... / Testing

Created by Wei Shangguan, last modified on May 21, 2020

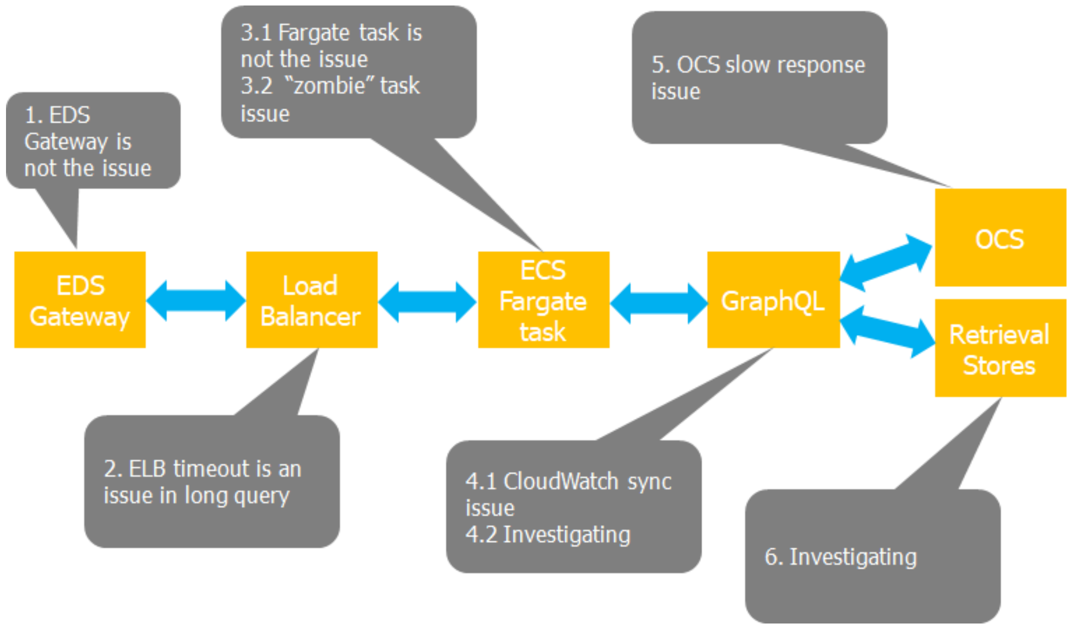
DAAPI is currently tested by EDS bulk team from EDS Gateway. But there are a lot of HTTP 502 and 504 errors in the response of the testing.

Research

Our research is focused on two parts:

1. There are several components in the end-to-end process. We need to address where the root cause happens
2. Within the component which causes the error, what we can do to improve or avoid the error?

The current status of the full picture is depicted below


3.1 Fargate task is not the issue
3.2 "zombie" task issue
5. OCS slow response issue
1. EDS Gateway is not the issue
EDS Gateway
Load Balancer
ECS Fargate task
GraphQL
OCS
Retrieval Stores
2. ELB timeout is an issue in long query
4.1 CloudWatch sync issue
4.2 Investigating
6. Investigating

Details about what we have done and are doing are:

Step 1. EDS Gateway

As HTTP 502 and 504 errors could be from the EDS Gateway or at the ELB level, DAAPI team configures the ELB to accept the inbound traffic from EDS bulk Glue directly.

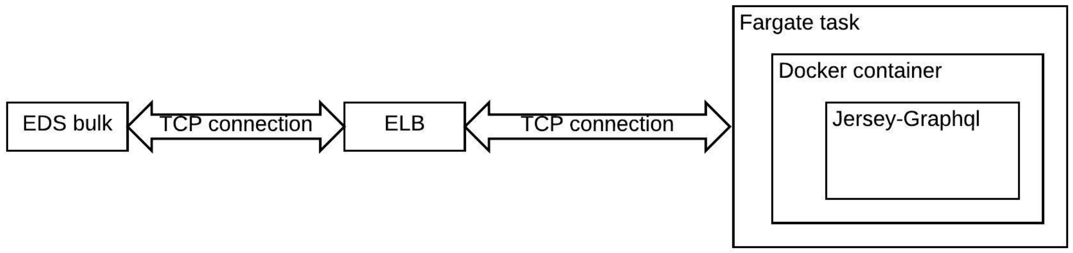
When EDS bulk visits the ELB directly, there are still a lot of HTTP 502 and 504 errors.

Conclusion: EDS Gateway is not the cause of the issue.

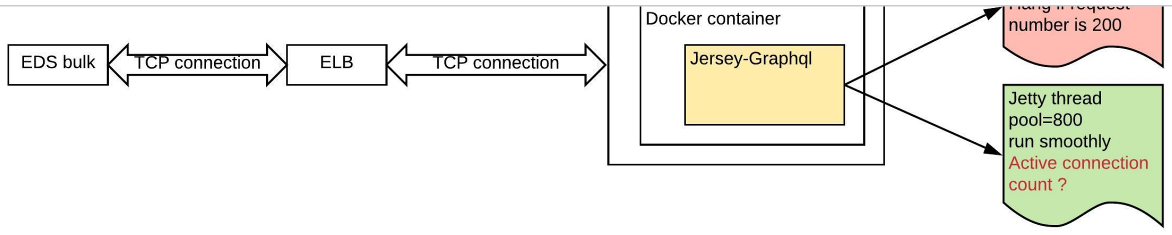
Step 2. From ELB to ECS

Excluding the EDS Gateway, the EDS bulk creates a HTTP (TCP) connection to ELB. Then ELB creates another HTTP (TCP) connection with the fargate task in ECS service.

A docker container is running in the fargate task with a Jersey webservice. Inside Jersey, there's the Graphql logical.



From local testing, that means to run the Graphql in a local machine. Then send HTTP request to the local server from JMeter. In the below diagram, the testing is on the "Jersey-Graphql" part.


Docker container
Hang if request number is 200
EDS bulk
TCP connection
ELB
TCP connection
Jersey-Graphql
Jetty thread pool=800 run smoothly
Active connection count ?

Step 2.1 Jetty

A Jetty service is inside the Jersey server. There's a thread pool inside Jetty. For each request, a thread from the Jetty thread pool is used to handle the request.
The default thread pool size is 200.

1. We set XRay enabled in the testing. The result shows the response time of OCS varies from hundreds of milliseconds to 8 or 9 second for a single query with one object ID.
2. To minimal the OCS impact, we created a mock OCS interface (along with the aux/sdl files) in the local testing. This mock OCS service could only accept two fixed queries. We can define the response time and the response content.

   mockquery.zip

   a. Also in the XRay result, we found some sequence operations in OCS request could be in parallel. We made this optimization.
   b. When the request number is closed or exceed to 200, the HTTP requests are all hang there with no response.
   c. When the Jetty thread pool size is 800, the local testing run smoothly. In current develop branch, the thread pool size is set to (200 to 1000).

Conclusion: Updating the concurrency of Jetty will reduce one piece of short-board. But this is still not the root cause.

Step 2.2 ELB

In the testing from EDS bulk team to direct connect to the ELB, there's a lot of ELB 504, 460 error and HTTP 504 error.

460 error means the TCP connection between EDS bulk and ELB is broke from the EDS bulk side. The most likely reason is HTTP connection timeout on EDS bulk side.

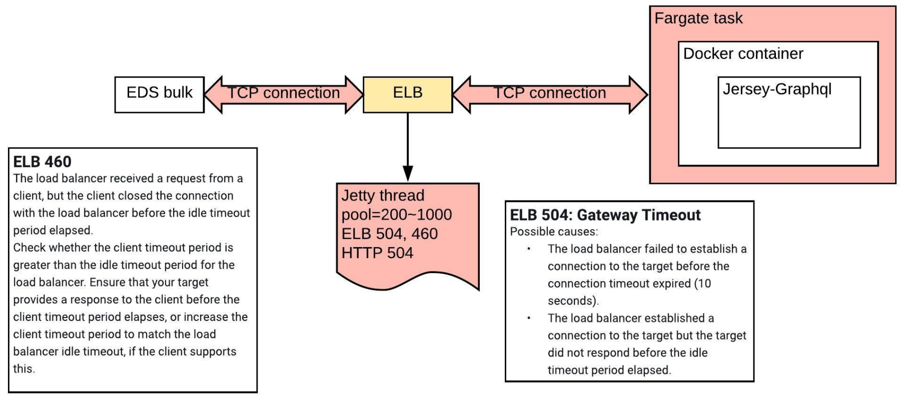
504 error, on the other side, means the TCP connection between ELB and ECS is disconnected or never created.

Then we **set the EDS bulk timeout and ELB timeout to a very large value**.

After this configuration, no such error but the query ran very very slow before response (CPU and memory are not high. the threads are all hang there)

1. The ability of a fargate task to handle multiple thread. The Graphql got slow and slow with increasing number of request. So we need to know if this is an issue of graphql or the fargate task model itself.
2. One thing need to be paid attention to is: even after the active connection count is getting down from a previous big amount, the Graphql will still response slow. Maybe some cache are there?

Conclusion: timeout will cause the requests to be terminated before get a result. but the Graphql itself got very slow with high query pressure.


Fargate task
Docker container
Jersey-Graphql
EDS bulk
TCP connection
ELB
TCP connection
ELB 460
The load balancer received a request from a client, but the client closed the connection with the load balancer before the idle timeout period elapsed.
Check whether the client timeout period is greater than the idle timeout period for the load balancer. Ensure that your target provides a response to the client before the client timeout period elapses, or increase the client timeout period to match the load balancer idle timeout, if the client supports this.
Jetty thread pool=200~1000 ELB 504, 460 HTTP 504
ELB 504: Gateway Timeout
Possible causes:
• The load balancer failed to establish a connection to the target before the connection timeout expired (10 seconds).
• The load balancer established a connection to the target but the target did not respond before the idle timeout period elapsed.

Step 3. Inside ECS service

To evaluate the query pressure on Graphql, we use JMeter to send queries to Graphql QA environment directly to see the performance.

Step 3.1 Concurrency performance

```
query($ids:[String!], $delta: DeltaArgs) {
    OrganizationReference(objectIds: $ids, delta: $delta) {
        Details {
```

```
        }
        ClassificationNAICSPrimarySchema {
            Industry {
                IndustryPermID
                IndustryUniqueName
            }
        }
        ClassificationTRBCPrimarySchema {
            Industry {
                IndustryPermID
                IndustryUniqueName
            }
        }
        OrganizationIdentifiers(filter: {OR: [{IdentifierTypeId: {EQ: "320634"}}, {IdentifierTypeId: {EQ:
"320035"}}]}) {
            IdentifierData {
                Identifier
            }
            IdentifierTypeId
        }
        Names(filter: {AND: [{OrganizationNameTypeCode: {EQ: "LNG"}},
{OrganizationNameLanguageId: {EQ: "505062"}}]}) {
            OrganizationName
            OrganizationNameLanguageId
            OrganizationNameTypeCode
            OrganizationNormalizedName
```

text file

We ran this query with 10 object ids. Changing the request thread number and task number of an ECS service to see the performance. The queries are sent repeatedly in each thread.

Thread number	Task number	Response time (sec)	
3	1	84	
60	20	697	

In the second line, the 60 threads (concurrent requests) are forwarded to 20 tasks randomly (with Round-Robbin algorithm). On average one task receives 3 threads as in the first line.

But the response time is much longer than the first testing.

Then we did a second round of testing with only 1 object id. Below are the result:

Thread number	Task number	Avg response time (sec)	Xray trace id
60	20	149	1-5eba2513-dbdbacaaef3bde5e23e9b98d
			1-5eba2496-bba4ded1638bb674b2df4f3e
			1-5eba23c4-a5e953c36a493f517c63ac11
3	1	59	1-5eba3f8e-2866ea151aa74a35d3028ab6
			1-5eba4018-1c3158b8b33464d810bad0c8
			1-5eba40e9-ccfa3f701d82d8b4fc89ec98

Similar result as with 10 ids. The more thread number (though the average thread/task is not changed), the worse performance.

Conclusion: this issue may be on ECS infrastructure level (Fargate), Graphql logical or under Graphql (say, OCS, ES). We will have further testing on the possible causes.

Step 3.2 Check the part under Graphql

To check if this is an issue on the data source (OCS and ElasticSearch), we will apply mock data source instead of the real ones.

As the query used in the series of testing just visit OCS source, we have a mock OCS server which could just response to the query used in this testing.

The mock OCS server is an API gateway + lambda (python 3.0)

Details at https://console.aws.amazon.com/apigateway/home?region=us-east-1#/apis/a4v1nog54c/stages/v1 and https://console.aws.amazon.com/lambda/home?region=us-east-1#/functions/a205941-graphql-mock-ocs?tab=configuration

Below are the results with 1 object id.

Thread number	Task number	Avg response time (sec)	Xray trace id
3	1	21	1-5eba50f9-b5fac32c8829944256952fca
			1-5eba516b-cf0e4b16168de6a6fa40d5bf
			1-5eba5205-aada3da4319458a4e84890e8
60	20	153	1-5eba5465-f666f3dded752d2972e93893
			1-5eba5466-7cbb6b1454260f9428039574
			1-5eba554c-2d91094a8d3e90389a07865b

Compare with the result of second round of step3.1, it doesn't get improved under 60 threads and 20 tasks.

Conclusion: OCS may or may be not the issue. Need more investigation. Also we need to go on checking the Fargate part and Graphql logical.

Step 3.3 More testing on OCS

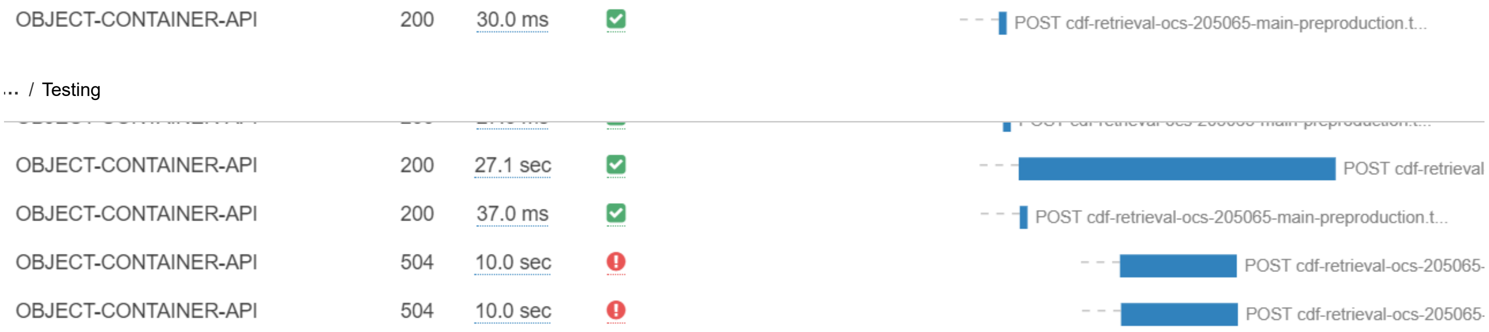
After the CloudWatch Async issue is resolved (finding 1 in Findings in performance testing), we ran the sample query with 1 object id+20 ECS tasks+60 threads.

The result is not stable. Statistics of the response of two times:

the 1st one: return after 217847 ms. XRay trace id=1-5ebe2741-d6ba5d1796cf6a8b4754891b

the 2nd one: return after only 12212 ms. XRay trace id=1-5ebe2898-9f84f5e22ff199e06efb57a2

For the first one, there are some long time response from OCS and timeout failure from OCS



OBJECT-CONTAINER-API	200	30.0 ms	✓	POST cdf-retrieval-ocs-205065-main-preproduction.t...
[illegible]	[illegible]	[illegible]	✓	[illegible]
OBJECT-CONTAINER-API	200	27.1 sec	✓	POST cdf-retrieval
OBJECT-CONTAINER-API	200	37.0 ms	✓	POST cdf-retrieval-ocs-205065-main-preproduction.t...
OBJECT-CONTAINER-API	504	10.0 sec	!	POST cdf-retrieval-ocs-205065-
OBJECT-CONTAINER-API	504	10.0 sec	!	POST cdf-retrieval-ocs-205065-

OCS team confirmed this is a known issue on their side.

Conclusion: OCS is an significant issue in Graphql performance. But we still need to check if there are any other issues in Graphql or ECS.

Step 3.4 ECS Fargate

To check if there's an issue in response to the high concurrency and long-time calculation on Fargate, we implement a new interface in Graphql to send a HTTP 200 response after 20 seconds. In the 20 seconds, keep the timestamp every 2 seconds.

Mock thread

```
@ApiType("PerfTesting")
@Singleton
@Path("/perf-testing")
public class PerfTestingController {

    @Inject
    public PerfTestingController() {}

    @ApiAlias("ThreadMock")
    @GET
    @Path("/mock-thread")
    public Response MockThreadHandler() throws Exception {
        String startTime = new Date().toString();
        String handlerId = String.format("%d", new Random().nextInt());
        String response = "{\n  \"from\": \"" + startTime + "\",\n  \"time\": [";
        for(int i = 0; i < 10; ++i) {
            try {Thread.sleep(2000);}
            catch (Exception e) {
                System.out.println(String.format("PerfThread %s: sleep failed", handlerId));
            }
            String current = new Date().toString();
            if(0 == i) response = response + "\n    \"" + current + "\"";
            else response = response + ",\n    \"" + current + "\"";
        }
        String endTime = new Date().toString();
        response = response + "\n  ],\n  \"to\": \"" + endTime + "\"\n}";
        return Response.ok(response).build();
    }

}
```

The statistic on ELB shows every thread will be returned in 20 seconds with 20 tasks and 100 threads.

Conclusion: Fargate is good at high concurrency performance. We need to focus on Graphql itself.

Step 3.5 Out of OCS

As OCS is an issue in step 3.3, we changed the query to see what happens on ElasticSearch source.

There're only limited number of dataset on ES now so our sample query doesn't contain a link inside.

The sample query, object ids and JMeter configurations are:

ES-test.jmx

We ran this query with 20 tasks and 120 threads. It ran most smoothly but still with some unexpected time spending.

Conclusion: we need to take more action on OCS and Graphql to see how much impact from each.

Findings

Findings in the process are listed and described at https://confluence.refinitiv.com/display/GRAPHQL/Findings+in+performance+testing

No labels