Lab #3
COMPSCI 677 - Fall 2022

Due 12/8/2022.

**Asterix and Multi-Trader Trouble: Replication, Fault Tolerance and Cache Consistency**

**General Instructions:**

- This project aims to familiarize you with some of the important canonical problems you have studied in this course-- replication, fault tolerance, and cache consistency.
- You may work in groups of two for this lab assignment.
- You may find that the evaluation criteria are less rigid than you're used to from other programming assignments. This is because ultimately we're just trying to make sure you've adequately grappled with and understood the concepts above, and do so in a way that mimics a research environment appropriate to a 600-level course. Therefore, you can be creative with this project. You are free to use any programming languages (C, C++, Java, python, etc) and any abstractions such as sockets, RPCs, RMIs, threads, events, etc. that might be needed. You have considerable flexibility to make appropriate design decisions and implement them in your program: just be sure you can justify your decisions to your graders.
- Ensure that peers do not share memory — if you're running on the same machine, peers should run in isolated processes and communicate as if they were running on different machines. For example, the python `threading` module does not meet this criteria, but the `multiprocessing` module does. Note, however, that you may use `threading` *within* a peer, to support, for example, its messaging protocols.
- Feel free to make use of online documentation for any languages or libraries you make use of, however, be sure to run any non-standard libraries you use by us, and be sure to package and include them in your submission if necessary. You should also be clear about what tools you make use of in your write-up.
- You should read this entire document before you begin: you'll be expected to document parts of your process of completing this assignment, so you'll have to know which parts before you complete them.

**A: The problem**
***A1: The story (useful for contextualizing) — Sequel to Lab 2***

The trading post model served the Gauls very well at the beginning. However, with the growing popularity of the market, the Gauls found that the trader became the new bottleneck. Sellers and buyers needed to wait for a long time to be served. Vitalstatix, the village chief, decreed that henceforth trading posts shall be replicated — there shall be multiple trading posts.

In the new market model, there will be one trader at each post. They shall use a common warehouse where all items for sale are stored. Any item that is sold is shipped from the

warehouse, back to the trader, and handed off to the buyer. New items that are offered for sale are added to the warehouse. Any trader can take buy or sell requests for any item. Sellers can deposit their goods with any trader. Buyers can buy from any trader. For this assignment, you don't need to worry about buyer budgets or making sure sellers get paid — at this point, the Gauls have attained a post-scarcity economy and don't care much about money.

***A2: General Technical Requirements***

In addition to buyers, sellers, and traders, you'll need a database server in your system — this acts as your central warehouse. You may implement the underlying database as a simple text file(s) that contains the type and number of items available for sale at the warehouse. The database "server" is a process that can access and update this file(s); the traders send the buy and sell orders to the warehouse. The database server provides a service to lookup the number of available items of a particular type and to take buy (decrement) and sell (increment) orders forwarded on from the trading posts.

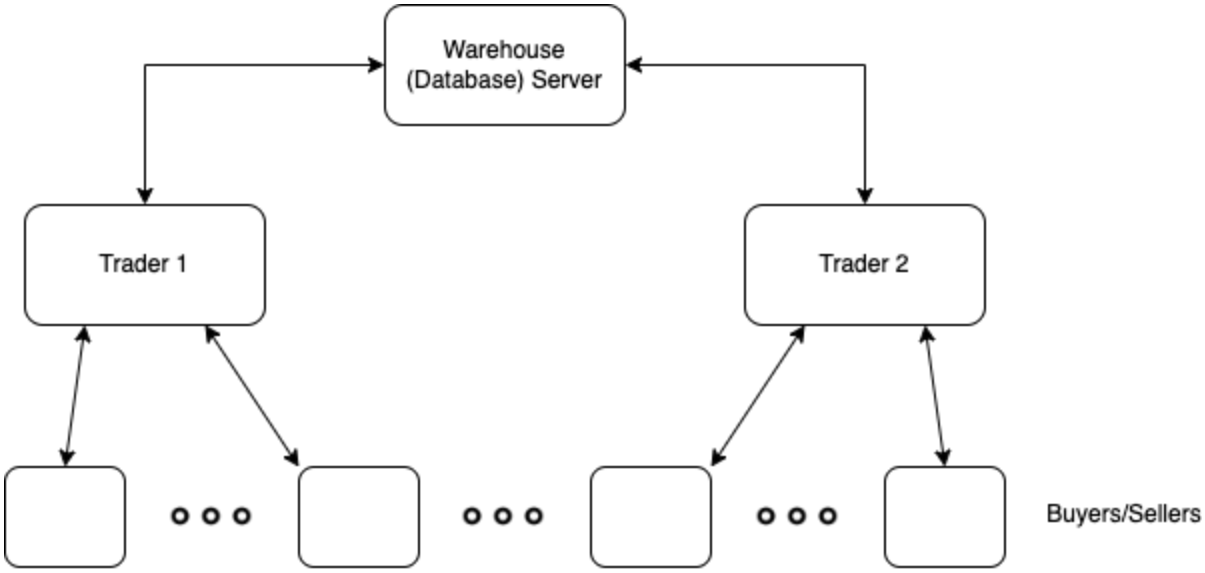
First, spawn the warehouse (database) process, and then a network of $N_p$ peers, where each peer is either a buyer or a seller. For this assignment, it will be easiest to define the number of buyers $N_b$ and the number of sellers $N_s$, where $N_p=N_b+N_s$. Assume that a list of all peers and their addresses/ports are specified in a configuration file.

Once the network is formed, have peers elect one coordinator (i.e. trader) for each trading post. This will result in $N_t$ traders. $N_t$ should be configurable for your system. The elected coordinator does not sell or buy anything once it is elected as the coordinator. In this assignment, traders are appointed for "life" — they cannot resign once appointed. For the "fault tolerance" portion of this assignment, however, they can permanently fail.

Buyers and sellers can randomly choose which trader to do business with.

As in previous labs, each seller should randomly select a particular item to sell. *Unlike* in previous labs, sellers in this simulation should accrue $N_g$ goods every $T_g$ seconds that they can then seek to sell to the trader. You may set $N_g$ and $T_g$ to constant values of your choosing.

The following figure depicts a network with two traders:




***A3: Cache vs. Cache-less***

The Gauls need you to implement both a cache-less, synchronized approach, and an approach where the traders are allowed to cache the warehouse inventory information in local memory. To maintain cache consistency, you may employ any *non-strict* (e.g. sequential, eventual, etc.) consistency approach of your choosing (see lecture 15, from week 8, to review consistency models). This will allow the Gauls to compare the two approaches and make an informed decision as to which implementation to use.

**For the synchronized approach**, you can ignore caching entirely. For every buy request made to a trader, the trader should send a buy command to the database server (i.e. the "warehouse"). In resolving the buy command, the database server needs to perform the following *synchronously*, with respect to the product in question:

```
if inventory is available for the requested product:
   decrement the inventory count by the amount requested
   respond to the trader saying the requested goods have shipped
else:
   respond to the trader saying the requested goods were unavailable
```

The trader should then inform the buyer of whether those goods were available and have been transferred.

You can achieve this synchronicity in a number of ways — for example, with a lock or queue for each product. Unlike in lab 2, you may resolve ordering of concurrent requests arbitrarily. There is no need for logical clocks — you can let the synchronizing lock/queue do its thing.

For every sell request made to a trader, the trader should send a sell command to the database server. In resolving the sell command, the database server needs to perform the following synchronously, again with respect to the product in question:

```
increment the inventory count by the amount being sold
respond to the trader saying the goods were stocked in the warehouse
```

To prevent both under- and over-selling, the buy and sell requests need to run synchronously for each product. Again, this can be accomplished with, for example, either a simple lock or a queue taking in buy and sell requests for a given product.

Again, you do not need to implement an ordering for requests like in lab 2 — you can let your system resolve concurrent requests arbitrarily, as in lab 1.

**For the approach using inventory information caches**, implement a *cache* at each trader that replicates the master inventory state of the warehouse. You're free to choose any of the non-strict consistency models discussed in class. You have considerable freedom to make design decisions here.
In this approach, the trader can use their cache to avoid always needing to synchronously communicate with the warehouse before responding to a buyer.

For example, the trader may use their cache to deny buy requests if the cache shows that no inventory is available — the problem is that the cache might be slightly out of date, and inventory may actually be available in the warehouse at the time of the buy request. This constitutes *under-selling*.

Alternatively, the trader can use their cache to accept buy requests and forward them on to the warehouse — the problem is that the cache might be slightly out of date, and in actuality all remaining inventory may have already been sold and shipped to other buyers. In this case, the warehouse must ultimately reject the buy request as it cannot ship the goods. This constitutes *over-selling*.

As long as the cache-based approach allows their trading posts to function more efficiently, the Gauls are prepared to live with the possibilities of over- and under-selling.

Different approaches to cache consistency can minimize these occurrences. It's up to you to decide which case to focus on minimizing and to describe design tradeoffs you explored.

Note that if you choose to implement *eventual* consistency, you may want to consider a globally and totally ordered ledger of transactions — as opposed to inventory counts — that can be merged across traders and the warehouse. Such an ordering of buy and sell transaction events can be achieved using logical clocks. The warehouse will still need final say on whether a product is oversold.

***A4: Fault Tolerance***

Should one of the elected Gaul traders fail, we would like another trader to take over their trading responsibilities.

For simplicity, execute this part of the assignment with only two traders, and only for your non-strict consistency model.

Implement a simple heartbeat protocol between the two traders, where each trader periodically asks the other one “are you there?” If no response is received within a timeout value, then that trader is assumed to have failed. In this case, the other trader sends a message to all peers indicating it is now the sole trader and all subsequent messages must be sent only to the active trader.

Think about how you might handle buy/sell requests that were in progress when the failure occurred. Any simple technique that allows you to restart such requests at the other trader is fine so long as the end result is consistent (e.g., a restarted buy request should not yield "duplicate" buy requests at the database server).

Note that, to test this, you will need to artificially force one of the traders to fail. Simply having a trader “exit” after a small time period is sufficient.

***A5: Optional Extra Credit Question***: In this optional component, we will use cloud computing to dynamically replicate the trader. Initially, start with one trader (one EC2 server). Monitor the load on the server (in terms of number buy and sell requests seen every minute). If the number of requests exceeds a (small) threshold, have the first trader dynamically start a new EC2 server and run a second trader on that machine. All peers must be informed of this new replica and a peer can randomly choose the trader they do business with. No fault tolerance functionality is needed in this extra credit question. For additional points, think of a design as to how the failure of a trader allows you to dynamically start a different trader on a separate EC2 server and add a small note to the design document as to how you may have implemented such a functionality).

**B. Evaluation and Measurement**

- Run an experiment comparing throughput of the cache-less and cache-based approaches. Assess the average amount of goods shipped, per second, from the warehouse under each approach. Wall time is fine here as long as your execution circumstances are similar — i.e. don’t run one approach when your machine is “busy” in the background, and the other when your machine is idle. Analyze the differences. See if you can identify different settings of your hyperparameters (e.g. number of traders, number of peers, buyer:seller ratio, etc.) that make the performance gains of the cache-based approach larger and/or smaller. Analyze your findings and offer an explanation of your results.

- Assess the rate of over-selling for your cache-based implementation. This can be measured by tracking the number of times a trader forwards a buy request to the warehouse that is ultimately denied by the warehouse due to a lack of inventory. Produce a plot where the x-axis is the ratio of buyers:sellers, and the y-axis is the incidence of over-selling as a percentage of total buy requests. Recall that sellers in this simulation should accrue $N_g$ goods every $T_g$ seconds that they can then seek to deposit with the trader. For this experiment, try to find values of $N_g$ and $T_g$ that make this plot "interesting." Analyze your findings and offer an explanation of your results.
- Run an experiment assessing the throughput of your system after a trader fails, and fault tolerance kicks in by allocating all traffic to the remaining trader. Assess the average amount of goods shipped, per second before and after a trader fails and fault tolerance kicks in.
- Run and report on other experiments you find interesting!

Note that if you are running on a single machine, you may need to limit the number of cores available to each replicated trader in order to emulate what you might expect from each trader running on a separate machine. Note that if you are spawning traders as python processes using the multiprocessing module, this happens by default due to the GIL. Similarly, for the fault tolerance experiment, you'll want to ensure that your remaining trader doesn't end up using all the CPU resources that the failed trader was originally using — this will skew results.

**C. What you will submit**

When you have finished implementing the complete assignment as described above, you will submit your solution in the form of a zip file that you will upload to Gradescope and a PDF document uploaded to a separate Gradescope assignment.

Each program must work correctly and be documented. The zip file you upload to Gradescope should contain:

- .txt files containing the output generated by running your program. Each trader should output to their own .txt file, and the warehouse should output to its own .txt file. Include a timestamp for each event. Events could include, for example: "11.02.2022 16:54:32.10 Trader 1 informed buyer 2 that no Salt was available", or "11.02.2022 16:54:32.10 Warehouse rejected buy request because inventory was oversold".
- Your source code, containing clear in-line documentation.
- A copy of the PDF you'll also be submitting to the other Gradescope assignment. This PDF should include your experimental results and analysis.
- A separate README file with instructions on how to run your program, experiments and test cases.

You'll also be submitting a PDF to a separate Gradescope assignment. Be sure to submit one assignment per group of two, and add group members to that submission. You may want to use this as an opportunity to learn LaTeX if you haven't already: if you ever plan on writing scientific

papers, you'll be using LaTeX to do so. The PDF you upload to Gradescope should contain all the following:

- **Design Description** — No more than 2-3 pages, including diagrams, describing the overall program design and design decisions considered and made. Identify trade-offs. If this document says "do X", you don't need to tell us *that* you did X: instead focus on *how* you did X, if it isn't too straightforward. We're looking for high-level design, such as files, classes, interesting methods, and maybe a couple of code snippets if you're particularly proud of them, plus anything you had difficulty with or didn't manage to accomplish.
- **Possible Improvements** — Also describe possible improvements and extensions to your program (and sketch with words and/or diagrams how they might be made). You also need to describe clearly how we can run your program - if we can't run it, we can't verify that it works.
- **Tests** — On a separate page, a description of the tests you ran on your program to convince yourself that it works correctly. Also describe any cases for which your program is known not to work correctly.
- **Experimental Results** — On another separate ~2 pages, include the experimental results described in section B, and your analysis of them. Include additional research questions and hypotheses about the system that these experimental results don't sufficiently answer. Offer your now-informed opinion on whether the Gauls should use the cache-less or cache-based approaches — defend your opinion. Is there another cache consistency implementation that you suspect might work better for the Gauls than the one you implemented?
- **Optional Separate Machine Deployment** — Finally, if you completed the optional part of this assignment, include your observations on another separate page.

**D. Grading policy**

Program Listing
- works correctly —---------- 40%
- in-line documentation —---------- 10%

Design Document
- Design Description —---------- 20%
- Possible Improvements —---------- 5%
- Tests —---------- 10%
- Experimental Results —---------- 15%

Optional separate-machine deployment ---------- +10%

Grades for this assignment will be capped at 100%. This cap applies before any late penalties.

After using your allotted late days for the course, grades for late submissions will be lowered 4 percent per 8 hours late. See the syllabus for the extraordinary circumstances late policy to see if that applies for your circumstances.