a string representing the percentage of server memory used. This is then compared to the request key, which is the minimum utilization percentage for degradation, and based on the comparison result either a success or failure is handled.

In order to compile Envoy, a powerful PC is needed. C++ code has a slow compilation speed to begin with, and Envoy itself is a massive project. Therefore, in order to compile our new code, we needed to start a CloudLab experiment. We were able to use the CloudLab experiment to build an Envoy binary and create a Docker image which we could use in our environment.

With these changes to the source code, the new health check had been successfully created. The next step was to configure the server(s) to retrieve the key from the Envoy configuration and set the Redis value at that key to its disk utilization. To accomplish this, we



used the Python programming language along with four Python libraries: redispy, shutil, sockets, and pyyaml (available via pip3).



```
set-redis-keys.py 2, M x Dockerfile-service docker-compose.yaml test_load_balancer.sh U start_service.sh M
set-redis-keys.py > ...
1 import redis
2 import shutil
3 import yaml
4 import socket
5
6 r = redis.Redis(host='localhost', port=6379)
7 stats = shutil.disk_usage("/")
8 percentage = stats.used / stats.total
9
10 with open('/code/front-envoy.yaml') as f:
11     dict = yaml.full_load(f)
12     key = dict["static_resources"]["clusters"][0]["health_checks"][0]["custom_health_check"]["typed_config"]["key"]
13
14     r.set(key, int(round(percentage * 100, 0)))
15
16 r.set("server", socket.gethostbyname(socket.gethostname()))
17 |
```

The Python server first connects to the local Redis server running on port 6379 (the default port for Redis). Then the code uses the `shutil disk_usage` method to get a custom object with three key-value pairs: `total` (total size of the disk in bytes), `used` (total bytes used), and `free` (total available bytes). To get the percent disk utilization, a simple division is performed (`used/total`).

To get the key from the Envoy configuration, the `yaml.full_load` function is used to load the configuration yaml file into a Python dictionary. The key is then accessed via normal Python dictionary operations as seen above. The code then rounds the percentage to a whole number and sets the key from the configuration to that value. Although it would potentially be more accurate to send a float value as opposed to rounding to an integer, the defined Redis response type only accepts numbers as integers and not floating point values.

The python code lastly sets the Redis “server” key to the IP address of the container. This is used during our testing to keep track of which server the Redis requests are being routed to during load balancing. The python sockets library allows us to easily retrieve the container’s IP address via the `socket.gethostbyname()` function.

## Configuring Communication Between the Servers

Now that we had the Redis health checker updated and a small Python application to set the Redis values, it was time to set up an environment to connect the two. For this we chose to use Docker-Compose, which is an extension for Docker that allows for multiple containers to be started and connected via a configurable network. Our particular Docker compose environment had two Docker containers: one representing the Envoy front proxy (a.k.a the load balancer) and one representing the server to be health checked. The Docker-Compose configuration can be seen below.

```
examples > front-proxy > 🐳 docker-compose.yml
1  version: "3.7"
2  services:
3
4    front-envoy:
5      build:
6        context: .
7        dockerfile: Dockerfile-frontenvoy
8      networks:
9        - envoymesh
10     ports:
11       - "8080:8080"
12       - "8002:8002"
13
14     service1:
15       build:
16         context: .
17         dockerfile: Dockerfile-service
18       volumes:
19         - ./service-envoy.yaml:/etc/service-envoy.yaml
20       networks:
21         - envoymesh
22       environment:
23         - SERVICE_NAME=1
24
25     networks:
26       envoymesh:
27         ipam:
28           driver: default
29           config:
30             - subnet: 172.28.0.0/16
31
```

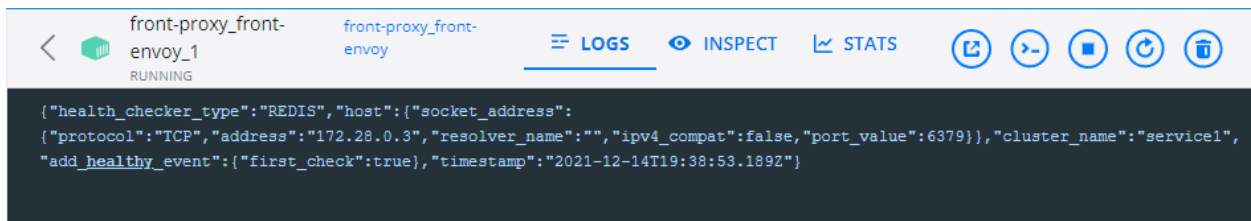
The file describes two services (front-envoy and service1) and their corresponding Dockerfiles. It also defines a network to connect the containers on the subnet 172.28.0.0/16.

The last step was to configure Envoy on the proxy container. The proxy configuration has two main components: first, a listener on port 8080 which is defined via filter chains as a Redis proxy. This allows us to query Redis on the proxy without installing Redis on the proxy itself. The second component is a cluster with a load balanced endpoint listening at port 6379, the default port for Redis. It also defines the custom health check, which is sent every 10 seconds to the server and logs results to the standard output. Savvy readers may notice the Redis key is configured here; our example configuration uses a threshold of 80 percent.

```
! front-envoy.yaml
1  static_resources:
2    listeners:
3      - address:
4          socket_address:
5            address: 0.0.0.0
6            port_value: 8080
7          filter_chains:
8            - filters:
9              - name: envoy.filters.network.redis_proxy
10                typed_config:
11                  "@type": type.googleapis.com/envoy.extensions.filters.network.redis_proxy.v3.RedisProxy
12                  stat_prefix: egress_redis
13                  settings:
14                    op_timeout: 5s
15                  prefix_routes:
16                    catch_all_route:
17                      cluster: service1
18
19    clusters:
20      - name: service1
21        type: STRICT_DNS
22        lb_policy: ROUND_ROBIN
23        load_assignment:
24          cluster_name: service1
25          endpoints:
26            - lb_endpoints:
27              - endpoint:
28                  address:
29                    socket_address:
30                      address: service1
31                      port_value: 6379
32          health_checks:
33            - timeout:
34                seconds: 300
35              interval:
36                seconds: 10
37              unhealthy_threshold: 1
38              healthy_threshold: 1
39              event_log_path: /dev/stdout
40              always_log_health_check_failures: true
41              custom_health_check:
42                name: envoy.health_checkers.redis
43                typed_config:
44                  "@type": type.googleapis.com/envoy.extensions.health_checkers.redis.v3.Redis
45                  key: '80'
46
47    admin:
48      address:
```

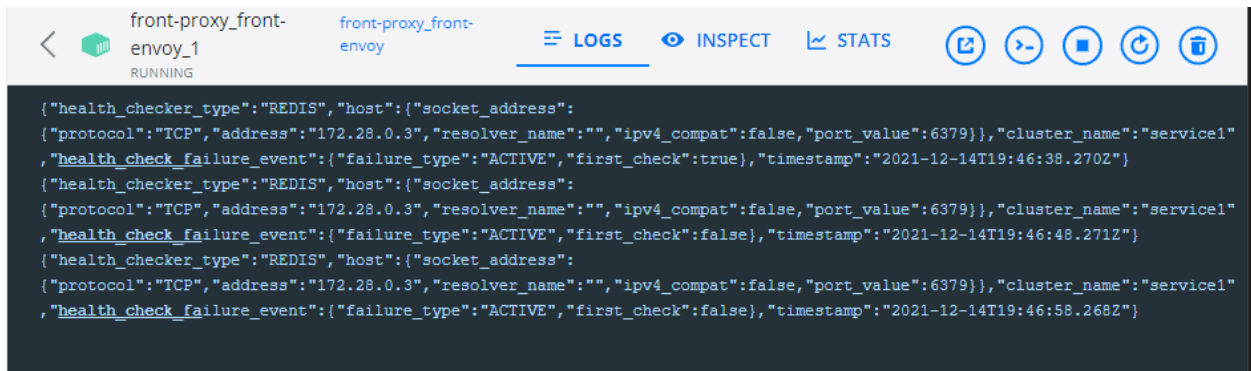
## Test Results

Once our environment was configured, it was time to test our experiment. The first thing we needed to test was whether or not our new Redis health check would properly label hosts as healthy and unhealthy. We did this by first determining the disk utilization manually (7%) and then running our Docker-Compose environment with keys higher and lower than that threshold. A higher key would be expected to yield a healthy response, and a lower key would be expected to yield an unhealthy response. As can be seen in the screenshots below, the health checker correctly labels hosts as healthy or unhealthy based on the given key and disk utilization.

A screenshot of the Envoy web interface showing the logs for the 'front-proxy\_front-envoy\_1' container. The 'LOGS' tab is selected. The log entry shows a successful health check event for a Redis host at 172.28.0.3:6379, with a timestamp of 2021-12-14T19:38:53.189Z. The event details include the health checker type (REDIS), host information, protocol (TCP), address, resolver name, IPv4 compatibility, port value, cluster name (service1), and the add\_healthy\_event details (first\_check: true, timestamp: 2021-12-14T19:38:53.189Z).

```
{ "health_checker_type": "REDIS", "host": { "socket_address": { "protocol": "TCP", "address": "172.28.0.3", "resolver_name": "", "ipv4_compat": false, "port_value": 6379 }, "cluster_name": "service1", "add_healthy_event": { "first_check": true, "timestamp": "2021-12-14T19:38:53.189Z" }
```

A healthy event triggered by a machine with a utilization of 7% and a key of 80%

A screenshot of the Envoy web interface showing the logs for the 'front-proxy\_front-envoy\_1' container. The 'LOGS' tab is selected. The log entry shows a series of health check failure events for a Redis host at 172.28.0.3:6379, with timestamps of 2021-12-14T19:46:38.270Z, 2021-12-14T19:46:48.271Z, and 2021-12-14T19:46:58.268Z. The event details include the health checker type (REDIS), host information, protocol (TCP), address, resolver name, IPv4 compatibility, port value, cluster name (service1), and the health\_check\_failure\_event details (failure\_type: ACTIVE, first\_check: true/false, timestamp: 2021-12-14T19:46:38.270Z, 2021-12-14T19:46:48.271Z, 2021-12-14T19:46:58.268Z).

```
{ "health_checker_type": "REDIS", "host": { "socket_address": { "protocol": "TCP", "address": "172.28.0.3", "resolver_name": "", "ipv4_compat": false, "port_value": 6379 }, "cluster_name": "service1", "health_check_failure_event": { "failure_type": "ACTIVE", "first_check": true, "timestamp": "2021-12-14T19:46:38.270Z" } }, { "health_checker_type": "REDIS", "host": { "socket_address": { "protocol": "TCP", "address": "172.28.0.3", "resolver_name": "", "ipv4_compat": false, "port_value": 6379 }, "cluster_name": "service1", "health_check_failure_event": { "failure_type": "ACTIVE", "first_check": false, "timestamp": "2021-12-14T19:46:48.271Z" } }, { "health_checker_type": "REDIS", "host": { "socket_address": { "protocol": "TCP", "address": "172.28.0.3", "resolver_name": "", "ipv4_compat": false, "port_value": 6379 }, "cluster_name": "service1", "health_check_failure_event": { "failure_type": "ACTIVE", "first_check": false, "timestamp": "2021-12-14T19:46:58.268Z" } }
```

A series of unhealthy events triggered by a machine with a utilization of 7% and a key of 5%

Because health check results are used by the Envoy load balancer to determine where to route server traffic, the behavior of the load balancer was also an effective testing metric for our work. As such, we also checked for appropriate traffic distributions across various combinations of healthy and unhealthy containers. To achieve this, we chose an arbitrary key

value to use as our threshold and manually set the 'disk utilization' value (an extra variable that we could edit for these tests) to either be above the threshold for an unhealthy response or below it for a healthy response via Redis. Then we ran a simple bash script that we wrote that would query the Redis CLI for the "server" key 100 times and count how many times each of the servers had the request routed to it.

We utilized Docker-Compose's scaling functionality to scale our system to 2,3, and 4 application servers. When all hosts were healthy, the load balancer evenly distributed the requests amongst all of the servers. We then would set one of the hosts to be "unhealthy" by manually updating it's Redis value at the threshold key, and we observed that the load balancer stopped routing traffic to that host and distributed the traffic evenly among the remaining healthy hosts. Interestingly, when only one host was healthy, the load balancer would then evenly distribute the traffic again. We theorize that this may be a feature of the Envoy load balancing algorithm, designed to prevent a server from being overloaded if it is the only healthy host on the network.

```
briangregg@LAPTOP-S5C6EVM3:/mnt/c/Users/brian/Documents/envoy_bgregg/examples/front-proxy$ bash ./test_load_balancer.sh
Server 172.28.0.3 was hit 26 times
Server 172.28.0.6 was hit 24 times
Server 172.28.0.5 was hit 25 times
Server 172.28.0.4 was hit 25 times
All done
briangregg@LAPTOP-S5C6EVM3:/mnt/c/Users/brian/Documents/envoy_bgregg/examples/front-proxy$ bash ./test_load_balancer.sh
Server 172.28.0.6 was hit 35 times
Server 172.28.0.5 was hit 32 times
Server 172.28.0.4 was hit 33 times
All done
briangregg@LAPTOP-S5C6EVM3:/mnt/c/Users/brian/Documents/envoy_bgregg/examples/front-proxy$ _
```

A test showing before and after marking host 172.28.0.3 as unhealthy via Redis

## Speculated Improvements

The way we designed this health checker is convenient in so much as it allows the server to define its own definition of "healthy". We chose disk utilization because it was a simple metric but other server side applications could choose to send memory utilization, CPU usage,

or other metrics. The health check would then work the same- comparing the threshold from the configuration to the passed value. However, our code is far from perfect and there are improvements than could be made given more time.

The first that comes to mind is to allow more granular metrics to be sent to the health checker (i.e, floating point values) by extending the Redis RespType class to include a response type for floats. This would allow the server to pass and compare more accurate metrics and therefore receive more accurate feedback regarding its health.

Another improvement that we wish we could have made is to set up a more robust HTTP application to properly test load balancing, as opposed to the simple Redis bash script that we used. We originally had set up a Python application via Flask that would serve information about the endpoint and it's health via HTTP calls. We intended to combine this with the Redis health checking to more thoroughly test our code; however we found the mixing of an HTTP proxy and a Redis proxy to be troublesome. In the end, we were forced to abandon the Flask application due to time constraints and settled for our bash script. However, we are confident that, given enough time, a solution to this problem could have been implemented (most likely via a creative use of filter chains in the proxy's Envoy configuration).

One final, more broad possibility is to move our custom health check into its own type separate from the Redis health check type. This was useful as a base for building our health check, but it does introduce some problems. Mainly, our new version of Envoy cannot consistently be incorporated into existing applications that use Envoy. In order to differentiate between EXISTS and SPEC health checks, we had to change the condition under which EXISTS health checks are identified; instead of looking for any key at all, the code now looks for any non-number key to create an EXISTS check. However, older applications using Envoy might have used number keys to create an EXISTS check. The only way to avoid this and make our addition to Envoy viable for all projects would be to build an entirely new health checking class separate from the existing Redis health check class.

## References

Envoy Project. (2016). Envoy documentation¶. Envoy documentation - envoy 1.21.0-dev-a7bf94 documentation. Retrieved December 14, 2021, from <https://www.envoyproxy.io/docs/envoy/latest/>.