thr Documentation

Release 0.0.1

Fabien MARTY

November 18, 2015

Contents

THR stands for *Tornado HTTP Router*, *Tornado HTTP over Redis* or *Toulouse HTTP Router*, your choice! It's an HTTP proxy based on Tornado and Redis allowing to do all sorts of things with your incoming HTTP traffic such as application-level QoS, request rewriting, etc.

This project is hosted on GitHub: https://github.com/thefab/thr

CHAPTER 1

Getting started

1.1 Hello THR

THR is made of two programs:

- `http2redis` takes incoming requests and performs actions on those requests according to rules. One of the most common actions consists in inserting requests into Redis queues that will get consumed by `redis2http`. When appending a request to a queue, `http2redis` specifies on which Redis key it expects to receive the response and waits for a response on that key.
- `redis2http` reads requests from incoming Redis queues, calls the actual underlying web services and writes responses back to the requested Redis key.

To get started, let's create a minimal web service that responds "Hello World" to any request:

```
from wsgiref.simple_server import make_server


def hello_world_app(environ, start_response):
    start_response(status='200 OK', headers=[('Content-type', 'text/plain')])
    return ["Hello World\n"]

HTTP_PORT = 9999
print("Serving app on http://localhost:{}".format(HTTP_PORT))
make_server('', HTTP_PORT, hello_world_app).serve_forever()
```

Let's create a minimal configuration file for http2redis:

```
from thr.http2redis.rules import add_rule, Criteria, Actions


# Match any incoming request and push it to thr:queue:hello
add_rule(Criteria(), Actions(set_redis_queue='thr:queue:hello'))
```

And here is a minimal configuration file for redis2http:

```
from thr.redis2http.queue import add_queue


# Pop requests from thr:queue:hello and forward them to our service
add_queue('thr:queue:hello', http_port=9999)
```

Start the app server:

```
$ python app_server.py
Serving app on http://localhost:9999
```

Start http2redis:

```
$ http2redis --config=http2redis_conf.py
Start http2redis on http://localhost:8888
```

Start redis2http:

```
$ redis2http --config=redis2http_conf.py
[I 150701 10:18:06 stack_context:275] redis2http started
```

Make an HTTP request:

```
$ curl http://localhost:8888
Hello World
```

The source code for this example can be found in directory `examples/hello`.

1.2 Setting up limits

One of the interesting things about THR is the ability to do rate-limiting based on various criteria. The source code for these examples can be found in directory `examples/limits`.

1.2.1 Fixed limit values

In order to demonstrate how THR can do rate-limiting, we musn't be limited by the backend ability to server multiple simultaneous requests. The basic single-threaded "Hello World" example from the previous section won't be suitable, so we prepare a minimal app that can be served with Gunicorn:

```
import time


def application(environ, start_response):
    time.sleep(1)
    data = "Hello World!\n"
    start_response(status='200 OK', headers=[
        ('Content-type', 'text/plain'),
        ('Content-length', str(len(data))),
    ])
    return [data]
```

We start this app with ten workers processes using Gunicorn:

```
$ gunicorn --workers 10 --bind 0.0.0.0:9999 app_server:application
[2015-07-10 17:17:28 +0000] [13971] [INFO] Starting gunicorn 19.2.1
[2015-07-10 17:17:28 +0000] [13971] [INFO] Listening at: http://127.0.0.1:8000 (13971)
[2015-07-10 17:17:28 +0000] [13971] [INFO] Using worker: sync
[2015-07-10 17:17:28 +0000] [13976] [INFO] Booting worker with pid: 13976
[...]
```

Now we add a limit to `redis2http` configuration file using the `add_max_limit()` function:

```
from thr.redis2http.queue import add_queue
from thr.redis2http.limits import add_max_limit


# Pop requests from thr:queue:hello and forward them to our service
add_queue('thr:queue:hello', http_port=9999)


def limit_foo(request):
    return request.headers.get('Foo') or 'none'

# Limit requests based on a header
add_max_limit('limit_foo_header', limit_foo, "bar", 2)
```

This says that we won't allow more that two simultaneous requests that have the HTTP header `Foo` with a value of `bar`.

After restarting `redis2http` with the new configuration, let's see how the limit affects performance. First, let's try ten concurrent requests that don't match the criteria and therefore shouldn't be affected by the limit. We use the Apache benchmarking tool to do that:

```
$ ab -c10 -n10 -H "Foo: baz" http://127.0.0.1:8888/|grep 'Time taken'
Time taken for tests:   1.045 seconds
```

Each request takes one second to be served, but since we are able to serve all requests at the same time, it still takes one second overall to serve ten requests. Now let's see what happens with requests that do match the limit criteria:

```
$ ab -c10 -n10 -H "Foo: bar" http://127.0.0.1:8888/|grep 'Time taken'
Time taken for tests:   5.055 seconds
```

Our limit of two simultaneous requests being now applied, it takes five seconds to serve ten requests.

1.2.2 Dynamic limit values

If instead of passing a value as the third argument to `add_max_limit()`, we repeat the second argument, then the limit will be applied on requests for which the function returns the same value. Let's change our `redis2http` configuration accordingly:

```
from thr.redis2http.queue import add_queue
from thr.redis2http.limits import add_max_limit


# Pop requests from thr:queue:hello and forward them to our service
add_queue('thr:queue:hello', http_port=9999)


# Just return header value
def limit_foo(request):
    return request.headers.get('Foo')


# Limit requests based on same header value
add_max_limit('limit_foo_header', limit_foo, limit_foo, 2)
```

The Apache benchmarking tool won't allow us to set dynamic headers so we're going to write a small Python script using the Tornado asynchronous client to send ten concurrent requests with ten different values for the `Foo` header:

```
from tornado import gen, ioloop, httpclient


@gen.coroutine
def make_requests():
    client = httpclient.AsyncHTTPClient()
    # Send 10 requests concurrently
    requests = [
        client.fetch('http://127.0.0.1:8888/', headers={
            'Foo': 'value_%s' % i
        }) for i in range(10)
    ]
    responses = yield requests  # Block until we've received all responses
    assert all(response.code == 200 for response in responses)


ioloop.IOLoop.current().run_sync(make_requests)
```

Let's measure its execution time:

```
$ time python dynamic_header_benchmark.py

real    0m1.235s
user    0m0.106s
sys 0m0.093s
```

Since each request has a different value for the `Foo` header, no limit is applied and all ten requests are served concurrently. If however we send the same header with each request, we observe that the limit of two simultaneous requests is applied:

```
$ ab -c10 -n10 -H "Foo: baz" http://127.0.0.1:8888/|grep 'Time taken'
Time taken for tests:   5.051 seconds
```

1.2.3 Statistics file

THR keeps real-time statistics by default in `/tmp/redis2http_stats.json`. You may watch this file while doing your tests to see what is going on:

```
$ cat /tmp/redis2http_stats.json
{
    "bus_reinject_counter": 12,
    "blocked_requests": 2,
    "running_requests": {
        "1886432e05954a23b471fd76eb2e1fb4": {
            "url": "http://localhost:9999/",
            "big_priority": 5,
            "method": "GET",
            "since_ms": 495
        },
        "13b6fc2a9d394d54ad55e7a432e306c9": {
            "url": "http://localhost:9999/",
            "big_priority": 5,
            "method": "GET",
            "since_ms": 491
        }
    },
    "expired_request_counter": 0,
```

```
    "running_bus_reinject_handler_number": 1,
    "epoch": 1437383921.08562,
    "stopping_mode": 0,
    "bus_reinject_queue_localhost:6379_size": 0,
    "total_request_counter": 6,
    "running_request_redis_handler_number": 1,
    "counters": {
        "limit_foo_header_globalblocks": 26,
        "limit_foo_header_globalvalue": 2,
        "limit_foo_header_limit": 2
    }
}
```

This shows you that six requests have been processed (`total_request_counter`), two requests are currently running (`running_requests`) and two requests are waiting to be processed (`blocked_requests`). The location of the statistics file can be customized with the `--stats_file` option of the `redis2http` command.

CHAPTER 2

http2redis configuration

http2redis must be configured with a Python script based on a simple API. You just need to know about one function and two classes to get started. The `add_rule()` function is used to create a new rule. `add_rule()` takes two mandatory parameters: an instance of the `Criteria` class and an instance of the `Actions` class. You may also use the optional `stop` keyword argument to tell THR that if a request matches the rule, it should ignore subsequent rules for that request.

Here is an example:

```
# myconfig.py

from thr.http2redis.rules import add_rule, Criteria, Actions

add_rule(Criteria(path='/forbidden'), Actions(set_status_code=403), stop=1)
add_rule(Criteria(path='/allowed'), Actions(set_status_code=200))
```

Using this configuration, any request made to `/forbidden` will trigger a 403 response code. Requests to `/allowed` should trigger a 200 response.

To use a configuration file, start `http2redis` with the `--config` argument:

```
$ http2redis --config=myconfig.py
Start http2redis on http://localhost:8888
```

Now you can send requests to verify that the configuration file is taken into acount:

```
$ curl -D - http://localhost:8888/forbidden
HTTP/1.1 403 Forbidden
[...]

$ curl -D - http://localhost:8888/allowed
HTTP/1.1 200 OK
[...]
```

CHAPTER 3

redis2http configuration

redis2http is configured with a Python script calling essentially two functions:

- `add_queue()`
- `add_max_limit()`

Here is an example of configuration file for redis2http:

```
from thr.redis2http.limits import add_max_limit
from thr.redis2http.queue import add_queue


def priority_hash(request):
    priority = int(request.headers.get("X-MyApp-Priority", "5"))
    return "low" if priority > 6 else "high"

# Pop requests from a Redis queue named thr:queue:hello and forward them to port 9999
add_queue('thr:queue:hello', http_port=9999)

# Limit rate of requests with an X-MyApp-Priority header greater than 6
add_max_limit("low_priority", hash_func=priority_hash,
              hash_value="low", max_limit=50)
```

To use a configuration file, start `redis2http` with the `--config` argument:

```
$ redis2http --config=redis2http_conf.py
[I 150701 16:43:28 stack_context:275] redis2http started
```

CHAPTER 4

API Reference

4.1 thr.http2redis

4.2 thr.redis2http

CHAPTER 5

Indices and tables

- genindex
- modindex
- search