CS667 Distributed Operating Systems

Spring 2019

Lab 3 Design Document

Team Name: STIMA

Name: Aggrey Muhebwa
Github Handle: amuhebwa
Student ID: 32055729

Name: Zeal Shah
Github Handle: zealshah95
Student ID: 31737150

Recap & Overview

In lab 2, we designed and implemented a layered web architecture that supported communication between different entities using HTTP REST API calls. The lab 2 system mimics an online book store where a client can browse the books in the store and can buy a book of interest. In lab 3, we have improved the performance and reliability of lab 2 book store. Performance has been improved by implementing replication, load balancing, and caching. Incorporation of fault tolerance and recovery mechanisms has enhanced the reliability of the system. Additionally, in order to ease the deployment of our application, we have dockerized our app and have uploaded it on Github for anyone to download and deploy.

Replication & Load Balancing

Two replicas of each catalog and order servers have been created- Catalog 1 & 2, and Order 1 & 2. Code and databases of the servers were replicated in the process. Every replica has it's own on-disk SQLite database. All the replicas are multi-threaded and so they can handle concurrent requests. These replicas have been created in order to improve the performance of our system. Instead of loading a single catalog or order server with client requests (as in previous lab), replicas are implemented to divide the requests load for achieving better performance. In order to balance the incoming requests-load fairly among the replicas, we have implemented a per-request load balancing technique in the front end server. Per-request load balancing basically, toggles/switches between two replicas whenever requests come in. For example, if the first incoming query request is forwarded to Catalog 1 replica then the second incoming query will be forwarded to Catalog 2. Order requests get forwarded to the order replicas in a similar alternating fashion. One assumption here is- since front end server acts as a load-balancer, it will always stay online and will never crash. Additionally, load balancing has been incorporated in order servers to improve performance for verification and buy requests.

Caching

In order to improve the response time of front end server to lookup query requests, we have implemented simple in-memory caching. Dictionary tool of Python has been used to implement caching. Whenever a lookup request comes in, the front end server looks for the data in cache. If data is not found in cache, the request gets forwarded to one of the catalog replicas. The response received from the catalog server is then cached for quickly serving subsequent lookup requests. Cache data can get outdated if the stock or price gets updated. So, in order to ensure cache

consistency- cache invalidation interface has been implemented as a server push mechanism. Invalidation request removes data for a specific item from the cache. Buy transaction, new stock addition, or price update operations cannot go through before invalidating cache. Back-end servers make a REST API call to the front end server to invalidate a cache entry and wait for an acknowledgement message before changing the price or stock value.

Database Consistency Mechanism

Catalog replicas and order replicas have to maintain their respective databases consistent. In order to achieve this, we make use of REST API calls. Operation- When Catalog 1 updates it's database, it makes a REST API call to Catalog 2, which makes Catalog 2 update it's database as well and vice versa. Same operation applies to order replicas. When one server makes the REST API call to its counterpart, the server sends the book number and the new stock/price value that needs to be updated in counterpart's database. We use the Deferred locking mechanism to ensure that once an operation is being performed on a table in an SQLite database, it is locked until all ACID (Atomicity, Consistency, Isolation, Durability) operations are guaranteed for the current transactions. As a further security measure to ensure that the database is not corrupt, we open the connection to the database, perform an operation and then close it for each of the transaction operations. While this is costly, it is guaranteed that the database won't be spoofed during any of the CRUD(Create, Read, Update , Delete) operations.

Fault Tolerance Implementation

Catalog and order replicas are vulnerable, they can crash, and so fault tolerance is necessary to ensure system's reliability. Load balancer function, in addition to providing load-balancing, also checks if the replica is alive or not and takes action accordingly. Front end server is equipped to detect failure of order and catalog replicas and to take necessary switching, re-issuing actions locally. Order replicas are equipped to detect failure of catalog replicas and to take necessary switching, re-issuing actions locally. Our design assumes that at-least one of each catalog and order replicas will remain online and will not crash. So, if one replica crashes, the other replica should remain online to serve the requests.
The fault tolerance behavior of our system can be described using three different cases:

- Case 1: A replica crashes as soon as a new request arrives at the front end server- In such a scenario, front end server's load balancer will first select a replica and will then check if the selected replica is online or not. If the selected replica is not online, then the request is forwarded to another replica which is online. Front end server can perform this switching for query and buy requests both.
  On the other hand, order servers, before sending a query or update request to the catalog server, check if that specific catalog server is online or not. If not, the request is sent to a different catalog replica.

- Case 2: A replica crashes while processing a request- Front end server can detect crashes of both order and catalog replicas. When a replica crashes in the middle of processing a request, the front end server calls the load-balancer function as soon as it detects the crash. Load-balancer function finds an alive replica and then the request gets re-issued to the selected online replica.
  Order servers behave in a similar fashion. If verification query or buy request serving replica fails in the middle of processing, the order server detects it and locally re-issues the query/buy request to an online catalog replica.

- Case 3: Failed replica comes back online- When a replica fails, the non-faulty replica starts storing all its updates in a temporary array. When the crashed replica tries to recover, it first synchronizes its database with its counterpart. A few seconds before recovery, the failed replica contacts its online counterpart and asks for all the updates to the database since it crashed. The non-faulty online replica sends the content stored in the temporary array to the recovering replica. Once the failed replica receives all the updates, it copies the updates to its database, recovers and comes online. As soon as the replica comes back online, the load balancers in front end and order servers detect its presence and start forwarding relevant requests to it.

HTTP REST API Implementation (in addition to Lab 2)

In addition to the API interfaces in lab 2, we have introduced a few new APIs for our book store in lab 3. They are as follows:

- *invalidate_cache(item_number)*: This interface is provided by the front end server. Back end servers- catalog and order, use this interface to invalidate front end server's cache entry for a specific book. Cache invalidation operation is performed right before changing the stock or price of a specific book. Specifically, buy operation and new stock addition cause the stock value of a specific book to be altered and so cache invalidation is performed right before these operations are performed. This interface sends an acknowledgement to the back end server stating that the cache has been invalidated, only after which any stock/price altering operation can be carried out by the back end server.

- **parallel db updates**: This interface is provided by both the replicas of catalog server (in form of **update_replicas API endpoint**) and both the replicas of order server (in form of **logstransactions_replicas API endpoint**). Whenever one replica updates it's database, it uses this interface to inform it's counterpart to make the same updates in it's database. The details regarding updates are sent in the API call itself. In general, this interface is used to maintain database consistency.

- **re-syncing replicas**: This interface is provided by both the replicas of catalog server (in form of **resync_replicasdbs API endpoint**) and both the replicas of order server (in form of **resync_log_replicas API endpoint**). Whenever a failed replica tries to recover, it uses this interface to sync its database with it's online counterpart. Only after the database has been updated, the recovering replica fully recovers and comes back online.

Dockerizing

We dockerize our servers so that they can run as independent micro-services on any platform irrespective of the underlying operating system. The **Dockerfile** contained in each of the folders corresponding to the servers contains all the requirements needed to create a new docker image that can be spun into a container. The three assumption we make are;The application is running on the latest version of flask, we are using python3 to run our application and our underlying platform is the latest version of Ubuntu Linux. For details on how we dockerized our servers, see the section **How to Run Dockerized Application** under **Implementation/Running the Program**

How Does it Work?

Operations like querying the book store or buying a book are still the same as discussed in Lab 2. The features added to the store in this lab like replication, caching, load balancing, and fault tolerance are transparent to the client. Client still makes requests in the same way as before but is unaware of the fact that the nodes are replicated and if the resource is being recovered from failure. Client sends a search request to the front end server. Load balancer in the front end server decided which of the two catalog replicas to forward the request to. The response received is then forwarded back to the client.

Client sends a lookup request to look for details associated with a specific book like the cost and stock availability. If the details are present in front end server's cache, they are forwarded to the client as a response. If the data is not found in cache, front end forwards the request to one of the two catalog servers (selected by the load balancer). Catalog server then queries it's database and sends back the response to front end which gets forwarded to the client. During this process, front end server caches the response to it's in-memory cache as well for faster future responses.

Based on the response received for the lookup request, the client sends a buy request to the front end server only if the book is in stock. Front end server's load balancer then selects one of the two order replicas and forwards the request to the selected replica. Once the order replica receives the buy request from front end, the load balancer in order server selects one catalog replica and sends the stock verification request to the selected catalog replica. Once the order server receives the confirmation that the book of interest is in stock, it again selects one replica to send the update request to, using it's in-built load balancer. Before sending the update request, the order server sends a cache invalidate request to the front end server. Only after receiving an acknowledgement for cache invalidation, the order sever sends an update request to decrement the stock count of the book, updates the database, and then marks the transaction as complete. Transaction is marked as failed in three situations. First, verification query returns empty stock value. Second, while processing the update request, some other client bought the last available book in stock. Third, if cache is not invalidated, it will not let the transaction go through and will print out a cache invalidation error message.

In the background of catalog servers, we have used timer threads for the purpose of adding new stock periodically. The threads periodically check if the stock of any book has gone to zero. If yes, then the update interface's add_stock function is used to add new stock of the book(s). Before adding new stock, catalog server sends a cache invalidate request to the front end server for that specific book. Only after the cache has been invalidated which is marked by receipt of an acknowledgement from front end server, the new stock gets added. Catalog server then updates the stock value in it's database.

Load balancer function of the front end server not only balances load but also detects checks if catalog/order replicas are online or not. Load balancer selects a replica, checks its status, and then returns IP address and port number if it's online. When a new lookup request data is not found in cache, load balancer of front end server selects one of the two replicas and checks if that server is alive or not. Only if the selected server is alive, the request is forwarded to that catalog replica. If the selected server is not alive, the request is forwarded to the other catalog replica. Similarly, whenever a buy request arrives, front end server's load balancer selects one of the two order replicas and checks if the selected one is alive. If the selected one is alive, only then is the request forwarded to it. Otherwise, the request is forwarded to it's counterpart.

For a case in which, a catalog/order replica crashes while processing a request, front end server detects it. As soon as the front end server detects that the replica processing a request has failed, it calls the load balancer. The load balancer then checks which replica is online and

returns it IP address and port number. Front end server then re-issues the request to a different online replica using the IP and port number.

Each order server is also provided with a similar load balancer that balances load as well as checks if the catalog replica is online or not and returns IP and port number of the online catalog replica. Each order server can detect failure and can take action locally- routing a new request to an online catalog replica and even re-issuing a previously issued request to a different replica in case of server failure. When a new buy request arrives at one of the order servers, load balancer of the order server selects one catalog replica to send the verification request to. The load balancer first checks if the selected replica is alive or not. If it is alive, only then the verification request gets forwarded to the selected catalog replica. If not, the request gets sent to the other replica. In the same way, when the order server wants to send an update request, the load balancer selects a catalog replica and check if it is alive. The update request gets sent to the selected catalog replica only if it is alive otherwise it gets sent to the other catalog replica.

In case, the catalog replica fails while processing a verification request or an update request, the order server detects the crash. As soon as the crash is detected, order server calls it's load balancer which checks and returns the IP and port of an online catalog replica. The verification/buy request then gets re-issued to another online catalog replica.

Whenever one replica makes an update to it's database, it makes a REST API call containing details of the update to it's counterpart. When the counterpart receives this request, it uses the details and makes the same update in it's database. This mechanism is used to maintain database consistency for catalog and order replicas. But the situation changes a bit when one of the replica fails and can no longer be contacted. Online replica can detect the failure of it's counterpart. As soon as a crash is detected, the online replica starts storing all the future updates locally in an in-memory array. When the crashed replica starts recovering, it first calls it's online counterpart for all the database updates since it crashed. On receiving this request, the online replica sends over all the entries present in the stored temporary array. Recovering replica then adds all the received entries to it's database and only then comes online with full recovery. As soon as the replica is online, all the load balancers will be able to detect it.

Design Trade-offs

- Dictionary has been used to perform in-memory caching.
- Caching has only been implemented for improving performance of lookup queries and not search queries.
- No additional features have been added to the cache like limit on the number of items or time outs.
- Transactions have not been implemented. Every server database is consistent with it's replica's database but the databases can have their own order of updates.
- Consensus protocol hasn't been implemented to ensure replica databases are written in the same order.
- One replica makes an update to it's database and sends over the details to its counterpart. In this case, the sending replica doesn't check if the changes were made to it's counterpart's database. So, the system doesn't provide a strong guarantee that the databases will be consistent.

- Replica fail switch-over mechanism built in the load balancer function is developed based on the assumption that we have two replicas for catalog server and two replicas for order server. This mechanism will be a bottleneck when the number of replicas increase and multiple replicas fail at the same time.
- Per-request load balancing mechanism has been used. The load balancer just toggles the request between two servers.
- If one replica fails- the online replica in addition to making updates in the database, it adds those updates to a temporary array. This temporary array is sent over to the recovering replica when called for. Temporary array means the information is stored in memory and not on-disk. In case even the online replica fails, the information in the temporary array will be lost.
- IPs and port numbers of all the machines have been hard-coded in the servers.
- In order to recover a crash server, it needs to be restarted which means you run the code once again.
- Our system assumes that at least one catalog server and one order server will always be online in order to guarantee reliability.
- GET is used to retrieve data and to make updates.

Possible Improvements

- Use more sophisticated tools to perform caching like pymemcache or cachetools.
- Implement caching for improving performance of search requests. Implement caching in back-end servers as well in order to improve response times.
- Add caching features like limiting number of items in cache or limiting the time for which an item stays in cache. This also calls for implementation of cache replacement policy.
- Implement a consensus protocol to ensure strong guarantee on consistency and order or updates in the replica databases.
- Re-design the database consistency mechanism so that it gives strong consistency guarantee and has a better performance.
- Improvise the load balancing and fault tolerance algorithms such that scaling to more than two replicas is not a problem. Instead of having the hard-coded portions for dealing with just two replicas, we plan to improve the code so that it can balance load and tolerate crashes for a system with any number of replicas. In simple words, we plan to make our code more generalized so that it doesn't depend on the assumption that every server has only two replicas.
- Implement per-session load balancing in place of per-request. This also necessitates the server to remain stateful for a specific user session.
- Make use of on-disk write ahead logs instead of using temporary array in the replica to track updates when it's counterpart fails. When the failed replica tries to recover, it can directly access the logs, make the necessary updates to it's database and then come online. Use of on-disk storage also ensures that if even the online replica fails, the log will not be lost.

- Figure out a mechanism to automatically restart a crashed server after some amount of time has passed. This will reduce a lot of effort on testing the code for fault tolerance where one has to manually crash and restart a server.
- Make use of POST command instead of GET to post updates.
- Figure out a cleaner and better way to measure latencies of different features like replica failure detection, replica recovery & re-synchronization time, etc. At present, the code is too cluttered.

Implementation/Running the Program

How to Run the Application?

NOTE:
(1)we use a configuration file **(*named config.py in the src folder of the project folder*) to specify the IP addresses of the different servers.**
Make sure that the configuration file is copied to each of the server folders before running.
(2) We changed the file structure so that it would be easy to dockerize the different servers. instead of putting all the files for creating servers under one parent directory (src), they are in child directories under src, that is src/catalog1 for catalog1, src/catalog2 for catalog2, etc.

Steps to run the setup on distributed machines:

- **Step 1:** Copy lab-3-stima folder to *EdLab 1*.
- **Step 2:** Ensure that flask, pandas and matplotlib modules are installed on *EdLab 1, EdLab 2, EdLab 3* and *EdLab 7*.
- **Step 3:** Change the IP addresses in the configuration files to reflect which machines you want to run them on, assuming you intend to run them on separate EdLab machines. You can leave the port numbers un-changed
- **Step 4:** Change into the sub-directory corresponding to the server you want to start, for example cd catalog1 . All the files should be run with Python3 as **cp .. \config.py &&** *python3* {*file_name.py*}. The reason behind this instruction is that in EdLab, the modules in step 2 get installed under python 3 instead of python 2.7. Also, note the first part of the command which copies the configuration file into the specific server folder
- **Step 5:** To run clients, change to the helper files, and follow repeat step (4).
- **Step 6:** Make sure that the servers are run on the EdLab machines corresponding to the IP addresses specified in the configuration files, otherwise you'll get errors.

How to Run Dockerized Application

- **Step 1:** Make sure that you have docker installed on the machine that you'll dockerize the servers from.
- **Step 2:** Ensure that flask, pandas and matplotlib modules are installed on the machine.

- **Step 3:** Change the IP addresses in the configuration files to reflect which machines you want to run them on, assuming you intend to run them on separate EdLab machines. You can leave the port numbers un-changed. **Note that if you are planning on running docker on localhost, docker for Mac OS doesn't recognize "localhost". Thus, you have use "host.docker.internal"**

- **Step 4:** change into the specific directory under *src* corresponding the the server you want to dockerize, eg catalog1, catalog2, order1, etc and run the command **cp .. \config.py && docker build -t *docker_image_name e.g catalog1server*** . Note the name of the docker image you have create. Repeat this for all the servers you wish to dockerize.

- **Step 5:** To create the docker container and put the server online, run the command **docker run -p portnumber:portnumber docker_image** e.g **docker run -p 12345:12345 catalog1server**. If you want to run the server as a background service, pass -d as a parameter when you run the above command.

- Step 6: To see the list of running dockerized servers, run the command **docker container ls -a**