Mqtt\_sc\_game

**What is the Supply chain game?**

The supply chain game is a game based on the beer game where players take control of one of a number of firms in a supply chain. This ranges from the lowest who supplies raw materials to the finished product. One thing to take note is that in this game a tier (if it is not the lowest) get's products in the form of raw materials from the tier right below it in terms of the supply chain. This firm will then use these materials to produce their finished goods which will then be passed on to the next tier(where these finished goods will become that tiers raw materials).

Each of the tiers will have 3 functions:

\*produce

\*order

\*deliver

These are the core functions the firms will use to interact with each other. Firm interaction will take place on a turn to turn basis where the length of the turns will be dictated by the admin(teacher/game coordinator). During these rounds, the player will be able to produce, order and deliver to and from other tiers. This will be followed by a 20-second break where the user will be allowed to get the deliveries from other tiers.

The game will continue to run until the admin quits the game. All the transactions made in the game will be captured in a MySQL database and will be able to use for data mining and analysis. A particular interest lies in the form of lean management in terms of how much stock they have at any given time as well as backlog and open deliveries. Using this data questions like the bullwhip effect can also be looked at.

**Game design**

**Requirements analysis**

**Functional requirements**

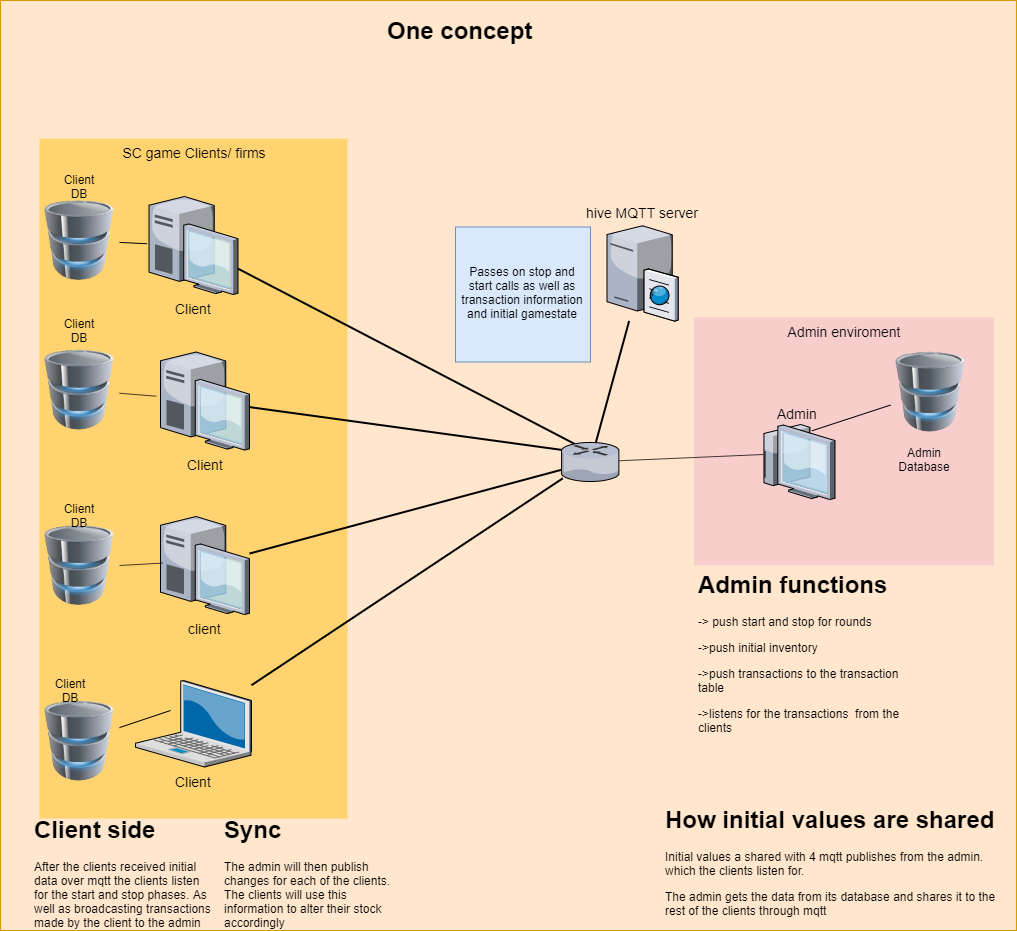
* Players should be able to log in as their assigned tier(firm)
* Players should be able to order raw materials from their suppliers each turn.
* Players should be able to deliver to their finished goods to customers(another tier) each turn using an RFID tag.
* Players should be able to produce products each turn, given that the user has enough production time as well as raw materials.
* Players should be able to get a delivery from another player using an RFID tag to write the delivery information.
* Players should get hint messages if they input incorrect data types or if they lack time or resources for a given action.
* Administrator of the game should be able to set the initial state of the game and how many products each of the users.
* Administrator should be able to set the game round length in seconds.
* Admin should be able to stop and start rounds.

**Non-functional requirements**

* Clients on the system should be able to communicate with each other without data loss.
* Orders production should be fast and responsive.
* System should be stable for at least 30 rounds.
* System should check user input to avoid crashes.
* System should pass along information.

**Game architecture**

The game's client communication architecture is built on a the hive MQTT framework in python. The for some purposes the clients will never interact directly with each other but will publish the needed information to the broker then the admin will gather information from the client pi's and distribute that information based on which client information is intended through a publish to the broker.



**Data Design**

The bulk of the data for the game is stored in a MySQL database. The game is set up in a way that each of the clients has their own database to write mission-critical data. The admin database on the other side contains data that will affect the initial starting state of the game and will function as both a main product and transaction ledger for the game. This means it will house all the products in the game as well as all the transactions that have happened.

**Client/Firms or tiers**

This data includes operational data such as what has been produced when the player produces products, what has been ordered by the player as well as what has been delivered by that tier to others. Further information that is added to that database includes the stock of that tier as well as the current amount of open orders and backlog the tier has. More general information also contained is a game table which tracks the unique games that are played and the round table which has a similar function to the former but instead has a many to one relationship to it since one game can consist of many rounds. Last to be mentioned but maybe the most important table would be the products. This table contains a list of products that is assigned to that specific tier. The information for this table is received from the administrator of the game at the beginning of each new game. A problem that arises at this point would be the fact that each time a new game starts the client would see that the product data has been already added to the database. Rest assured that the database does not allow for duplicates so it will just skip the addition of said data.

**Administrator**

As specified above the administrator keeps data concerning the starting state of the game itself.

This data includes*:*

* A full list of products that exist in the game(overall of the tiers)
* The production times of different products for each tier.
* Data concerning what prerequisites are needed to produce certain products
* The initial amount of products that each tier starts with.
* The round length of each game

Other than the initial state data the Administrator also acts as a transactional ledger. This means that the Admin's database houses a table that is a collection of all the transactions that happens within the game.

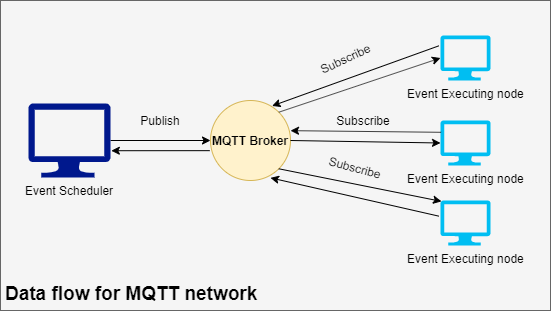
Transactions meaning:

* Production information when each tier produces products.
* Information concerning orders from tiers to others
* Lastly information from the deliveries of products from one tier to the next

One thing to note is that one must manually add the product information in the specific format stated otherwise the system will have problems and become unstable.

**MQTT Architecture:**

**Better description will be added**

****

**Further additions:**

* **Data design:** One improvement on the system will be how the data is packaged and sent from one client through the broker to another. The current way of how data is sent is making use of a class that contains one variable with an array attribute. The class is invoked whenever data needs to be sent and then the array is filled with a matrix of data. This data can range from data concerning product information to orders and deliveries.

Once the data is in the data matrix, the object will be converted into a JSON package. This is needed since one cannot send arrays over an MQTT message.

When the MQTT message is received by the user of that data, a function will decode it into usable information using the object class that is used to package it into a JSON package.

In the current version of the game, I opted for the use of this matrix structure for the reason that it is quite versatile and can handle a lot of different information without needing multiple different classes for each of the various operations. In this sense, the data processing is done on the side of where the data ends up by an appropriate function.

The idea is to make many different classes for each different function case when it comes to packaging up data and sending it over the network. This will help with adding a better description of how this information is broken up. This will also allow the system to appease better OOP standards.

* **Multi for everything:**

The sentiment to have everything run on its own thread is echoed by the fact that some operations may cause others to be interrupted having the game then hanging and not knowing what to do. In reality, this will make the game freeze and unusable which will cause data loss and having to re-run the whole system instead of just that single client.

* More will be added later