Changes to Overall Design

The overall design has now changed, and the front end is responsible for significantly more functions as opposed to the previous lab. The front-end now has a client-side cache that implements a cache and along with it there are two separate load balancer implementations about which we will cover in this document.

1. Client-Side cache – We decided to implement our own thread-safe client-side cache for caching results. The cache class supports the functions that you would see in a cache – get, put and erase. When the front-end server gets a read request, it checks if the cache already contains a result if it does then it returns that otherwise the front-end queries the catalog server and stores the result in the cache on return. There is also an erase functionality that the catalog servers can call when they receive an update on an object so that they can delete that entry from the cache. Caching provides a significant performance boost on subsequent reads and the results are covered in the performance and testing document.
2. Load Balancers – Another major component of our design are the load balancers. We created a generic thread-safe Load Balancer class and then created specializations for the Catalog Load balancer and the Order Server Load balancer. The important bit is that the load balancers run as separate services and they are not only responsible for routing the requests, but they also act as leaders of their respective kind of servers. Functions that the load balancer performs are –
    a. **Instance class** – Load balancers have a private implementation of instance classes which contain information like the host and port of the active instance. This instance also maintains an update queue that can be used to push updates to the instance.
    b. **Maintaining a list of instances** – The load balancer maintains a list of active instances that can server the requests that it receives. Our load balancer also has support for adding a new instance of catalog/order server at any time and that instance would be consistent with other instances and be added to the list of instances.
    c. **Request Routing** – The load balancer performs round robin request routing for read requests and performs a write request on all active instances.
    d. **Heartbeat fault-tolerance** – The load balancer has an inbuilt functionality to check for dead servers every 0.1 seconds. Once it determines that a server is dead it removes the server from the list of active instances.
    e. **Maintaining consistency of writes across all servers** – The consistency model that we have selected is strict consistency. Since we have only one load balancer it is responsible for ordering the writes to the catalog servers. This is done using a shared critical section where each server is updated before the critical section is finished. The distributed cache on the front-end via a server push is also updated during this critical section.
    f. **Fault-recovery to a consistent state** – Since the load balancer acts as a leader it is responsible for fault-recovery of a server to a consistent state. This is done by comparing the state of the recovering node to that of an existing active instance and applying all the updates to the node. This way any new node or a recovering node will always be in a consistent state.

As before all the servers communicate using HTTP protocols and REST services. Logging is enabled for each server and they each log to files. We used python logger to do this because it is thread safe and thus logs would be written in a thread safe manner. Flask provides a multithreaded server so our servers can work on requests concurrently and because we use sqlite over a file we don't need to lock any threads inside and sqlite can manage the locking for the database internally more efficiently. Moreover our implementations of Load Balancer and Caching are thread-safe.

Maintaining Consistency

We have selected the catalog load balancer as the leader for the catalog servers and as covered above the load balancer is responsible for maintaining consistency across all the active node. In case a node has failed it will be made consistent on recovery using a consistent-cut protocol.

Fault tolerance and recovery –

We have two mechanisms for fault tolerance. The assumption is that the lab only wants us to work with 2 instances of each order and catalog server.

1. The first mechanism is a heartbeat functionality that detects if an instance is live or not. If it detects that an instance is dead then it removes it from the list of active instances.
2. The second mechanism that we have implemented is a fallback mechanism in case a request comes in when the heartbeat hasn't detected a failure yet. In that case the request will be routed to the other active instance. But if both the instances are down then the system cannot function.

How it works

1. Client – The client side still just reads in a request and displays the response in a proper format. It performs some validation checks to make sure that the request read in is correct.
2. Front end Server – The front end server now takes in requests and sends it to the respective load balancers and sends the request back to the client. It also has caching for improved performance. For a lookup or a search request the front end sends it to the catalog load balancer and for a buy request it sends it to the order load balancer.
3. Catalog Load Balancer – The catalog load balancer receives requests from the Order Load balancer and the Front-end server. It routes these requests to active instances of the catalog server using a round-robin algorithm. It can also add a new instance at any time using an add instance two phase commit protocol.
4. Order Load Balancer – Order Load balancer works in a very similar way to the catalog load balancer. It accepts requests and routes them to the order servers using a round robin algorithm.

5. Catalog Server: The catalog server deals with maintaining the catalog database. On initializing, it will also initialize the database with a certain number of books or obtain the current consistent version from the load balancer. It has two threads running, one is the flask server, and another is a background server periodically check if there is any book that is out of stock, if any book is out of stock it raises a message to the admin to restock them. The microservice supports query and update. Query just queries the database and returns the results in JSON format. Update updates the database if possible and returns an error if it cannot.

6. Order Server: The order server supports simple buy message, which will query from the catalog server first, and if the stock is sufficient, then send update message to the catalog server, if the catalog server return selling successful message, return bought message to the frontend server.

Docker

We converted all the services to docker images and uploaded them on UMass box. They contain updated codebase and work out of the box. We have included information that needs to be used to run these images in the README file on the docker folder. Output images are included in the output folder under the docs folder.

Tradeoffs

The tradeoff for our application are the load balancers being a single point of contention and failure.

1. Single point of contention – The load balancers are a single point of contention as they are thread safe and thus each request locks through the load balancer and subsequent requests have to wait.
2. Single point of failure – The load balancers are also a single point of failure. Since there is no leader re-election method the leader can't be re-elected and thus the load balancer has to be running at all times.

Potential Improvement –

1. We can use Raft consensus to remove load balancers as the single point of failure. An easier approach to make the performance better can also be to use Read and Write Locks to support multiple reads. Raft will allow the load balancer to be completely lock free.
2. The cache we currently have is only a client side cache but we can make it a separate entity and that can help it scale up to multiple servers. Which might be a better scenario if there are multiple front end servers reading from it.

   For performance and testing reports please refer to the performance and testing doc.
   For how to run please refer to the github repo.
   For docker please refer to the README in the docker folder of the repo.