Tester les performances d'un service REST

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 5 janvier 2022

https://www.bortzmeyer.org/benchmark-http.html

J'ai eu récemment à tester un service REST, pour ses performances mais aussi pour sa robustesse, et ça a été l'occasion de faire un rapide tour des outils libres existants. Voici quelques notes.

Précisons bien le cahier des charges : le service à tester était du REST. Certes, il tournait sur HTTP mais ce n'était pas un site Web avec plein de traqueurs, de JavaScript, de vidéos ou de fenêtres polichinelles à faire disparaitre avant de voir quelque chose. Il n'utilisait que HTTP 1.1, je n'ai donc pas regardé ce que ces outils donnaient avec HTTP/2. Et je voulais des outils libres, fonctionnant autant que possible en ligne de commande.

On peut tester les performances d'un serveur HTTP simplement avec curl (et sa très pratique option `--write-out`) mais on voudrait ici des tests répétés, qui matraquent le serveur.

J'ai trouvé six outils différents, et voici mon expérience. Mais d'abord quelques notes générales si vous n'avez pas l'habitude de ce genre d'outils de test :

— Les résultats dépendent évidemment du réseau entre le client (le logiciel de test) et le serveur testé. On obtient évidemment des meilleures performances lorsqu'ils sont sur la même machine, mais c'est moins réaliste. Dans mon cas, le testeur était chez Free et le testé sur Nua.ge <https://www.bortzmeyer.org/nuage.html>, ce qui veut dire entre autres que je ne maitrisais pas complètement les fluctations du réseau. (Mais j'ai aussi fait des tests en local.)
— Le parallélisme compte : une machine même lente peut saturer un serveur HTTP si elle fait de nombreuses requêtes en parallèle. Typiquement, le nombre de requêtes par seconde augmente lorsqu'on augmente le parallélisme, puis diminue au-delà d'un certain seuil.
— Pour atteindre un bon niveau de parallélisme, le client de test doit ouvrir beaucoup de fichiers (en fait, de descripteurs de fichiers), et des erreurs comme « "*socket : Too many open files (24)*" » seront fréquentes. Sur Unix, vous pouvez augmenter la limite du nombre de descripteurs ouverts avec `limit descriptors 65535` (mais attention, cela va dépendre de plusieurs choses, options du noyau, être root ou pas, etc). Comme le dit un message du logiciel Locust « "*It's [la limite] not high enough for load testing, and the OS didn't allow locust to increase it by itself.*" ». Les différents logiciels de test ont souvent une section entière de leurs documentation dédiée à ce problème (voir par exemple celle de Locust <https://github.com/locustio/locust/wiki/Installation#increasing-maximum-number-of-open-files-limit> ou bien celle de Gatling <https://gatling.io/docs/gatling/reference/current/core/operations/>).

— Le goulet d'étranglement qui empêche d'avoir davantage de requêtes par seconde peut parfaitement être dans la machine de test, trop lente ou mal connectée. (C'est pour cela qu'il est sans doute préférable que les programmes de test soient écrits dans un langage de bas niveau.)
— Même avec la plus rapide des machines et le meilleur programme de test, une seule machine peut être insuffisante pour tester une charge importante du serveur. Plusieurs logiciels de test ont un mécanisme pour répartir le programme de test sur plusieurs machines (voir par exemple dans la documentation de Locust <https://docs.locust.io/en/stable/running-locust-distributed.html>).
— Enfin, d'une manière générale, la mesure de performances est un art difficile <https://www.bortzmeyer.org/mesurer-temps-execution.html> et il y a plein de pièges. Au minimum, quand vous publiez des résultats, indiquez bien comment ils ont été obtenus (dire « le serveur X peut traiter N requêtes/seconde » sans autre détail n'a aucune valeur).

Maintenant, les différents outils. On commence par le vénérable ApacheBench <https://httpd.apache.org/docs/2.4/programs/ab.html> (alias ab). Largement disponible, son utilisation de base est :

```
ab -n 5000 -c 100 http://tested.example.org/
```

Le `-n` permet de spécifier le nombre de requêtes à faire et le `-c` le nombre de requêtes en parallèle. ApacheBench produit un rapport détaillé :

```
Time taken for tests:   8.536 seconds
Complete requests:      5000
Failed requests:        0
Requests per second:    585.74 [#/sec] (mean)
Time per request:       170.724 [ms] (mean)
Time per request:       1.707 [ms] (mean, across all concurrent requests)

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        5   93 278.9     39    3106
Processing:     5   65 120.1     43    1808
Waiting:        4   64 120.2     43    1808
Total:         14  158 307.9     87    3454

Percentage of the requests served within a certain time (ms)
  50%     87
  66%    104
  75%    123
  80%    130
  90%    247
  95%   1027
  98%   1122
  99%   1321
 100%   3454 (longest request)
```

Le serveur a bien tenu, aucune connexion n'a échoué. D'autre part, on voit une forte dispersion des temps de réponse (on passe par l'Internet, ça va et ça vient ; en local - sur la même machine, la dispersion est bien plus faible). Le taux de requêtes obtenu (586 requêtes par seconde) dépend fortement du parallélisme. Avec `-c 1`, on n'atteint que 81 requêtes/seconde.

ab affiche des messages d'erreur dès que le serveur HTTP en face se comporte mal, par exemple en fermant la connexion tout de suite (« *"apr_socket_recv : Connection reset by peer (104)"* » ou bien « *"apr_pollset_poll : The timeout specified has expired (70007)"* » et, dans ces cas, n'affiche pas du tout les statistiques.

Ce manque de robustesse est regrettable pour un logiciel qui doit justement pousser les serveurs à leurs limites.

Par défaut, ab ne réutilise pas les connexions HTTP. Si on veut des connexions persistentes, il existe une option `-k`.

Deuxième logiciel testé, Siege `<https://github.com/JoeDog/siege>` (j'en profite pour placer le mot de poliorcétique, que j'adore) :

```
%  siege -c 100 -r 50 http://tested.example.org/
Transactions:         5000 hits
Availability:       100.00 %
Elapsed time:         5.09 secs
Response time:         0.07 secs
Transaction rate:       982.32 trans/sec
Concurrency:        66.82
Successful transactions:        5000
Failed transactions:            0
Longest transaction:         3.05
Shortest transaction:         0.00
```

(Le nombre de requêtes, donné avec `-r` est par fil d'exécution et non pas global comme avec ab. Ici, 50*100 nous redonne 5 000 comme dans le test avec ab.) On voit qu'on a même pu obtenir un taux de requêtes plus élevé qu'avec ab alors que, comme lui, il crée par défaut une nouvelle connexion HTTP par requête. En éditant le fichier de configuration pour mettre `connection = keep-alive` (pas trouvé d'option sur la ligne de commande pour cela), on obtient cinq fois mieux :

```
Transactions:         5000 hits
Availability:       100.00 %
Elapsed time:         0.92 secs
Response time:         0.02 secs
Transaction rate:      5434.78 trans/sec
Concurrency:        85.12
Successful transactions:        5000
Failed transactions:            0
Longest transaction:         0.46
Shortest transaction:         0.00
```

Si vous augmentez le niveau de parallélisme, vous aurez peut-être le message d'erreur :

```
WARNING: The number of users is capped at 255.  To increase this
         limit, search your .siegerc file for 'limit' and change
         its value. Make sure you read the instructions there...
```

Il faut alors éditer le fichier de configuration (qui ne s'appelle pas `.siegerc`, sur ma machine, c'était `/.siege/siege.conf`) et changer la valeur. (Rappelez-vous l'avertissement au début sur les limites imposées par le système d'exploitation.)

Siege a aussi des problèmes de robustesse et vous sort des messages comme « *"[alert] socket : select and discovered it's not ready sock.c :351 : Connection timed out"* » ou « *"[alert] socket : read check timed out(30) sock.c :240 : Connection timed out"* ».

Notons que Siege permet d'analyser le HTML reçu et de lancer des requêtes pour les objets qu'il contient, comme le CSS, mais je n'ai pas essayé (rappelez-vous, je testais un service REST, pas un site Web).

Troisième logiciel testé, JMeter <https://jmeter.apache.org/>. Contrairement à ApacheBench et Siege où tout était sur la ligne de commande et où on pouvait démarrer tout de suite, JMeter nécessite de d'abord définir un "*Test Plan*", listant les requêtes qui seront faites. Cela permet de tester Et, apparemment, on ne peut écrire ce plan qu'avec l'outil graphique de JMeter (« "*GUI mode should only be used for creating the test script, CLI mode (NON GUI) must be used for load testing*" »). J'étais de mauvaise humeur, je ne suis pas allé plus loin.

Quatrième logiciel, Cassowary <https://github.com/rogerwelin/cassowary>. Contrairement aux trois précédents, il n'était pas en paquetage tout fait pour mon système d'exploitation (Debian), donc place à la compilation. Cassowary est écrit en Go et la documentation disait qu'il fallait au moins la version 1.15 du compilateur mais, apparemment, ça marche avec la 1.13.8 que j'avais :

```
git clone https://github.com/rogerwelin/cassowary.git
cd cassowary
go build ./cmd/cassowary
```

Et on y va :

```
% ./cassowary run -u http://tested.example.org/   -n 5000 -c 100

 TCP Connect.....................: Avg/mean=17.39ms  Median=16.00ms p(95)=36.00ms
 Server Processing...............: Avg/mean=15.27ms  Median=13.00ms p(95)=22.00ms
 Content Transfer................: Avg/mean=0.02ms  Median=0.00ms p(95)=0.00ms

Summary:
 Total Req.......................: 5000
 Failed Req......................: 0
 DNS Lookup......................: 8.00ms
 Req/s...........................: 5903.26
```

Par défaut, Cassowary réutilise les connexions HTTP, d'où le taux de requêtes élevé (on peut changer ce comportement avec `--disable-keep-alive`). Lui aussi affiche de vilains messages d'erreur (« "*net/http : request canceled (Client.Timeout exceeded while awaiting headers)*" ») si le serveur HTTP, ayant du mal à répondre, laisse tomber certaines requêtes. Et, ce qui est plus grave, Cassowary n'affiche pas les statistiques s'il y a eu ne serait-ce qu'une erreur.

Cinquième logiciel utilisé, k6 <https://github.com/grafana/k6>. Également écrit en Go, je n'ai pas réussi à le compiler :

```
% go get go.k6.io/k6
package hash/maphash: unrecognized import path "hash/maphash" (import path does not begin with hostname)
package embed: unrecognized import path "embed" (import path does not begin with hostname)
```

Les deux derniers logiciels sont, je crois, les plus riches et aussi les plus complexes. D'abord, le sixième que j'ai testé, Gatling <https://gatling.io/>. On le télécharge, on `unzip gatling-charts-highcha` et on va dans le répertoire ainsi créé. On doit d'abord enregistrer les tests à faire. Pour cela, on lance l'enregistreur de Gatling. L'interface principale est graphique (je n'ai pas vu s'il y avait une interface en ligne de commande) :

```
%  ./bin/recorder.sh
```

Il fait tourner un relais HTTP qu'on doit ensuite utiliser depuis son client HTTP. Par exemple, si j'utilise curl :

```
% export http_proxy=http://localhost:8000/
% curl http://tested.example.org/
```

Et Gatling enregistrera une requête HTTP vers `http://tested.example.org/`. Le plan est écrit sous forme d'un programme Scala (d'autres langages sont possibles) que Gatling exécutera. Il se lance avec :

```
% ./bin/gatling.sh
```

Et on a une jolie page HTML de résultats. Après, les choses se compliquent, Gatling est très riche, mais il faut écrire du code <https://gatling.io/docs/gatling/tutorials/advanced/>, même pour une tâche aussi simple que de répéter une opération N fois.

Enfin, le septième et dernier, Locust <https://locust.io/>. (Après le champ lexical de la guerre, avec Siege et Gatling, celui d'une plaie d'Égypte...) Locust est écrit en Python et il a été simple à installer avec `pip3 install locust`. Il dispose d'une documentation détaillée <https://docs.locust.io/en/stable/installation.html>. Comme JMeter, il faut écrire un plan de test mais, contrairement à JMeter, on peut le faire avec un éditeur ordinaire. Ce plan est écrit lui-même en Python et voici un exemple très simple où le test sollicitera deux chemins sur le serveur :

```
from locust import HttpUser, task
import requests

class HelloWorldUser(HttpUser):
        @task
        def hello_world(self):
            self.client.get("/")
            self.client.get("/hello")
```

Une fois le plan écrit, on peut lancer Locust en ligne de commande :

```
% locust --host http://tested.example.org --headless --users 100 --spawn 10
```

On ne peut pas indiquer le nombre maximal de requêtes (seulement le temps maximal), il faut interrompre le programme au bout d'un moment. Il affiche alors :

```
  Name                                                                                          # reqs      # fails  |     Avg     Min     M
--------------------------------------------------------------------------------------------------------------------------------------------
 GET /                                                                                          20666     0(0.00%)  |      87       7    54
```

On voit que le nombre de requêtes par seconde est faible. Une des raisons est que Locust est très consommateur de temps de processeur : celui-ci est occupé à 100 % pendant l'exécution, et est le facteur limitant. Locust avertit, d'ailleurs, « *"CPU usage above 90% ! This may constrain your throughput and may even give inconsistent response time measurements ! "* ». Bref, il ne pousse pas le serveur à ses limites. Locust réutilise les connexions HTTP, et je n'ai pas trouvé comment faire si on voulait tester sans cela. Cette réponse sur StackOverflow `<https://stackoverflow.com/questions/62192486/how-to-close-tcp-connection-in-python-locust-requests-session>` ne marche pas pour moi, les requêtes sont bien faites, mais pas comptées dans les statistiques.

Mais l'intérêt de Locust est de lancer un serveur Web qui permet d'obtenir de jolis graphes :

On peut aussi utiliser cette interface Web pour piloter les tests.

En conclusion ? Je ne crois pas qu'il y ait un de ces logiciels qui fasse tout ce que je voulais et comme je voulais. Donc, j'en garde plusieurs.

`https://www.bortzmeyer.org/benchmark-http.html`