Distributed and Operating Systems *CS 677, Spring 2020*

Lab 03 - Pygmy.com - Design Document

1 Important classes and interfaces

1.1 *UIServer*

UIServer is the front-end server. It is a microservice that exposes **four** REST endpoints:

- /search/***topic***: This endpoint accepts ***GET*** requests with the topic passed as a **URL paramater**. The results are printed in a JSON format to the user as shown in Figure 1.
- /lookup/***book-id***: This endpoint accepts ***GET*** requests to lookup details of a specific book. The ***book-id***, a unique ***alphanumeric*** identifier for every book in the catalog, should be passed as **URL paramater**. The result for the lookup include the name, topic, cost, stock and unqiue-id of the book, as shown in Figure 2.
- /buy/***book-id***: This endpoint accepts ***POST*** requests with the ***book-id*** as a **URL parameter**. This endpoint assumes a default buy-count of 1 unit i.e. if this endpoint is used, the count defaults to 1. The result of this operation is as shown in Figure 3. Specifically, the result returned by the UIServer is a JSON packet with the following information enclosed: nique order id (the UIServer tags every buy request with a unique order id before forwarding the request to OrderServer/CatalogServer), stock of the item (remaining after this buy operation), the timestamps in milliseconds (at which the request was serviced by UIServer, OrderServer and CatalogServer), code (0 for success, -1 for failure), CatalogServer status, OrderServer status, UIServer status, etc.
- /multibuy: /multibuy is an additional endpoint that is exposed to service buy requests with a count > 1. /multibuy requires a JSON packet as argument, with 2 fields, ***book-id*** and ***count***. The motivation behind retaining the /buy endpoint was for ease of performing tests on the endpoints. The output of /multibuy is exactly same as /buy.

1.2 *OrderServer*

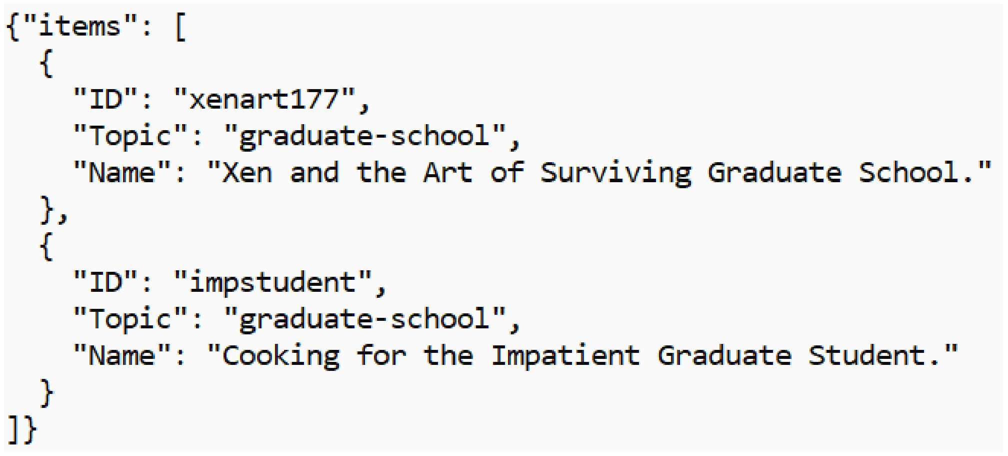
The ***OrderServer*** exposes just one endpoint: ***/buy***. This endpoint requires a JSON packet with the book-id, count, and the UIServer timestamp (the timestamp at which this request was serviced by the UIServer).

1.3 *CatalogServer*

CatalogServer is the backbone of ***pygmy.com***. It exposes three endpoints:

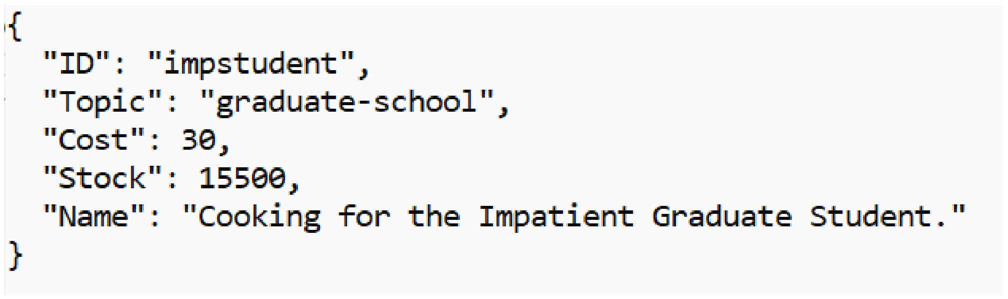
- /query/topic/***topic***: This endpoint is where /search requests are forwarded to by the UIServer. CatalogServer scans the inventory and returns, in JSON format, minimal information (name, unique book-id) about all books that fall under the queried topic.
- /query/book/***book-id***: This endpoint is where /lookup requests are forwarded to by the UIServer. CatalogServer scans the inventory and returns, in JSON format, all information (name, unique book-id, cost, stock) of a book, if it finds the exact book-id in the inventory. If the queried book-id is not present in the inventory, it returns an appropriate error-message citing the reasons.

- /update: This endpoint requires a JSON packet that **should** contain, **at least**, the following information: book-id, count (to update the stock by. Both increments and decrements are supported), the order-id. If the update was a success, the response is sent in JSON format with the updated stock and the **code** field set to 0, else 1.



```
{"items": [
  {
    "ID": "xenart177",
    "Topic": "graduate-school",
    "Name": "Xen and the Art of Surviving Graduate School."
  },
  {
    "ID": "impstudent",
    "Topic": "graduate-school",
    "Name": "Cooking for the Impatient Graduate Student."
  }
]}
```

Figure 1: Output of ***/search/graduate-school***



```
{
  "ID": "impstudent",
  "Topic": "graduate-school",
  "Cost": 30,
  "Stock": 15500,
  "Name": "Cooking for the Impatient Graduate Student."
}
```

Figure 2: Output of ***/lookup/impstudent***

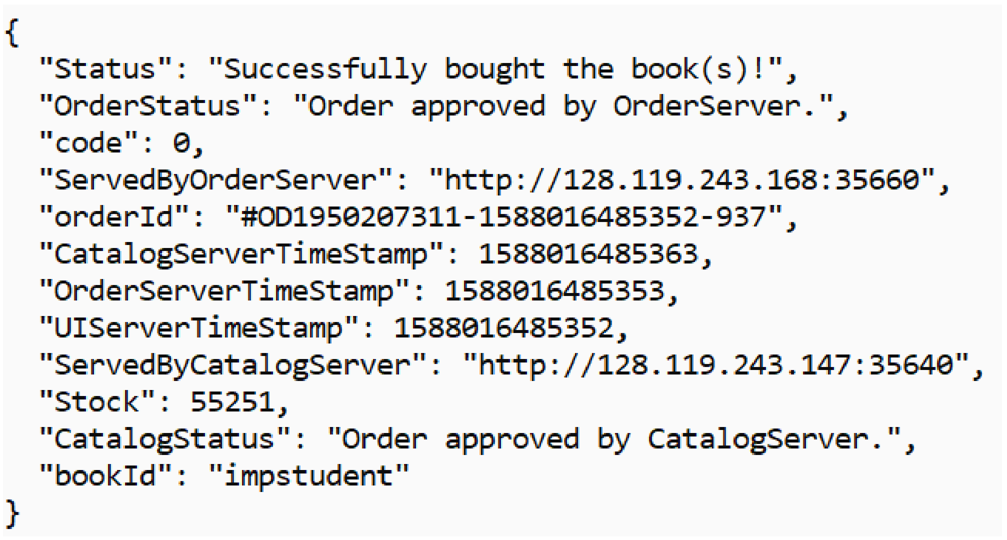
The 3 components mentioned above are mostly same as in lab-2. The major changes that were brought in to support caching, fault-tolerance, replication consistency are documented below.

1.4 *FrontEndCacheManager*

The FrontEndCacheManager uses the **cache2k** library (https://cache2k.org/) for providing a lightweight cache functionality. This class maintains two caches: ***topicLookupCache*** that caches responses for ***/search*** topic-queries and ***bookLookupCache*** that caches responses for ***/lookup*** queries on a specific book.

topicLookupCache doesn't invalidate cache entries since responses to topic-queries don't change over time on **pygmy.com**. However, we need to support cache invalidation for ***bookLookupCache*** since a **buy** operation can invalidate the responses in the cache (specifically, the stock of the book in the inventory).
For this purpose, we devise a server-push invalidation mechanism. The UIServer now exposes a new REST endpoint: ***/invalidate***. After CatalogServer executes an order, it calls UIServer's ***/invalidate***, passing the book-id as parameter. When UIServer receives a HTTP request on ***/invalidate***, it asks the FrontEndCacheManager to remove the cached entry for that specific book-id. The next time a ***/lookup*** query on the same

book comes in at UIServer, we perform a read-through operation i.e. if the entry is missing in the cache, the lookup request is appropriately forwarded to one of the catalog-servers (as decided by the load-balancer) and the response is then cached to ***bookLookupCache***.



```
{
  "Status": "Successfully bought the book(s)!",
  "OrderStatus": "Order approved by OrderServer.",
  "code": 0,
  "ServedByOrderServer": "http://128.119.243.168:35660",
  "orderId": "#OD1950207311-1588016485352-937",
  "CatalogServerTimeStamp": 1588016485363,
  "OrderServerTimeStamp": 1588016485353,
  "UIServerTimeStamp": 1588016485352,
  "ServedByCatalogServer": "http://128.119.243.147:35640",
  "Stock": 55251,
  "CatalogStatus": "Order approved by CatalogServer.",
  "bookId": "impstudent"
}
```

Figure 3: Output of ***/buy/impstudent***

1.5 *HeartbeatMonitor, LoadBalancer, PygmyJob*

These classes are part of the newly introduced ***scheduler*** package in the new **pygmy.com**. **PygmyJob** is a schedulable entity that represents a user-request (such as search/lookup/buy). Every user request is transformed into a PygmyJob which is then scheduled/forwarded to a OrderServer or CatalogServer by the LoadBalancer. The LoadBalancer maintains the list of servers that have successfully processed requests in the past. The **HeartbeatMonitor** tracks every PygmyJob until completion. If a PygmyJob is not completed within a specific timeout period (20 seconds for **buy** operation and 2 seconds for **search, lookup**), then the HeartbeatMonitor deems the server to which the job was forwarded to as faulty and asks the load-balancer to remove the faulty server from being forwarded any jobs in future (until the faulty server comes back up using a handshake mechanism). The delayed PygmyJobs are then picked up by a recovery-thread periodically and forwarded to the non-faulty server. The combined functionality (of HeartbeatMonitor, LoadBalancer and PygmyJob) is present at both the UIServer and the OrderServer.

Note: I used a Round-robin load-balancer where every alternate request is forwarded to the same server.

2 How does pygmy.com work?

- ***search/lookup***: When a **search** (by-topic) or **lookup** (by-id) request arrives at the UIServer, the UIServer first checks if the response for topic/book is already present in the cache. If the cache has an entry, we return without further calls. If it doesn't find a cached response, the UIServer creates a PygmyJob and registers it with the HeartbeatMonitor.
- ***buy***: The UIServer creates a PygmyJob and registers it with the HeartbeatMonitor.

2.1 Buy request in more detail

- When a PygmyJob is registered with HeartbeatMonitor, the start time is noted and then the load-balancer forwards it to the next server in the round-robin queue (CatalogServer if search/lookup request, or OrderServer if buy request).
- A **recovery** thread that runs at CatalogServer regularly checks if there are PygmyJobs that have not been processed by the server they were forwarded to. If a PygmyJob is not processed with a specific timeout, the load-balancer is asked to remove the faulty server from the round-robin scheduler and then the PygmyJob is queued again freshly for processing, this time to a new replica server.
- When the CatalogServer processes a search/buy request, it calls on the **invalidate** REST endpoint of UIServer so that the cached response is invalidated. Search and lookup requests do not require the CatalogServer replica to be holding the token since these requests need not be replicated on both replicas. After processing a request, CatalogServer acknowledges the completeion of processing to the UIServer. This acknowledgement is used as a heartbeat mechanism and allows the replica server to remain in UIServer's load-balancer.
- When the OrderServer receives a buy-request, it uses its own load-balancer to schedule PygmyJob to different replica servers alternately.
- When a buy-request reaches CatalogServer, the request will not be immediately processed. Instead, it is queued into a **TaskQueue** and the CatalogServer waits until it has the token. When the CatalogServer gets a token, it polls **one** task from the **TaskQueue** and then processes. It also asks the replica server to replicate the same buy request (replication requests also do not need a token since replication is being done at the command of another replica). Replication is ordered using the /replicate endpoint of CatalogServer. Once a task is processed and replicated, the CatalogServer sends an acknowledgement back to the OrderServer (this allows the CatalogServer to remain in OrderServer's load-balancer). After this, the token is transferred to the other replica server.

2.2 Write Ahead Logging

Both CatalogServer and OrderServer have their own WriteAhead Loggers that log the records into an on-disk file. For CatalogServer, the WAL is in format that supports recovery of CatalogServer to a consistent in the event of a failure. Figure 4 shows the format of CatalogServer's WAL. The first line is an **INITDB** record which tells the CatalogServer to initialize the stock from an on-disk file. All buy operations result in **UPDATE** records with sufficient information (bookId, count by which stock was updated, timestamp, unique orderId) to recover to consistent state. Both WAL logs are written to a **.WAL** file in the root directory of the source, by default (**CatalogServer.WAL** and **OrderServer.WAL**). A few sample WAL files have been uploaded at **docs/wal** folder of the repo.

2.3 State Machine Replication

This section deals with how replciation-consistency is achieved on both CatalogServer replicas. I used a distributed token-ring algorithm for replciation consistency. Essentially, **we want both replicas to execute every request in the same order**.

When a CatalogServer replica starts, it contacts the UIServer on a REST endpoint (**/catalog/add**). The UIServer hands out a token to the very first CatalogServer that contacted it. A **buy** request can only be executed by a server that has the token. When a replica acquires a token, it checks if any pending orders are present in the Task-Queue and polls just one out of it and executes it. It also asks the other replica to replicate the same order (using the **/replicate** REST endpoint of CatlogServer). Once the order is replicated, the token is transferred to the other replica and the cycle continues.

The token-ring algorithm is designed to minimize starvation by forcing the token-bearer to release the token after executing just one order.

What happens when the replica that had the token crashed? If a replica doesn't have a token, it waits for **one** second. As soon as one second has passed without token, it sends a HTTP request on /heartbeat endpoint of the replica server to see if fthe other replica is still alive. If the replica server is alive, we never forcibly acquire the lock. But if the replica doesn't respond to the heartbeat, the token is forcibly acquired.

This design ensures that all buy-requests are executed in the same order on both replica servers.

2.4 Fault tolerance

Our implementation of pygmy.com can tolerate faults in server even if one OrderServer replica and one CatalogServer replica are down at the same time. We show how the fault-tolerance works in the worst possible case i.e. both OrderServer and CatalogServer replica (one of each) crashed at the same time:

- Assume that the UIServer forwards a /buy request to the OrderServer replica that had just died moments ago.
- Once the **recovery** thread of UIServer detects that the /buy request has not been processed in the last **20 seconds**, it pings the OrderServer that was assigned this task using a /hearbeat REST endpoint.
- Since the OrderServer replica, that was assigned the /buy request, is dead, it will not respond to the heartbeat of UIServer. The UIServer will now remove this faulty OrderServer from its load-balancer and immediately migrate the /buy request to the correctly running OrderServer replica.
- Once the /buy request gets forwarded to the new OrderServer replica, lets assume that it forwarded the /buy request to the CatalogServer that just died moments ago.
- Once the **recovery** thread of OrderServer detects that the /buy request has not processed in the last **7 seconds**, it pings the CatalogServer that was assigned this task using a /hearbeat REST endpoint.
- Since the CatalogServer replica, that was assigned the /buy request, is dead, it will not respond to the heartbeat of OrderServer. The OrderServer will now remove this faulty CatalogServer from its load-balancer and immediately migrate the /buy request to the correctly running CatalogServer replica.
- The non-faulty CatalogServer replica will wait for the token. If the token was last seen with the CatalogServer replica that crashed, the running replica forcibly acquires the token after ensuring that the other replica is indeed dead.
- Once the token is acquired, the CatalogServer replica performs the transaction and acknowledges the completion to the OrderServer that issued the request, after having logged the transaction into its WAL.
- The OrderServer, after receving the acknowledgement of completion, logs the order into its own WAL and then returns the response to UIServer.

So that's how my implementation tolerates faults in the system. We will now see how recovery of a crashed CatalogServer works.

2.5 CatalogServer recovery

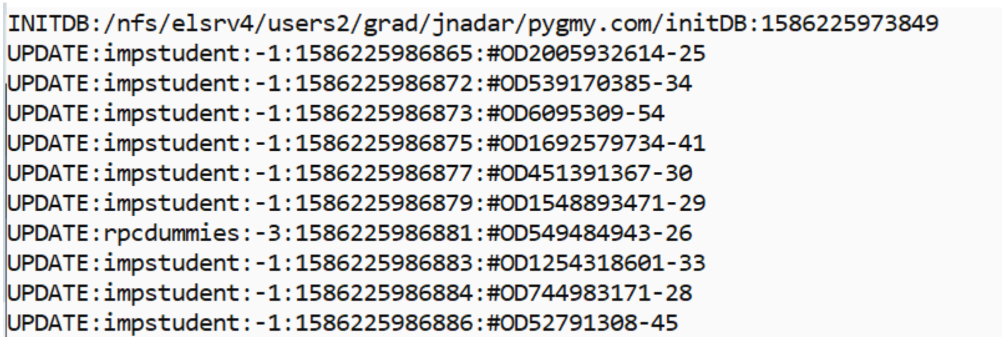
When a CatalogServer crashes, it's state needs to brought to a stable/consistent one before it can start processing buy-requests again.

To recover a CatalogServer, it has to be restarted in **"recovery"** mode (passed as a command-line argument). **When started in recovery mode**, the CatalogServer first contacts the other replica at the **/recovery/initiate** endpoint. On receiving the request to initiate recovery process, the replica (that was up and running) sends the contents of its Write Ahead Log to the server that requested the recovery. The contents of WAL are self-sufficient enough such that the state of the database can be replicated by replaying each buy request in the WAL. After recovery is done, the replica that just recovered uses the **/recovery/-complete** REST endpoint to tell the other server that recovery is indeed complete.

An important thing to note is that while recovery is under progress, no orders are executed on both the replica servers. Once recovery is complete, the replica server that helped with the recovery (i.e. the server that was up and running all the while) will forcibly acquire the token and start processing the requests from its **TaskQueue** and then the cycle continues!

Notes:

- The three servers are multithreaded. This is because SparkJava, used to provide the REST endpoints, has a threadpool of 8 threads by default. **search** and **lookup** are complelety cncurrent in nature due to this threadpool. Threads that execute **buy** operation, on the other hand, have to first wait for the token, then acquire database's lock before updating the inventory.
- The write ahead logging mechanism is thread safe and logs events in the order they come. This is achieved by holding write-locks while writing to the WAL file.



```
INITDB:/nfs/elsrv4/users2/grad/jnadar/pygmy.com/initDB:1586225973849
UPDATE:impstudent:-1:1586225986865:#OD2005932614-25
UPDATE:impstudent:-1:1586225986872:#OD539170385-34
UPDATE:impstudent:-1:1586225986873:#OD6095309-54
UPDATE:impstudent:-1:1586225986875:#OD1692579734-41
UPDATE:impstudent:-1:1586225986877:#OD451391367-30
UPDATE:impstudent:-1:1586225986879:#OD1548893471-29
UPDATE:rpcdummies:-3:1586225986881:#OD549484943-26
UPDATE:impstudent:-1:1586225986883:#OD1254318601-33
UPDATE:impstudent:-1:1586225986884:#OD744983171-28
UPDATE:impstudent:-1:1586225986886:#OD52791308-45
```

Figure 4: Sample WAL of CatalogServer

3 How to run?

3.1 Deployment of pygmy.com servers

To run the program, I have developed a couple of deployment scrips. Here are the steps to be followed (from any edlab machine terminal of choice) :

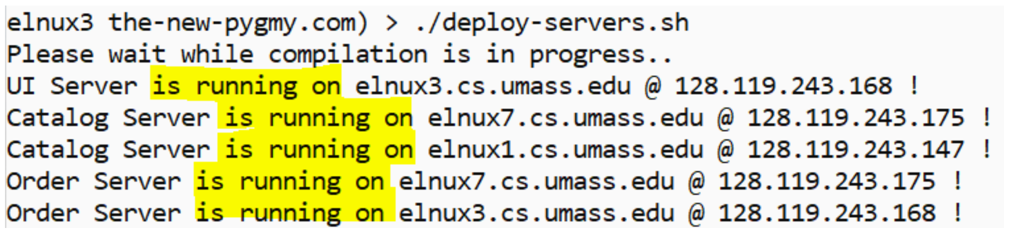
- Download the source folder or clone from the Github repo.
- ***cd*** into the root directory of the source.
- From the root of the source, execute:
  chmod +x *.sh
  Ensure that all shell scripts (.sh) files in the root have execute permissions.
- Run this only if passwordless-ssh has not been setup yet. Execute:
  ./ssh-setup.sh
  This script sets up ssh key-pairs for the grader to deploy the 3 servers to remote edlab machines without repeatedly entering the passwords. Some points to note here:
  - ***Do not*** change the default location of the public/private RSA keys.
  - If prompted to overwrite existing keys at the default location, choose *yes* or *y*.
  - Use only the empty passphrase when prompted.
  - Enter your own edlab password when asked. Since the ssh-keys are being copied to 3 different machines, you may have to enter the password 3 times.

  Note: The ssh-setup script needs to be run only once.
- Execute:
  ./deploy-servers.sh clean (this cleans up all existing servers that may be running on this machine. This doesn't affect other teams' servers.)
  ./deploy-servers.sh (this deploys all 5 servers on pre-configured machines.)
  This script automatically deploys the 5 servers to 3 different edlab machines. The CatalogServer is deployed to elnux7.cs.umass.edu and elnux1.cs.umass.edu, the UIServer to elnux3.cs.umass.edu, and the OrderServer to elnux7.cs.umass.edu and elnux3.cs.umass.edu. The ports are pre-configured and chosen from the port-range allocated to every group.
  The remote machines are configurable and the 5 pygmy.com servers can run on any machine (the hardcoding is done for the ease of grader). If everything goes fine, you should see the output as shown in Figure 5.

Note: If you see **"is not running"** instead of the highlighted **"is running on"**, please re-deploy the servers by running ***./deploy-servers.sh clean*** and then ***./deploy-servers.sh***. I used SparkJava for REST endpoints and sometimes, the Jetty server in the SparkJava library doesn't cleanup ports in a timely manner which leads to failure in starting of server the next time. Usually this is resolved by cleaning up all instances and deploying them again.

The CatalogServer gets its initial inventory from initDB file in the root directory of the source. We stock the items enough to pass all the tests. If the stocks or items need to be changed, please follow the format of the initDB file (the format is clearly mentioned in the initDB file itself).

The WAL logs for the CatalogServer and OrderServer will be present in the root directory of the source at ***CatalogServer1.WAL***, ***CatalogServer2.WAL*** and ***OrderServer1.WAL***,***OrderServer2.WAL*** once the servers are up and running.



```
elnux3 the-new-pygmy.com) > ./deploy-servers.sh
Please wait while compilation is in progress..
UI Server is running on elnux3.cs.umass.edu @ 128.119.243.168 !
Catalog Server is running on elnux7.cs.umass.edu @ 128.119.243.175 !
Catalog Server is running on elnux1.cs.umass.edu @ 128.119.243.147 !
Order Server is running on elnux7.cs.umass.edu @ 128.119.243.175 !
Order Server is running on elnux3.cs.umass.edu @ 128.119.243.168 !
```

Figure 5: Output of ***deploy-servers.sh*** **script**

3.2 Client program/User Interface

This section is applicable only after the 3 servers are correctly deployed as shown in the previous section.
We created a script that provides a command-line interface to execute ***search***, ***lookup*** and ***buy*** operations. To start the user-interface, open a new terminal (on edlab), and then execute:
./user-interface.sh
The prompts are very simple to follow. The unique-id of the book will be shown in the ***search*** result, which can then be used for lookup/buy operations. A sample output of running this script is placed in **docs/sample-output.txt**.

3.3 Running experiments

Cache Consistency

The experiment itself is detailed in the Performance report. To run this experiment, execute, from the root folder of git repo, on an edlab machine: ***./test-cache-consistency.sh***

A sample output from this experiment is placed at **docs/cache-consistency/** directory.

Fault tolerance and recovery

The experiment itself is detailed in the Performance report. To run this experiment, execute, from the root folder of git repo, on an edlab machine: ***./test-fault-tolerance.sh***

A sample output from this experiment is placed at **docs/fault-tolerance** directory. Please make a note of few **very important** observations below:

- The **fault-tolerance-CatalogServer1.WAL** is the WAL file of the catalog-server replica that was not killed during the experiment. The **fault-tolerance-CatalogServer2.WAL** is the WAL file of the catalog-server replica that **was** killed and restarted in this experiment. One can easily verify that the UPDATE/BUY operations in both the files are in the same order (only timestamps will differ) proving that the restarted catalog-server recovered to a stable state by replaying all transaction from the replica that was up and running!
- The **fault-tolerance-catalog-server1.out** is the output of the first catalog-server replica that was **not killed**. One can see the logs **"Initiating recovery of faulty replica server.."** after timestamp **04-27-2020; 13:51:54.620** and **"Recovered faulty replica server.."** just before timestamp **04-27-**

2020; 13:52:24.910. This shows that restarted catalog-server replica approached the running replica and **completed the recovery process**!

- If you observe that during the test, a particular /buy request took more than 30 seconds or so, do note that this is expected. It is because the CatalogServer only migrates a job after 20 seconds of a request not being processed and similarly the OrderServer migrates a job to another CatalogServer only after 7 seconds once a server is found to be dead.

4 Dockerization of pygmy.com

Note: all the docker container-images are present at the **Docker hub** repository:
https://hub.docker.com/r/shibingeorge/pygmy.com/tags.

There are 4 images that are tagged in the above dockerhub repo (uiserver, catalogserver, orderserver, user-interface). The time of upload of these images can be seen at the same link.

4.1 Design

Our implementation of pygmy.com is such that the OrderServers always listen on a specific port and so do UIServer and the CatalogServer Servers. This design choice necessitates that we run the two replicas of OrderServer on two different machines so that they different IP addresses (and the same applies to CatalogServer). This was easy to achieve on edlab since we could run an instance each of OrderServer an CatalogServer on the same machine and the other 2 replicas of the servers on another machine. However, since we had to run Dockerized pygmy.com on the same machine, this would result in a port conflict (all docker containers on a machine get assigned the same IP address by default).

To avoid making changes to the code/port-configurations, I decided to follow another approach. Docker allows you to create a custom subnet which you can use to distribute unique IP addresses to different containers even on the same machine. Once we created a subnet, we assigned different different IP addresses in the subnet to the 5 different machines!

To reduce the size of images, I used Alpine-Linux containers running JRE-13. The docker containers were tested on an Ubuntu 14.04 OS, running **Docker version 18.06.1-ce, build e68fc7a**.

4.2 How to run?

I developed a shell script that:

- pulls the images from the docker-hub repository.
- creates a docker subnet
- deploys all 5 server instances (1 UIServer, 2 CatalogServer, 2 OrderServer) with unique IP addresses
- Also launches an interactive docker container from which you can run shell-based user-interface to issue commands to dockerized pygmy.com (The IP addresses of all pygmy.com servers are hardcoded in this container for ease of execution).

To deploy docker container on a linux machine, with docker already installed, run the following commands from root of the git repo, **from the machine** on which you want to deploy the containers on:

- **./docker-cleanup.sh**
  (to cleanup any existing images/containers from my repository - this doesn't affect other teams' containers).
- **./docker-run-pygmy.sh**
  (this pulls the images from dockerhub and deploys the containers)
- At this point, you should see a bash shell prompt. This bash shell is running on a docker container specifically meant for issuing commands to the pygmy.com servers. **This bash shell is not running directly on the host OS but inside teh container**.
  From the bash shell, execute the following:
  - **cd /**
  - **./docker-user-interface.sh**

  Once the user-interface appears, it is very easy to follow the prompts to issue /search, /lookup and /buy requests to the UIServer! Note that the above **2** commands are run from inside the docker container. Do not attempt to run these commands directly from the local machine/Host OS on which docker containers are running.

Note:

- Sometimes, due to large number of docker containers running on the same machine and routing of request within the subnet by Docker, I see requests timing out. If this happens, I just restart all containers again i.e. start again from **./docker-cleanup.sh**.

The **Dockerfiles** I used to build the images are present in **dockerfiles/** directory of the git repo, under the respective folders.

5 Tradeoffs and caveats

- The /buy requests take time to process due to the token ring algorithm that I used to ensure state machine replication.

6 Possible improvements

- A better replication algorithm so that /buy requests can be independently executed on both servers, so that response time is better.