

AIT - Laboratoire 3 - Load Balancing

Authors: Olivier Koffi, Nathanaël Mizutani

Introduction

In this lab, we will use HAProxy in different cases.

At first, we will test the proxy with its default configuration, that means round-robin mode without sticky-sessions

Then, we will configure the proxy to manage sticky-sessions.

After that, we will experiment the `DRAIN` and `MAINT` modes.

Finally, we will configure and compare different balancing modes.

Task 1: Install the tools

1. Explain how the load balancer behaves when you open and refresh the URL <http://192.168.42.42> in your browser. Add screenshots to complement your explanations. We expect that you take a deeper a look at session management.

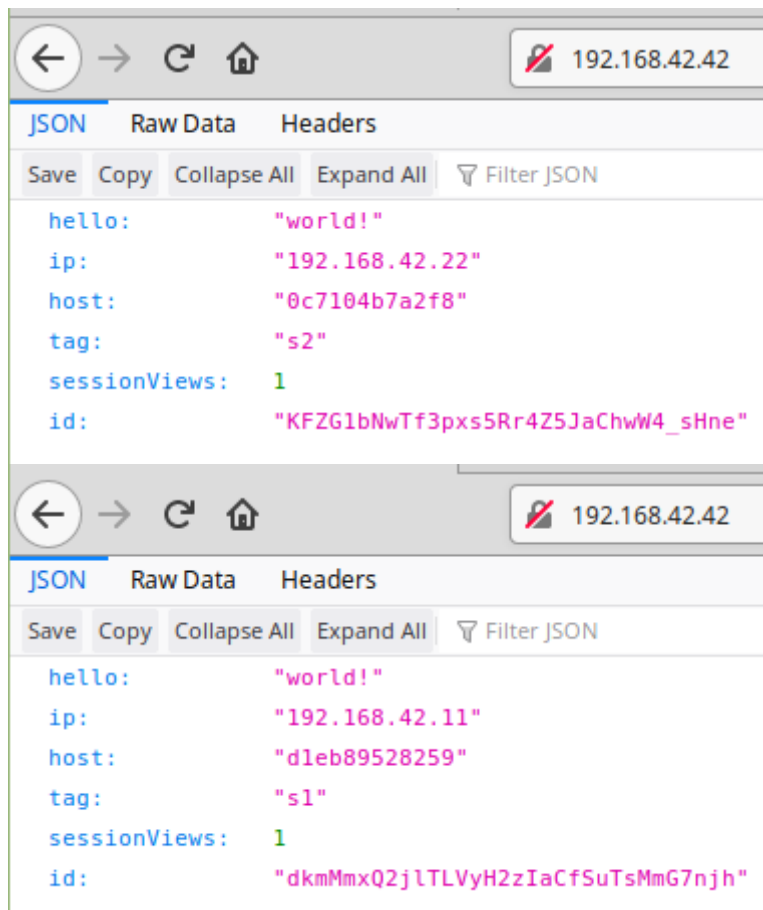
Answer

When we refresh the browser on the URL <http://192.168.42.42> we can see that the request is sent alternatively on each web server. HAProxy seems to work with a round robin configuration by default.

We also can see that the `NODESESSID` is refreshed for each request. This is because each server receives the token from the other one and doesn't recognize it. So it sets a new cookie containing a new `NODESESSID`. The same thing happens on the second server.

In fact, we have a setup that demands sticky sessions but HAProxy is configured in a stateless mode by default.

You can see below the results of two successive requests:



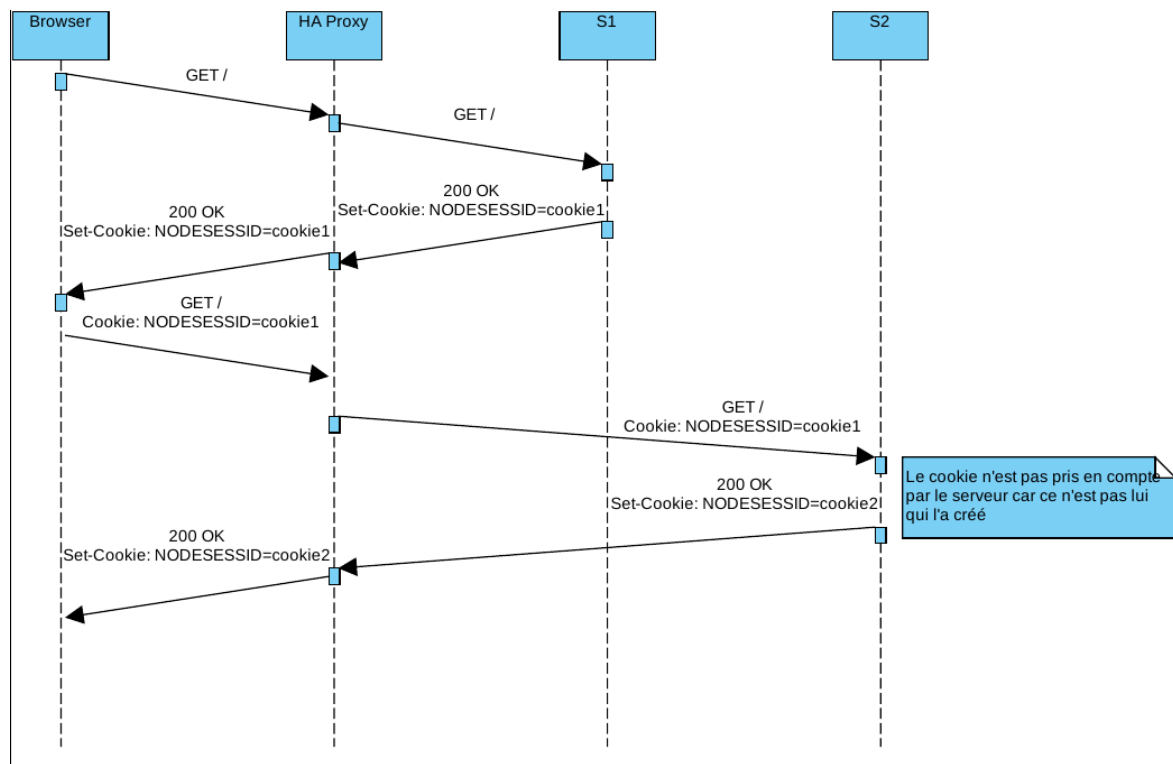
2. Explain what should be the correct behavior of the load balancer for session management.

Answer

HAProxy should be configured to be statefull. That means it should forward existing sessions on the correct server. This is what we call sticky sessions.

3. Provide a sequence diagram to explain what is happening when one requests the URL for the first time and then refreshes the page. We want to see what is happening with the cookie. We want to see the sequence of messages exchanged (1) between the browser and HAProxy and (2) between HAProxy and the nodes S1 and S2.

Answer



4. Provide a screenshot of the summary report from JMeter.

Answer

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
GET /	1000	1	1	8	0.64	0.00%	231.4/sec	112.00	49.15	495.6
S2 reached	500	0	0	1	0.36	0.00%	118.7/sec	0.00	0.00	0
S1 reached	500	0	0	1	0.37	0.00%	119.0/sec	0.00	0.00	0
TOTAL	2000	1	0	8	1.02	0.00%	462.7/sec	111.98	49.14	247.8

5. Run the following command:

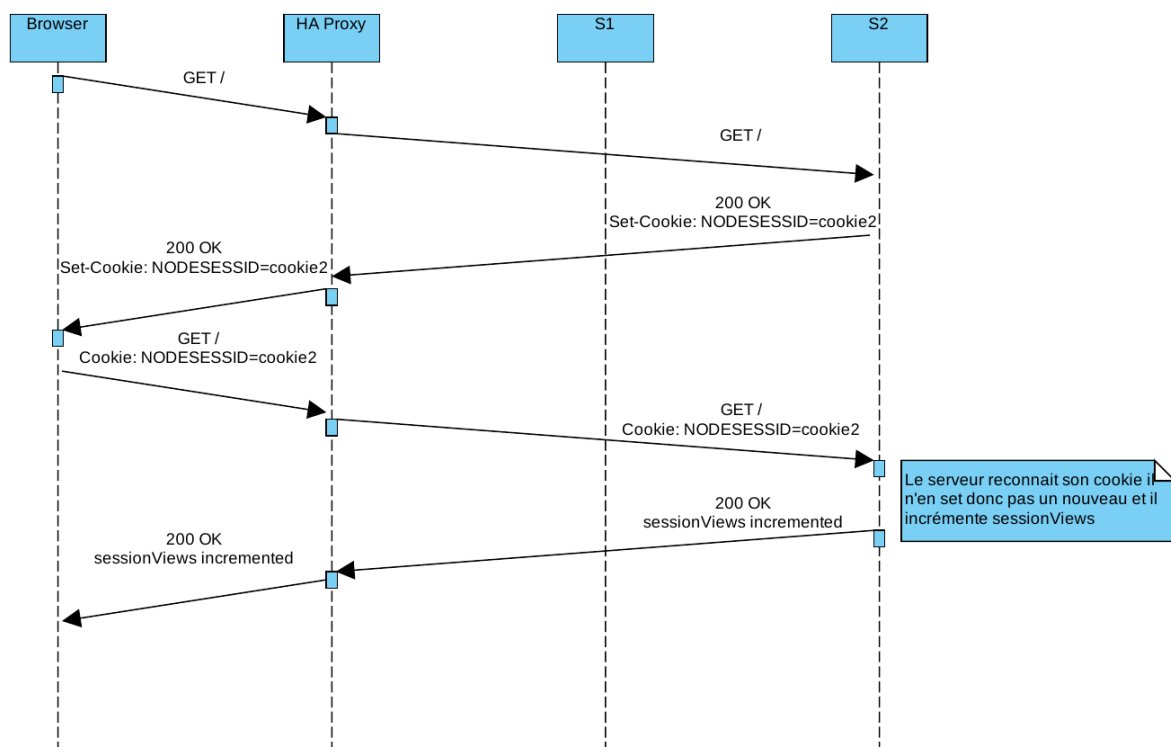
```
$ docker stop s1
```

Clear the results in JMeter and re-run the test plan. Explain what is happening when only one node remains active. Provide another sequence diagram using the same model as the previous one.

Answer

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
GET /	1000	4	1	20	2.13	0.00%	103.5/sec	37.60	21.72	372.0
S2 reached	1000	0	0	9	0.75	0.00%	109.4/sec	0.00	0.00	0
TOTAL	2000	2	0	20	2.45	0.00%	207.0/sec	37.59	21.72	186.0

Now HAProxy forwards all traffic on S2 because it's the only server up. S2 now recognizes the cookie `NODESESSID`, because it's the one it has set. So there is no need for a new cookie. The session is established, so the `sessionViews` is incremented at each new request. But it's not due to the fact that HAProxy is configured to manage sticky sessions, it's rather because S2 is the only server up, so all the traffic is always forwarded there.



Task 2: Sticky sessions

1. There is different way to implement the sticky session. One possibility is to use the `SERVERID` provided by HAProxy. Another way is to use the `NODESESSID` provided by the application. Briefly explain the difference between both approaches (provide a sequence diagram with cookies to show the difference).

Answer

With `SERVERID`, HAProxy is configured to add a cookie (named `SERVERID`) in the server response to identify the server. When the client send another request, the proxy reads the `SERVERID` cookie and forwards the request to the correct server. HAProxy inserts a `SERVERID` cookie in the server's response only if it doesn't exist in the client's request, if it already exists it just forwards requests and responses.

With `NODESESSID`, it's almost the same operation. The application on the server sets a `NODESESSID` and then the proxy concatenate the server id in the cookie. When the client sends another request, the proxy extracts the cookie from the request and reads it in order to redirect the request to the correct server.

Choose one of the both stickiness approach for the next tasks.

We'll use the `SERVERID` mechanism for the following manipulations.

2. Provide the modified `haproxy.cfg` file with a short explanation of the modifications you did to enable sticky session management.

Answer

```

backend nodes
# Define the protocol accepted
# http://cbonte.github.io/haproxy-dconv/configuration-1.5.html#4-mode
mode http

# Define the way the backend nodes are checked to know if they are alive or down
# http://cbonte.github.io/haproxy-dconv/configuration-1.5.html#4-option%20httpchk
option httpchk HEAD /

# Define the balancing policy
# http://cbonte.github.io/haproxy-dconv/configuration-1.5.html#balance
balance roundrobin

# Modifications
cookie SERVERID insert indirect

# Automatically add the X-Forwarded-For header
# http://cbonte.github.io/haproxy-dconv/configuration-1.5.html#4-option%20forwardfor
# https://en.wikipedia.org/wiki/X-Forwarded-For
option forwardfor

# With this config, we add the header X-Forwarded-Port
# http://cbonte.github.io/haproxy-dconv/configuration-1.5.html#4-http-request
http-request set-header X-Forwarded-Port %[dst_port]

# Define the list of nodes to be in the balancing mechanism
# http://cbonte.github.io/haproxy-dconv/configuration-1.5.html#4-server
server s1 ${WEBAPP_1_IP}:3000 check cookie s1
server s2 ${WEBAPP_2_IP}:3000 check cookie s2

```

The line starting with `cookie` indicates to HAProxy that it must insert a `Set-Cookie` named `SERVERID` into the response's header when a client sends a request for the first time (or rather if the client has not a `SERVERID` cookie already set).

We also added `cookie s1` for server s1 and `cookie s2` for server s2 to set the content of the `SERVERID` cookie that will be stored in the client's browser.

3. Explain what is the behavior when you open and refresh the URL <http://192.168.42.42> in your browser. Add screenshots to complement your explanations. We expect that you take a deeper a look at session management.

Answer

The screenshot shows a web browser at `localhost` displaying a JSON response: `{ "hello": "world!", "ip": "192.168.42.11", "host": "02a592e7b945", "tag": "s1", "sessionViews": 10, "id": "NL7J9_gVA3FAhLeYrFFuT0AqLSWLJ5pf" }`. Below the response, the Chrome DevTools network tab shows a request to `http://192.168.42.42:80`. The request details are visible, showing the following headers:

```

GET / HTTP/1.1
Host: 192.168.42.42
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:70.0) Gecko/20100101 Firefox/70.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Cookie: NODESESSID=s%3ApIY2wPggtU1a35DpJmPCeY_aXKx7ddkt.7Ys88oiffEsZzRS0thEHsFNMV09m6M0%2Bcx5B0N%2F9Ccg; SERVERID=s1
Upgrade-Insecure-Requests: 1
If-None-Match: W/"81-TTsD23bWSfBY1A9ILCsK806E4KQ"

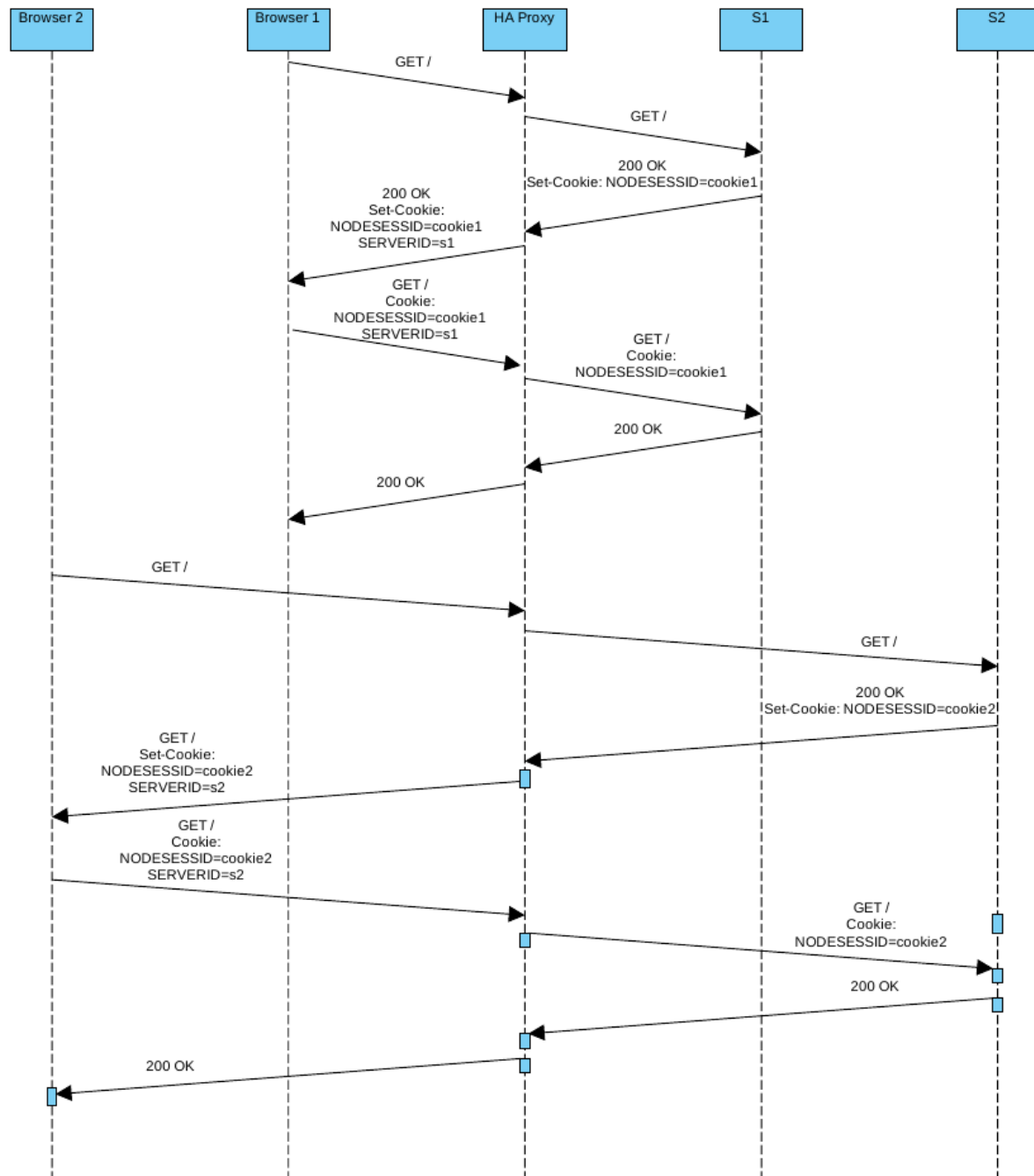
```

Now, when the page is refreshed several times, the `NODESESSID` is still the same and `sessionViews` is incremented. That shows that HAProxy is now configured to manage sticky sessions.

In the above image, we can see that the `SERVERID` cookie is set to s1. That's because HAProxy set it up in the first response and now as long as the client's browser keeps the `SERVERID` cookie, HAProxy will receive it in each request and forwards messages to s1.

4. Provide a sequence diagram to explain what is happening when one requests the URL for the first time and then refreshes the page. We want to see what is happening with the cookie. We want to see the sequence of messages exchanged (1) between the browser and HAProxy and (2) between HAProxy and the nodes S1 and S2. We also want to see what is happening when a second browser is used.

Answer



5+6. Provide a screenshot of JMeter's summary report. Is there a difference with this run and the run of Task 1? Give a short explanation of what the load balancer is doing.

- Clear the results in JMeter.
- Now, update the JMeter script. Go in the HTTP Cookie Manager and ~~uncheck~~ verify that the box Clear cookies each iteration? is unchecked.
- Go in Thread Group and update the Number of threads. Set the value to 2.

Summary Report

Name: Summary Report

Comments:

Write results to file / Read from file

Filename

Browse...

Log/Display Only:

Errors

Successes

Configure

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
GET /	2000	1	1	7	0.62	0.00%	652.3/sec	237.01	147.08	372.1
S2 reached	1000	0	0	1	0.28	0.00%	398.1/sec	0.00	0.00	.0
S1 reached	1000	0	0	2	0.32	0.00%	390.9/sec	0.00	0.00	.0
TOTAL	4000	0	0	7	0.81	0.00%	1304.2/sec	236.93	147.03	186.0

Answer

This question could be tricky because it looks like there are no differences between the test above and the test from the task 1 but in fact, there is more than meets the eye.

In the task 1, there is only one thread (user) sending 1000 requests and HAProxy simply dispatches each request in round robin mode. That means each successive request goes respectively on s1, then s2. But there is no sticky sessions. A new `Set-Cookie: NODESESSID` is added in the header of the response of each request !

In this test, there are now 2 threads (users) sending 1000 requests each but this time sticky sessions is configured. That means user1 sent 1000 requests on s1 and user2 sent 1000 requests on s2.

s1 and s2 added a `Set-Cookie` in the header of the response only at the first request from user1 and user2 respectively. The `sessionViews` has been incremented to 1000 for each user.

Task 3: Drain mode

1+2. Take a screenshot of the Step 5 and tell us which node is answering.

Answer

[illegible]

The page has been refreshed 30 times. We can see that it is the node s2 who is answering.

2. Based on your previous answer, set the node in DRAIN mode. Take a screenshot of the HAProxy state page.

Answer

```
> oki@oki-HP-ProBook-450-G5:~$ socat - tcp:192.168.42.42:9999
prompt

> set timeout cli 1d

> set server nodes/s2 state drain
```

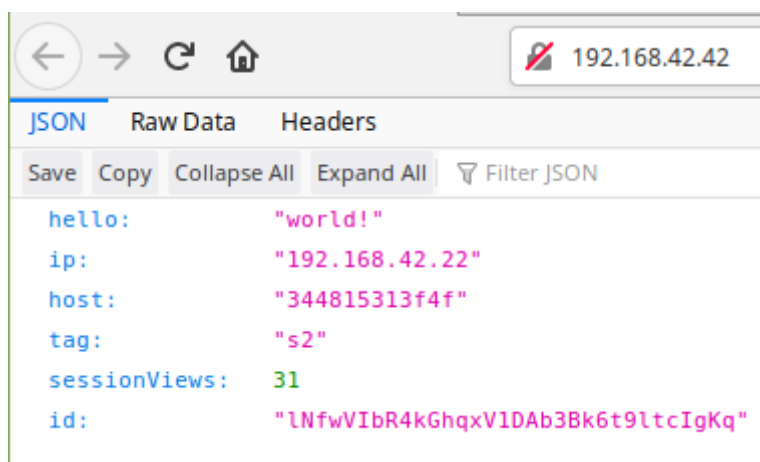
192.168.42.193

<

We can see that the s2 node line changed to blue (active or backup SOFT STOPPED for maintenance)

3. Refresh your browser and explain what is happening. Tell us if you stay on the same node or not. If yes, why? If no, why?

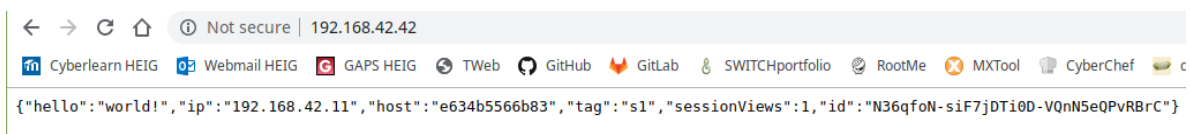
Answer



When we refresh the browser, we stay on the same node. It is as expected because as explained during the course, **DRAIN mode** lets current sessions continue to make requests to the node that is in **DRAIN mode** and will redirect all other traffic to the other nodes.

4. Open another browser and open `http://192.168.42.42`. What is happening?

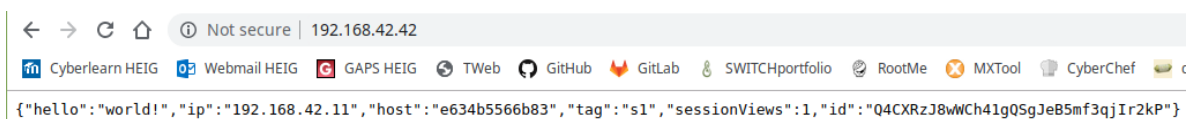
Answer



As expected, all other traffic is redirected to the other node s1.

5. Clear the cookies on the new browser and repeat these two steps multiple times. What is happening? Are you reaching the node in DRAIN mode?

Answer



The node in **DRAIN mode** (s2) is never reached because only the sessions that already existed when it was set to **DRAIN mode** are redirected to it. Again, all other traffic will be redirected on s1.

6. Reset the node in READY mode. Repeat the three previous steps and explain what is happening. Provide a screenshot of HAProxy's stats page.

Answer

```
> oki@oki-HP-ProBook-450-G5:~$ socat - tcp:192.168.42.42:9999
prompt

> set timeout cli 1d

> set server nodes/s2 state drain

> set server nodes/s2 state ready
```

[illegible]

Now, we can see that s2 is reached one time in two with the new browser when the cache is cleaned between each request. This is the normal round robin mode.

7. Finally, set the node in MAINT mode. Redo the three same steps and explain what is happening. Provide a screenshot of HAProxy's stats page.

Answer

```
> oki@oki-HP-ProBook-450-G5:~$ socat - tcp:192.168.42.42:9999
prompt

> set timeout cli 1d

> set server nodes/s2 state drain

> set server nodes/s2 state ready

> set server nodes/s2 state maint
```

The screenshot shows a web browser with the address bar displaying '192.168.42.42'. The developer tools are open, showing the 'JSON' tab. The JSON response is as follows:

```

{
  "hello": "world!",
  "ip": "192.168.42.11",
  "host": "e634b5566b83",
  "tag": "s1",
  "sessionViews": 2,
  "id": "bdGVtW3qn62d46IKPldJA6lwsUX0cxKM"
}

```

The browser's address bar also shows 'Not secure' and a list of bookmarks including Cyberlearn HEIG, Webmail HEIG, GAPS HEIG, TWeb, GitHub, GitLab, SWITCHportfolio, RootMe, MXTool, and CyberChef.

3. Set a delay of 2500 milliseconds on `s1`. Same than previous step.

```
oki@oki-HP-ProBook-450-G5:~/Documents/HEIG/S5/1_AIT/Labos/Labo3$ curl -H "Content-Type: application/json" -X POST -d '{"delay": 2500}' http://192.168.42.11:3000/delay
{"message": "New timeout of 2500ms configured."}oki@oki-HP-ProBook-450-G5:~/Documents/HEIG/S5/1_AIT/Labos/Labo3$
```

Now we can see that s1 is never reached. All traffic goes on s2.

4. In the two previous steps, are there any error? Why?

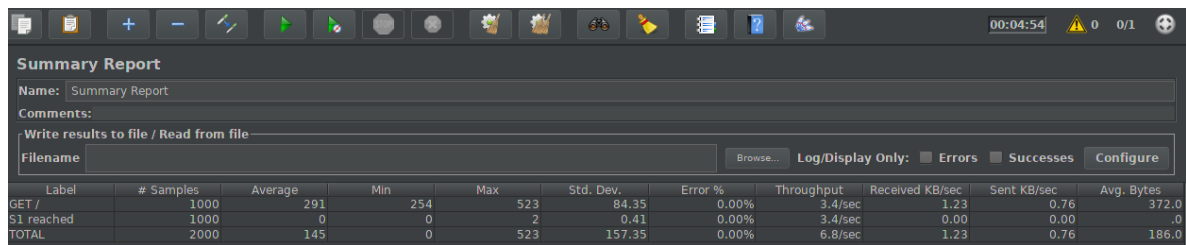
[illegible]

5. Update the HAProxy configuration to add a weight to your nodes. For that, add `weight [1-256]` where the value of `weight` is between the two values (inclusive). Set `s1` to 2 and `s2` to 1. Redo a run with 250ms delay.

```
# Define the list of nodes to be in the balancing mechanism
# http://cbonte.github.io/haproxy-dconv/configuration-1.5.html#4-server
server s1 ${WEBAPP_1_IP}:3000 cookie s1 weight 2 check
server s2 ${WEBAPP_2_IP}:3000 cookie s2 weight 1 check
```

6. Now, what happened when the cookies are cleared between each requests and the delay is set to 250ms ? We expect just one or two sentence to summarize your observations of the behavior with/without cookies.

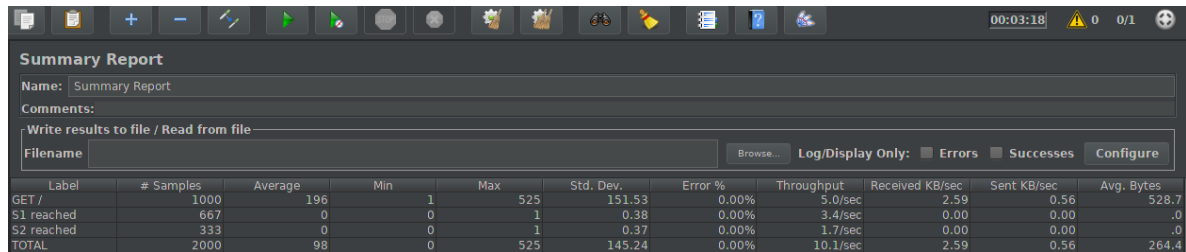
With cookie :



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
GET /	1000	291	254	523	84.35	0.00%	3.4/sec	1.23	0.76	372.0
S1 reached	1000	0	0	2	0.41	0.00%	3.4/sec	0.00	0.00	.0
TOTAL	2000	145	0	523	157.35	0.00%	6.8/sec	1.23	0.76	186.0

- We can see that there is no differences in terms of response time because HAProxy is configured to manage sticky sessions so all traffic is send to s1; because there is a session between the client and s1.

Without cookie :



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
GET /	1000	196	1	525	151.53	0.00%	5.0/sec	2.59	0.56	528.7
S1 reached	667	0	0	1	0.38	0.00%	3.4/sec	0.00	0.00	.0
S2 reached	333	0	0	1	0.37	0.00%	1.7/sec	0.00	0.00	.0
TOTAL	2000	98	0	525	145.24	0.00%	10.1/sec	2.59	0.56	264.4

- Now HAProxy behaves like a "normal" load balancer because there is a new session at each request.
S1 weights 2 and S2 weights 1, that means on three requests two will be sent to S1 and one to S2.
So the average response time is a bit better because S2 has "0" [ms] delay.
But a better configuration would be to set a heavier weight to S2 because it has a better response time than S1. It would also improve the average response time.

Task 5: Balancing strategies

1. Briefly explain the strategies you have chosen and why you have chosen them.

Answer

- **static-rr** :
Each server is used in turns, according to their weights.
This algorithm is similar to round robin except that it is static, which means that changing a server weight on the fly will have no effect. It also uses slightly less CPU to run (around -1%).
We chose this one to compare performances with an algorithm that has better performances with long sessions.
- **leastconn** :
This system configures HAProxy to choose the server with the lowest number of active connections. If several servers have the same load, then Round-robin between them is applied.
This algorithm is recommended when very long sessions are expected.
We chose this one to compare the performances with an algorithm that has better performances with short sessions.

2. Provide evidences that you have played with the two strategies (configuration done, screenshots, ...)

Answer

static-rr:

Configuration

```
backend nodes
# Define the protocol accepted
# http://cbonte.github.io/haproxy-dconv/configuration-1.5.html#4-mode
mode http

# Define the way the backend nodes are checked to know if they are alive or down
# http://cbonte.github.io/haproxy-dconv/configuration-1.5.html#4-option%20httpchk
option httpchk HEAD /

# Define the balancing policy
# http://cbonte.github.io/haproxy-dconv/configuration-1.5.html#balance
balance static-rr

# Modifications
cookie SERVERID insert indirect

# Automatically add the X-Forwarded-For header
# http://cbonte.github.io/haproxy-dconv/configuration-1.5.html#4-option%20forwardfor
# https://en.wikipedia.org/wiki/X-Forwarded-For
option forwardfor

# With this config, we add the header X-Forwarded-Port
# http://cbonte.github.io/haproxy-dconv/configuration-1.5.html#4-http-request
http-request set-header X-Forwarded-Port %[dst_port]

# Define the list of nodes to be in the balancing mechanism
# http://cbonte.github.io/haproxy-dconv/configuration-1.5.html#4-server
server s1 ${WEBAPP_1_IP}:3000 cookie s1 weight 1 check
server s2 ${WEBAPP_2_IP}:3000 cookie s2 weight 1 check
```

Without sticky-session

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
GET /	100000	1	0	18	1.26	0.00%	356.4/sec	183.97	39.67	528.6
S1 reached	50000	0	0	8	0.26	0.00%	178.2/sec	0.00	0.00	.0
S2 reached	50000	0	0	9	0.26	0.00%	178.2/sec	0.00	0.00	.0
TOTAL	200000	0	0	18	1.25	0.00%	712.7/sec	183.97	39.67	264.3

With sticky-session

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
GET /	100000	1	0	18	1.34	0.00%	374.9/sec	136.88	84.20	373.9
S1 reached	100000	0	0	9	0.26	0.00%	374.9/sec	0.00	0.00	.0
TOTAL	200000	0	0	18	1.31	0.00%	749.7/sec	136.88	84.20	186.9

leastconn:

Configuration

```
backend nodes
# Define the protocol accepted
# http://cbonte.github.io/haproxy-dconv/configuration-1.5.html#4-mode
mode http

# Define the way the backend nodes are checked to know if they are alive or down
# http://cbonte.github.io/haproxy-dconv/configuration-1.5.html#4-option%20httpchk
option httpchk HEAD /

# Define the balancing policy
# http://cbonte.github.io/haproxy-dconv/configuration-1.5.html#balance
balance leastconn

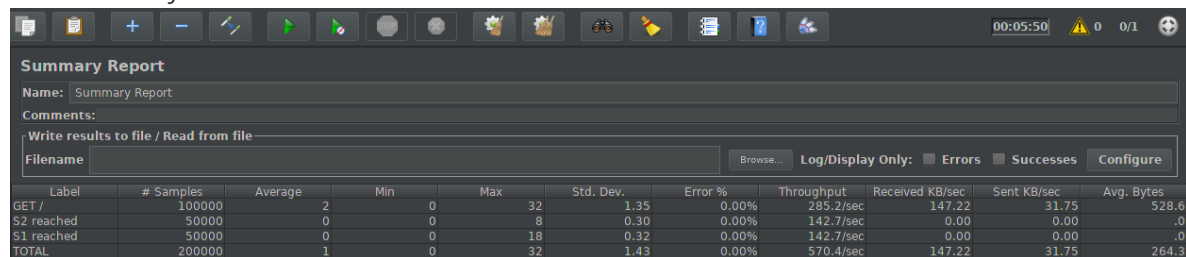
# Modifications
cookie SERVERID insert indirect

# Automatically add the X-Forwarded-For header
# http://cbonte.github.io/haproxy-dconv/configuration-1.5.html#4-option%20forwardfor
# https://en.wikipedia.org/wiki/X-Forwarded-For
option forwardfor

# With this config, we add the header X-Forwarded-Port
# http://cbonte.github.io/haproxy-dconv/configuration-1.5.html#4-http-request
http-request set-header X-Forwarded-Port %[dst_port]

# Define the list of nodes to be in the balancing mechanism
# http://cbonte.github.io/haproxy-dconv/configuration-1.5.html#4-server
server s1 ${WEBAPP_1_IP}:3000 cookie s1 weight 1 check
server s2 ${WEBAPP_2_IP}:3000 cookie s2 weight 1 check
```

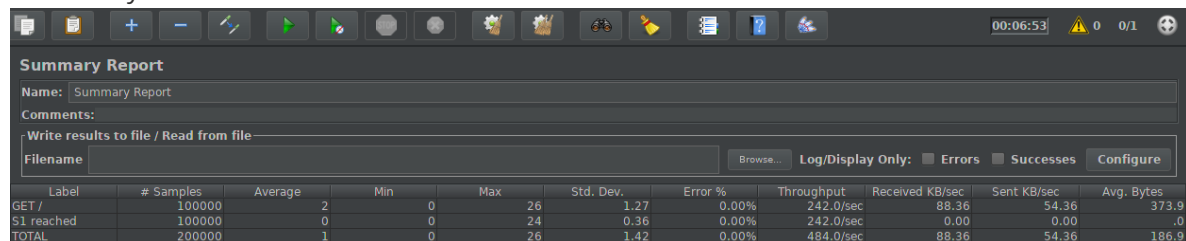
Without sticky-session



The screenshot shows the HAProxy statistics page with a 'Summary Report' section. The report includes a table with the following data:

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
GET /	100000	2	0	32	1.35	0.00%	285.2/sec	147.22	31.75	528.6
S2 reached	50000	0	0	8	0.30	0.00%	142.7/sec	0.00	0.00	.0
S1 reached	50000	0	0	18	0.32	0.00%	142.7/sec	0.00	0.00	.0
TOTAL	200000	1	0	32	1.43	0.00%	570.4/sec	147.22	31.75	264.3

With sticky-session



The screenshot shows the HAProxy statistics page with a 'Summary Report' section. The report includes a table with the following data:

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
GET /	100000	2	0	26	1.27	0.00%	242.0/sec	88.36	54.36	373.9
S1 reached	100000	0	0	24	0.36	0.00%	242.0/sec	0.00	0.00	.0
TOTAL	200000	1	0	26	1.42	0.00%	484.0/sec	88.36	54.36	186.9

3. Compare the both strategies and conclude which is the best for this lab (not necessary the best at all).

Answer

For the tests, we put the total number of requests at 100'000 to better see performances. As we expected, the `static-rr` mode has a better response time average. This is because the round-robin algorithm is really easy and HAProxy doesn't need to do any complex operation before forwarding the request and the response. Moreover because we have only short session time for our application, `rr-static` is a better solution than `leastconn`.

Conclusion

This lab allowed us to get familiar with the HAProxy environment and its features. We also found out some other balancing modes (e.g. first, source, uri) that can be used for specific infrastructure types or requirements.

HAProxy configuration file is easy to administrate and the community edition is free. This tool could be used as reverse proxy for a small or medium infrastructure.