Report on the 2IMN10 Architecture of Distributed Systems Lab Exercise

Sam Baggen, *1295632* Quint Bakens, *1454315* and Dmytro Mudragel, *2022060*

Abstract—In the fast-paced realm of financial markets, staying competitive demands swift access to accurate stock price data. Using Docker, RPyC and Nginx, this report shows what a system looks like that is able to deal with high loads and faults. While also showcasing which considerations one needs to make when implementing such a system. Each of the phases of the development process is highlighted, first by making it work, then by scaling it and last by making it robust.

Index Terms—Computer Society, IEEE, Computer System Architecture, Distributed System, Docker, Nginx, RPyC.

1 Introduction

In the fast-paced realm of financial markets, staying competitive demands swift access to accurate stock price data. This report delves into the world of Docker-based distributed systems designed for stock price monitoring, offering flexibility, scalability, and high availability. We will explore how Docker, a containerization technology, provides the ideal foundation for this innovative approach. Our objective is to discuss the architecture, components, and deployment of such a system and highlight its potential. The report is structured as follows: section 2 discusses how to make the system work in a bare-bones manner. section 3 shows how to scale the system such that it is able to handle large loads while scaling it in a sustainable manner. section 4 highlights how to make the system robust, being able to handle faulting servers while keeping the system up and running.

2 Phase 1 - Architectural Model and Implementation

The architecture for our stock price monitoring system is shown in Figure 1. The system comprises a client and a server with a cache. The client sets up a connection with the server and requests stock prices. The server checks if it has the stock prices cached, while also checking if the cached prices are not too old, and returns the requested prices either from its cache or from the Financial Service Provider API. While this system is designed to easily switch APIs, where only the server needs to be adjusted, we have opted for this proof of concept to use the Alpha Vantage API due to its ease of use.

During the design of our system, we specifically looked at optimizing the performance and specifically the response time. Therefore, we have opted to use a cache on the server side. This saves unnecessary API requests and allows for faster responses. As shown in Figure 2, the difference in response time from a cached price (for instance Ahold Delhaize's stock) and an API fetched price (for instance Walt Disney Co's stock) is about three orders of magnitude. Similarly, we only set up an RPyC connection once instead of opening and closing it for each request. The performance difference between opening a connection once and opening and closing a connection for each request is about a factor 3, with the average response time increasing from 0.3 milliseconds (opening the connection once) to 1 millisecond (opening and closing the connection for each request). These averages are for the most performant case where everything is cached, requesting a stock from the API will dwarf the amount of time saved by reusing an RPyC connection. Yet, every bit of performance counts for a scale of thousands of requests.

Both the client and the server are Docker containers, connected together on a Docker network. The advantage of a Docker network is the automatic DNS resolution of the container names, not needing to bother with IP addresses which might change over time. Additionally, Docker is OS agnostic, meaning that this system does not make underlying assumptions on the host machines apart from the Docker engine requirement. Hence, one is able to deploy this system on all kinds of machines, whether they are running Windows, MacOS, or a Linux distribution. For testing purposes, this is ideal, though one change is required before moving this system into production, which is the registration of a domain name for the server. This way, the client only needs to remember the domain and not the IP address, allowing it to change transparently in the future, reducing the amount of maintenance for the client.

If for some reason the API request fails and sends an unexpected response, the server will try to catch the error and return a negative price of $\$ -1$ instead. A negative price is not a realistic value, and we believe it is better to return a negative price (indicating an error) rather than failing silently and not returning anything. In this way, the client knows that the server has seen and parsed the request, instead of the timeout error that RPyC may throw when we do not reply at all, which might give the client the wrong idea that there may be a network issue between them and the server.

• *Department of Mathematics and Computer Science and Engineering at Eindhoven University of Technology, The Netherlands.*

Manuscript received October 25, 2023

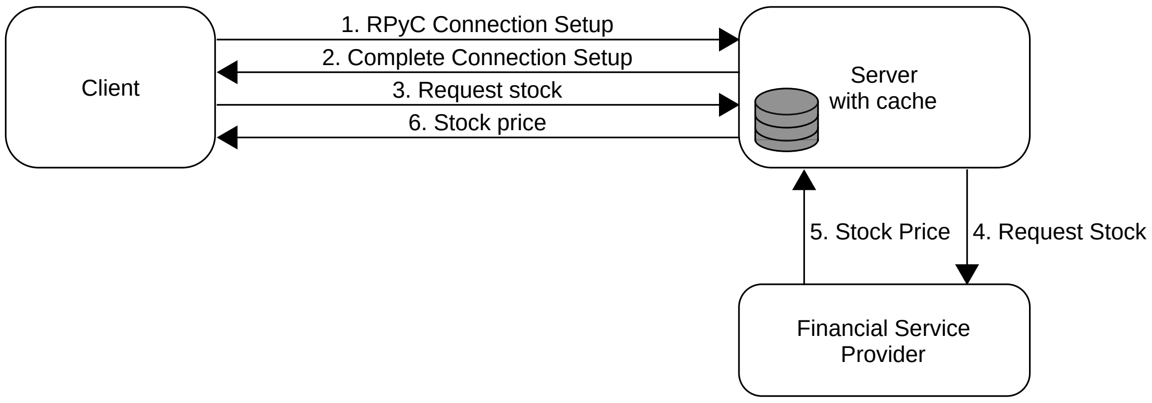
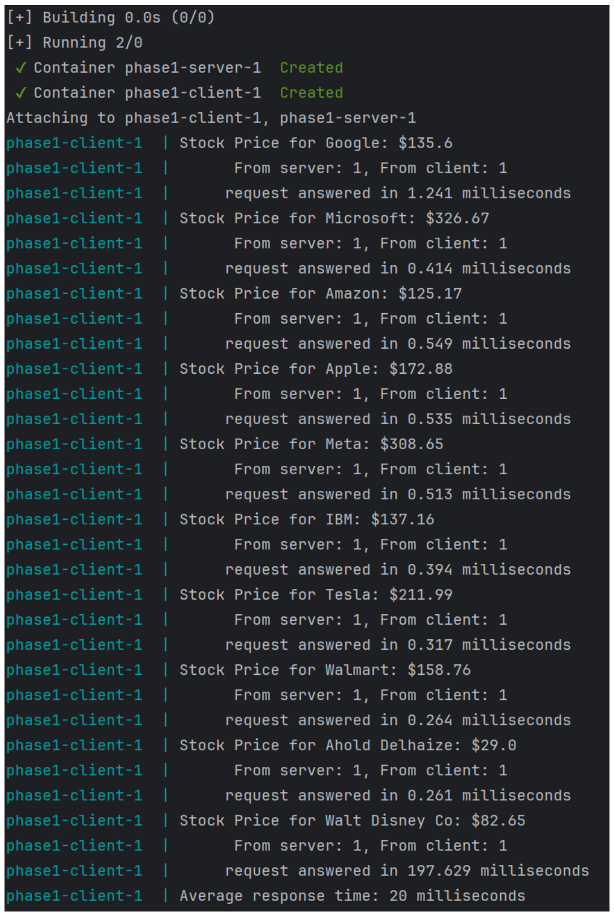


Fig. 1: Architecture of the Stock Price System for Phase 1.



```
[+] Building 0.0s (0/0)
[+] Running 2/0
 ✓ Container phase1-server-1  Created
 ✓ Container phase1-client-1  Created
Attaching to phase1-client-1, phase1-server-1
phase1-client-1  | Stock Price for Google: $135.6
phase1-client-1  |      From server: 1, From client: 1
phase1-client-1  |      request answered in 1.241 milliseconds
phase1-client-1  | Stock Price for Microsoft: $326.67
phase1-client-1  |      From server: 1, From client: 1
phase1-client-1  |      request answered in 0.414 milliseconds
phase1-client-1  | Stock Price for Amazon: $125.17
phase1-client-1  |      From server: 1, From client: 1
phase1-client-1  |      request answered in 0.549 milliseconds
phase1-client-1  | Stock Price for Apple: $172.88
phase1-client-1  |      From server: 1, From client: 1
phase1-client-1  |      request answered in 0.535 milliseconds
phase1-client-1  | Stock Price for Meta: $308.65
phase1-client-1  |      From server: 1, From client: 1
phase1-client-1  |      request answered in 0.513 milliseconds
phase1-client-1  | Stock Price for IBM: $137.16
phase1-client-1  |      From server: 1, From client: 1
phase1-client-1  |      request answered in 0.394 milliseconds
phase1-client-1  | Stock Price for Tesla: $211.99
phase1-client-1  |      From server: 1, From client: 1
phase1-client-1  |      request answered in 0.317 milliseconds
phase1-client-1  | Stock Price for Walmart: $158.76
phase1-client-1  |      From server: 1, From client: 1
phase1-client-1  |      request answered in 0.264 milliseconds
phase1-client-1  | Stock Price for Ahold Delhaize: $29.0
phase1-client-1  |      From server: 1, From client: 1
phase1-client-1  |      request answered in 0.261 milliseconds
phase1-client-1  | Stock Price for Walt Disney Co: $82.65
phase1-client-1  |      From server: 1, From client: 1
phase1-client-1  |      request answered in 197.629 milliseconds
phase1-client-1  | Average response time: 20 milliseconds
```

Fig. 2: Screenshot of the output of our system functioning for Phase 1.

3 Phase 2 - Scalability & Load Balancing

The architecture introduced in section 2 suffers from one major problem: scalability. The more clients ask for stock prices, the more load the server needs to handle. Without assigning more resources to the server, eventually, it will not be able to serve all requests fast enough (or at all). Therefore, we either need to assign more resources to the server or adjust the architecture to be able to deal with more servers to serve all the requests. While assigning more resources to the server will help with the issue, there are only so many resources available to one machine. Think about the maximum amount of RAM memory a CPU is able to deal with (the Intel Xeon W5-2455X CPU has a maximum memory capacity of 2 TB [1]) or the physical limits of bandwidth of the internet connection port of the server (like 1 Gbps). In contrast, deploying more servers is not limited by such constraints (practically only by space on Earth and the budget for deploying more servers) and therefore is the better solution for the long term. Hence, we have adjusted the architecture to be able to handle more servers being part of the system. The modified architecture is shown in Figure 3.

As shown in Figure 3, the only change to our architecture compared to Phase 1 is the addition of a load balancer and the deployment of multiple servers. The load balancer component is responsible for balancing the load across the servers. Taking our consideration for setting up an RPyC connection once, instead of opening and closing it for each request, the load balancer only balances the load for the incoming connection requests. The requests for the stock price are instead sent to the responsible server directly. The advantages of this manner of load balancing include: a lesser load on the load balancer itself, due to fewer incoming requests that need to be load balanced compared to load balancing every request; a lesser load on the network due to not every request needing to be forwarded from the load balancer to the server but instead delivered directly to the server, nearly halving the amount of traffic moving through the network; and a faster response time for requests, because the request is delivered directly to the server instead of having to take a detour through the load balancer before being delivered to the server. While this may lead to an imperfectly balanced load, as one client might send more requests than another client, we believe that this disadvantage is outweighed by the advantages of only load balancing the setup requests.

Figure 4 shows the system in action for Phase 2. Due to space constraints, we have only provided an excerpt of the complete system output, while still showcasing the important parts. Namely, the system responds to requests,

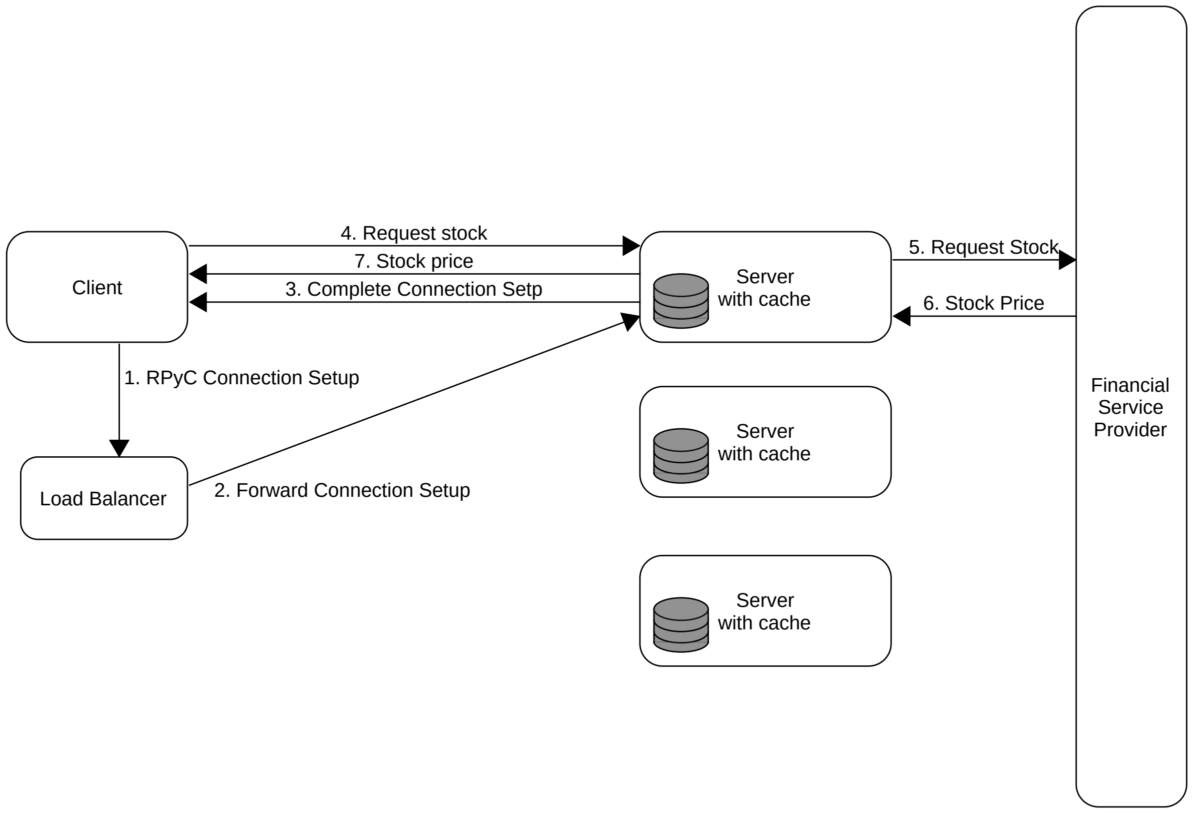


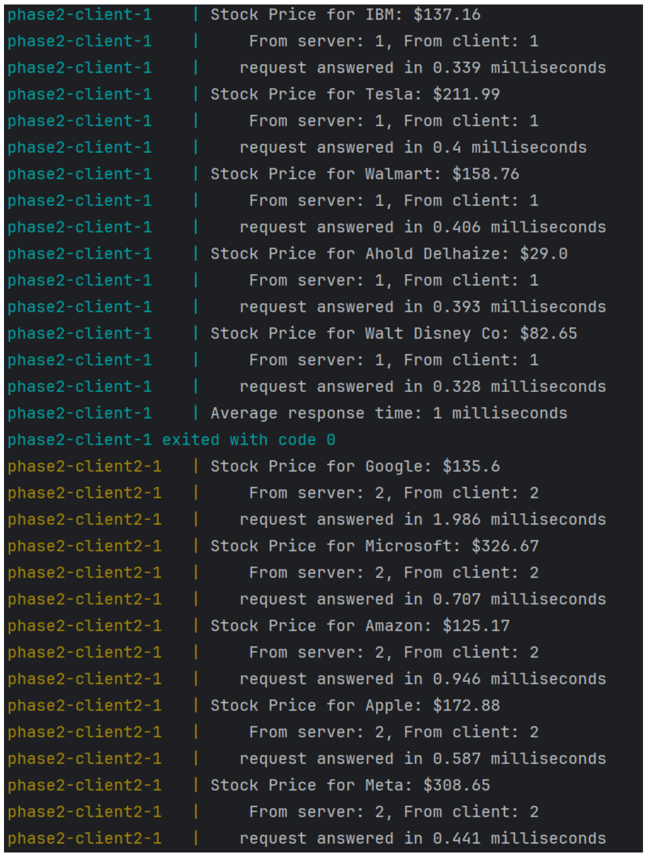
Fig. 3: Architecture of the Stock Price System for Phase 2 and 3.

the system responds from multiple servers and the system responds to multiple clients. For this proof of concept, the load is balanced by the Nginx load balancer. We have tested two load balancing methods: round-robin and random. In the following discussion, load will refer to the number of requests a server handles, while connection load refers to the amount of connections a server handles. This is an important distinction given the way our clients reuse the RPyC connection for multiple requests.

The round-robin load balancing method will balance the load in a round-robin manner. If a request is forwarded to server 1, then the next request will be forwarded to server 2, and the one after that to server 3. This continues until server n has received a request, with n being the total amount of servers available for answering requests. After server n has received a request, the next request will be sent to server 1 again and the whole sequence starts again. This method guarantees a spread load, assuming every client sends the same amount of requests. Unfortunately, this assumption may be false, and using the round-robin method will lead to an imperfectly balanced load. Therefore, round-robin leads to a balanced connection load but possibly an imbalanced load.

The random load balancing method will balance the load randomly. If a request is received by Nginx, it will draw a uniformly distributed random number from 1 to n, with n being the total amount of servers available for answering requests, and forward the request to the server corresponding to the randomly drawn number. This method is not predictable at all, and with a small amount of clients will most likely lead to an imbalanced load, but given enough clients it will distribute the load nearly uniformly. This is due to dependant probabilities. Let the number of servers is n and the number of clients is c. Then the chance of assigning one client to server 1 is $\frac{1}{n}$. The chance of assigning two clients to server 1 is $\frac{1}{n} \cdot \frac{1}{n} = \frac{1}{n^2}$. Continuing this series we get the following probability of assigning all clients to server 1: $\frac{1}{n^c}$, which decreases the more clients there are and converges to 0 if c goes to ∞. Given the distribution is uniform, it will be more likely that all servers handle roughly the same amount of clients than one server handling significantly more clients, resulting in a balanced connection load. However, this method still suffers from the same problem as the round-robin method, which is that not every client will send the same amount of requests. However, due to the unpredictability of the random load balancing method, it may be that the load is better distributed (due to chance) or distributed worse (again due to chance).

All in all, it does not really matter which load balancing method is used, as both will (given enough clients) distribute the connection load evenly among the servers. One may prefer the round-robin method due to its predictability, or opt for the random method because of its unpredictability.



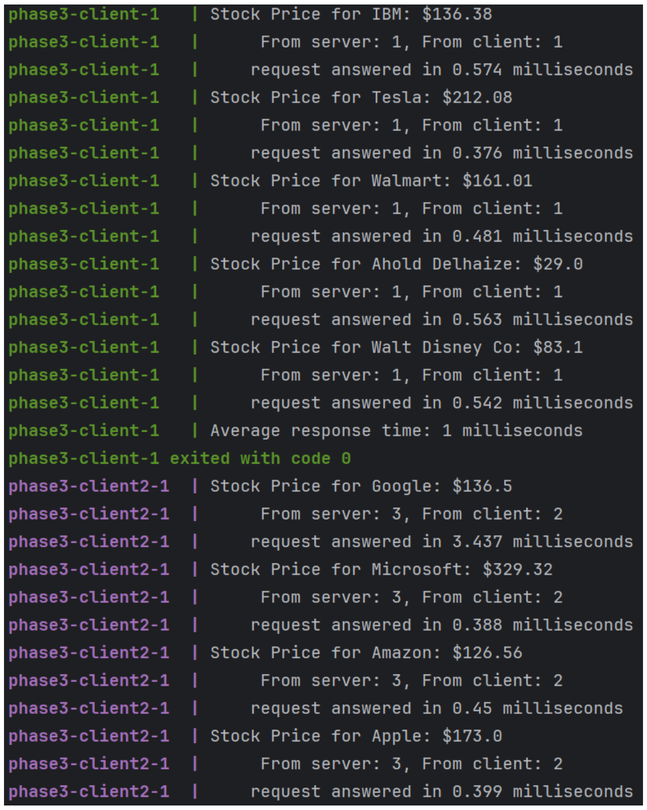
```
phase2-client-1    | Stock Price for IBM: $137.16
phase2-client-1    |     From server: 1, From client: 1
phase2-client-1    |    request answered in 0.339 milliseconds
phase2-client-1    | Stock Price for Tesla: $211.99
phase2-client-1    |     From server: 1, From client: 1
phase2-client-1    |    request answered in 0.4 milliseconds
phase2-client-1    | Stock Price for Walmart: $158.76
phase2-client-1    |     From server: 1, From client: 1
phase2-client-1    |    request answered in 0.406 milliseconds
phase2-client-1    | Stock Price for Ahold Delhaize: $29.0
phase2-client-1    |     From server: 1, From client: 1
phase2-client-1    |    request answered in 0.393 milliseconds
phase2-client-1    | Stock Price for Walt Disney Co: $82.65
phase2-client-1    |     From server: 1, From client: 1
phase2-client-1    |    request answered in 0.328 milliseconds
phase2-client-1    | Average response time: 1 milliseconds
phase2-client-1 exited with code 0
phase2-client2-1   | Stock Price for Google: $135.6
phase2-client2-1   |     From server: 2, From client: 2
phase2-client2-1   |    request answered in 1.986 milliseconds
phase2-client2-1   | Stock Price for Microsoft: $326.67
phase2-client2-1   |     From server: 2, From client: 2
phase2-client2-1   |    request answered in 0.707 milliseconds
phase2-client2-1   | Stock Price for Amazon: $125.17
phase2-client2-1   |     From server: 2, From client: 2
phase2-client2-1   |    request answered in 0.946 milliseconds
phase2-client2-1   | Stock Price for Apple: $172.88
phase2-client2-1   |     From server: 2, From client: 2
phase2-client2-1   |    request answered in 0.587 milliseconds
phase2-client2-1   | Stock Price for Meta: $308.65
phase2-client2-1   |     From server: 2, From client: 2
phase2-client2-1   |    request answered in 0.441 milliseconds
```

Fig. 4: Screenshot of part of the output of our system functioning for Phase 2.

4 PHASE 3 - LOAD BALANCING WITH FAULT TOLERANCE

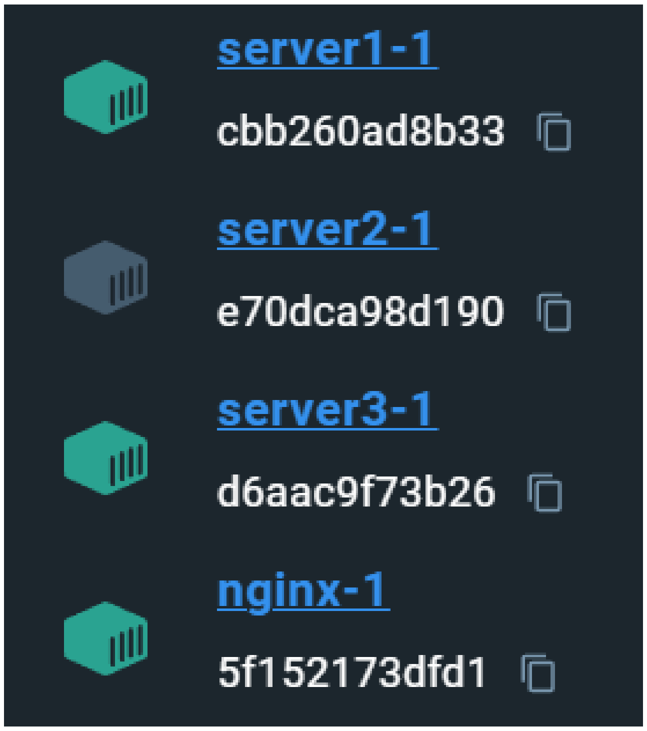
A final refinement for the architecture of Phase 2 is fault tolerance. With the Nginx configuration used for Phase 2, a failed node would remain in the pool of available servers, leading to roughly one-third of the incoming connection requests remaining unanswered and eventually timing out. Hence, we have added a maximum timeout of 30 seconds to the forwarding of a request. It will then try again, and if the request times out again, the server will be removed from the pool of available servers ensuring that no requests will be forwarded towards the failed server. This way, if a server fails it will only affect a small number of the incoming connection requests that fail instead of roughly one-third.

The fault tolerance mechanism is shown in action in Figure 5 and Figure 6. Figure 6 shows that we have simulated a failure for server 2 by turning it off. Then we show in Figure 5 that server 2 is taken out of the round robin rotation, skipping it and instead forwarding the connection setup to server 3. Additionally, as shown in Figure 7, when server 2 returns to service by turning it back on, it is inserted into the pool of available servers again without the need to restart the Nginx container. This leads to less downtime of the system, as restarting the load balancer would mean that requests are not forwarded to the servers which means that the system is down for all intents and purposes. However, since the restart of the load balancer is not required, the system is able to tolerate faulting servers, automatically



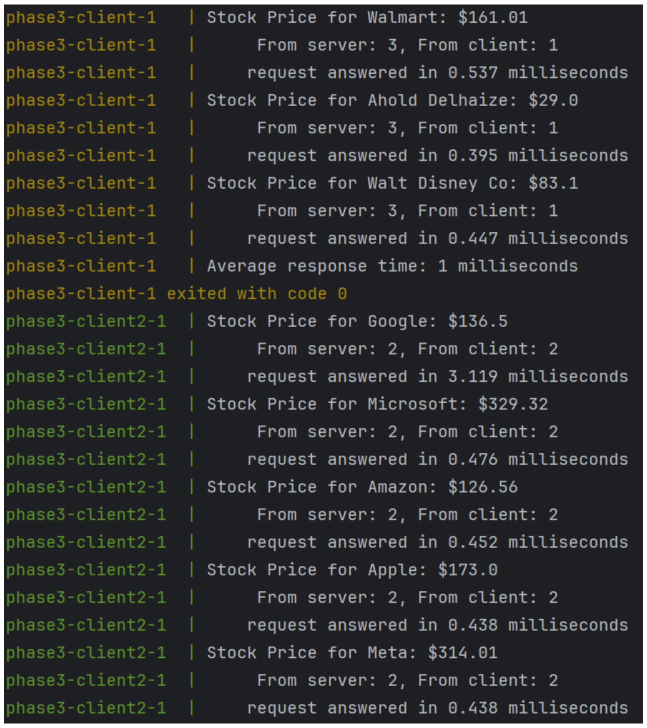
```
phase3-client-1    | Stock Price for IBM: $136.38
phase3-client-1    |     From server: 1, From client: 1
phase3-client-1    |    request answered in 0.574 milliseconds
phase3-client-1    | Stock Price for Tesla: $212.08
phase3-client-1    |     From server: 1, From client: 1
phase3-client-1    |    request answered in 0.376 milliseconds
phase3-client-1    | Stock Price for Walmart: $161.01
phase3-client-1    |     From server: 1, From client: 1
phase3-client-1    |    request answered in 0.481 milliseconds
phase3-client-1    | Stock Price for Ahold Delhaize: $29.0
phase3-client-1    |     From server: 1, From client: 1
phase3-client-1    |    request answered in 0.563 milliseconds
phase3-client-1    | Stock Price for Walt Disney Co: $83.1
phase3-client-1    |     From server: 1, From client: 1
phase3-client-1    |    request answered in 0.542 milliseconds
phase3-client-1    | Average response time: 1 milliseconds
phase3-client-1 exited with code 0
phase3-client2-1   | Stock Price for Google: $136.5
phase3-client2-1   |     From server: 3, From client: 2
phase3-client2-1   |    request answered in 3.437 milliseconds
phase3-client2-1   | Stock Price for Microsoft: $329.32
phase3-client2-1   |     From server: 3, From client: 2
phase3-client2-1   |    request answered in 0.388 milliseconds
phase3-client2-1   | Stock Price for Amazon: $126.56
phase3-client2-1   |     From server: 3, From client: 2
phase3-client2-1   |    request answered in 0.45 milliseconds
phase3-client2-1   | Stock Price for Apple: $173.0
phase3-client2-1   |     From server: 3, From client: 2
phase3-client2-1   |    request answered in 0.399 milliseconds
```

Fig. 5: Screenshot of part of the output of our system functioning for Phase 3. Nginx uses round-robin load balancing here while server 2 has been turned off to simulate a failure.



```
server1-1
cbb260ad8b33
server2-1
e70dca98d190
server3-1
d6aac9f73b26
nginx-1
5f152173dfd1
```

Fig. 6: Screenshot of the running containers of our system, showcasing that server 2 has been turned off to simulate a failure.



```
phase3-client-1   | Stock Price for Walmart: $161.01
phase3-client-1   |      From server: 3, From client: 1
phase3-client-1   |     request answered in 0.537 milliseconds
phase3-client-1   | Stock Price for Ahold Delhaize: $29.0
phase3-client-1   |      From server: 3, From client: 1
phase3-client-1   |     request answered in 0.395 milliseconds
phase3-client-1   | Stock Price for Walt Disney Co: $83.1
phase3-client-1   |      From server: 3, From client: 1
phase3-client-1   |     request answered in 0.447 milliseconds
phase3-client-1   | Average response time: 1 milliseconds
phase3-client-1 exited with code 0
phase3-client2-1  | Stock Price for Google: $136.5
phase3-client2-1  |      From server: 2, From client: 2
phase3-client2-1  |     request answered in 3.119 milliseconds
phase3-client2-1  | Stock Price for Microsoft: $329.32
phase3-client2-1  |      From server: 2, From client: 2
phase3-client2-1  |     request answered in 0.476 milliseconds
phase3-client2-1  | Stock Price for Amazon: $126.56
phase3-client2-1  |      From server: 2, From client: 2
phase3-client2-1  |     request answered in 0.452 milliseconds
phase3-client2-1  | Stock Price for Apple: $173.0
phase3-client2-1  |      From server: 2, From client: 2
phase3-client2-1  |     request answered in 0.438 milliseconds
phase3-client2-1  | Stock Price for Meta: $314.01
phase3-client2-1  |      From server: 2, From client: 2
phase3-client2-1  |     request answered in 0.438 milliseconds
```

Fig. 7: Screenshot of part of the output of our system functioning for Phase 3. Server 2 is back operational and inserted in a different spot in the round robin rotation than during normal start-up.

moving them out of the pool of available servers, and reinserting them into the pool of available servers when they are operational again. Figure 7 also shows that server 2 is inserted into the round robin load balancing in a different spot than during normal start-up, as the usual order is server 1, then server 2, and then server 3. But now the order is server 1, then server 3, and then server 2.

The advantage of fault tolerance in this manner is the great reduction in downtime, as mentioned before. However, there is another aspect of reduced downtime we have not mentioned before. This system will in fact gradually reduce the quality of service when failures occur. As each node fails, less computing capacity remains available for all of the requests. Unless all nodes fail simultaneously, the system will remain up. However, the response times to requests will gradually go up as the load is redistributed to the working servers, away from the failing ones. The more servers fail, the higher the load on the remaining servers. Yet, the system only goes down when all servers fail. Combined with the increasing response times, we can conclude that this system will gradually reduce the quality of the service until it is no longer able to provide the service at all. Still, this gives maintainers and system administrators the time to investigate the already failed servers and potentially resolve the failures, increasing the capacity for serving requests again or at least maintaining the same capacity for serving requests if other servers fail. Therefore, the more servers are employed, the less likely it becomes that the complete system goes down, instead the service quality is reduced due to the higher response times to requests.

5 Conclusion

In conclusion, we have shown the design of a stock price monitoring system, how it is able to scale in a sustainable manner, and how it is fault tolerant. Furthermore, we have discussed different load-balancing methods and the considerations for choosing between them. Lastly, we have also discussed the advantages of a fault-tolerant load balancer, reducing downtime in case of a server fault.

Acknowledgments

The authors would like to thank the coffee machine, for getting us the caffeine required to write the code for the system and this report.

References

[1] Intel, “Intel xeon w5-2455x processor 30m cache, 3.20 ghz product specification,” March 2023. [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/233420/intel-xeon-w52455x-processor-30m-cache-3-20-ghz.html