Bazar Multi-tier Online Bookstore Part -2-

Distributed Operating Systems

10636456

10th. Dec. 2022

Masa koni 11820155

Lama Darawsheh 11819574

Program Design:

we modified part 1 to implement: replication, load balancing and caching.

- Frontend server responsible for forwarding requests to the two servers, and implements load balancing (using round-robin algorithm). It also has an in-memory cache to enhance the performance, for this we used a hash map for caching requests.
- To implement replication we made two replicas of both Catalog server and order server, and ran them on a VM with two catalog servers and two order server all with different ports.
- As for consistency, we used a push-based method, in which any of the catalog servers notify the other whenever a patch request for (quantity or cost) is received or whenever a book is purchased so that the other server make the changes to its database, too and this way both servers are always up to date.
- Also, whenever a change happen at any of the servers, it sends an invalidate request to the frontend server so that the frontend server removes the modified book from its cache in case it exists.

technologies: Node.js & express, mongoDB, Virtual box with Ubuntu running on it.

On MongoDB we made a database with a collection of: _id, itemNumber, name, cost, topic and numberOfItems.

Catalog servers:

- 192.168.56.101:3000
- 192.168.56.101:3005

Order servers:

- 192.168.56.101:3003
- 192.168.56.101:3004

How it works:

1- The client only talks to the frontend server, and sends requests to localhost on port 5000 using a browser or postman.

http://localhost:5000/

2- If the frontend server receives:
   a. A get requests and checks if it has the required information in cache, if it's not, it contacts one of the two replicas of the catalog servers and it chooses which server to contact using round robin.
   b. A patch request, it forwards it to one of the replicas of the catalog server and it chooses which server to contact using round robin.
   c. Purchase book request, it forwards it to one of the replicas of the order server and it chooses which server to contact using round robin.

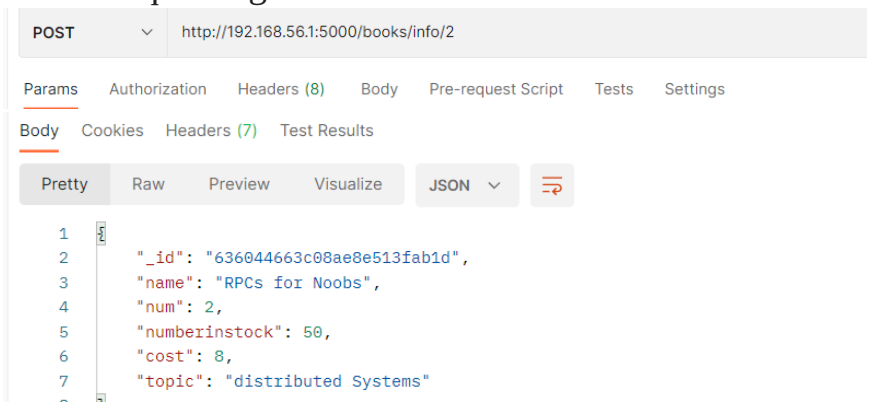
3- If the catalog server replicas receive a get request, it returns the result from its database.

4- When any of the catalog server replicas receive a patch request (either from the frontend server or from the order server when a book is purchased), it makes the required changes to the database and then sends two requests:
   A. notify the other replica to make the same change.
   B. notify the frontend server to remove this book from the cache if it exists.

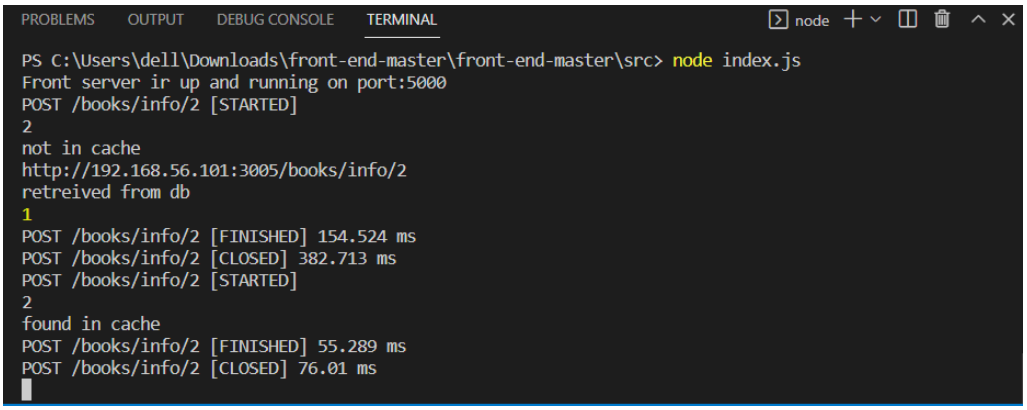
5- When any of the replicas of the catalog servers receives a notification message from the other replica, it applies the required changes

Here is a list of the available requests and their responses:

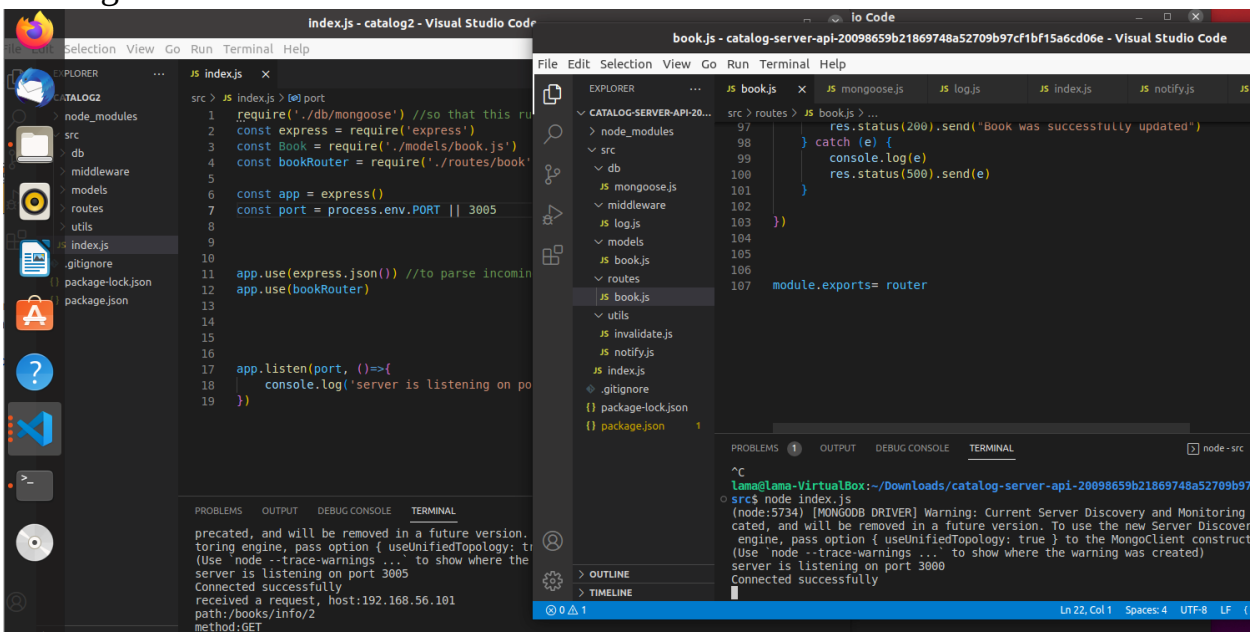
1. **Get information by id:** Request: localhost:5000/books/info/[id]
   - Requesting information of book 2



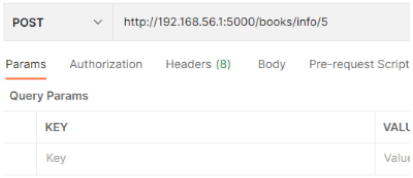
Frnot-end server chechs if book in cache, addes it if it is not

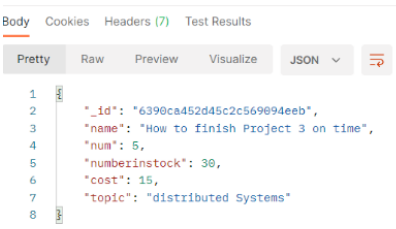


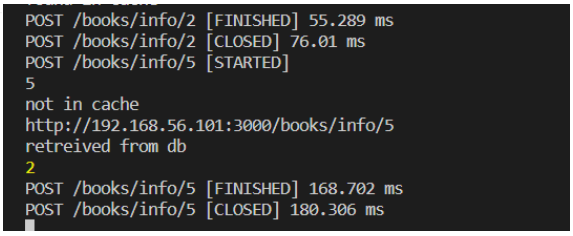
Catalog servers



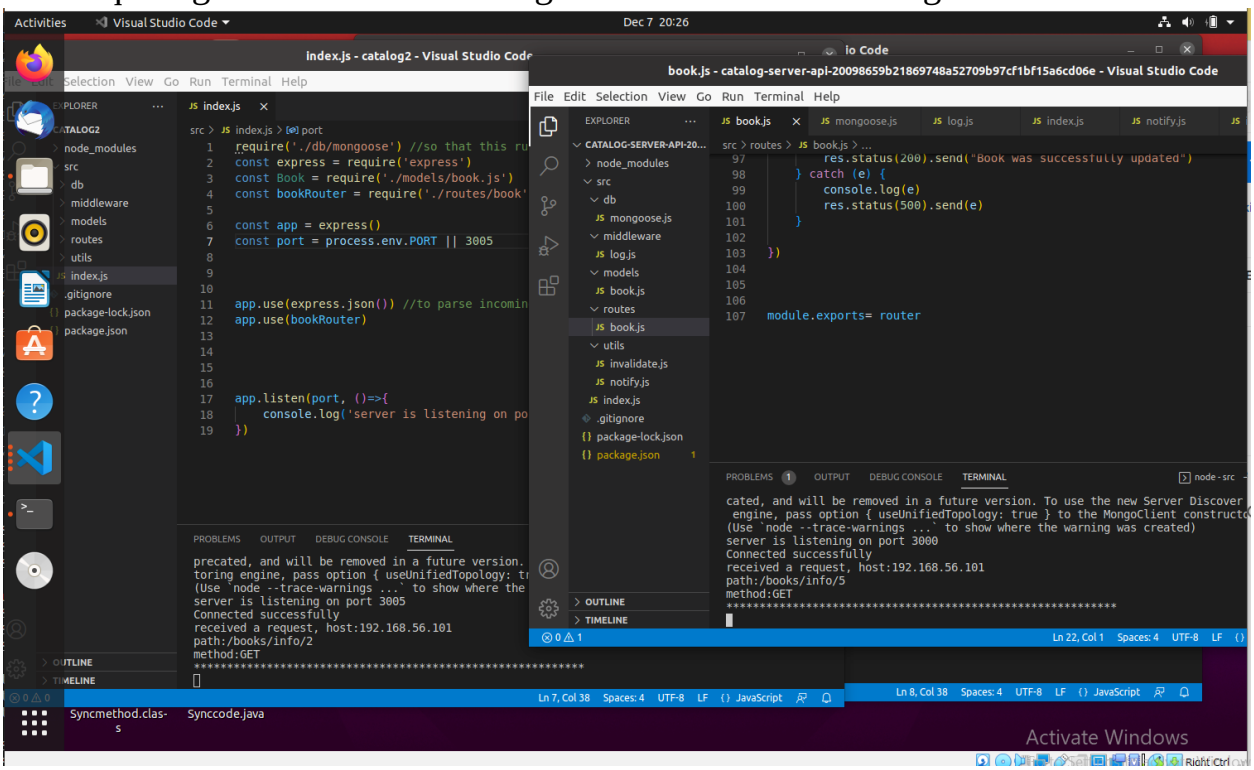
2. **Info of book 5,** front server get it from catalog



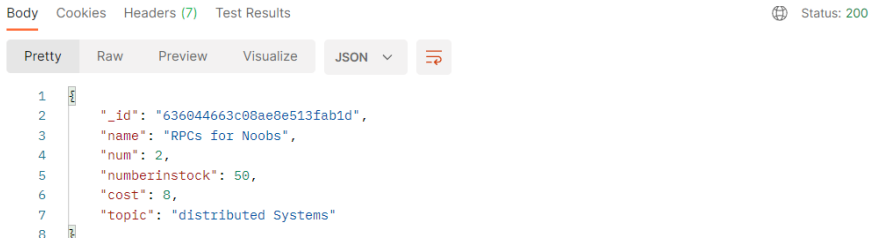


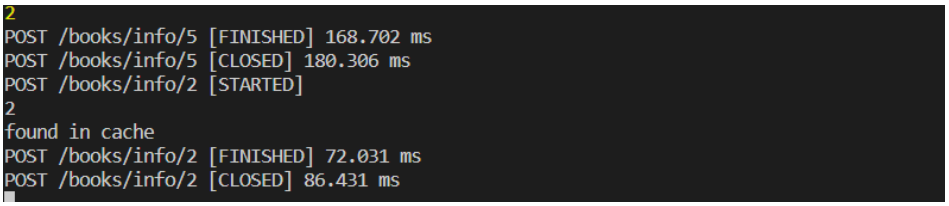


One request got redirected to catalog 1 and the other to catalog 2



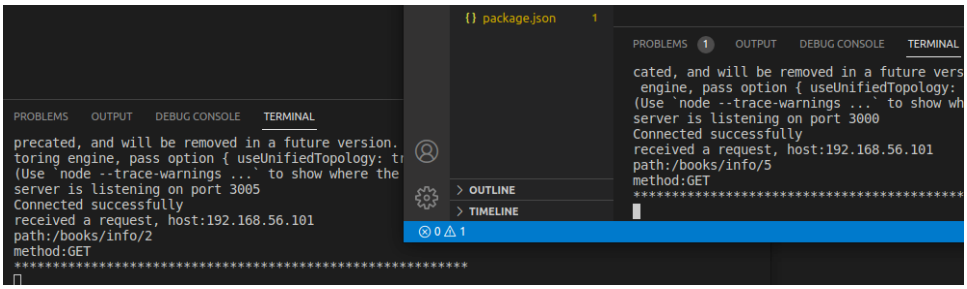
3. Request book 2 after adding it to cache.



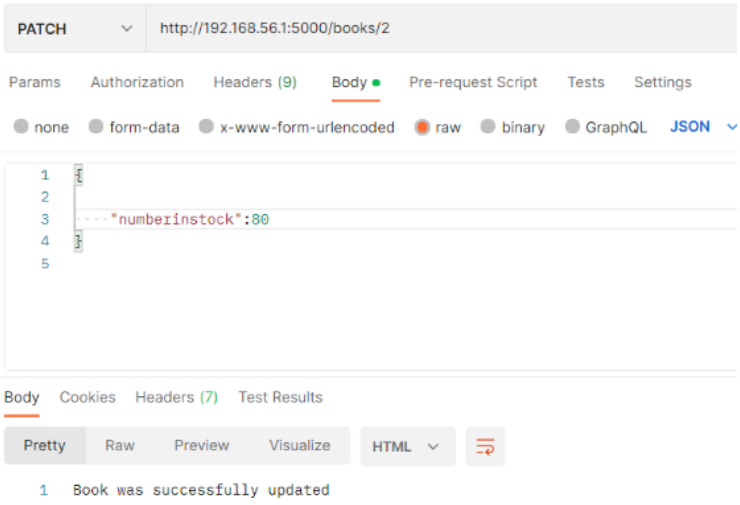


frontend server

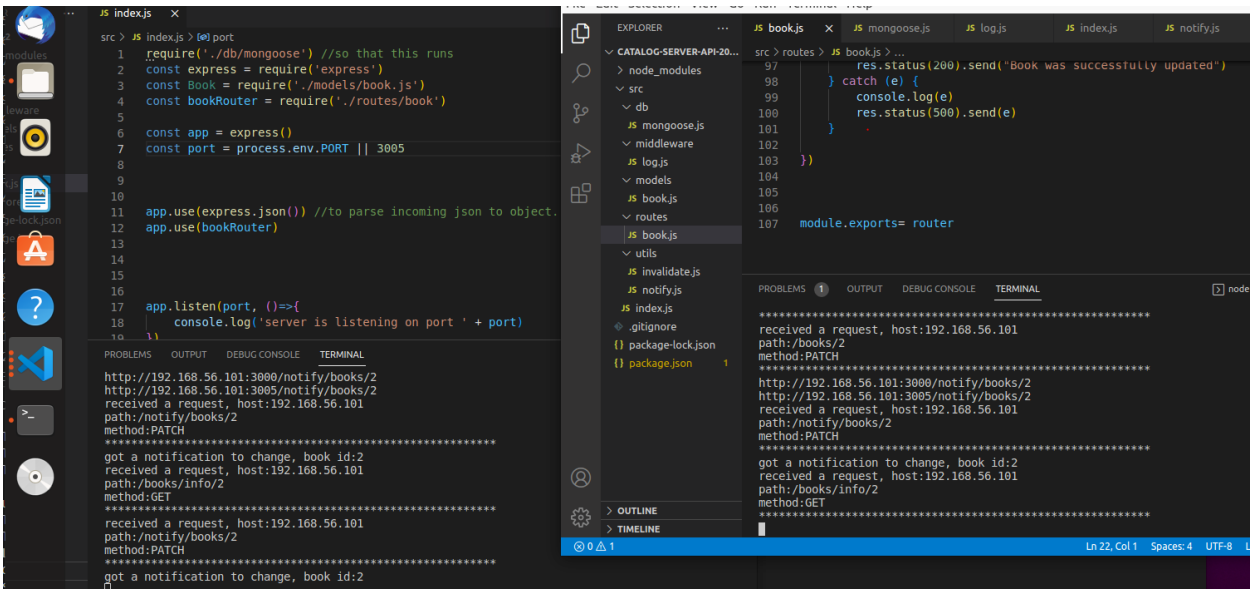
catalog servers did not receive any request



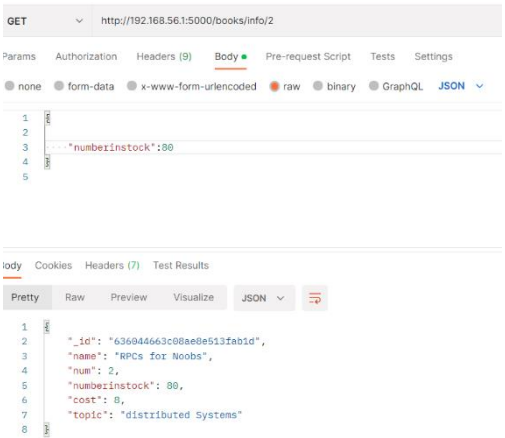
4. **Update numberinStock:** frontend deletes book 2 from cache

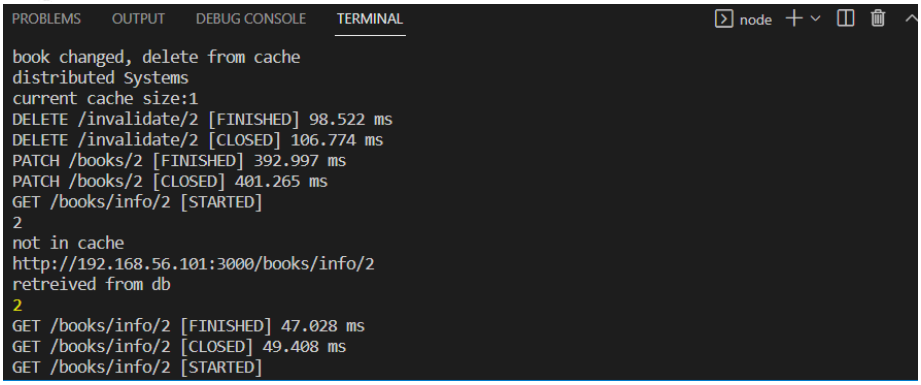


Catalog server: the server that made the patch notifies the other server and sends an invalidate request to frontend server.

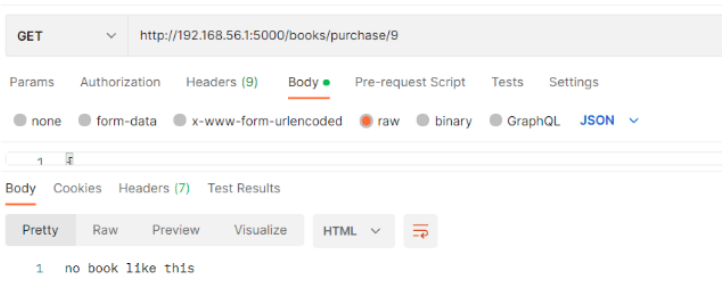


Frontend deletes record in cache and add a new record when we request the updated information

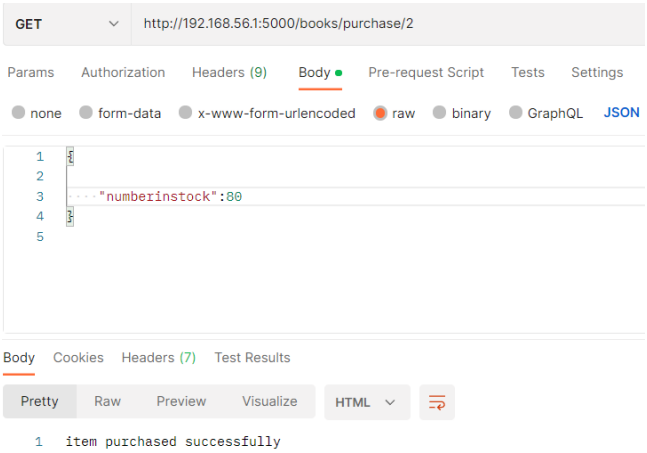




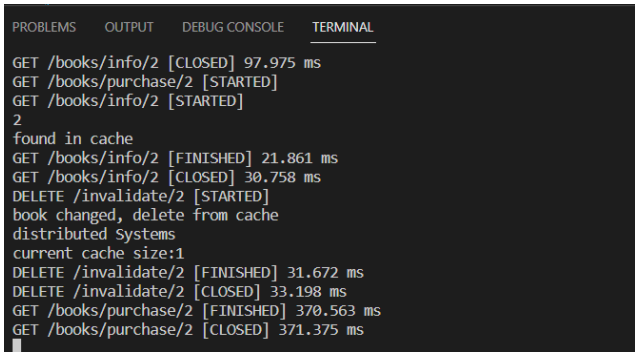
5. **Purchase book:** Request: localhost:5000/books/purchase/[id]
Before sending the purchase request.
If no book found with this id:



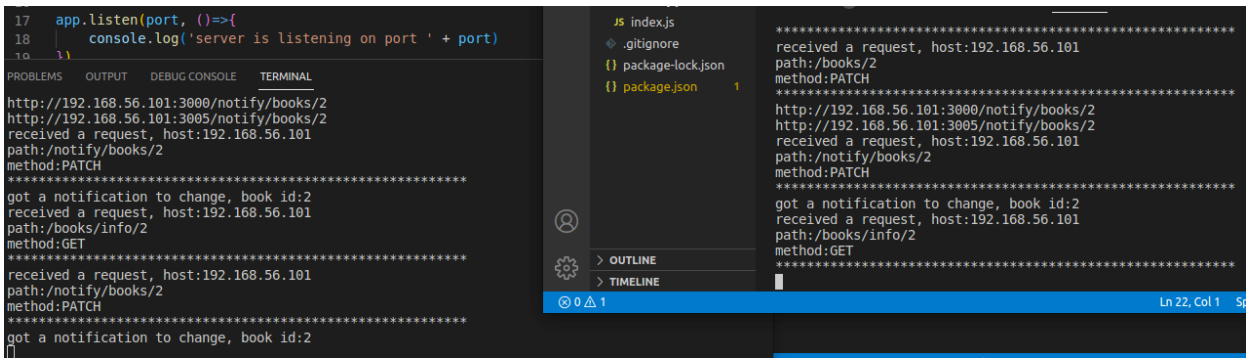
- Sending a request to buy book 2



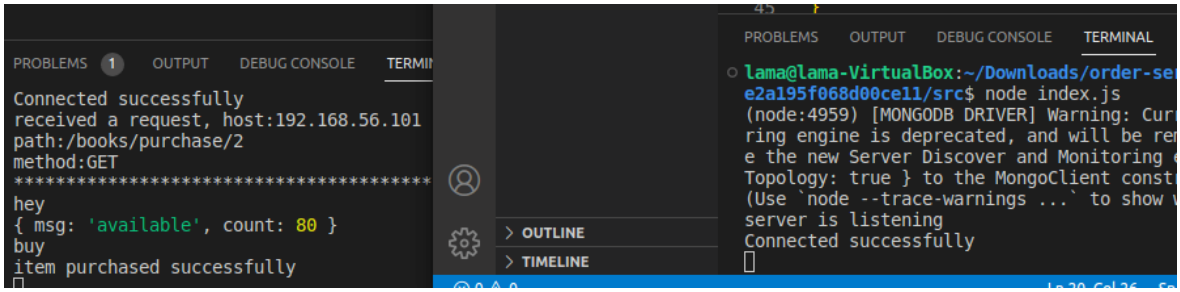
Frontend



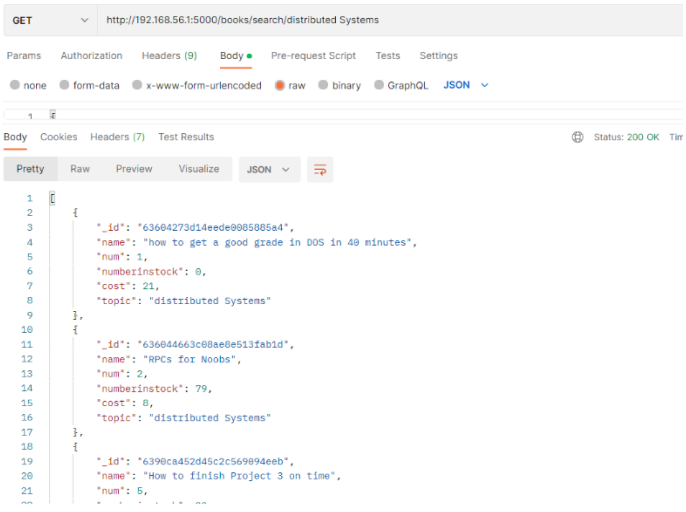
Catalog servers received a get and a patch request from the local order server and sent a request to notify the other replica

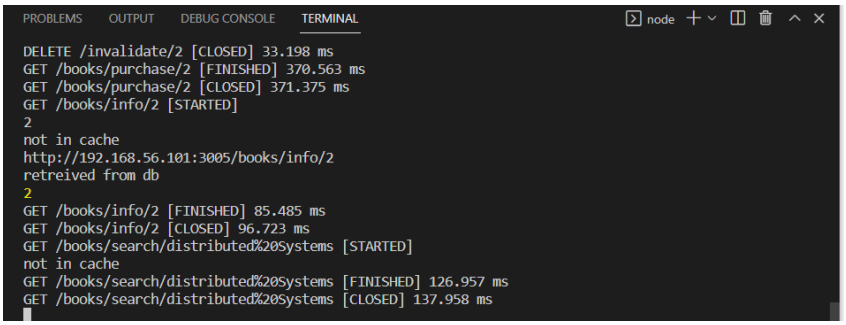


Order servers: one server received the purchase and bought the book the other did not.

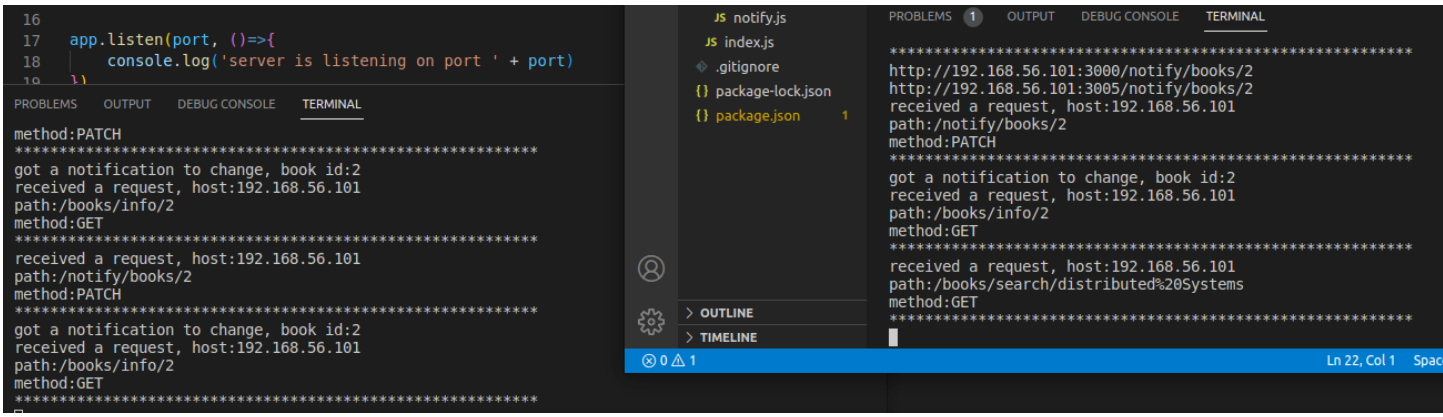


6. **Search by topic:** Request: localhost:5000/books/search/[topic]
   - Searching for a book under title distributed systems





One catalog sever got the search request



Design tradeoffs:

There may be an overhead due to the communication between servers. the frontend server can become a bottleneck to the system.

Replication increased the communication overhead between the replicas. But it increased availability and reliability

Caching increased performance, and we chose hash map and not a separate caching server to reduce the communication overhead.

Using round-robin algorithm for choosing which replica to contact has the following trade-off: it's simpler than any other algorithm but other algorithms may give better results.

Improvements and extensions:

Making a better level of consistency, as we only used messaging between replicas because we only have 2 replicas, however in a real system, instead of messaging acknowledgments, timers, …etc are used.

How to run Program:

1. Install node.js and Express framework on the two machines.
2. Build a database on monogodb with a collection containing the books.
3. Open the frontend terminal and start the server by typing: npm start
4. Open the catalog terminal and start the server by typing: npm start
5. Open the order terminal and start the server by typing: npm start
6. Start sending requests from postman/browser on localhost:5000.

Code:

Frontend server: https://github.com/MasaKni/front-end

Catalog server: https://github.com/lamadarawsheh/CatalogServer

Order server: https://github.com/lamadarawsheh/orderServer