VRIJE
UNIVERSITEIT
BRUSSEL

CLOUD COMPUTING REPORT

Gérard Lichtert
0557513
gerard.lichtert@vub.be

January 1, 2025

Sciences and Bio-Engineering Sciences

Contents

1 Architecture and choices

Prior to beginning with the actual components of the project there is some information that should be mentioned. All but the Matching engine use TypeScript, however it is transpiled to JavaScript when containerized and all components but the Exchange Dashboard use Fastify for http communications. I chose Fastify because it is one of the fastest http libraries available in Node and I wanted to get familiarized with it. I make use of a MySQL database to move the state from stateful applications to the database as well as keep track of the state of the orders.

1.1 Client Gateway

For the Client Gateway, as mentioned earlier, I use Fastify to create a http server. This libary also allows for requests to be pre-validated as they come in using fluent-json-schema, denoting the required types and other requirements of the incoming request and returning a Bad Request response otherwise. The requests that are validated are forwarded to the Order Manager. Note that when the cluster is deployed, this application is scaled horizontally and is behind a load balancer.

1.2 Order Manager

To implement the Order Manager I made use of Fastify as my http server and I also make use of mysql2/promise, which is a library allowing me to connect to a database and execute queries asynchronously. When the Order Manager receives a request, it strips and inserts the order in the database. With the returned secnum of the order, the order is then forwarded to the Matching Engine as well as the Market Data Publisher. When the order is saved in the database, it returns a 201 response to the gateway, which returns the same status to the client. This application can also be horizontally scaled as its state is now delegated to the database.

1.3 Matching Engine

Just like the previous applications, the Matching Engine also makes use of the Fastify library to run a http server. This application is implemented in JavaScript because otherwise I had to do type gymnastics due to the Engine being implemented in JavaScript. The matching engine receives orders from the Order Manager sends it to a Subject, which in turn executes the order in the Engine, matching it with existing orders. Whenever orders are matched, it collects the executions, grouping them by secnum, updates the orders in the database and sends these 'executions' to the Order Manager, which in turn forwards them to the Market Data Publisher. Since there is only one instance of the Matching Engine running, it has a recovery system, should it crash.

It retrieves the unfullfilled orders and places them back in the subject, whilst keeping track of the

orders currently in the Engine. Should an order be processed twice, it will not impact the state of the orders, since there's a constraint on the database keeping the amount left on the order in check.

1.4 Market Data Publisher

The Market Data publisher also uses Fastify to host a http server. It also uses it to host a websocket connection using socket.io for real time updates to the Exchange Dashboard. I chose for socket.io as it allows for multiple clients to be connected to the publisher The Publisher holds rooms for each symbol of the stock, and thus emits events to their respective rooms. Whenever the publisher receives either a request for an order or an order-fill it sends it to a Subject, which checks the structure of the content and then emits the corresponding event.

Note that when a client connects to the publisher, it queries the MySQL database for the current state of the engine for a specific symbol of a stock and then returns the data, along with the average per minute. The Publisher is also capable of querying the database to "restore" the status of a client, should it become inconsistent with the engine.

1.5 Exchange Dashboard

The Exchange Dashboard is different from the other applications, as it does not use Fastify to host a http server. Instead it is a React application. I chose React as it is a good choice for dynamic front-end applications. React allows me to use charting libraries to create charts and update them dynamically, as React re-renders the page efficiently using the virtual-DOM. To prevent the application from re-rendering everything I used some React hooks to optimize the re-renders and isolate them to their components respectively. The client side creates a socket connection with the publisher and subscribes itself to a symbol. Once the client is subscribed, it receives events from the publisher and updates its local data. Additionally, should the local data get corrupted, it self-heals and corrects incorrect data by requesting correct data from the Publisher.

Furthermore The Exchange Dashboard is hosted on a Nginx server as this came recommended.

2 Deployment instructions

To deploy the cluster, first you need to build all the containers. For this you need to have docker-desktop running on windows or equivalent on other operating systems. For ease of use and to not build each container individually I created a docker-compose manifest. To use this execute the following command: **docker-compose build**. This should build all the containers. Afther it finishes we can start deploying the cluster.

To deploy the cluster, I wrote a series of manifests located in the /k8s directory. It also includes

manifests for automatic scaling. Whilst you can apply them all in one go I recommend applying the mysql manifest first, as the Engine will crash otherwise till the database is up and running. To do this first run: **kubectl apply -f ./k8s/1.mysql.yaml**. After running the command, wait about $\approx$ 30 seconds, and then apply the rest of the manifests.

You can apply the rest of the manifests using **kubectl apply -f ./k8s/**. Once the cluster is up and running, you can specify the requests per seconds in the client_order_streamer, and run **npm install** within the client_order_streamer directory. Once the streamer is ready you can run the streamer with **node ./client_order_streamer/main.js**. You can check the results of the cluster in real time at http://localhost:8080/