Lab 3 Design Documentation

Introduction

This project implements a toy store that is deployed in the cloud on AWS and that ensures fault tolerance and transparency for the client and implements a front end cache. This project consists of a 3 micro service toy store server and a client. The toy store has a frontend micro service which runs an HTTP server that can be accessed by clients and has a cache for storing toys that have been recently queried already. The toy store has 2 backend micro services: a catalog service and an order service. The front end forwards toy queries to the catalog service, and forwards order requests to the order service. The client connects over HTTP to the front end service and places orders, checks toy information, and checks orders. The server implements fault tolerance by having 3 replicas of the order service running at the same time. If one of the replicas goes down the frontend will use one of the other two replicas. This is all done on the server side, and the client is not aware that this is happening, ensuring transparency for the client.

Goals and Objectives

The goals for this project are to:

1. Design a distributed toy store system in which 3 micro services interact with each other to serve one functioning toy store.
2. Implement a front end cache so that the front end can skip querying the catalog and serve a client's request from the frontend's stored memory.
3. Implement fault tolerance to ensure when one micro service goes down, the entire toy store is able to continue running.
4. Ensure client transparency so that when a micro service does fail the client does not notice the downed service.

5. Deploy a server on the cloud using AWS so that it can be reached and used by clients everywhere.

Design Details

The solution consists of the 3 micro services: frontend service, catalog service, and order service.

The frontend service is the HTTP server that clients will connect to. The service opens an HTTP port and receives incoming HTTP requests to query a toy, buy a toy, or check an order number. The frontend service keeps a running cache where it stores the last N toy queries. If a toy is queried that is in the cache already, then the reply is served from the cache and no request is sent to the catalog. If the toy does not exist in the cache then the request is sent to the catalog and the reply from the catalog is added to the cache. If the cache is full, then the least recently used item in the cache is discarded. When the frontend starts up, it checks on which order service replicas are currently running and assigns the one with the highest ID to be the new order service leader. When an order request or an order check request is received, the front end attempts to send the request to the order service leader. If the leader is not running, it will catch the connection error and instead perform the leader selection by, again, polling to see which order service replicas are up and running, then selecting the new leader to be whichever one has the highest ID. After a leader that is running is found, it sends the order or order check request to the new leader. Since this process happens automatically on the frontend, the client never sees that the connection to the downed leader did not work, and instead will only receive the reply once the new leader is selected.

The catalog service can receive a query request from the front end and reply with the quantity and price from the toy store database. The catalog can receive a request from the order service asking if there is enough stock of the requested toy, and will return a code that either says there is enough stock, is not enough stock, or that the toy does not exist. The catalog service will restock the store every 10 seconds to ensure the store can fulfill order requests. Whenever the catalog receives an order or restocks an item, the catalog will send an invalidation request to the front end informing the front end that if these toys exist in the front end cache, to delete them since the information in the cache is no longer accurate with the toy store database.

The order service can create up to 3 replicas of itself. When a replica is started, it will first request from the other replicas the order logs that it has missed while it was down. Other replicas who are online already will receive this request and reply with their own order logs starting at the last order number the requesting replica asked for. If it is the only replica running, or the other replicas don't have any new order logs for it, the order service begins. The order service receives requests for orders and to check orders from the front end. Orders are fulfilled by querying the catalog to ensure there is enough stock for that toy and then logging the order. To ensure all replicas have the same order logs, each time an order is placed on one replica, it sends the order information to each other replica. When a replica receives order information from another replica, it logs the order to ensure all replicas have the same order logs. When the current leader replica goes down, the front end will send a health check to each other replica, replicas simply reply with a response saying I am healthy, and the replica with the highest ID becomes the new leader.

The client only connects to the frontend. The client can make query requests, buy requests and check orders. The frontend handles all errors and crashes that occur on the server side. The client has no knowledge of when an order service replica is down or that a new replica leader is selected. All the client receives is the order information that it requested.

Deployment on AWS. The steps for AWS deployment can be found in the Evaluation.pdf file which shows the step by step instructions for how the toy store was deployed. The toy store is deployed on an AWS instance such that clients can access the store from anywhere at any time.

Testing

The code contains a test folder. The tests include tests for the order service, the catalog service and the frontend service. The catalog service is tested by querying various items that do, or do not exist and ensuring the correct response or error message is returned. The order service is tested by placing orders and checking that the correct order information or error code is returned, the replica implementation is tested by placing an order on one replica and placing an order check on the other replicas and ensuring they all have returned the same order information. The frontend service is tested by sending various buy and query requests for items that do or do not exist or do or do not have enough stock, and ensuring the correct order information or error code is returned. The transparency and redundancy is tested through the final test where, halfway through the test, the user must manually kill the leader replica and the test checks whether the client was able to notice a difference in the response.

Improvements

One improvement I could have made to the project would be to use the cache to check on orders before sending them to the order service. The order service queries the catalog to see if there is enough stock of the toy before placing the order. I could have had the front end perform the stock check against its cache before sending the order to the order service. Then, if an item is out of stock in the cache, the front end never has to send the order service the buy request which can reduce the response time for buy requests.