Distributed and Operating Systems

Spring 2019

Lab 1: Bazar.com: A Multi-tier Online Book Store

- **A: The problem**

  You have been tasked to design Bazar.com - the World's smallest book store. Bazar.com carries only four books for sale:

  1. How to get a good grade in DOS in 40 minutes a day.
  2. RPCs for Noobs.
  3. Xen and the Art of Surviving Undergraduate School.
  4. Cooking for the Impatient Undergrad.

  The store will employ a two-tier web design - a front-end and a back-end - and use microservices at each tier. The front-end tier will accept user requests and perform initial processing. The backend consists of two components: a catalog server and an order server.

  The catalog server maintains the catalog (which currently consists of the above four entries). For each entry, it maintains the number of items in stock, cost of the book and the topic of the book. Currently all books belong to one of two topics: distributed systems (first two books) and undergraduate school (the last two books). The order server maintains a list of all orders received for the books. The front-end is implemented a single microservice, while the catalog and order services in the backend are implemented as two separate microservices. In this case, a micro-service can be viewed as a separate process that accepts requests.

  The front end server supports three operations:

  - ***search****(topic)* - which allows the user to specify a topic and returns all entries beloging to that category (a title and an item number are displayed for each match.
  - ***info****(item_number)* - which allows an item number to be specified and returns details such as number of items in stock and cost
  - ***purchase****(item_number)* - which specifies an item number for purchase.

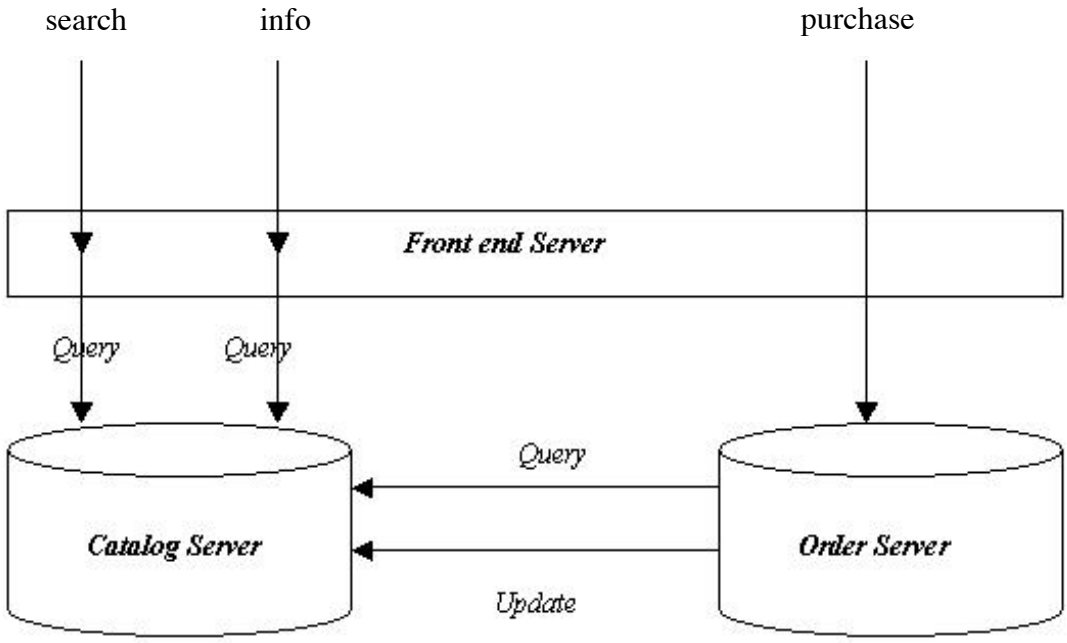
The first two operations trigger queries on the catalog server. The purchase operations triggers a request to the order server.

The catalog server supports two operations: *query* and *update*. Two types of queries are supported: query-by-subject and query-by-item. In the first case, a topic is specified and the server returns all matching entries. In the second case, an item is specified and all relevant details are returned. The update operation allows the cost of an item to be updated or the number of items in stock to be increased or decreased.

The order server supports a single operation: *purchase(item_number)*. Upon receiving a purchase request, the order server must first verify that the item is in stock by querying the catalog server and then decrement the number of items in stock by one. The purchase request can fail if the item is out ot stock.

Assume that new stock arrives periodicaly and the catalog is updated accordingly.

An pictorial representation of the system is as shown in the figure below.



B: How to tackle the problem

You will use a web framework to write the code. We strongly recommend using either Java, PHP or Python for this lab and we also

strongly recommend use of a lightweight micro web framework such as Flask for Python or Spark / Ninja for Java, Lumen for PHP. These are very lightweight web frameworks that are easier to learn than full-fledged web frameworks such as Laravel/Django / Struts etc. We would strongly discourage you from using these heavyweight frameworks since one of the goals of this lab is to write small/lightweight micro-services. Like before, you still have creative freedom to use a different language or a web framework but you will need to discuss your design choice before hand with us.

A second requirement of the lab is to implement the above interfaces for each component as a HTTP REST interface (the above web frameworks have this functionality built into it). For example, rather than implementing *search(item)* using an RPC interface, you should use a REST call of the form

```
SERVER_IP:80/search/itemName
```

or simply

```
SERVER_IP:80/search
```

In the former case, the argument item name is part of the request URL itself, while in the second case, the argument can be passed as a json object as part of the request. In either case, the response should be returned as a JSON object. All interface calls, namely search, info, purchase, query, update should be exposed as HTTP REST calls by each component.

REST Interface As noted above, your system should use a REST client/server architecture. This means creating several endpoints that correspond to the interfaces provided above.

5. A sample endpoint for the *search(item)* interface may be of the following form:

   serverIP:80/search/itemName

6. Clients can make this api call to the server and will receive back a json object of the following form:

```
{
  "items": {
    "RPC for Noobs": 345,
        "Cooking for the Impatient Undergraduate Student": 359
      }
}
```

Your code needs to support concurrent requests. However modern web frameworks have built-in support for accepting multiple concurrent requests and use threads or async processing, and thus you get concurrency for "free" without writing low level threads code.

The order and catalog requests need to maintain data in a persistent manner (i.e., on disk). Normally a web framework will maintain persistent data in a database. Your code should minimally use a simple text file (e.g., CSV file) to maintain the catalog and order log, but it is also fine to use a very simple database such as sqlite. Please refrain from using heavyweight databases such mysql or postgresql / mongodb etc for this lab. Simple text files or sqlite database should suffice.

Your multi-tier application should have the ability to run the three components on different machines in a distributed fashion. You can achieve this by implementing each component (e.g., front-tier, order server, catalog server) as a separate flash micro-service that communicate with each other using HTTP REST calls.

You will use github for a source code control repository. Please make sure you use multiple commits and provide detailed commit messages. You will be evaluated on your use of effective commits. One of the goals of the lab is to teach to you properly test distributed code.

We do not expect elaborate use of github rather we want you to become familiar with these tools and start using them for distributed programming (or your own work.)

No GUIs are required. Simple command line interfaces are fine.

C. Evaluation and Measurement

This lab uses a client-server model, and the server itself uses a multi-tier and microservies design.

- Deploy your system on three virtual machines, you can use any virtual machine software you prefer: VirtualBox or VMWare. Create three virtual machine instances where each instance runs one of the 3 services you will develop. It is preferably if your virtual machines where linux based (Ubuntu for example) Because later (in the next homework) you will use docker instances instead

of the Virtual Machine instances. BE SURE NOT TO USE PORT 80 for your code since it may conflict with processes run by other (also port 80 is reserved and typically not allowed for user processes). Run a client on your host operating system and show that your code works properly by making different types of requests and printing appropriate log messages at the client and the components.

D. What you will submit

When you have finished implementing the complete assignment as described above, you will submit your solution to github. We expect you would have used github throughout for source code development for this lab; please use github to turn in all of the following (in addition to your code)

- Source code with inline comments/documentation.
- A copy of the output generated by running your program. When it receives a book, have your program print a message "bought book book_name ". When a client issues a query (info/search), having your program print the returned results in a nicely formatted manner.
- A seperate document of approximately two to three pages describing the overall program design, a description of "how it works", and design tradeoffs considered and made. Also describe possible improvements and extensions to your program (and sketch how they might be made). Describe any cases for which your program is known not to work correctly. You also need to describe clearly how we can run your program. **Please submit the design document and the output in the docs directory in your repository.**

Appendix

Here is a listing of 3 of the required REST APIs for this homework:

GET

`https://CATALOG_WEBSERVICE_IP/search/distributed%20systems`

Reply Example:

```
[
    {
        "id": 1,
        "title": "How to get a good grade in DOS in 40 minutes a day"
    },
    {
        "id": 2,
        "title": "RPCs for Noobs",
    },

]
```

GET

`https://CATALOG_WEBSERVICE_IP/info/2`

Reply Example:

```
{
  "title": "RPCs for Noobs",
  "quantity": 5,
  "price": 50
}
```

PUT?/POST?

`https://ORDER_WEBSERVICE_IP/purchase/2`