Creating measurements on RIPE Atlas through the API

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

First publication of this article on 16 April 2013

https://www.bortzmeyer.org/ripe-atlas-api.html

The network of Atlas <http://atlas.ripe.net/> probes managed by the RIPE-NCC exists for a long time. These small probes are installed in many places <http://atlas.ripe.net> all over the world and perform periodic measurements, which are the basis of Internet health checks, interesting statistics and many papers <https://labs.ripe.net/>. There have been many improvments recently, the addition of UDM (User Defined Measurements), where you can decide your own measurements via a Web interface, then an API to retrieve results in a structured format, and now an API to start measurements. It is currently beta.

Two warnings before you read further away : the UDM are not available for the general public. You need an account at RIPE-NCC and some **credits** earned from the Atlas system. And a second caveat : the API is in flux (and the measurement creation API is very recent) and therefore it is quite possible that the code published in this article will not work in the future. My goal is to show that it is possible, not to make the definitive documentation. So, always keep an eye on the official documentation <https://atlas.ripe.net/doc/api> (the measurement creation API is documented separately <https://atlas.ripe.net/alpha/measurement-api/> because it is still unstable.)

First, let's create in Python a script to run a measurement in several geographical areas (the world and four big regions). We will ping www.bortzmeyer.org with IPv6. The Atlas API is a simple REST one, taking parameters in JSON and giving back results in the same format. We will use the urllib2 <http://docs.python.org/2.7/library/urllib2.html> package in Python's standard library and first create a Request because we will need non-standard headers :

```
url = "https://atlas.ripe.net/api/v1/measurement/?key=%s" % key
request = urllib2.Request(url)
request.add_header("Content-Type", "application/json")
request.add_header("Accept", "application/json")
```

The two HTTP headers are added because Atlas only speaks JSON. So, we need to define the parameters in JSON after reading the documentation <https://atlas.ripe.net/alpha/measurement-api/>. First, create a Python dictionary (this is Python code, not JSON, even if it is similar) :

```
data = { "definitions": [
           { "target": "www.bortzmeyer.org", "description": "Ping my blog",
           "type": "ping", "af": 6, "is_oneoff": True} ],
         "probes": [
             { "requested": 5, "type": "area", "value": "WW" } ] }
```

And let's change it at each iteration :

```
for area in ["WW", "West", "North-East", "South-East", "North-Central", "South-Central"]:
    data["probes"][0]["value"] = area
```

And start the measurement with an HTTP `POST` request and the Python dictionary encoded as JSON as a parameter :

```
conn = urllib2.urlopen(request, json.dumps(data))
```

Atlas will send us back a JSON object giving, not the actual results, but the ID of this measurement. We will have to retrieve it later, through the Web interface or via the API, as explained in the next paragraphs. But, for the time being, let's just display the measurement ID. This requires parsing the JSON code :

```
results = json.load(conn)
print("%s: measurement #%s" % (area, results["measurements"]))
```

And that's all. The program will display :

```
% python ping.py
WW: measurement #[1007970]
West: measurement #[1007971]
North-East: measurement #[1007972]
South-East: measurement #[1007973]
North-Central: measurement #[1007974]
South-Central: measurement #[1007976]
```

There are two things I did not explain here : the error handling and the API key. To create a measurement, you need an API key, which you get from the Web interface. In my case, I store it in `$HOME/.atlas/auth` and the script reads it there and adds it to the URL of the request. For the error handling, I suggest you see the actual script, (en ligne sur `https://www.bortzmeyer.org/files/atlas-ping.py`).

The above script did only half of the work. It creates a measurement but does not retrieve and parse it. Let's now do that. Measurements can take a long time (in the previous example, because of the parameter `is_oneoff`, the measurement was done only once ; if it is repeated, it lasts of course much longer) and there is no callback in Atlas, you have to poll. You get a JSON object with a member named `status` which is not thoroughly documented `<https://atlas.ripe.net/doc/data_struct>` but with trial and errors and help from nice people, you can decipher it. The possible values are :

```
   0: Specified
   1: Scheduled
   2: Ongoing
   4: Stopped
   5: Forced to stop
   6: No suitable probes
   7: Failed
   8: Archived
```

Now, Let's poll :

```
over = False
while not over:
    request = urllib2.Request("%s/%i/?key=%s" % (url, measure, key))
    request.add_header("Accept", "application/json")
    conn = urllib2.urlopen(request)
    results = json.load(conn)
    status = results["status"]["name"]
    if status == "Ongoing" or status == "Specified":
            print("Not yet ready, sleeping...")
            time.sleep(60)
    elif  status == "Stopped":
            over = True
    else:
            print("Unknown status \"%s\"\n" % status)
            time.sleep(120)
```

So, for measurement 1007970, we retrieve `https://atlas.ripe.net/api/v1/measurement/1007970/` and parse the JSON, looking for the status. Once the above `while` loop is done, the results are ready and we can get them at `https://atlas.ripe.net/api/v1/measurement/1007970/result/` :

```
request = urllib2.Request("%s/%i/result/?key=%s" % (url, measure, key))
request.add_header("Accept", "application/json")
conn = urllib2.urlopen(request)
results = json.load(conn)
total_rtt = 0
num_rtt = 0
num_error = 0
for result in results:
   ...
```

Here, we will do only a trivial computation, finding the average RTT of all the tests for all the probes. Just remember that some probes may fail to do the test (it is unfortunately much more common with IPv6 tests) so we have to check there is indeed a `rtt` field :

```
    for result in results:
        for test in result["result"]:
            if test.has_key("rtt"):
                total_rtt += int(test["rtt"])
                num_rtt += 1
            elif test.has_key("error"):
                num_error += 1
            else:
                raise Exception("Result has no field rtt and not field error")
...
print("%i successful tests, %i errors, average RTT: %i" % (num_rtt, num_error, total_rtt/num_rtt))
```

And that's all, we have a result :

```
Measurement #1007980, please wait the result (it may be long)
...
12 successful tests, 0 errors, average RTT: 66
```

The entire script is (en ligne sur `https://www.bortzmeyer.org/files/atlas-ping-retrieve.py`).

Now, let's try with a different type of measurements, on the DNS, and a different programming language, Go. The preparation of the HTTP POST request is simple, using the net/http <`http://golang.org/pkg/net/http/`> standard package :

```
client := &http.Client{}
data := strings.NewReader(DATA)
req, err := http.NewRequest("POST", URL + key, data)
req.Header.Add("Content-Type", "application/json")
req.Header.Add("Accept", "application/json")
```

And its execution also :

```
response, err := client.Do(req)
body, err := ioutil.ReadAll(response.Body)
```

But I skipped one step : what is in the `DATA` constant ? It has to be JSON content. Go does not have a way to write a JSON-like object as simply as Python, so, in this case, we cheat, we create the JSON by hand and put it in a string (the proper way would be to create a Go map, with embedded arrays and maps, and to marshall it into JSON) :

```
DATA string = "{ \"definitions\": [ { \"target\":
      \"d.nic.fr\", \"query_argument\": \"fr\",
      \"query_class\": \"IN\", \"query_type\": \"SOA\",
              \"description\": \"DNS AFNIC\", \"type\": \"dns\",
      \"af\": 6, \"is_oneoff\": \"True\"} ],  \"probes\":
              [  { \"requested\": 5, \"type\": \"area\", \"value\": \"WW\" } ] }"
)
```

Now, we just have to parse the JSON content sent back with the standard package encoding/json <`http://golang.org/pkg/encoding/json/`>. Go is a typed language and, by default, type is checked before the program is executed. In the REST/JSON world, we do not always know the complete structure of the JSON object. So we just declare the resulting object as `interface{}` (meaning untyped). Does it disable all type checking ? No, it just postpones it until run-time. We will use type assertions to tell what we expect to find and these assertions will be checked at run-time :

```
err = json.Unmarshal(body, &object)
mapObject = object.(map[string]interface{})
```

In the code above, the type assertion is between parenthesis adter the dot : we assert that the `object` is actually a map indexed by strings, and storing untyped objects. We go on :

```
status := mapObject["status"].(map[string]interface{})["name"].(string)
if status == "Ongoing" || status == "Specified" {
fmt.Printf("Not yet ready, be patient...\n")
time.Sleep(60 * time.Second)
} else if status == "Stopped" {
over = true
} else {
fmt.Printf("Unknown status %s\n", status)
time.Sleep(90 * time.Second)
}
```

There was two other type assertions above, one to say that `mapObject["status"]` is itself a map and one to assert that the field `name` contains character strings. We can now, once the polling loop is over, retrieve the result, parse it and display the result :

```
err = json.Unmarshal(body, &object)
arrayObject = object.([]interface{})
total_rtt := float64(0)
num_rtt := 0
num_error := 0
for i := range arrayObject {
mapObject := arrayObject[i].(map[string]interface{})
result, present := mapObject["result"]
if present {
rtt := result.(map[string]interface{})["rt"].(float64)
num_rtt++
total_rtt += rtt
} else {
num_error++
}
}
fmt.Printf("%v successes, %v failures, average RTT %v\n", num_rtt, num_error, total_rtt/float64(num_rtt))
```

And it displays :

```
Measurement #1008096
...
4 successes, 0 failures, average RTT 53.564
```

And this is the (temporary) end. The Go program is (en ligne sur `https://www.bortzmeyer.org/files/atlas-dns.go`).

Thanks to Daniel Quinn and I[Caractère Unicode non montré [1]]igo Ortiz de Urbina for their help and tricks and patience. A good tutorial on running UDM and analyzing their results is Hands-on : RIPE Atlas <http://cnds.eecs.jacobs-university.de/users/nmelnikov/aims2013-ripe-atlas.html> by Nikolay Melnikov.

1. Car trop difficile à faire afficher par LaTeX