Implementing and Evaluating A Distributed Stock Price Service with Docker - Group 16

Juan Luengo (1555383), Ties Tienhoven (1896504) & Wouter Elenbaas (1370421)

***Abstract*—In the lab assignment described in this paper, a client-server system in Docker is set up to simulate a distributed stock price service. During the design process, a plain single-client, single-server architecture is extended to multiple clients and multiple servers. An NGINX Docker container is added to distribute the stock price requests over the server containers. From the Yahoo Finance API, the stock value is returned to the server and eventually to the client. The single-client, single-server design is compared with a multi-client, multi-server design using a performance test. The results show that the latency for a batch of 100 requests is halved after extending the system to multiple servers. The system has become scalable because of the NGINX load balancer. With additional server health checks in the NGINX container, the system also becomes resilient against dysfunctional server containers.**

***Index Terms*—NGINX, RPyC, Client-Server style, Docker compose**

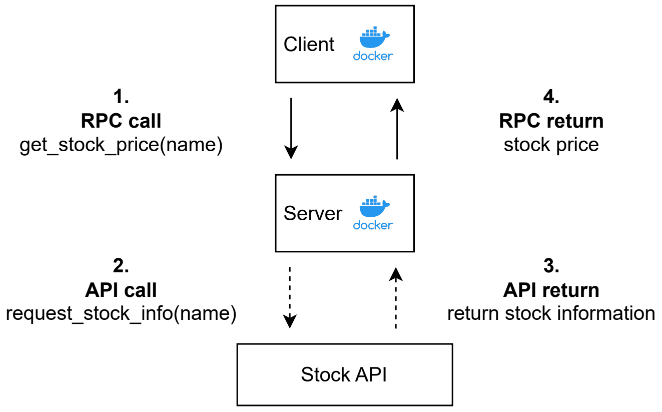


Fig. 1. Architecture of Stock Price Service System after completing phase 1 of the lab assignment

I. Introduction

To retrieve the real-time stock price of a particular company, one can use a stock price API that provides the exact price for a requested stock's symbol. This lab report shows an implementation of a stock price service where a client application requests the stock price from a server that returns the real-time value as given by the API service. In order to test the performance of such a client-server system, a design is created inside Docker. This program allows isolating the system's components into Docker containers, to simulate a real-world simulation where the system's applications run on different devices. The client container uses the Remote Python Call (RPyC) library to connect to the server container. RPyC uses object-proxying to make Python functions in the server container available to the client container as if they were local functions. This lab assignment consists of 3 phases, in which the stock price service on Docker is gradually extended.
During the first phase, a simple single-client, single-server system is set up. In the second phase, the number of servers is extended and the requests to these servers are distributed by a separate NGINX container. NGINX serves as a load balancer for the servers and allows the system to scale up. The last and third phase consists of a fault-tolerant test, where NGINX performs health checks on the servers to account for defective server containers.

II. Phase 1:

The simple single-client, single-server architecture created in this phase is shown in Figure 1. All involved communication processes are numbered in order of occurrence. Furthermore, the diagram shows two entities running inside Docker and one external component outside the system connected with dashed lines.
The terminal output from running the server and client is given in Figures 3 and 4 of Appendix A. The client executes requests with the interval and to the IP address inserted in the command window. In this interval, it chooses a random new stock symbol from the S&P 500 stock index. This symbol is included in the RPyC call to the server, which consequently executes an API call to the Yahoo Finance API to retrieve the corresponding stock value. This value is returned to the client, which prints it to the terminal as shown in the command window screenshots.

III. Phase 2:

For this phase, NGINX will be acting as a load balancer that helps in distributing incoming client requests across multiple server replicas, improving the system's ability to handle a larger volume of requests and allowing scalability. Figure 2 gives an overview of every process involved in the system. Different load-balancing methods such as Round Robin, Least Connections, and IP Hash are supported by NGINX, each with its different way of distributing requests. But for this assignment, we will be using only 2. The configuration for load balancing is done through the `nginx.conf` file, wherein you specify the load balancing method and other necessary configurations based on your RPC implementation, ensuring the right distribution mechanism for your service.

When a client sends a request to get the price of a stock, the request first reaches the NGINX load balancer. NGINX, depending on the configured load-balancing method, decides which server replica will handle the request. With three server

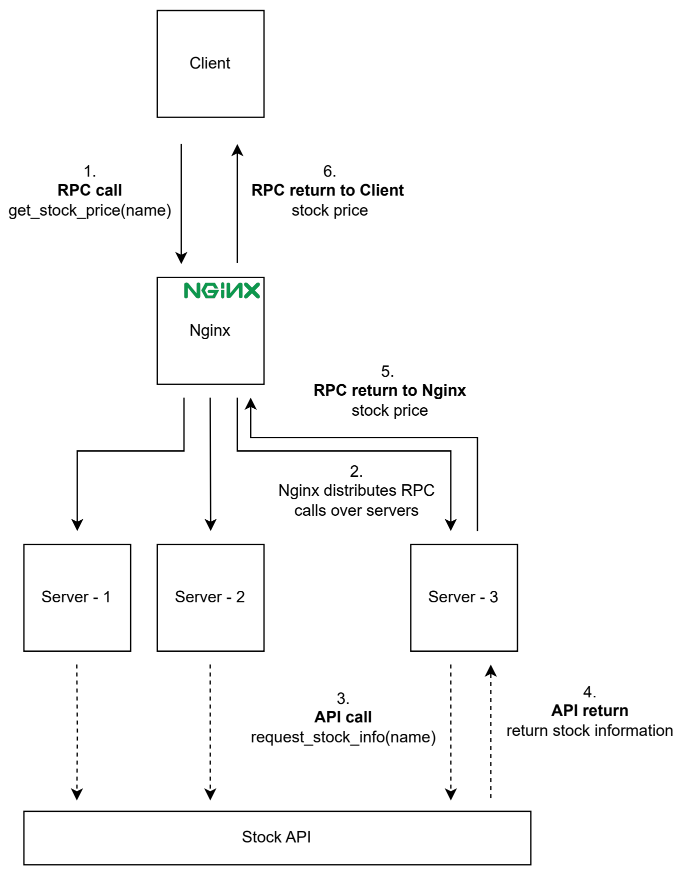


Fig. 2. Architecture of Stock Price Service System after completing phase 2 of the lab assignment

replicas, NGINX ensures that the load is evenly distributed and thus prevents a single server from becoming a bottleneck.

In this phase, the service is deployed using Docker Compose. This simplifies the preparation of multi-container Docker applications. In the `docker-compose.yml` file, the settings are specified for the client and server containers. For the client, we created a non-root user, set the working directory to `/app`, and defined the entry point to execute `src/main.py` with a command line argument of `"3"`. In the same way, the server's Dockerfile sets up a non-root user, designates the working directory, and has a command to execute `src/main.py`, but also shows port `18861` for communication. With Docker Compose, you encapsulate these configurations, allowing you to build and deploy your client-server setup with a single command. This approach not only makes deployment quicker but also makes sure to have a consistent environment for our distributed stock price service, helping in the scalability and maintenance of the application.

A. *Performance analysis*

To evaluate the scalability of our system, we have measured the processing time for a batch of requests and compared the results between phase 1 and phase 2. Our tests consist of one client requesting information about the first 100 stocks in our list of stocks, this way each run of this test requests the same symbols. These 100 requests are made asynchronously and are practically all made at the same time. Then by starting a timer before the first request and stopping it after we get a response for each of the 100 requests we can measure the processing time of our server.

TABLE I
BATCH TEST RESULTS

Run	1	2	3	4	5	Avg
P1	10.32	9.95	9.83	10.71	10.14	10.19
P2-Round Robin	4.16	4.17	4.14	4.17	4.16	4.16
P2-Least Conn	4.46	4.43	4.43	4.23	4.27	4.36

We have run this test first on our phase 1 implementation as a reference. Then, we ran this on our phase 2 implementation with two different load balancing strategies. The results are shown in Table I. This table shows that our phase 1 (P1 in table) implementation takes an average time of 10.19 seconds to respond to these 100 requests. Comparing this to our phase 2 implementations (P2-Round robin and P2-Least Conn in table) result we can see that the load balancing introduced in phase 2 cuts down this processing time to almost a third of the time. Thus we can conclude that our system does scale with multiple servers and a load balancer between our clients and servers.

Note that the scalability of the system is also dependent on the chosen stock API, we are using the Yahoo finance API which has a relatively high request limit per minute thus our servers don't have to wait on the API-key requests limit. This could have been a problem when using an API like Alpha Vantage since this has a limit of 5 requests per minute.

IV. PHASE 3:

For this phase, we will implement fault tolerance in one of the nodes to see if the program can still run without it, and check how NGINX redistributes all the workload. As a start, we will be running the program as usual and stopping one of the docker containers which is running a server to simulate a failure in a server.

In the development of our program, we have experimented with two distinct load-balancing methods: Round-Robin and Least Connections. The difference between the two can be found in the network traffic and application behavior. The Round-Robin method is really simple. It moves incoming requests sequentially across all available nodes without regard to the workload on the node at the moment. On the other hand, the Least Connections method intelligently routes incoming requests to the nodes with the least number of active connections, which is key, especially when the nodes have different processing capabilities (some can compute faster than others). We have run our system with these strategies independently, enabling us to observe and analyze the differences in load distribution and fault tolerance behavior.

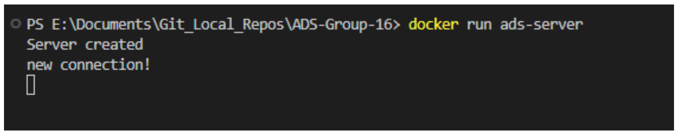
Within the NGINX stream context, we defined a server block to listen on port 18861, directing the traffic to an upstream group named `ads_servers`. This group involves three server nodes, each with a specified maximum failure and a timeout duration post-failure. The fault tolerance used in our setup is initiated when any server node crosses the point of failure. The `max_fails` and `fail_timeout` parameters are really important for monitoring the health of the nodes and ensuring the continuity of service by routing the traffic away from the failed nodes.

We tested the fault tolerance of this setup by simulating node failures, where one or more nodes were shut down by us. The observation of the behavior after the shutdown revealed that the program was working as intended. An automatic redistribution of load across the remaining healthy nodes, therefore making the system fault tolerant. Through Docker, we were able to monitor the status of the nodes, providing evidence demonstrated in the appendix of the fault tolerance mechanism's functionality in real-time.

V. Conclusion

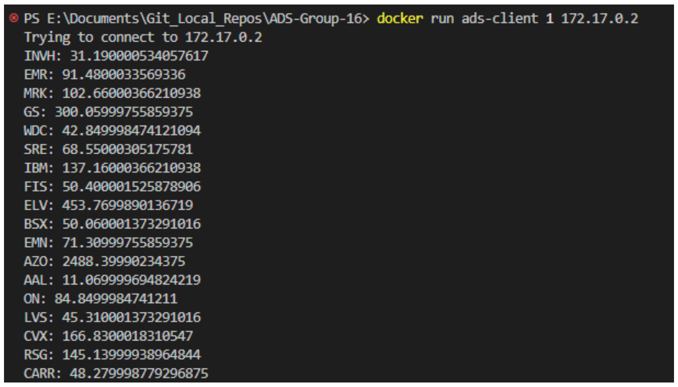
As we have seen throughout this report we were able to build the stock service system. First, we successfully created a setup with one client and one server. Then, we were able to expand this using NGINX and introduce load balancing which made the system scalable as demonstrated in our performance testing. Lastly, we were able to implement fault-tolerance where node failures were handled successfully and our system kept operating as expected.

Appendix A
Terminal outputs



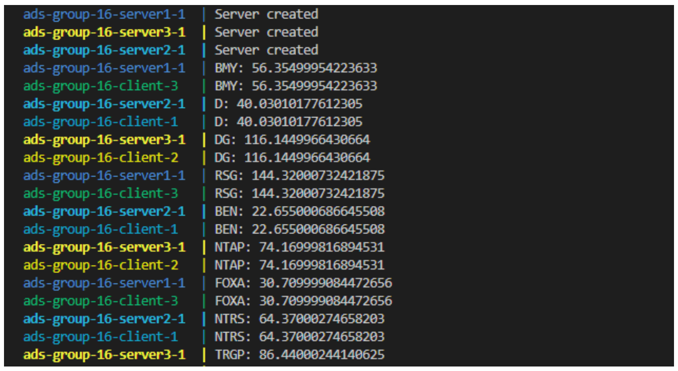
```
○ PS E:\Documents\Git_Local_Repos\ADS-Group-16> docker run ads-server
  Server created
  new connection!
```

Fig. 3. Phase 1: Terminal output of the server container without command line arguments



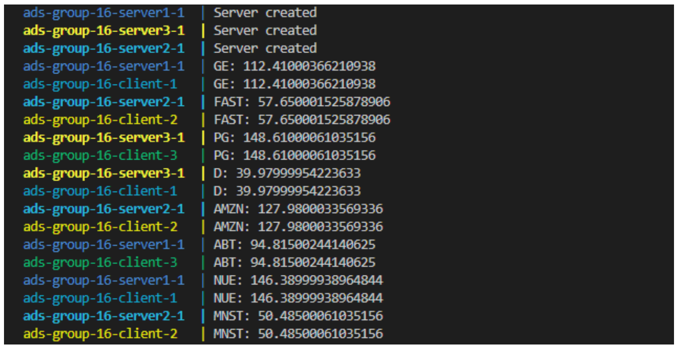
```
⊗ PS E:\Documents\Git_Local_Repos\ADS-Group-16> docker run ads-client 1 172.17.0.2
  Trying to connect to 172.17.0.2
  INVH: 31.190000534057617
  EMR: 91.4800033569336
  MRK: 102.66000366210938
  GS: 300.05999755859375
  WDC: 42.849998474121094
  SRE: 68.55000305175781
  IBM: 137.16000366210938
  FIS: 50.400001525878906
  ELV: 453.7699890136719
  BSX: 50.060001373291016
  EMN: 71.30999755859375
  AZO: 2488.39990234375
  AAL: 11.069999694824219
  ON: 84.8499984741211
  LVS: 45.310001373291016
  CVX: 166.8300018310547
  RSG: 145.13999938964844
  CARR: 48.279998779296875
```

Fig. 4. Phase 1: Terminal output of the client container with two command line inputs: a 1-second request interval and the server container IP address



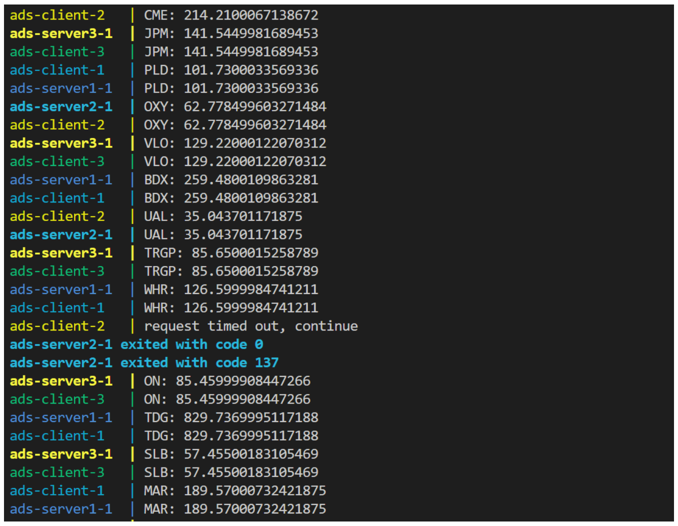
```
ads-group-16-server1-1  | Server created
ads-group-16-server3-1  | Server created
ads-group-16-server2-1  | Server created
ads-group-16-server1-1  | BMY: 56.35499954223633
ads-group-16-client-3   | BMY: 56.35499954223633
ads-group-16-server2-1  | D: 40.03010177612305
ads-group-16-client-1   | D: 40.03010177612305
ads-group-16-server3-1  | DG: 116.1449966430664
ads-group-16-client-2   | DG: 116.1449966430664
ads-group-16-server1-1  | RSG: 144.32000732421875
ads-group-16-client-3   | RSG: 144.32000732421875
ads-group-16-server2-1  | BEN: 22.655000686645508
ads-group-16-client-1   | BEN: 22.655000686645508
ads-group-16-server3-1  | NTAP: 74.16999816894531
ads-group-16-client-2   | NTAP: 74.16999816894531
ads-group-16-server1-1  | FOXA: 30.709999084472656
ads-group-16-client-3   | FOXA: 30.709999084472656
ads-group-16-server2-1  | NTRS: 64.37000274658203
ads-group-16-client-1   | NTRS: 64.37000274658203
ads-group-16-server3-1  | TRGP: 86.44000244140625
```

Fig. 5. Phase 2: Terminal output resulting from the *Docker compose up* command. The Nginx container is set to *Round Robin*, causing the servers to execute in consecutive order.



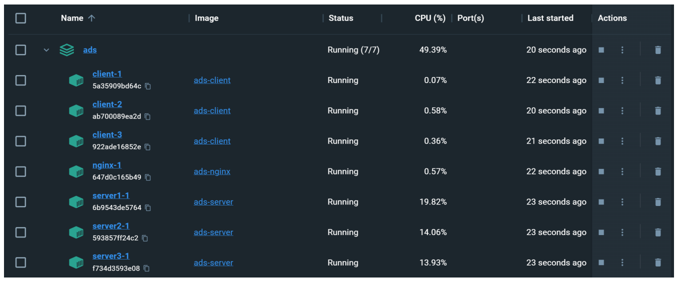
```
ads-group-16-server1-1  | Server created
ads-group-16-server3-1  | Server created
ads-group-16-server2-1  | Server created
ads-group-16-server1-1  | GE: 112.41000366210938
ads-group-16-client-1   | GE: 112.41000366210938
ads-group-16-server2-1  | FAST: 57.650001525878906
ads-group-16-client-2   | FAST: 57.650001525878906
ads-group-16-server3-1  | PG: 148.61000061035156
ads-group-16-client-3   | PG: 148.61000061035156
ads-group-16-server3-1  | D: 39.97999954223633
ads-group-16-client-1   | D: 39.97999954223633
ads-group-16-server2-1  | AMZN: 127.9800033569336
ads-group-16-client-2   | AMZN: 127.9800033569336
ads-group-16-server1-1  | ABT: 94.81500244140625
ads-group-16-client-3   | ABT: 94.81500244140625
ads-group-16-server1-1  | NUE: 146.38999938964844
ads-group-16-client-1   | NUE: 146.38999938964844
ads-group-16-server2-1  | MNST: 50.48500061035156
ads-group-16-client-2   | MNST: 50.48500061035156
```

Fig. 6. Phase 2: Terminal output resulting from the *Docker compose up* command.The Nginx container is set to *Least connections*, causing the servers to execute in order of least present connections.



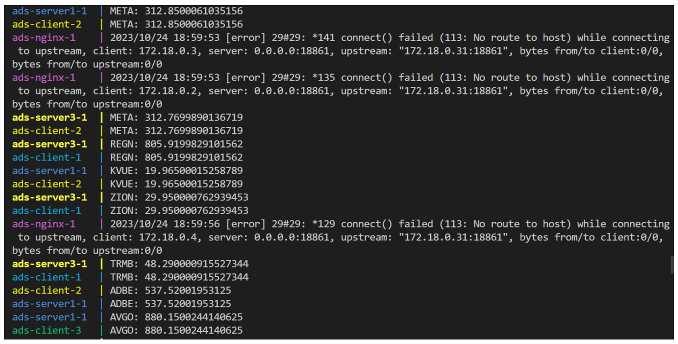
```
ads-client-2    | CME: 214.2100067138672
ads-server3-1   | JPM: 141.5449981689453
ads-client-3    | JPM: 141.5449981689453
ads-client-1    | PLD: 101.7300033569336
ads-server1-1   | PLD: 101.7300033569336
ads-server2-1   | OXY: 62.778499603271484
ads-client-2    | OXY: 62.778499603271484
ads-server3-1   | VLO: 129.22000122070312
ads-client-3    | VLO: 129.22000122070312
ads-server1-1   | BDX: 259.4800109863281
ads-client-1    | BDX: 259.4800109863281
ads-client-2    | UAL: 35.043701171875
ads-server2-1   | UAL: 35.043701171875
ads-server3-1   | TRGP: 85.6500015258789
ads-client-3    | TRGP: 85.6500015258789
ads-server1-1   | WHR: 126.5999984741211
ads-client-1    | WHR: 126.5999984741211
ads-client-2    | request timed out, continue
ads-server2-1 exited with code 0
ads-server2-1 exited with code 137
ads-server3-1   | ON: 85.45999908447266
ads-client-3    | ON: 85.45999908447266
ads-server1-1   | TDG: 829.7369995117188
ads-client-1    | TDG: 829.7369995117188
ads-server3-1   | SLB: 57.45500183105469
ads-client-3    | SLB: 57.45500183105469
ads-client-1    | MAR: 189.57000732421875
ads-server1-1   | MAR: 189.57000732421875
```

Fig. 7. Phase 3: Terminal output from the *Docker compose up* command before and when we introduce a server failure on server 2. With the load balancing setup with Round Robin



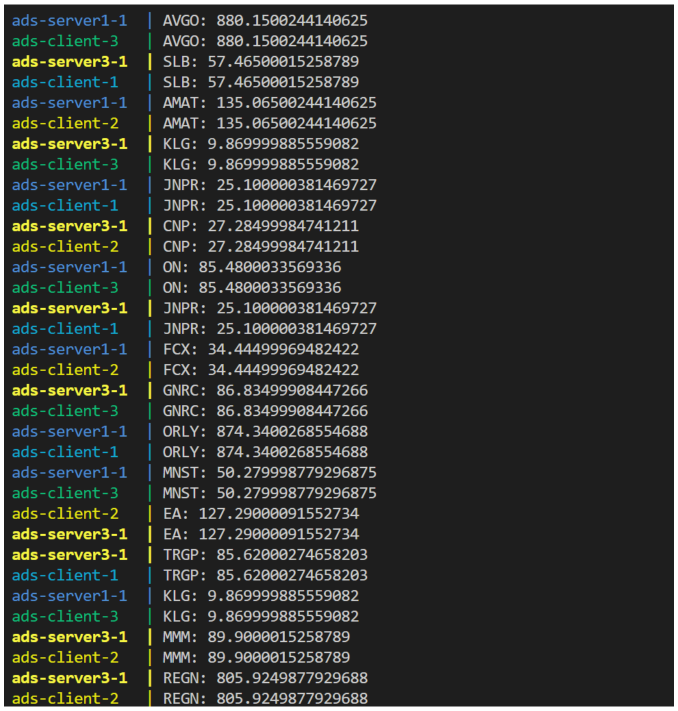
Name	Image	Status	CPU (%)	Port(s)	Last started	Actions
ads		Running (7/7)	49.39%		20 seconds ago	
client-1 5a35909bd64c	ads-client	Running	0.07%		22 seconds ago	
client-2 ab700089ea2d	ads-client	Running	0.58%		20 seconds ago	
client-3 922ade16852e	ads-client	Running	0.36%		21 seconds ago	
nginx-1 647d0c165b49	ads-nginx	Running	0.57%		22 seconds ago	
server1-1 6b9543de5764	ads-server	Running	19.82%		23 seconds ago	
server2-1 593857ff24c2	ads-server	Running	14.06%		23 seconds ago	
server3-1 f734d3593e08	ads-server	Running	13.93%		23 seconds ago	

Fig. 8. Phase 3: The docker container status before we introduce a server failure



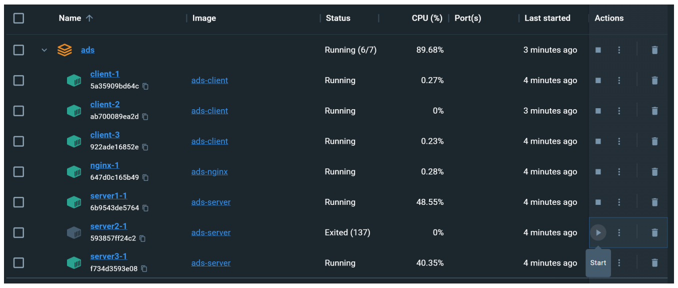
```
ads-server1-1  | META: 312.8500061035156
ads-client-2   | META: 312.8500061035156
ads-nginx-1    | 2023/10/24 18:59:53 [error] 29#29: *141 connect() failed (113: No route to host) while connecting
 to upstream, client: 172.18.0.3, server: 0.0.0.0:18861, upstream: "172.18.0.31:18861", bytes from/to client:0/0,
bytes from/to upstream:0/0
ads-nginx-1    | 2023/10/24 18:59:53 [error] 29#29: *135 connect() failed (113: No route to host) while connecting
 to upstream, client: 172.18.0.2, server: 0.0.0.0:18861, upstream: "172.18.0.31:18861", bytes from/to client:0/0,
bytes from/to upstream:0/0
ads-server3-1  | META: 312.7699890136719
ads-client-2   | META: 312.7699890136719
ads-server3-1  | REGN: 805.9199829101562
ads-client-1   | REGN: 805.9199829101562
ads-server1-1  | KVUE: 19.96500015258789
ads-client-2   | KVUE: 19.96500015258789
ads-server3-1  | ZION: 29.950000762939453
ads-client-1   | ZION: 29.950000762939453
ads-nginx-1    | 2023/10/24 18:59:56 [error] 29#29: *129 connect() failed (113: No route to host) while connecting
 to upstream, client: 172.18.0.4, server: 0.0.0.0:18861, upstream: "172.18.0.31:18861", bytes from/to client:0/0,
bytes from/to upstream:0/0
ads-server3-1  | TRMB: 48.290000915527344
ads-client-1   | TRMB: 48.290000915527344
ads-client-2   | ADBE: 537.52001953125
ads-server1-1  | ADBE: 537.52001953125
ads-server1-1  | AVGO: 880.1500244140625
ads-client-3   | AVGO: 880.1500244140625
```

Fig. 9. Phase 3: The terminal output from the *Docker compose up* command at the moment where NGINX starts to realise server 2 is down



```
ads-server1-1  | AVGO: 880.1500244140625
ads-client-3   | AVGO: 880.1500244140625
ads-server3-1  | SLB: 57.46500015258789
ads-client-1   | SLB: 57.46500015258789
ads-server1-1  | AMAT: 135.06500244140625
ads-client-2   | AMAT: 135.06500244140625
ads-server3-1  | KLG: 9.869999885559082
ads-client-3   | KLG: 9.869999885559082
ads-server1-1  | JNPR: 25.100000381469727
ads-client-1   | JNPR: 25.100000381469727
ads-server3-1  | CNP: 27.28499984741211
ads-client-2   | CNP: 27.28499984741211
ads-server1-1  | ON: 85.4800033569336
ads-client-3   | ON: 85.4800033569336
ads-server3-1  | JNPR: 25.100000381469727
ads-client-1   | JNPR: 25.100000381469727
ads-server1-1  | FCX: 34.44499969482422
ads-client-2   | FCX: 34.44499969482422
ads-server3-1  | GNRC: 86.83499908447266
ads-client-3   | GNRC: 86.83499908447266
ads-server1-1  | ORLY: 874.3400268554688
ads-client-1   | ORLY: 874.3400268554688
ads-server1-1  | MNST: 50.279998779296875
ads-client-3   | MNST: 50.279998779296875
ads-client-2   | EA: 127.29000091552734
ads-server3-1  | EA: 127.29000091552734
ads-server3-1  | TRGP: 85.62000274658203
ads-client-1   | TRGP: 85.62000274658203
ads-server1-1  | KLG: 9.869999885559082
ads-client-3   | KLG: 9.869999885559082
ads-server3-1  | MMM: 89.9000015258789
ads-client-2   | MMM: 89.9000015258789
ads-server3-1  | REGN: 805.9249877929688
ads-client-2   | REGN: 805.9249877929688
```

Fig. 10. Phase 3: The terminal output from the *Docker compose up* command after the server failure. Here we can see NGINX successfully balancing the load over the remaining two servers



Name	Image	Status	CPU (%)	Port(s)	Last started	Actions
ads		Running (6/7)	89.68%		3 minutes ago	
client-1 5a35909bd64c	ads-client	Running	0.27%		4 minutes ago	
client-2 ab700089ea2d	ads-client	Running	0%		3 minutes ago	
client-3 922ade16852e	ads-client	Running	0.23%		4 minutes ago	
nginx-1 647d0c165b49	ads-nginx	Running	0.28%		4 minutes ago	
server1-1 6b9543de5764	ads-server	Running	48.55%		4 minutes ago	
server2-1 593857ff24c2	ads-server	Exited (137)	0%		4 minutes ago	
server3-1 f734d3593e08	ads-server	Running	40.35%		4 minutes ago	Start

Fig. 11. Phase 3: The docker container status after we have introduced a container failure