COMPSCI-677: Lab 3 Design Doc

Will Lillis and Pranav Shekar

May 1, 2023

1 Overview

We plan to implement a Two-Tiered Stock Bazaar using a microservice architecture. The microservices implemented will be the Front End Service, Catalog Service, and Order Service. The Order Service utilizes replication, and the frontend utilizes caching to service certain requests. All microservices utilize Python's builtin threading library in order to service requests concurrently. The entire application is dockerized and then deployed to an AWS ec2 `t2.micro` instance.

2 Implementation

2.1 Frontend Service

The front end service is the only way clients can interact with the system. The front end microservice will expose three interfaces to clients in the form of HTTP URLs. A threaded flask server will handle incoming client requests on a thread-per-request basis. As the clients are only supposed to be able to lookup information on stocks, buy/sell stocks, or enquire about past orders, the three exposed interfaces will be the `Lookup`, `Order`, and `Get Order` interfaces. Upon accessing the `Lookup` interface, the front end microservice will first check if its local cache is populated with the necessary information. The cache is implemented using a dictionary object. This was done for the sake of simplicity, as using a dictionary allowed us to access the cache in the same way the catalog service accesses its in-memory market data structure. If the data is found in the cache, the front end immediately returns it to the client. Otherwise, it issues a gRPC `Lookup` call to the Catalog service. The cache's consistency is maintained via a server push technique: every time the catalog service processes a valid `Update` call, it issues a cache invalidation HTTP request to the front end for the affected stock. Whether the information comes from the cache or the catalog service, the front end service completes some post processing and returns the data to the client as a JSON response. In the case of an error, a gRPC error code is returned, which is then translated to an appropriate HTTP status code to pass to the client. Using this method of error passing seemed like the most straightforward method, as it simply utilizes an included feature in the gRPC framework that we are already using. We decided to implement the cache via an in-memory data structure in the front end instead of adding an additional caching service. We decided to implement caching in this manner for performance reasons, as we assumed that the entire point of caching in this case was to decrease the response time for a given `Lookup` request. We decided to keep the cache local in order avoid the additional network overhead that would be incurred if the cache was kept on another process, which would have to be accessed by the front end via a gRPC call.

Upon accessing the `Order` interface, the front end microservice will issue a gRPC `Order` call to the leader replica (more on this in the `Order Service` subsection) of the Order service. If this call fails, the front end then initiates a new leader election and re-issues the call after finding a new leader. Upon the completion of the order call, the front end will complete some processing on the returned data, and then hand it off to the client. To the client, there is no difference between the call being processed on the first try compared to a call failing, a leader being reelected, and the call being completed on a retry.

Upon accessing the `Get Order` interface, the front end microservice will issue a gRPC `GetOrder` call to the Order service. If this call fails, the front end then initiates a new leader election and re-issues the call just as it does with failed `Order` requests. Upon receiving a reply, the front end will complete some post processing on the returned data, and then hand it off to the client.

In addition to client-facing functionality, the front end also fills other tasks for the application. As mentioned above, the front end handles detection for a downed order replica leader. If the front end detects that the leader is down, it initiates a leader election and informs all replicas of the results via a `signal_order_leader` gRPC call. This down detection is completed only by checking if a gRPC call to the Order service leader replica failed. We decided on this over a polling approach because it was much simpler and didn't introduce additional network traffic.

The front end also implements a `health_check` gRPC call, in which it can check if an order replica is currently "up."

2.2 Catalog Service

The Catalog service manages the stock market in our system. It keeps both an in-memory record of the market, as well as a backup on disk (a `.csv` file). The in-memory market is stored using a dictionary object. We decided to use this, as it allowed for straightforward access to various stocks and their attributes by name, rather than enforcing an indexing scheme on a less flexible object. On the start of the service, we will check if an appropriate market `.csv` file exists. If so, we will load in the market using the contents of the file. If not, an in-memory market is created with default values for all stocks. A new `.csv` file is then created and the market's starting contents are written in. Access to both the in-memory and disk market records are protected by a single mutex to prevent issues related to concurrent access.

The catalog service exposes two interfaces: `Lookup` and `Update`. The `Lookup` interface is accessible from the front end, simply taking in a stock name and returning information (price, quantity available, current trading volume, max trading volume) for said stock. The `Update` interface is accessed by the Order service. Upon receiving an update request (buy/sell action of a particular quantity for a single stock), the `Update` interface first grabs the mutex protecting the market dictionary object, ensuring exclusive access. It then makes the necessary changes to the in-memory stock record, reflects them in the `.csv` file, releases the lock, and returns back to the Order service. The `Order` service then handles the rest of the logic and returns its response to the front end. At this point we note that the `Update` interface performs no checking regarding whether a given update request is valid (i.e. a trade doesn't exceed the max trading volume). Without completing these checks itself, the catalog service depends on the Order service to do so. We decided to implement `Update` in this manner based off of feedback from Lab 2. Additionally, it makes the application more modular and organized.

We decided to have the service update the on-disk market record directly after completing the transaction in memory. We did this since it guarantees that any transaction returned to the client will be reflected in persistent storage, thus avoiding an potential conflicts that could arise over differing records.

The catalog service also works to maintain the front end cache's consistency. Upon completing a valid `Update` request, the catalog service issues a `invalidate_cache` call to the front end for the affected stock in the form of an HTTP request. We had first thought about creating another endpoint through gRPC for invalidating the cache, but that would involve having the frontend host both a gRPC server as well as the Flask app, and decided that a simple HTTP request through Flask resulted in a much cleaner implementation.

2.3 Order

The Order microservice takes in all `Order` and `GetOrder` requests from the front end. It exposes the `Order` and `Get Order` interfaces in order to allow this functionality. In addition to simply completing orders, the service also commits every valid transaction to an on-disk log (again, in the form of a .csv file). We decided to enact this logging immediately after the order is completed in the market record (via the `Catalog` service's `Update` interface). Therefore, the transaction's information is recorded before any information is returned to the front end service (and in turn, the client). This decision was made because it ensures that information stays consistent between the clients and the system. We chose this approach over periodic logging since a transaction that was completed in the market record and returned to the client may not be recorded in the log in the event of a service going down.

The `Order` interface first issues a `Lookup` request to the catalog service, and utilizes the returned information to determine whether a given order request is valid. If an issue is found, an error is returned to the front end. Otherwise, an `Update` request is issued to the catalog service. Finally, the transaction is committed to the on-disk log.

The `Order` microservice utilizes replication to increase its fault tolerance. At the application's start time, a single replica (out of three) is selected as the leader. The leader replica is the only replica that communicates with the front end, making the replication mostly transparent to the front end service and completely transparent to clients. After servicing an `Order` request, the leader replica then instructs its follower replicas to commit the same information to their own logs through a gRPC request. If the front end detects that the leader replica is down, another replica is selected to take its place. In the event that a replica (leader or follower) crashes and is restarted, it contacts the other replicas via gRPC calls in order to gain the transaction logs it missed during its downtime.

2.4 Proto

We plan to create a separate folder for the proto file and its generated information so that it can be accessed from the micro services in the same fashion. We made this decision to not have to recreate all the generated proto files

multiple times in each microservice folder. This created some difficulty when creating Docker images that will be mentioned in the next section.

For the gRPC rpcs, requests, responses, we decided to base our structure mostly similar to our implementation of Lab 1's RPCs with slight modifications following the lab's instructions.

3 Docker

We decided to dockerize our application in order to ease the deployment onto an AWS instance. Using the docker-compose and Dockerfiles from the previous lab, there was very little we have to do to continue support for our application with Docker.

When dealing with the replicas of Order service, we chose to create three separate services instead of utilizing the `replicas` in docker-compose. This is because there seemed to be no way for each of the replicas to have unique ports assigned to them, where having three separate services defined made assigning ports and replicas fairly straightforward.

Using docker also eased a few other problems that we wanted to tackle. First, we wanted to ensure that services started up in a specific order, with catalog first, then the order replicas, and finally frontend, and this was easily managed with the `depends-on` attribute in docker-compose. Secondly, we did not want to have to manage multiple ssh sessions into the ec2 instance, one for each service, so using docker-compose to have one session control the entire application made managing the application on the instance simple.

4 Outside References

The following outside references were used to complete this project:

- HTTP status code reference
- gRPC error code reference
- Adding files with docker
- Command line arguments
- Python unit testing
- Killing processes via listening port
- gRPC Repeated fields to recover multiple missing logs in a single call
- Setting up docker compose on the aws instance
- Shell scripting basics