AIT - Lab Report - Load balancing

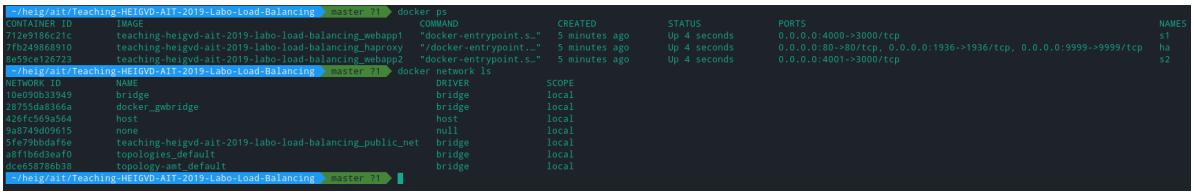
by Bonzon Tiffany, Scherer Laurent & Thoeny Laurent

Introduction

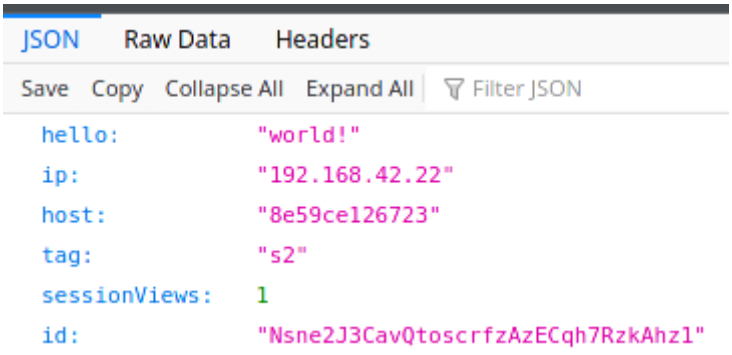
In this lab, we will see different configurations for HAProxy. First we will examine a classic round-robin with no sticky sessions, then we will add sticky sessions so that a client with an open session will always make its requests on the same server. Later on, we will experiment with the DRAIN and MAINT modes, with delay on our servers and finally we will try two different configuration modes for the proxy: first and leastconn

Task 1: Install the tools

After running `docker-compose up --build` we can verify that our containers are started.



We can then verify that we're able to access the load balancer address properly



Then we start the JMeter script and verify that the output is split accordingly

Label	# Samples	Average	Min	Max
GET /	1000	10	1	51
S2 reached	500	0	0	1
S1 reached	500	0	0	1
TOTAL	2000	5	0	51

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.

Using the tag from the response, we can see that the server that respond change every time we make a request. A quick look at load balancing strategies tell us that we're facing a *Round-Robin* load balancer. The screenshots above show that exactly half of the request sent by JMeter were handled by both servers, with that proof and a quick look at the config in `haproxy.cfg` we have confirmed our hypothesis.

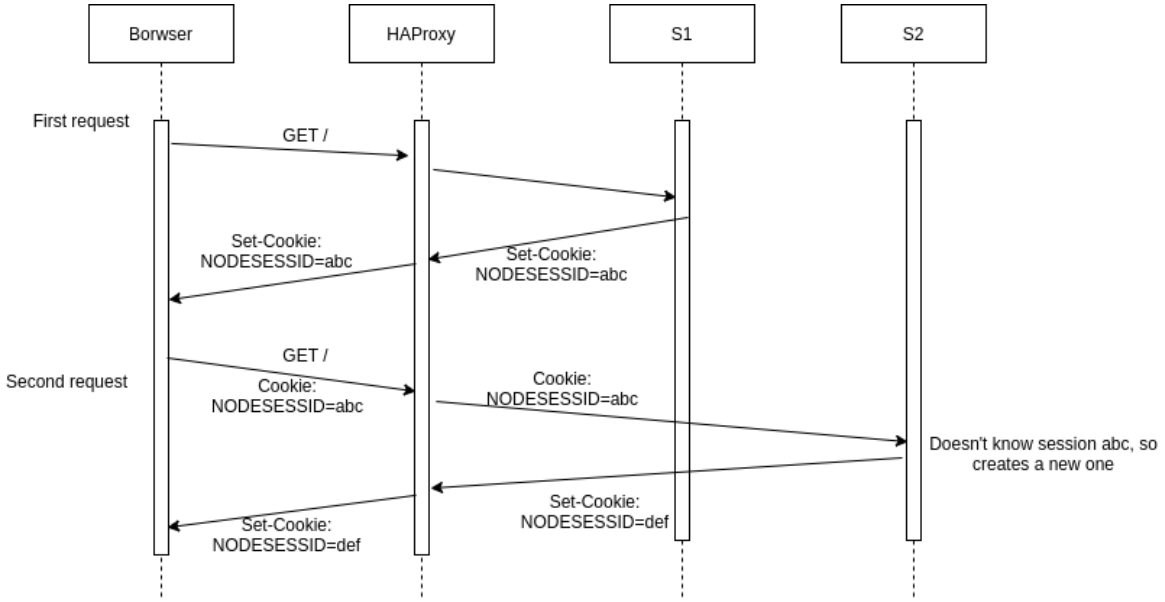
```
# Define the balancing policy
# http://cbonte.github.io/haproxy-dconv/2.2/configuration.html#balance
balance roundrobin
```

Looking at the *NODESESSID* cookie we received, we can see that a new session is established at every connection, this means the session management is stateless, there is no guarantee that the server answering your second query will be the same that answered the first. It can become a problem if we implement stateful services in our servers.

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

The load balancer should implement *sticky sessions*, a mechanism that allows the load balancer to know when a connection was established by the client before, allowing the client to be served by the same server again. For example, Traefik uses cookies to allow the sticky sessions, the cookie contains the IP address of the server concerned by a connection.

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.



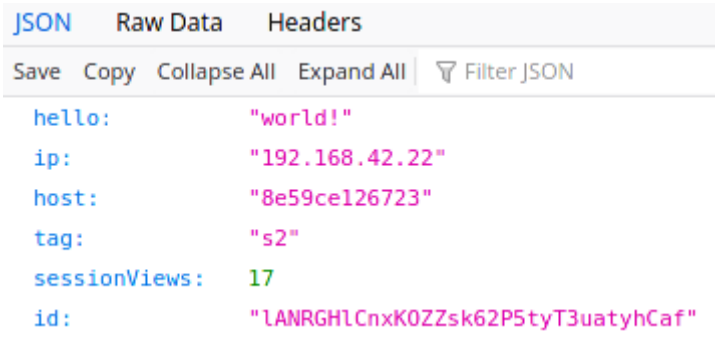
Currently, the proxy doesn't take the cookie into consideration, it simply redirects every request on either S1 or S2 according to its round-robin policy. As the browser sends its session ID each time and the request is sent to the other server, the session is recreated by each server each time.

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

Label	# Samples	Average	Min	Max
GET /	1000	10	1	51
S2 reached	500	0	0	1
S1 reached	500	0	0	1
TOTAL	2000	5	0	51

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.

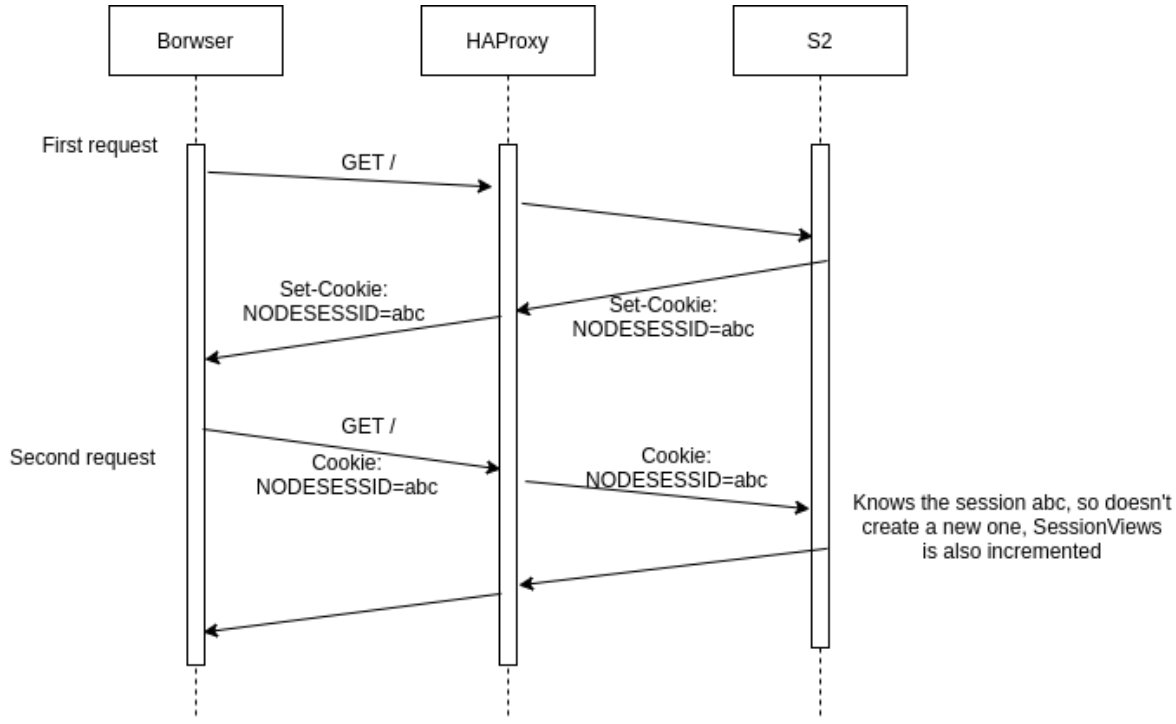
This time, when we refresh, we're always on the second server, this allows us to see that the value of `sessionViews` is now incremented, the value of the cookie doesn't change anymore, we've preserved a session with the same server.



You can see the report of the results below, the result is obvious, every request is now directed to the same server.

Label	# Samples	Average	Min	Max
GET /	1000	12	1	54
S2 reached	1000	0	0	1
TOTAL	2000	6	0	54

And below again, you can see the sequence diagram updated to show the situation.

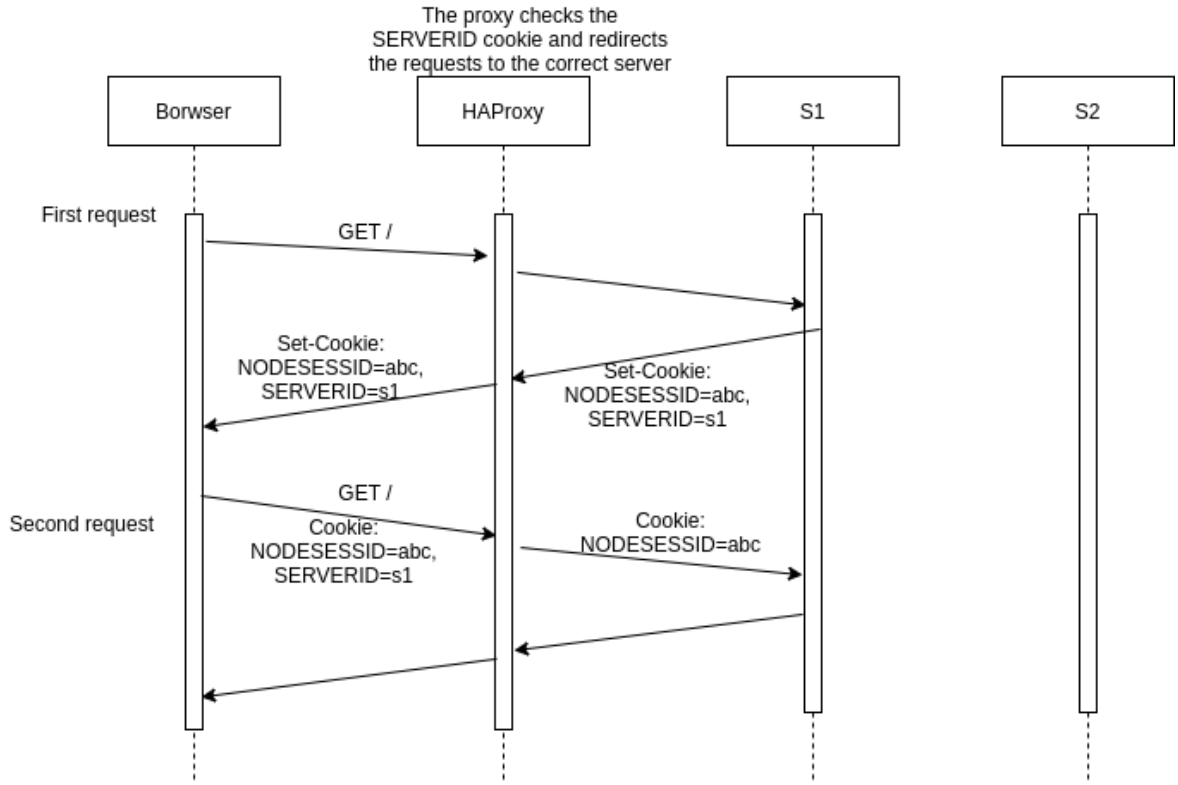


Task 2: Sticky sessions

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).

The difference is that an application level session ID (`NODESESSID` in this case) might not be understood by the load balancer, also there could be multiple services on a server that don't share the same ID but should be considered as a whole.

We do prefer using the ID at the server level and will go on with an implementation of the `SERVERID` load balancing. Note that the `SERVERID` cookie is only used and set by the proxy.



This time, compared to the first diagram of task#1, the proxy sets the `SERVERID` cookie, and redirects the requests accordingly. A second browser would have its requests redirected to S2 (as seen in the diagram below).

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

We declare a cookie based on this documentation and we add the cookie keyword to the server declarations based on this documentation.

```
backend nodes
    ...
    # Add the cookie, based on the doc quoted on the report
    cookie SERVERID insert indirect nocache

    # Define the list of nodes to be in the balancing mechanism
    # http://cbonte.github.io/haproxy-dconv/2.2/configuration.html#4-server
    server s1 ${WEBAPP_1_IP}:3000 check cookie s1
    server s2 ${WEBAPP_2_IP}:3000 check cookie s2
```

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.

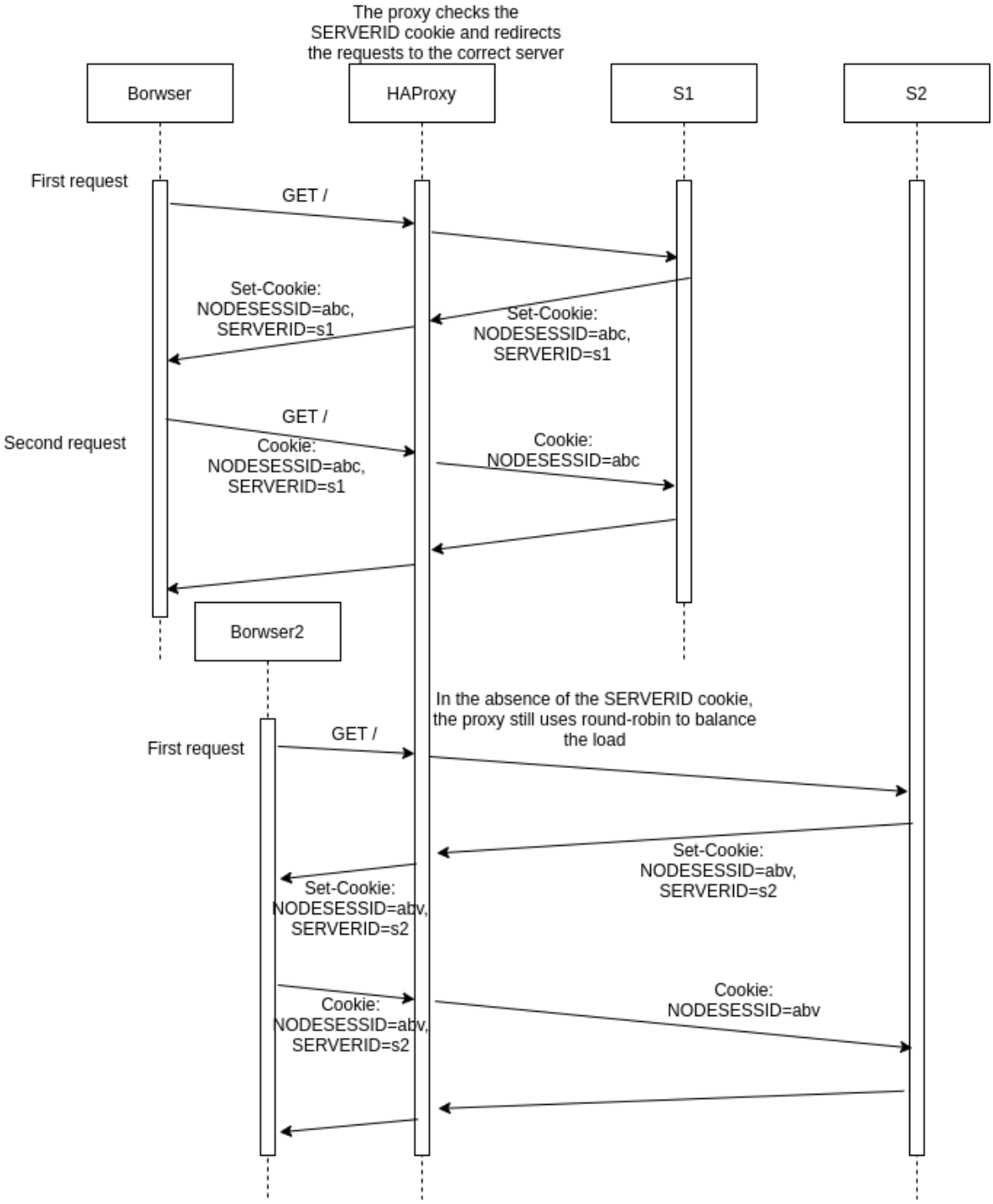
After restarting both servers and the proxy, we can verify the config from the browser, this time if we refresh the page many time we will see that our connexion has been preserved with the same server and that the `sessionViews` value is incremented once again.

Name	Value	Domain	Path
NODESESSID	s%3AzQke9grNAfwnA7JH7br3cEDAJNdSsIho.UyDMgBJXWYNSO6v46QWHiT%2B4zrGYt1%2FM5ofKIYYkLGc	192.168.42.42	/
SERVERID	s2	192.168.42.42	/

What was happening before is that the application server verified the `NODESESSID` value and assigned a new one if the existing value wasn't recognized. Since we went from server A to server B every time, the value of the `NODESESSID` was changed every time. After that, when we did shutdown a server and connected to the same multiple time in a row, we saw that the session ID allowed Node to increment the variable.

What we implemented now is the second cookie, `SERVERID`, this cookie is created by our proxy when a request is forwarded to a server and contains the name of said server. When we make another request, the proxy checks for an existing cookie, if the cookie exists and contains a valid value (the name of a server that is currently online) then the request will be forwarded accordingly. Otherwise a new cookie is created.

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.



5. Provide a screenshot of JMeter's summary report. Is there a difference with this run and the run of Task 1?

There is a clear difference, now every request was sent to the same endpoint, it's obviously thanks to the sticky session we implemented.

Label	# Samples	Average	Min	Max
GET /	1000	13	1	57
S1 reached	1000	0	0	2
TOTAL	2000	6	0	57

6. Provide a screenshot of JMeter's summary report. 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 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.

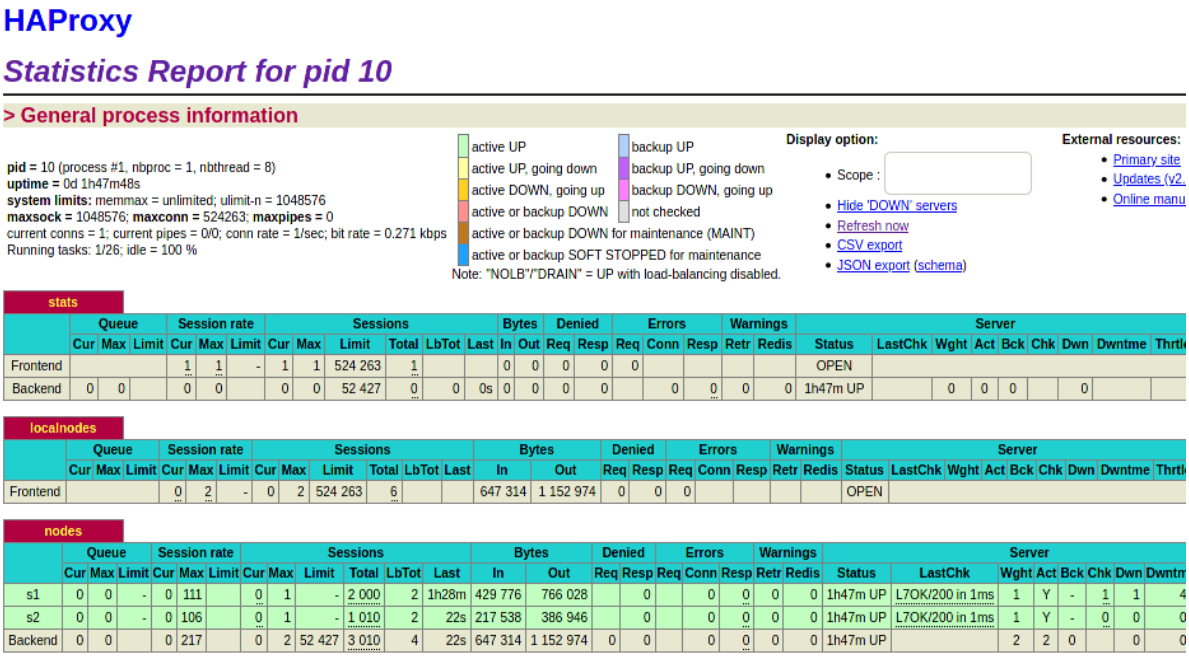
We modified the number of threads to be 2.

The first thread was directed toward Server A, the second thread was directed toward Server B thanks to the round robin load balancing, then every request made by each of the threads was sent to the same server as the first they made, thanks to the session stickiness.

Label	# Samples	Average	Min	Max
GET /	2000	10	1	70
S2 reached	1000	0	0	1
S1 reached	1000	0	0	1
TOTAL	4000	5	0	70

Task 3: Drain mode

We did verify our access to the stats panel by going to `http://192.168.42.42:1936/`



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

The screenshot is available above, we can see that the server `s2` answered our latest query.

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

We did set the corresponding server in drain mode, you can see the commands as well as the state from the proxy panel in the screenshots below.

```
~/heig/ait/Teaching-HEIGVD-AIT-2019-Labo-Load-Balancing master !1 ?1 socat - tcp:192.168.42.42:9999
prompt

> set timeout cli 1d

> set server nodes/s2 state drain

>
```

HAProxy

Statistics Report for pid 10

> General process information

pid = 10 (process #1, nbproc = 1, nbthread = 8)
uptime = 0d 2h09m07s
system limits: memmax = unlimited; ulimit-n = 1048576
maxsock = 1048576; **maxconn** = 524263; **maxpipes** = 0
current conns = 1; current pipes = 0/0; conn rate = 1/sec; bit rate = 0.271 kbps
Running tasks: 1/28; idle = 100 %

active UP
active UP, going down
active DOWN, going up
active or backup DOWN
active or backup DOWN for maintenance (MAINT)
active or backup SOFT STOPPED for maintenance
backup UP
backup UP, going down
backup DOWN, going up
not checked
Note: "NOLB"/"DRAIN" = UP with load-balancing disabled.

Display option:

- Scope :
- Hide 'DOWN' servers
- Refresh now
- CSV export
- JSON export (schema)

Exter

stats

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server						
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn
Frontend				1	1	-	1	1	524 263	2			788	44 314	0	0	0					OPEN						
Backend	0	0		0	0		0	0	52 427	0	0	0s	788	44 314	0	0		0	0	0	0	2h9m UP		0	0	0		0

localnodes

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server						
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn
Frontend				0	2	-	0	2	524 263	7			647 724	1 153 356	0	0	0					OPEN						

nodes

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server					
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	C
s1	0	0	-	0	111		0	1	-	2 000	2	1h49m	429 776	766 028		0		0	0	0	0	2h9m UP	L7OK/200 in 2ms	1	Y	-	
s2	0	0	-	0	106		0	1	-	1 011	2	1m43s	217 948	387 328		0		0	0	0	0	7s DRAIN	L7OK/200 in 5ms	1	Y	-	
Backend	0	0		0	217		0	2	52 427	3 011	4	1m43s	647 724	1 153 356	0	0		0	0	0	0	2h9m UP		1	1	0	

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?

Yes, we did stay on the same node, it was the expected behavior since we were already on an established communication with the node. The new traffic, however, will be directed towards the other servers.

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

We opened another browser and it connected to `s1` as expected. We did refresh the page a few times after clearing the cookie and also connected from private navigation, every test resulted the same way, a connection was established to `s1`. It means the new traffic can't reach `s2` from the load balancer, as expected.

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.

First we set the server back with `set server nodes/s2 state ready` then we proceeded to the same tests as before, this time every time we established a new connection without a cookie we were redirected toward a new server. Below is the screenshot from the stats panel.

HAProxy

Statistics Report for pid 10

> General process information

pid = 10 (process #1, nbproc = 1, nbthread = 8)
uptime = 0d 2h19m10s
system limits: memmax = unlimited; ulimit-n = 1048576
maxsock = 1048576; maxconn = 524263; maxpipes = 0
current conns = 1; current pipes = 0/0; conn rate = 1/sec; bit rate = 0.271 kbps
Running tasks: 1/28; idle = 100 %

active UP
active UP, going down
active DOWN, going up
active or backup DOWN
active or backup DOWN for maintenance (MAINT)
active or backup SOFT STOPPED for maintenance
backup UP
backup UP, going down
backup DOWN, going up
not checked
Note: "NOLB"/"DRAIN" = UP with load-balancing disabled.

Display option:
- Scope :
- Hide 'DOWN' servers
- Refresh now
- CSV export
- JSON export (schema)

External resources:
- Primary site
- Updates (v2.2
- Online manua

stats

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server									
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
Frontend				1	1	-	1	3	524 263	4			1 662	88 813	0	0	0					OPEN								
Backend	0	0		0	0		0	0	52 427	0	0	0s	1 662	88 813	0	0		0	0	0	0	2h19m UP		0	0	0		0		

localnodes

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server									
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
Frontend				0	2	-	0	3	524 263	12			663 053	1 169 942	0	0	2					OPEN								

nodes

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server								
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	T
s1	0	0	-	0	111		0	1	-	2 012	6	8s	435 742	772 037		0		0	0	0	0	2h19m UP	L7OK/200 in 3ms	1	Y	-	1	1	4s	
s2	0	0	-	0	106		0	1	-	1 030	5	13s	227 311	397 457		0		0	0	0	0	1m31s UP	L7OK/200 in 3ms	1	Y	-	0	0	8m39s	
Backend	0	0		0	217		0	2	52 427	3 042	11	8s	663 053	1 169 494	0	0		0	0	0	0	2h19m UP		2	2	0		0	0s	

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.

We set the server using the same command as above, then tried every step from the previous tests and we couldn't reach a connection with the server `s2`. Then we tried to refresh the page from our original browser (that contains a cookie referring to `s2`) and were connected to `s1` with a new cookie.

Below is a screenshot of the stats from the proxy panel.

HAProxy

Statistics Report for pid 10

> General process information

pid = 10 (process #1, nbproc = 1, nbthread = 8)
uptime = 0d 2h23m33s
system limits: memmax = unlimited; ulimit-n = 1048576
maxsock = 1048576; maxconn = 524263; maxpipes = 0
current conns = 1; current pipes = 0/0; conn rate = 1/sec; bit rate = 0.271 kbps
Running tasks: 1/28; idle = 100 %

active UP
active UP, going down
active DOWN, going up
active or backup DOWN
active or backup DOWN for maintenance (MAINT)
active or backup SOFT STOPPED for maintenance
backup UP
backup UP, going down
backup DOWN, going up
not checked
Note: "NOLB"/"DRAIN" = UP with load-balancing disabled.

Display option:
- Scope :
- Hide 'DOWN' servers
- Refresh now
- CSV export
- JSON export (schema)

External resources:
- Primary site
- Updates (v2.
- Online manu

stats

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server									
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtl
Frontend				1	1	-	1	3	524 263	6			2 536	133 297	0	0	0					OPEN								
Backend	0	0		0	0		0	0	52 427	0	0	0s	2 536	133 297	0	0		0	0	0	0	2h23m UP		0	0	0		0		

localnodes

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server									
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtl
Frontend				0	2	-	0	3	524 263	15			673 535	1 181 547	0	0	3					OPEN								

nodes

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server								
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwn
s1	0	0	-	0	111		0	1	-	2 033	8	1m34s	446 224	783 431		0		0	0	0	0	2h23m UP	L7OK/200 in 1ms	1	Y	-	1	1	
s2	0	0	-	0	106		0	1	-	1 030	5	4m36s	227 311	397 457		0		0	0	0	0	2m6s MAINT		1	Y	-	0	1	10m
Backend	0	0		0	217		0	2	52 427	3 063	13	1m34s	673 535	1 180 888	0	0		0	0	0	0	2h23m UP		1	1	0		0	

At the end of this step we restored the server to its ready state.

Task 4: Round robin in degraded mode

Adding a delay

```
# Gets s1 IP Address
$ docker inspect -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' s1
192.168.42.11
# Adds a 300ms delay
$ curl -H "Content-Type: application/json" -X POST -d '{"delay": 300}'
http://192.168.42.11:3000/delay
{"message":"New timeout of 300ms configured."}
```

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received ...	Sent KB/sec	Avg. Bytes
GET /	1074	34	1	666	94.33	0.00%	39.6/sec	15.28	9.00	395.2
S1 reached	74	0	0	1	0.42	0.00%	2.8/sec	0.00	0.00	.0
S2 reached	1000	0	0	1	0.41	0.00%	79.0/sec	0.00	0.00	.0
TOTAL	2148	17	0	666	68.85	0.00%	79.2/sec	15.28	9.00	197.6

We can see the effect of the delay on S1 with the JMeter results shown above

1. Make sure a delay of 0 milliseconds is set on `s1`. Do a run to have a baseline to compare with in the next experiments.

Removing the previously added delay

```
# Removes previously added delays
$ curl -H "Content-Type: application/json" -X POST -d '{"delay": 0}'
http://192.168.42.11:3000/delay
{"message":"New timeout of 0ms configured."}
```

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received ...	Sent KB/sec	Avg. Bytes
GET /	2000	11	1	70	17.76	0.00%	139.5/sec	53.84	31.46	395.1
S1 reached	1000	0	0	1	0.39	0.00%	74.4/sec	0.00	0.00	.0
S2 reached	1000	0	0	1	0.38	0.00%	72.4/sec	0.00	0.00	.0
TOTAL	4000	6	0	70	13.85	0.00%	279.1/sec	53.83	31.46	197.5

2. Set a delay of 250 milliseconds on `s1`. Relaunch a run with the JMeter script and explain what is happening.

```
# Adds a 250ms delay
$ curl -H "Content-Type: application/json" -X POST -d '{"delay": 250}'
http://192.168.42.11:3000/delay
{"message":"New timeout of 250ms configured."}
```

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received ...	Sent KB/sec	Avg. Bytes
GET /	786	22	1	554	62.21	0.00%	80.6/sec	31.10	18.16	395.3
S1 reached	31	0	0	1	0.40	0.00%	3.4/sec	0.00	0.00	.0
S2 reached	755	0	0	1	0.35	0.00%	81.7/sec	0.00	0.00	.0
TOTAL	1572	11	0	554	45.36	0.00%	161.1/sec	31.12	18.17	197.8

We can see that, while S2 still processes around 72 requests per second, S1 handles around 3.4 requests per second, which is approximately 290ms per request.

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

```
# Adds a 2500ms delay
$ curl -H "Content-Type: application/json" -X POST -d '{"delay": 2500}'
http://192.168.42.11:3000/delay
{"message":"New timeout of 2500ms configured."}
```

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received ...	Sent KB/sec	Avg. Bytes
GET /	2000	13	1	55	18.52	0.00%	128.0/sec	49.37	28.85	395.1
S2 reached	2000	0	0	1	0.33	0.00%	128.0/sec	0.00	0.00	.0
TOTAL	4000	6	0	55	14.72	0.00%	255.9/sec	49.37	28.85	197.5

This time S1 doesn't answer any requests. This is due to the 2.5 seconds delay. We can see that HAProxy considers that S1 is down

stats

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server								
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
Frontend				1	1	-	1	1	524 263	1			0	0	0	0	0					OPEN								
Backend	0	0		0	0		0	0	52 427	0	0	0s	0	0	0	0		0	0	0	0	3h40m UP		0	0	0		0		

localnodes

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server								
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntme	Thrtle
Frontend				0	2	-	0	4	524 263	30			2 391 480	4 232 189	0	0	0					OPEN								

nodes

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings		Server						
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	D
s1	0	0	-	0	95		0	1	-	3 010	6	15m10s	653 351	1 153 215		0		0	0	0	0	12m52s DOWN	* L7TOUT in 2001ms	1	Y	-	4	
s2	0	0	-	0	194		0	2	-	8 038	11	3m10s	1 738 129	3 078 974		0		0	0	0	0	3h40m UP	L7OK/200 in 4ms	1	Y	-	0	
Backend	0	0		0	194		0	2	52 427	11 048	17	3m10s	2 391 480	4 232 189	0	0		0	0	0	0	3h40m UP		1	1	0			

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

There aren't any errors shown, as HAProxy detected S1 as DOWN and simply stopped sending requests to it

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 a 250ms delay.

```
# Define the list of nodes to be in the balancing mechanism
# http://cbonte.github.io/haproxy-dconv/2.2/configuration.html#4-server
server s1 ${WEBAPP_1_IP}:3000 check cookie s1 weight 2
server s2 ${WEBAPP_2_IP}:3000 check cookie s2 weight 1
```

After modifying the config and rebuilding the images, we can add the delay to both servers

```
$ curl -H "Content-Type: application/json" -X POST -d '{"delay": 250}'
http://192.168.42.11:3000/delay
{"message":"New timeout of 250ms configured."}
```

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received ...	Sent KB/sec	Avg. Bytes
GET /	1190	58	1	565	113.57	0.00%	20.2/sec	7.79	4.52	395.1
S2 reached	1000	0	0	1	0.39	0.00%	74.0/sec	0.00	0.00	.0
S1 reached	190	0	0	1	0.42	0.00%	3.3/sec	0.00	0.00	.0
TOTAL	2380	29	0	565	85.36	0.00%	40.4/sec	7.79	4.52	197.6

We can see that with cookies and a 250ms delay on S1, weights have virtually no effects

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

For that, we go into JMeter > HTTP Cookie Manager, and check the `Clear cookies each iteration?` checkbox

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received ...	Sent KB/sec	Avg. Bytes
GET /	123	376	1	823	279.37	0.00%	5.2/sec	2.92	0.58	575.5
S1 reached	82	0	0	1	0.43	0.00%	3.5/sec	0.00	0.00	.0
S2 reached	41	0	0	1	0.40	0.00%	1.8/sec	0.00	0.00	.0
TOTAL	246	188	0	823	272.78	0.00%	10.4/sec	2.92	0.58	287.7

We can see that S1 handles 2 times more requests that S2 even with the 250ms delay.

- Summary:
  - With cookies
    - As cookies are conserved, the load balancer always sends requests from the same thread to the same server. Here the weights have no effects.
  - Without cookies
    - As configured, S1 will receive 2 times more requests that S2. So S2 will be "slowed" due to its lower weight with the round-robin policy

Task 5: Balancing strategies

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

We did choose to compare the strategies `leastconn` and `first` with the `roundrobin` we have already used, our choice is mainly because the two strategies can be seen as "opposites" to each other and have realistic use cases. We do give further details on the strategies in our comparison below.

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

First, we did modify our configuration to implement the `first` policy.

```
# Set the strategy
balance first

# We keep the sticky session options
cookie SERVERID insert indirect nocache

# maxconn parameter was added for both server
server s1 ${WEBAPP_1_IP}:3000 check cookie s1 maxconn 9
server s2 ${WEBAPP_2_IP}:3000 check cookie s2 maxconn 9
```

Then we used JMeter, we set 10 threads to connect 100 times and asked it to clear cookie at each iteration.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %
GET /	1000	17	1	62	19.51	0.00%
S1 reached	954	0	0	8	0.53	0.00%
S2 reached	46	0	0	1	0.38	0.00%
TOTAL	2000	8	0	62	16.32	0.00%

As expected, most of the connections were send to the first server, but around 5% of them were sent to the second one, this happened every time our 10th thread sent a request when the first 9 were already connected.

We then made the same test but without asking JMeter to clear the cookie at each iteration.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %
GET /	1000	17	1	68	19.54	0.00%
S1 reached	900	0	0	9	0.53	0.00%
S2 reached	100	0	0	1	0.38	0.00%
TOTAL	2000	8	0	68	16.35	0.00%

This time, as we expected, we can see that the first 9 threads sent 100 request to our first server and stayed connected, therefore the 10th thread connected to our second server and sent its 100 requests to him.

HAProxy

Statistics Report for pid 10

> General process information

pid = 10 (process #1, nbproc = 1, nbthread = 8)
uptime = 0d 0h00m23s
system limits: memmax = unlimited; ulimit-n = 1048576
maxsock = 1048576; **maxconn =** 524263; **maxpipes =** 0
current conns = 1; current pipes = 0/0; conn rate = 1/sec; bit rate = 0.271 kbps
Running tasks: 1/26; idle = 100 %

stats

	Queue			Session rate			Sessions						
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In
Frontend				1	1	-	1	4	524 263	1			0
Backend	0	0		0	0		0	0	52 427	0	0	0s	0

localnodes

	Queue			Session rate			Sessions						
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In
Frontend				0	0	-	0	0	524 263	0			

nodes

	Queue			Session rate			Sessions						Bytes	
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out
s1	0	0	-	0	0		0	0	9	0	0	?	0	0
s2	0	0	-	0	0		0	0	9	0	0	?	0	0
Backend	0	0		0	0		0	0	52 427	0	0	?	0	0

On the screenshot above, you can see that our statistics page shows us the session limit was indeed fixed at 9 for both our nodes.

Now moving on to the second policy, `leastconn`.

```
# Set the strategy
balance leastconn

# We keep the sticky session options
cookie SERVERID insert indirect nocache

# maxconn parameter was added for both server
server s1 ${WEBAPP_1_IP}:3000 check cookie s1
server s2 ${WEBAPP_2_IP}:3000 check cookie s2
```

We will run the same JMeter tests as above, 10 users and 100 threads, first while clearing the cookies, then without it.

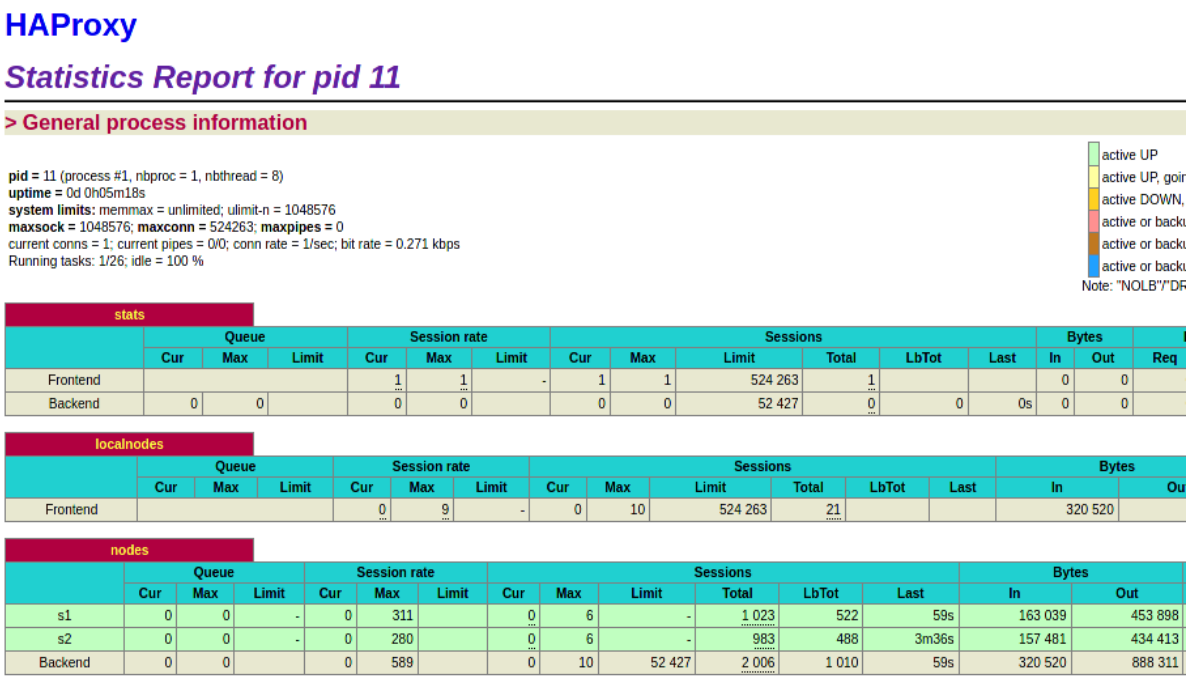
Label	# Samples	Average	Min	Max	Std. Dev.	Error %
GET /	1000	14	1	60	18.69	0.00%
S2 reached	483	0	0	3	0.36	0.00%
S1 reached	517	0	0	2	0.36	0.00%
TOTAL	2000	7	0	60	14.98	0.00%

Label	# Samples	Average	Min	Max	Std. Dev.	Error %
GET /	1000	14	1	58	18.93	0.00%
S1 reached	500	0	0	3	0.35	0.00%
S2 reached	500	0	0	1	0.30	0.00%
TOTAL	2000	7	0	58	15.25	0.00%

As we expected, the results are way more balanced this way, exactly 50% when the session is preserved through cookies and a little difference above, probably because 13 more connections were established while one more was still active on the other server. The second screenshot shows that `roundrobin` is correctly implemented when the number of connections is the same on all the servers.

We can verify that by accessing our browser and accessing the page `http://192.168.42.42/` multiples times with cookies disabled, we can see that we're bouncing between our servers.

Finally here is a screenshot of the stats page from this config, where you can see the load balancer kept track of the number of sessions established.



HAProxy

Statistics Report for pid 11

> General process information

pid = 11 (process #1, nbproc = 1, nbthread = 8)
uptime = 0d 0h05m18s
system limits: memmax = unlimited; ulimit-n = 1048576
maxsock = 1048576; **maxconn =** 524263; **maxpipes =** 0
current conns = 1; current pipes = 0/0; conn rate = 1/sec; bit rate = 0.271 kbps
Running tasks: 1/26; idle = 100 %

active UP
active UP, goir
active DOWN,
active or backu
active or backu
active or backu
Note: "NOLB"/"DR

stats

	Queue			Session rate			Sessions						Bytes		
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req
Frontend				1	1	-	1	1	524 263	1			0	0	
Backend	0	0		0	0		0	0	52 427	0	0	0s	0	0	

localnodes

	Queue			Session rate			Sessions						Bytes	
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Ou
Frontend				0	9	-	0	10	524 263	21			320 520	

nodes

	Queue			Session rate			Sessions						Bytes	
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out
s1	0	0	-	0	311		0	6	-	1 023	522	59s	163 039	453 898
s2	0	0	-	0	280		0	6	-	983	488	3m36s	157 481	434 413
Backend	0	0		0	589		0	10	52 427	2 006	1 010	59s	320 520	888 311

This conclude our analysis of both strategies, our conclusions can be read in the point below.

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

It's interesting to see that the two strategies have radically different goals and applications, the `first` strategy will put every connection on the first server of the poll until said server has reached it's maximum number of allowed connections, then move onto the second server, etc ...

This strategy allows to exploit every server to the maximum of its capacity and to adapt the size of the server cluster/poll according to the number of the connections. If you have 10 servers with 1000 max connections during peak hours but a maximum of 1500 connections (during the night, for example) then you could turn off / re-purpose 8 of the servers. It is useful if you have tool that automate the size of your server poll based on the servers activity. This configuration fits better with long sessions (RDP and IMAP are quoted as examples in the documentation)

On the other hand, the `leastConn` strategy will take advantage of every server available to its capacity by always assigning a new connection to the less busy server of the poll (`roundrobin` is used when they are equal), this allows the load to be equally distributed among our servers. This policy is better when used with very long sessions (LDAP and SQL are among the examples quoted in the documentation)

We don't think any of those two strategies would fit the lab better than the `roundrobin` policy who was used, mostly because the sessions were especially shorts during the tests and examples while the two strategies aforementioned are better for longer ones. However it's important to know about such possibilities when you're about to decide on a load balancer implementation.

Conclusion

In this lab we have taken a look at load balancing, understood the session stickiness trough implementation and compared a few strategies that can be used while setting up load balancing. Running the tests allowed us to get a better understanding of every of those strategies as well as the different behaviors when we change the state of a node.

The two strategies we did study in the last ask allowed us to see different use cases for load balancing and allow us to take proper decisions if we have to implement load balancing for longer sessions later on.