Group A – Project Documentation

Table of Contents

# [1.1 Web Client 1](#_Toc441774863)

# [1.2 Workload Generator 1](#_Toc441774864)

# [1.3 Web Server 2](#_Toc441774865)

# [1.4 Transaction Server 2](#_Toc441774866)

# [1.5 Client/Transaction Database 3](#_Toc441774867)

# [1.6 Project Webpage 3](#_Toc441774868)

# [1.7 Auditing/Reporting Database 3](#_Toc441774869)

# [2.1 Workload Generator & Load Balancer 5](#_Toc441774870)

# [2.2 Web Server 5](#_Toc441774871)

# [2.3 Transaction Server 5](#_Toc441774872)

# [2.4 Trigger Manager 5](#_Toc441774873)

# [2.5 Transaction Database 5](#_Toc441774874)

# [2.6 Cache 6](#_Toc441774875)

# 1.0 Functional Requirements

## 1.1 Web Client

Implement in HTML5

Required functions

Note: web client implementation can be left until the end of the project.

Adding a user

In the tool bar at the top of each page is a "Sign Up" link that leads to a page where one can enter their desired user name.

Cookie to store logged in username?

Logging in (no authentication)

In the tool bar at the top of each page is a "Log in" link that leads to a page where one can enter their user name.

Again, cookies?

Account summary

Show transaction history, status of accounts, current buy/sell triggers

Getting a stock quote

Only give access to quote page if logged in (most services would require this).

Creating buy/sell order

Show account balance and owned stocks. Inputs for stock name and amount (in $) to buy/sell.

Confirm buy/sell order

Buy/Sell page could have a confirmation box on another layer that appears when the web server gets the quote, has a countdown timer, and initiates a re-quote if the timer runs out once. Finally, give an option to cancel the trade.

Create Buy/Sell trigger

Similar to buy/sell order page, with the addition of the trigger threshold.

Admin functions?

Perhaps have a special admin user that adds the dump commands to the user page? It's one way to trigger the generation of the logs. Full dump only supposed to be run from an admin account.

## 1.2 Workload Generator

Description

The workload generator was initially written in Python due to group familiarity with the language. Performance was not considered to be a key factor at this stage of development. Devising a functional system was the primary focus.

Advancements in the code may take the form of translation to C, and adding threads to parallelize the commands (keeping each users' commands in order).

Method of operation

The workload generator opens the provided workload file, reads each line sequentially, builds a command to send for processing, and connects to the web server to simulate a form submission from a web client.

An alternative being considered is to have the workload generator communicate directly to the transaction server.

## 1.3 Web Server

Takes commands from the workload generator or web client, sends to the transaction server for execution, and forwards the result to the web client.

## 1.4 Transaction Server

Accepts commands from the web client, executes the required operations to fulfill the command.

Example:

The user executes a QUOTE command. The transaction server would:

Receive the command from the web server

Contact the quote server for a quote

The received quote would be sent back to the web server to be sent to the client.

Report the quote request to the database for logging

The transaction server will make use of connection pooling to efficiently pass commands to the database. The initial thought is to have multiple threads each with a connection that can consume requests from a queue maintained on the transaction server. The difficulty will most likely lie in maintaining a paper-trail to return the request to the web client.

## 1.5 Client/Transaction Database

Maintains records of user accounts and transactions, and responds to requests for information.

All incoming requests are tied to a username. If a user does not exist, a record will be generated for them and then the request processed.

User account table: user name, account balance

Transaction table: user name, action (buy/sell), stock name, number of shares, share price (and/or crypto-key from quote server).

If user names are not unique then they will need to be given a unique identifier when they enter the system. This can be accomplished with a randomized number from a large number set.

## 1.6 Project Webpage

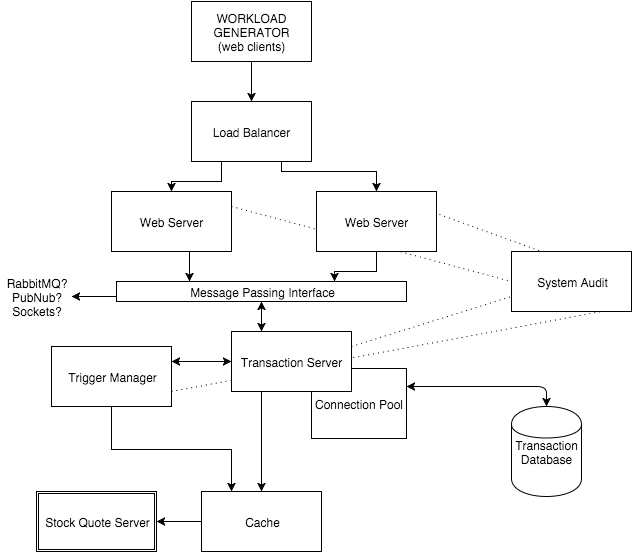
The team project webpage can be found at:  https://seng462project.wordpress.com/

This page contains a link to the codebase, as well as blog posts with updates on the status and performance of this project.

## 1.7 Auditing/Reporting Database

Receives information from all servers about received requests such that a single request can be tracked all the way through the system.

# 2.0 Architecture Analysis

Figure 2.1 below shows the basic structure of the system. For the purpose of this project the workload generator will replace the web client. The workload generator will feed the workload files directly into the web server. The Web Server will run basic sanitization on the input by ensuring the right number of parameters for a given request.

**Figure 2.1: First Architecture Diagram**

The web server will then pass the requests to the transaction server through a basic socket interface. The Transaction server handles the final sanitation of the request before executing and delegating the connections to the trigger manager, transaction database or the quote server (cache).

## 2.1 Workload Generator & Load Balancer

The workload generator is written in python to parse the workload file and submit requests to the web-server. The final goal for the workload generator is to parse the workload files and separate commands by users and hand them to the load balancer. The load balancer would then be responsible for starting threads to read the commands directly into the web servers.

## 2.2 Web Server

Our initial web server is written in python using wsgi to interface with apache. However, our target implementation will use a basic C server that uses socket communication to communicate with the transaction server. The web server will directly consume from the workload generator (or indirectly though the load balancer once multiple web servers have been implemented). The input read will be validated for the proper number of arguments before it is passed through the socket interface to the transaction server.

## 2.3 Transaction Server

The transaction server will be performing the most complex computations and should therefore be the most efficient. The target implementation will be written in C, and will accept the messages passed via socket interface from the web server. These messages will then be processed, with potential calls being made the transaction database, trigger manager, or stock quote server cache. Any actions will also be forwarded to the System audit to be recorded.

## 2.4 Trigger Manager

The trigger manager is less sensitive to inefficiencies as it will simply be periodically pinged whenever stock prices are updated in the cache. When a stock price changes, the trigger manager will be notified so that it can initiate any triggers that match the new price level. The trigger manager will also be communicating with the transaction server to actually complete these actions, as well as to receive any new trigger requests from customers (via the web server).

## 2.5 Transaction Database

The transaction database is where all records of user accounts, transaction histories, and triggers will be stored. Every request will be hitting the transaction database with at least one write or read, so it is essential that the database is able to handle multiple connections as efficiently as possible. Postgres provides the best balance of speed, stability, and features that would enable us to handle this level of traffic through an SQL database.

## 2.6 Cache

The quote server is both slow and expensive to access, so we will be implementing a cache to reuse quotes for as long as they are valid. The cache will likely be built into the Transaction Server, and the implementation is still to be determined. So far our discussion has come up with 3 cases where a cache will help improve efficiency.

Case 1) Quote first then set a trigger, trigger then doesn't need to poll for the price for the next ~60s

Case 2) Trigger first then buy/sell/quotes after it has been set, which will have to do a quote because of the business logic, this updates the cache and the trigger server is good for the next ~60s

Case 3) Buy/Sell then more quote/buy/sell on the same stock before a commit buy/sell.

# 3.0 Distribution Analysis

//To be decided at a later date

# 4.0 Stability Analysis

//To be decided at a later date