**mediaMicroService**

Running **sudo docker container ls** tells us that

mediaMicroservice runs 33 containers

**Structure of files:**

A docker-compose file would define a container like this:

Movie-id-service: : tag name

image: yg397/media-microservices : the code inside the container

that would be running

hostname: movie-id-service : vm name, container name

# ports:

# - 10002:9090

restart: always : If the container errors while running

It will restart in a forever loop

entrypoint: MovieIdService : the entrypoint for this container

**NOTE: hostname will serve as the mapping of container ip address in the code. In the file   
config/service-config.json, the addresses mentioned correspond to these hostnames.**

In the src folder we have the following services:

CastInfoService/ MovieIdService/

MovieReviewService/ PlotService/

ReviewStorageService/ UniqueIdService/

UserService/ ComposeReviewService/

MovieInfoService/ PageService/

RatingService/ TextService/

UserReviewService/

Any service will have the RPC stack defined by the thrift file **media\_service.thrift**

The corresponding code for RPC communication is then generated using the thrift commands (Refer **Appendix 2** in the end). Each service will generate 3 sets of code in python, cpp and lua. For example,

gen-py/media\_service/UserService.py

gen-cpp/UserService.cpp

gen-lua/media\_service\_UserService.lua

The actual business logic for a service is written in the src/ folder. For example src/UserService.

Most of the services are doing pretty much the same things so you would see the same logical steps for some services down below.

Note: I assume that the same structure and logic will be used in the other microservices as well like social network and hotel reservation.

**Inner workings of**  
**“./wrk -D exp -t 10 -c 10 -d 30s -L -s ./scripts/media-microservices/compose-review.lua http://localhost:8082/wrk2-api/review/compose -R 1**

**”**

Running the above command creates 10 threads with 10 connections in total. The experiment runs for a duration of 30seconds and the requests are generated @1rps.  
  
Following are the end to end steps that the above command would do:

Step1)   
wrk2/scripts/media-microservices/compose-review.lua  
  
In this part of code, a random movie and a random user index are selected and with a random rating of the movie and a random review text. The code then creates an HTTP POST request to

"<http://localhost:8082/wrk2-api/review/compose>" endpoint in the nginx server with suitable headers and body.

Step2)

nginx-web-server/conf/nginx.conf:76:

In this file, the endpoint is defined as   
location /wrk2-api/review/compose {

content\_by\_lua '

local client = require "wrk2-api/review/compose"

client.ComposeReview();

';

}

Grepping the “ComposeReview()” we jump to  
nginx-web-server/lua-scripts/wrk2-api/review/compose.lua

**SPAN1)** This is where our first span is being defined for the nginx microservice container. (I believe social-network was running 3 nginx containers, whereas we just have 1 container for mediamicroservice).

This particular part of code spawns 4 threads for 4 parallel tasks.

ngx.thread.spawn(\_UploadUserId, req\_id, post, carrier),

ngx.thread.spawn(\_UploadMovieId, req\_id, post, carrier),

ngx.thread.spawn(\_UploadText, req\_id, post, carrier),

ngx.thread.spawn(\_UploadUniqueId, req\_id, carrier)

These tasks are defined in the same file on top. Following steps goes through each tasks one by one.

Step 3) \_UploadUserId

For this task, the **nginx** server contacts **user-service** on port 9090 at **UploadUserWithUsername** endpoint

The next part of code is defined in the Thrift file which auto-generates a lua script, a cpp code and a python code, all doing the same task.

Associated Files:

media\_service.thrift

gen-py/media\_service/UserService.py

gen-cpp/UserService.cpp and gen-cpp/UserService.h

gen-lua/media\_service\_UserService.lua

The control jumps to line 357 in src/UserService/UserHandler.h where subtask1 is executed in

void UserHandler::UploadUserWithUsername

Subtask1: MmcGetUserId

Mmc is memcache

src/UserService/UserHandler.h line 357

The code tries to cache the user in the memcache.

Data such as password, salt and ID are cached in Memcached

The code first checks if there is an entry in the Memcache for the particular username. If it is not there it grabs the user from the mongodb and places it in the memcache db. It will throw exceptions and errors wherever necessary.

This part of code defines 4 spans  
  
**SPAN2)** "UploadUserWithUsername" child of **SPAN1)**

Time taken to upload user to memcache

**SPAN3)** "MmcGetUserId" child of **SPAN2)**

Time taken to check if the user already exist in memcache

**SPAN4)** "MongoFindUser" child of **SPAN2)**

Time taken to check if the mongodb has the user and grab it.

**SPAN5)** “MmcSetUserId” child of **SPAN2)**

Time taken to set the user in memcache

**Depending upon if the user was cached or not this part of code will have high variability.**

Step 4) Store user in the Memcache as part of the Compose Review component

Between Span4 and Span5 the function calls the compose\_client->UploadUserId

**This is when the user-service Microservice will contact the compose\_review\_service.**

Subtask2: UploadUserId

The control then goes to line 363 of src/ComposeReviewService/ComposeReviewHandler.h

void ComposeReviewHandler::UploadUserId(

Where it defines this spans

**SPAN6)** "UploadUserId" child of **SPAN2)**

Time taken to store the userid of the user who pushed the review in memcache.

*If this thread is the last one uploading the review components,*

*it is in charge of compose the request and upload to the microservices in*

*the next tier.*

Step 5) \_UploadMovieId

Similar to userService,

For this task, the **nginx** server contacts **movie-id-service** on port 9090 at **UploadMovieId**

Endpoint

The next part of code is defined in the Thrift file which auto-generates a lua script, a cpp code and a python code, all doing the same task.

Associated Files:

media\_service.thrift line 78

gen-py/media\_service/MovieIdService.py

gen-cpp/MovieIdService.cpp and gen-cpp/MovieIdService.h

gen-lua/media\_service\_MovieIdService.lua line 264

Line 66 of the src/MovieIdService/MovieIdHandler.h provides the next span of action

**SPAN7)** "UploadMovieId" child of **SPAN1)**

Uploads the movieId to memcache

**SPAN8)** "MmcGetUserId" child of **SPAN7)**

Time taken to check if the movieId already exist in memcache

**SPAN9)** "MongoFindMovieId" child of **SPAN7)**

Time taken to check if the mongodb has the movieId and grab it.

**SPAN10)** “MmcSetMovieId” child of **SPAN2)**

Time taken to set the movieId in memcache

**Depending upon if the movieId was cached or not this part of code will have high variability.**

Once that is finished, UploadMovieId is called from the composeClient. compose\_client->UploadMovieId

And then the rating\_client->UploadRating is called to upload the rating the same way.

This finishes the **SPAN7)**

RatingService

The next part of code is defined in the Thrift file which auto-generates a lua script, a cpp code and a python code, all doing the same task.

Associated Files:

media\_service.thrift

gen-py/media\_service/UserService.py

gen-cpp/UserService.cpp and gen-cpp/UserService.h

gen-lua/media\_service\_UserService.lua

Here we have spans like

“UploadRating”

“RedisInsert”

Step 6) \_UploadText

Similar to userService,

For this task, the **nginx** server contacts **text-service** on port 9090 at **UploadText** endpoint

The next part of code is defined in the Thrift file which auto-generates a lua script, a cpp code and a python code, all doing the same task.

Associated Files:

media\_service.thrift line 94

gen-py/media\_service/UserService.py

gen-cpp/UserService.cpp and gen-cpp/UserService.h

gen-lua/media\_service\_TextService.lua line 133

vim src/TextService/TextHandler.h

TextMap represents SpanContexts as key:value string pairs.

This part of code defines 1 span  
**SPAN10)** "UploadText" child of **SPAN1)**

Total Time taken to upload text using compose review functionality.

**SPAN11)** "UploadText" child of **SPAN10)**

Time taken to upload text using compose review functionality.

Some kind of counter is being maintained in a separate table of memcache for each request\_id.

Then the text is uploaded into memcache for each request\_id

UploadText

void ComposeReviewHandler::UploadText( line 584

In this function the text is stored into memcache

**Memcache usage:**

(Inside **src/ComposeReviewService/ComposeReviewHandler.h** )

The memcache store that they have defined has 5 components.

NUM\_COMPONENTS = 5

key\_unique\_id.c\_str(),

key\_movie\_id.c\_str(),

key\_user\_id.c\_str(),

key\_text.c\_str(),

key\_rating.c\_str()

These components are filled up for each request\_id and are served by 5 separate threads asynchronously.

They maintain a counter of the number of components filled up so far out of 5 for each request. The last thread that fills up the 5th component, which can be anything is responsible   
  
 *If this thread is the last one uploading the review components,*

*// it is in charge of compose the request and upload to the microservices in*

*// the next tier.*

By calling \_ComposeAndUpload

**gen-cpp/media\_service\_types.h** contains all the different data models used in mediaMicroservices

memchached\_mget is a multi-get operation which gets the required 5 components of the request and composes a new review with added timestamp = system.currentTime.

This new review is then sent to the review\_storage\_service, user\_review\_service and movie\_review\_service one by one.

* review\_storage\_client->StoreReview(req\_id, new\_review, writer\_text\_map);
* user\_review\_client->UploadUserReview(req\_id, new\_review.user\_id,

new\_review.review\_id, new\_review.timestamp, writer\_text\_map);

* movie\_review\_client->UploadMovieReview(req\_id, new\_review.movie\_id,

new\_review.review\_id, new\_review.timestamp, writer\_text\_map);

Where this span is calculated

**SPAN12)** "StoreReview" child of **SPAN10)**

And then a binary blob of movie review created and stored in the mongo db and the next span is measured.

**SPAN13)** "MongoInsertReview" child of **SPAN12)**

Similarly, we have the spans in the user-review-service  
SPAN14) “UploadUserReview”

SPAN15) “MongoFindUser”  
SPAN16) “MongoInsert”

SPAN17) “MongoUpdate”

SPAN18) “RedisUpdate”

Similarly, we have the spans in the movie-review-service  
SPAN19) “UploadMovieReview”

SPAN20) “MongoFindMovie”  
SPAN21) “MongoInsert”

SPAN22) “MongoUpdate”

SPAN23) “RedisUpdate”

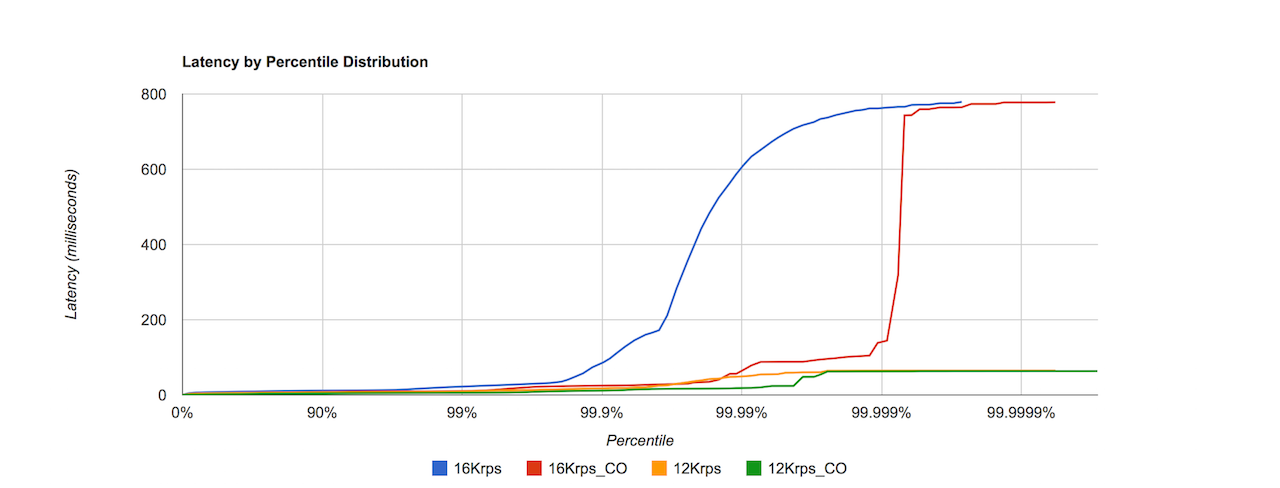
We also have UploadUniqueId in the ComposeReviewService with the corresponding span  
“UploadUniqueId”

Under the test folder the authors have written the tests for all the services including the required readReview service in python.  
**test/testReviewStorageService.py**

The authors also seem to be writing a script to test various workflows in **wrk2/run.sh**

For now, the authors have written load generators in **wrk2/scripts/media-microservices/** only for compose review functionality. Future work from the authors is expected to populate this folder.

The authors also computed the differences between wrk and wrk2. Wrk2 generates the request in a steady rate and do not wait for the response and thus the problem of coordinated omission is averted. Below they have generated the graph of latency difference observed in the two cases. This graph is stored in **wrk2/CoordinatedOmission/wrk2\_CleanVsCO.png**



**Appendix 1**

**Workings of Jaeger and openTracing:**  
  
<https://github.com/opentracing/opentracing-lua>

General terminologies  
<https://opentracing.io/specification/>  
  
The Traces in OpenTracing are defined by their Span and their Span context. Each colored line in the Jaeger UI is basically a span and the whole set of spans for one end-to-end request is what is known as a Trace.  
  
Now, each span is related to another Span by the childOf or FollowsFrom relationship.  
Clicking on the “Trace JSON” button will pull out the complete JSON for that particular trace where you can see how different Spans are related.

**Appendix 2**

**Workings of Apache Thrift:**<https://thrift-tutorial.readthedocs.io/en/latest/intro.html>  
  
Thrift is a language-independent software stack for client-server RPC code generation.  
The above link is a tutorial that I went through to understand how the authors would have generated the files for the project.

Understanding this, I realized that the **authors have generated the same stack of code for the microservices in** **Lua, CPP and Python.**

Working on newer image for mediamicroservice

**sudo docker rmi $(sudo docker images -f "dangling=true" -q)**

removes unnecessary images

**sudo docker images --all**

shows available images

**Building image**

Change ARG NUM\_CPUS=4 in Dockerfile

**sudo docker build --no-cache -t media-microservices -m 4g .**

command to build image for mediamicroservice

**New files created/modified for read reviews**

* /home/ubuntu/DeathStarBench/mediaMicroservices/nginx-web-server/conf/nginx.conf
* /home/ubuntu/DeathStarBench/mediaMicroservices/nginx-web-server/lua-scripts/wrk2-api/review/read.lua
* /home/ubuntu/DeathStarBench/mediaMicroservices/src/MovieReviewService/MovieReviewHandler.h

require rebuild of image

sudo docker-compose build --no-cache

require picking image locally, so need to change docker-compose

sudo docker logs -f --tail 100 be9c4861809b

* /home/ubuntu/DeathStarBench/mediaMicroservices/wrk2/scripts/media-microservices/read-review.lua
* /home/ubuntu/DeathStarBench/mediaMicroservices/scripts/read\_review.sh

sudo docker logs -f --tail 100 be9c4861809b

docker container logs --follow 3f840a82aabe

look for container id for mediamicroservices\_movie-review-service\_1

docker container logs --follow container\_id

sudo docker system prune -f

sudo docker-compose down

sudo docker build --no-cache -t media-microservices -m 4g .

sudo docker-compose up -d