Project Software Documentation

End-To-End Development and Analysis of a Production Day Trading System

SENG 468

University of Victoria

Alwien Dippenaar
V00849850
B.Sc Computer Science
alwiend@uvic.ca

Cam
V00877439

Braydon
V00741364
B.Eng Software Engineering
braydonb@uvic.ca

Professor: Stephen W. Neville

Table of Contents

Project Plan

Create a day trading system that will support a large number of remote clients through a centralized transaction microservice processing system. Each of the clients will log in through a web browser and then perform a number of stock trading and account management activities, this includes but not limited to the following:

- View their account information
- Add money to their account
- Get a stock Quote
- Buy a number of shares in a stock
- Sell a number of shares in a stock they own
- Set an automated sell point for a stock
- Set an automated buy point for a stock
- Review their complete list of transactions
- Cancel a specified transaction prior to its being committed
- Commit a transaction

With all these transactions taking place synchronously, logging all transactions for the system and for each user is essential for audit trails and account summaries.

The technologies that we will use to implement the day trading system are docker which will be used to build the system, MySQL Database, C# with Visual Studio for development.

Milestone	Expected Finish Date
Run 10 user workload	Feb. 7th
Run 45 user workload	Feb. 11th
Run 100 user workload	Feb. 25th
Run 1000 user workload	Mar. 10th
Run final workload	Mar. 21st
Web Client	Apr. 1st
Database backup	Mar. 21st?
Caching Service	Mar. 3rd
Load Balancing	Mar. 3rd
AutoRecovery	Mar. 27th
Final Documentation	Apr. 1st

Initial Requirements

The client will interact with the web server which will in turn interact with the microservices within the transaction server which will interact with the database and quote server. All major actions that occur on the system will be sent to an audit server.

Adding money to an account:

The Add command will be implemented by creating the user if they don't exist and then adding money into their account using an Update statement. Concurrency the user's money will be handled by the database.

Get a stock quote:

The Quote command will be implemented by checking a cache before connecting to the quote server to keep costs low.

Buying and selling stocks:

The buy and sell commands are each accompanied with a 60 second timer that will validate the transaction, if time has elapsed for longer than 60 seconds then those transactions are voided and the money/stock value held for the transaction is returned to the user's account. To initiate a buy, the user must have greater than or equal to the amount of stock they wish to buy in their account, the same goes for selling stocks.

Triggers:

After a trigger is initiated, the user must cancel and remake the trigger in order to change the values.

Two ways of solving the trigger business requirements were considered. Once the number of triggers are beyond the scale that a single server can process, it is necessary for the triggers to be distributed. One architecture involves storing all triggers in the database and loading them back onto the trigger servers periodically. However, it was thought that this would either require a table scan every time a trigger server wants to process triggers, or trigger server assignments would have to be precomputed. The former would limit scalability and the later might limit the flexibility of assigning triggers.

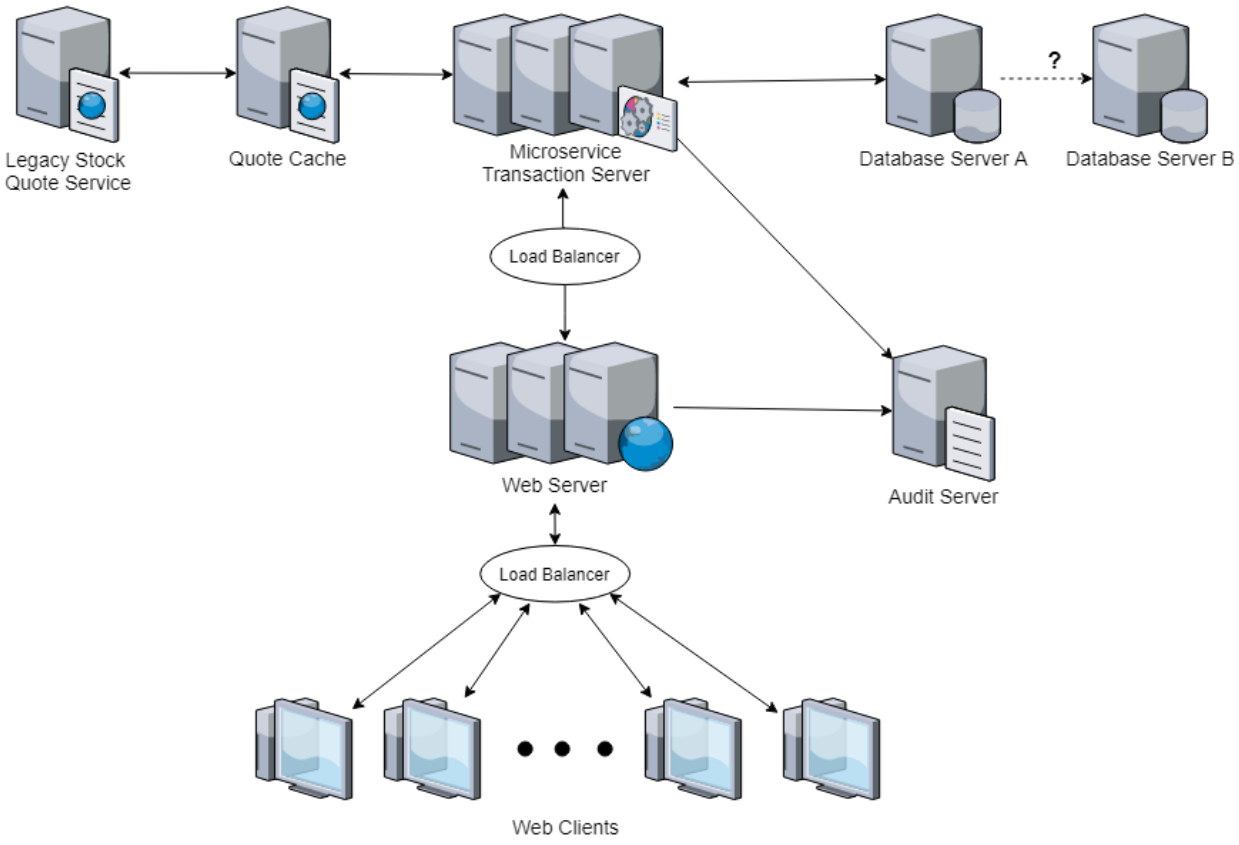
Ultimately, it was decided that each trigger server would store its triggers locally and only update the server of individual trigger changes so triggers could be reloaded in the event of a failure. This means every request for a particular trigger must be routed to the same trigger server. This will be implemented in the load balancer.

A Trigger server will be implemented as a socket server that takes requests to update triggers. Periodically, a background thread will active the triggering process, which will result in the trigger server requesting quotes and completing triggers that have passed their threshold value. When a trigger is updated or being processed, it will be locked, so any other thread that attempts to use it must wait until the lock is released. This will prevent issues like a user receiving confirmation that a trigger was canceled and in the meantime, the trigger being activated and completed.

Account Info and Transactions:

The information required in these commands can be retrieved from the database using simple SQL Select queries. They will then be formatted by the web server.

Architecture



The system will consist of the following components, an SQL database, a microservice transaction server, an audit server, web servers, a web client and load balancers were appropriate. For testing, a multi-container work load generator system will be developed.

Web Server:

The web server will provide an api for the workload generator to send requests to and will generate a human-usable webclient using HTML, CSS and JavaScript. The web server's main responsibility is to do basic request content validation, mapping requests the appropriate transaction microservice and generating HTML pages.

Transaction Microservices:

The requests from the web client will be sent to the appropriate transaction microservice server for processing. When different microservices rely on the same data (eg. the amount of money in a user's account), that data will be stored in the database and updated via transactional queries to ensure consistency and correctness. To scale, multiple instances of each microservice will be run. The following transaction microservices will be available:

- Add Service
    - Adds money to users accounts
- Quote Service
    - Gets a quote for a stock
- Buy Service
    - Handles buy and sell requests
- Trigger Service
    - Handles setting and executing triggers for buying and selling based on stock price thresholds
- Account Service
    - Creates summaries of user account info and transactions

Database Server:

A MySQL server with tables track money, stock amounts, transactions, triggers, etc. To increase the durability of the system, a backup database will operate that mirrors the contents of the main database. To increase scalability, a read replica can be setup to offload some processing to a secondary system.

Audit Server:

Each server will be setup to send a record of its activities to the Audit Server.

Workload Generator:

The workload generator will provide inputs to the system for emulation testing of the other systems. The workload generator will be a system that works over multiple containers. Each individual generator will

have the same workload file and a set of users. The generator will filter the workload file for commands that contains its users and run those commands in sequential order for each user.

Errors:

When a service encounters a problem (eg. network error, insufficient funds or shares), the system will revert any changes it has made and send an error response. In the interest of time, it will not necessarily be visible to the user what type of error occurred, only that the request was not processed correctly. For this reason, database changes should generally be made before changing the local copy of that state. Local state interactions should be sufficiently tested such that they do not cause errors.

Networking:

The web server will use HTTPS for secure communication with the web client and workload generator. Inter-server communication will be done using sockets and JSON serialization with the exception of database connections. Internal servers will not accept connections from computers outside of our network. We acknowledge that user authentication is an issue that would need to be solved in a production system, but it is not expected to be a major performance bottleneck and will not be included in the prototype.