Replication, Caching, and Consistency for an online Book store CS677 lab3 project

Group member: Shahrooz Pouryousef
University of Massachusetts Amherst
shahrooz@cs.umass.edu

ABSTRACT

This report explains our design, implementation, and evaluation for rearchitecting an online store to handle a higher workload. We first explain how we add replication and caching to improve request processing latency. Then, we explain how our system tries to be fault-tolerant. We explain the functionality that we have added to the front-end node to detect back-end node failures. We evaluate the end-to-end response time with and without caching and show caching can reduce the request processing latency. Specifically, caching can reduce the end-to-end response time for GET requests. However, it can increase the end-to-end response time for other queries such as DELETE as we need to take care of cache consistency by exchanging messages between different microservices. Based on our evaluation results, when we add the fault-tolerant functionality to the system, the end-to-end response is increased. The reason is that the system needs to keep check the available replicas and this adds the overhead of exchanging the extra messages between components in the system.

1 IMPLEMENTATION DESIGN CHOICES

We have fully implemented the caching and replication and fault-tolerant functionality for our online book store. Here we explain our design choices for our implementation.

- For load balancing, we use a round-robin algorithm.
- For server-push technique we use a simple GET request with a specific record ID from the catalog and order microservices to the front end microservice.
- For caching, we save the data in memory.
- To check if a replica is available, we send a simple GET request with a specific record ID to the target replica. In another word, this GET request is our heartbeat message.

2 RUNNING THE SYSTEM

You need to run each microservice and each replication of the microservice with its IP addresses and port number. We have defined the IP address and port number that each microservice will be running on in the "each_server_IP.txt" file. For example, for running the front-end microservice, you need to run the command "python3 order_microservice.py 127.0.0.4 5004" where 127.0.0.4 is the IP address and 5004 is the port number that this replication of the order microservice will be running on. For running the frond-end microservice, you do not need to specify the IP address and port number. Just running the command "python3 front_end_caching" is enough. Note that, if you want to have multiple replications of each microservice, you have to define the IP address and the port number in the "each_server_IP.txt" file and run that replication separately manually in the terminal.

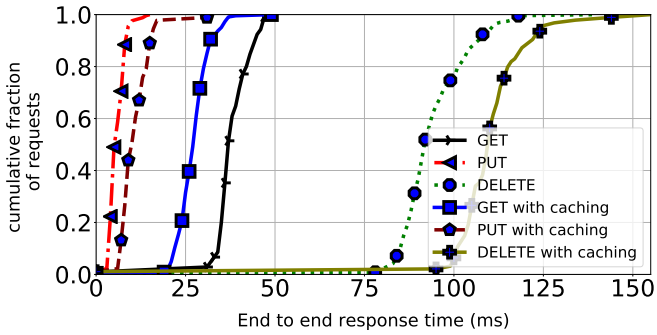

Figure 1: A CDF on the end-to-end response time for a request with and without caching. Caching can reduce the end-to-end response time for GET requests. However, it can increase the end-to-end response time for other queries such as DELETE as we need to take care of cache consistency by exchanging messages between different microservices.

3 EVALUATION

In this section, we evaluate the end-to-end response time with and without using caching in the system.

3.1 End-to-end response time with caching

The goal of this experiment is to measure the end-to-end response time for each client's request with and without caching. We repeat the experiment 1000 times. Each request identifies a run of the system where only one of the clients wants to send a request to the front-end tier microservice.

Figure 1 shows a CDF on the end-to-end response time cross all requests in our experiment for the case that we have used one single machine for all the microservices. Caching can improve (reduce) the end-to-end response time for GET requests. Because of added overhead for consistency of caching in the system among the replications, the end-to-end response time for other requests has increased.

3.2 End-to-end response time with replication

The goal of this experiment is to evaluate the end-to-end response time when we have multiple replications of microservices and we need to check the replicas and make the system fault-tolerant.

Figure 2 shows the CDF for 1000 requests with and without replication. As we can see, making the system fault-tolerant adds some overhead to the system and increases the end-to-end response time. When we have multiple replicas in the system and add the fault-tolerant functionality to the system, the end-to-end response is increased. For example, the end-to-end response time for 50 percent

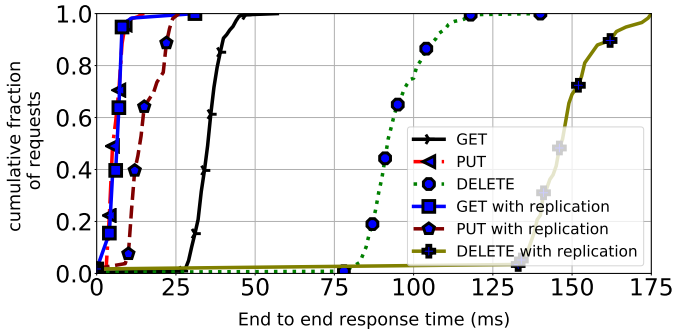

Figure 2: A CDF on the end-to-end response time for a request with and without replication. When we have multiple replicas in the system and add the fault tolerant functionality to the system, the end-to-end response is increased.

of DELETE requests with replication is around 150 milliseconds. This value without replication is 90 milliseconds and with only caching functionality in figure 1 is 120 milliseconds.

4 AN ELECTRONIC COPY OF OUTPUT

Figure 3 shows an electronic copy of the output generated by running my program.

```
 * Serving Flask app "front_end_caching" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:5001/ (Press CTRL+C to quit)
127.0.0.1 - - [30/Apr/2021 01:02:42] "GET /4444 HTTP/1.1" 200 -
127.0.0.1 - - [30/Apr/2021 01:02:42] "GET /2022 HTTP/1.1" 200 -
127.0.0.1 - - [30/Apr/2021 01:02:42] "DELETE //1 HTTP/1.1" 200 -
127.0.0.1 - - [30/Apr/2021 01:02:45] "PUT / HTTP/1.1" 200 -
127.0.0.1 - - [30/Apr/2021 01:02:48] "PUT / HTTP/1.1" 200 -
we do not have data for this reguest in our cache!
127.0.0.1 - - [30/Apr/2021 01:02:51] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [30/Apr/2021 01:02:54] "GET /4444 HTTP/1.1" 200 -
127.0.0.1 - - [30/Apr/2021 01:02:54] "GET /2024 HTTP/1.1" 200 -
127.0.0.1 - - [30/Apr/2021 01:02:54] "DELETE //3 HTTP/1.1" 200 -
127.0.0.1 - - [30/Apr/2021 01:02:57] "GET /4444 HTTP/1.1" 200 -
127.0.0.1 - - [30/Apr/2021 01:02:57] "GET /2023 HTTP/1.1" 200 -
127.0.0.1 - - [30/Apr/2021 01:02:57] "DELETE //2 HTTP/1.1" 200 -
127.0.0.1 - - [30/Apr/2021 01:03:00] "PUT / HTTP/1.1" 200 -
we got the data for this GET request from our in-memory cache!
127.0.0.1 - - [30/Apr/2021 01:03:03] "GET / HTTP/1.1" 200 -
we got the data for this GET request from our in-memory cache!
127.0.0.1 - - [30/Apr/2021 01:03:06] "GET / HTTP/1.1" 200 -
we got the data for this GET request from our in-memory cache!
127.0.0.1 - - [30/Apr/2021 01:03:09] "GET / HTTP/1.1" 200 -
we got the data for this GET request from our in-memory cache!
127.0.0.1 - - [30/Apr/2021 01:03:12] "GET / HTTP/1.1" 200 -
we got the data for this GET request from our in-memory cache!
127.0.0.1 - - [30/Apr/2021 01:03:15] "GET / HTTP/1.1" 200 -
we got the data for this GET request from our in-memory cache!
127.0.0.1 - - [30/Apr/2021 01:03:18] "GET / HTTP/1.1" 200 -
we got the data for this GET request from our in-memory cache!
127.0.0.1 - - [30/Apr/2021 01:03:21] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [30/Apr/2021 01:03:24] "PUT / HTTP/1.1" 200 -
```

Figure 3: An electronic copy of the output