FLAME

FACILITY FOR LARGE-SCALE ADAPTIVE MEDIA EXPERIMENTATION

Dynamic service delivery using network-aware graph analytics and endpoint controls

Nikolay Stanchev

IT Innovation

2019-06-05

Introduction

- *Estimating end-to-end delay measurements through graph-based analytics*
- *Building a network topology graph*
- *Building a temporal graph from time-series measurements of a media service*
- *Querying for round-trip time calculations*
- *Writing round-trip-time calculations as time-series measurements*
- *Automating the full graph monitoring pipeline*
- *Creating a state-change alert policy based on the new metric*

FLAME CLMC – Cross Layer Management and Control

CLMC combines measurements from different layers of the FLAME platform and allows for the construction of new metrics which could give a better estimate of the performance of a given service.

A particular use case is the estimation of the end-to-end delay of a media service, that is the delay that a client would experience while using this service – a metric which could be broken down to two different factors:

- network-related measurements
- service-related measurements

Measurement model of the end-to-end service delay

Overall, the assumption is that the total delay of a service function could be measured using the following formula:

total_delay = forward_network_delay + service_delay + reverse_network_delay

which could be extended to the following:

total_delay = forward_latency
+ forward_data_delay (dependent on request size and bandwidth)
+ service_delay
+ reverse_data_delay (dependent on response size and bandwidth)
+ reverse_latency

Full details of these calculations can be found at https://gitlab.it-innovation.soton.ac.uk/FLAME/consortium/3rdparties/flame-clmc/blob/master/docs/total-service-request-delay.md (contributed by Stephen Phillips).

Important assumption of this simplified model is that services can measure the processing time for requests which is **forward_data_delay + service_delay + reverse_data_delay** (out-of-the-box support for nginx, tomcat, etc.)

Building the network topology graph

CLMC utilises the northbound API of the SDN controller (Floodlight in current implementation) to build up the network topology graph and retrieve the switch-to-switch (a.k.a. service routers) network latencies.

The graph is then stored in Neo4j and is supposed to be managed by the platform provider.

This is implemented as a REST-like API with the following three endpoints:

- ***POST http://platform/clmc/clmc-service/graph/network*** – builds up the network topology graph and creates new nodes and links if needed
- ***PUT http://platform/clmc/clmc-service/graph/network*** – builds up the network topology graph, but also updates the latency measurement of already existing links
- ***DELETE http://platform/clmc/clmc-service/graph/network*** - completely deletes the network topology graph

The *Neo4j* browser could be used to view the graph and explore latency measurements between service routers – ***http://platform/clmc/neo4j/browser***

Building a temporal graph for a media service

A temporal graph is simply a graph representing the state of a media service in a given time window, e.g. from start of today until end of tomorrow.

CLMC builds such graphs to calculate the round-trip time from a specific user equipment to a specific service function endpoint.

In order to build this graph, three metrics must be measured for a given service function:

- **response_time** (or service delay) – how much time it takes to process a request (seconds), that is from the moment the first byte of the request is read until the moment last byte of the response is sent.
- **request_size** – the size of a request to the service function (bytes)
- **response_size** – the size of a response from the service function (bytes)

Building a temporal graph for a media service

An example with a *Tomcat*-based service:

- use the *Tomcat Telegraf* input plugin for monitoring – includes fields like ***bytes_sent***, ***bytes_received*** and ***processing_time*** in measurement ***tomcat_connector***.
- ***processing_time*** is the total time spent processing incoming requests measured since the server has started.
- ***bytes_sent*** and ***bytes_received*** measured using the same approach.
- ***request_count*** gives the number of requests since the server has started.

- ● bytes_received
- ● bytes_sent
- current_thread_count
- current_threads_busy
- error_count
- max_threads
- max_time
- ● processing_time
- ● request_count

Building a temporal graph for a media service

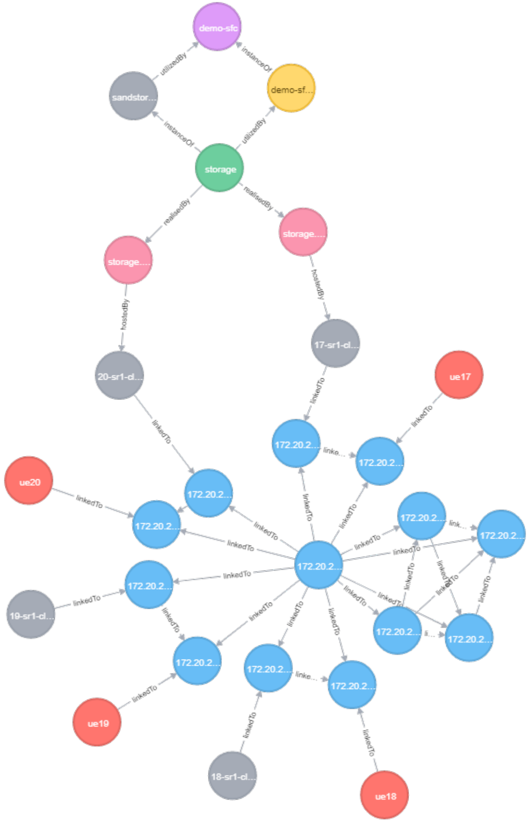
For CLMC to understand how to build a media service graph, the graph must be described in JSON format. The description must define the following:

- the time window for the temporal part of the graph (service function endpoint nodes)
- the media service (a.k.a. service function chain) identifiers
- the service function packages the media service is using
- a partial influx query for obtaining the measurement values described in the previous slide (**request/response size** and **service delay**), basically an aggregation function with a field name, e.g. *mean(processing_time)*
- the measurement name where the fields from these partial queries reside, e.g. *tomcat_connector*

Building a temporal graph for a media service

The JSON description is then sent to CLMC as the body of a POST request to **/clmc/clmc-service/graph/temporal**

```
{
    "from": 1549881060,
    "to": 1550151600,
    "service_function_chain": “demo-sfc",
    "service_function_chain_instance": “demo-sfc_1",
    "service_functions": {
        “sandstorage": {
            "response_time_field": "(last(processing_time) - first(processing_time)) / ((last(request_count) – first(request_count)) * 1000)",
            "request_size_field": "(max(bytes_received) - min(bytes_received)) / (last(request_count) – first(request_count))",
            "response_size_field": "(max(bytes_sent) - min(bytes_sent)) / (last(request_count) – first(request_count))"
            "measurement_name": "tomcat_connector"
        }
    }
}
```



Building a temporal graph for a media service

The JSON description is fixing a time window (from **1549881060** to **1550151600**, UNIX timestamps) and defining how to query the average response time, request size and response size of all endpoints that use the ***sandstorage*** service function package.

Since Tomcat's measurement model is reporting continuously increasing metrics (the value since the server has started), the partial influx queries look a bit more complicated.

For example, the average response time query: *(last(processing_time) - first(processing_time)) / ((last(request_count) – first(request_count)) * 1000)*

Simple scenario:

- processing_time measurements (measured in milliseconds) received in the time period – 41629, 41641, 41793, 41839
- requests_count measurements received in the time period – 102, 103, 108, 110

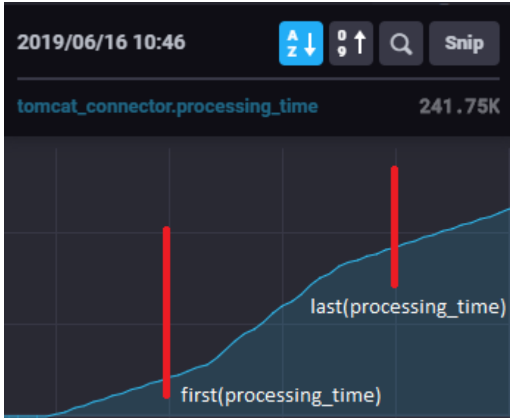
- last(processing_time) - first(processing_time) = 210 (milliseconds used to process all requests in this time period)
- last(request_count) – first(request_count) = 8 (total of 8 requests processed in this time period)
- (last(processing_time) - first(processing_time)) / (last(request_count) – first(request_count)) = 26.25

The query defined above will evaluate to 26.25ms = 0.02625s and will give us the the average delay of the service per request. The same reasoning is used for *bytes_sent* and *bytes_received* to calculate the average request size and average response size.

Building a temporal graph for a media service

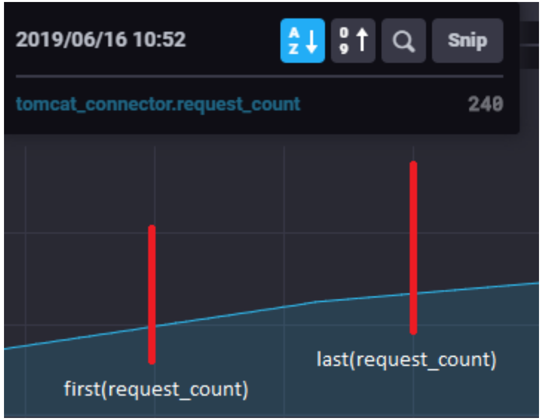
Visualisation is always more helpful than simply looking at the numbers:

- measurement time window for ‘processing_time’
- measurement time window for ‘request_count’



2019/06/16 10:46

tomcat_connector.processing_time 241.75K



2019/06/16 10:52

tomcat_connector.request_count 240

From time-series to graph data

CLMC monitoring data model:

CLMC measurement

- Media service global tags – *flame_sfc, flame_sfp, flame_sfe, flame_location, etc.*
- Through the context given by these measurements CLMC can extract the graph nodes and relationships from the time-series data.

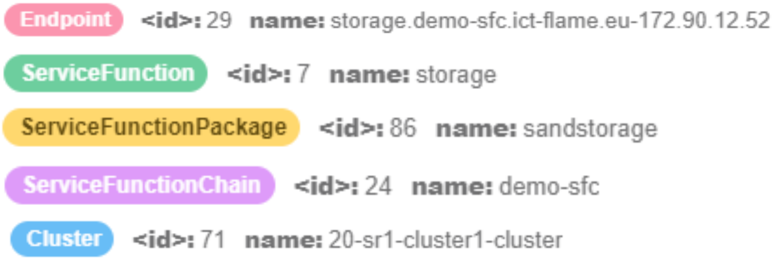
From time-series to graph data

What happens in the background is that a time-series data query is executed and the results are used to generate the media service graph in the Neo4j database:

```
SELECT {0} AS mean_response_time, {1} AS mean_request_size, {2} AS mean_response_size FROM "{3}"."{4}".{5} WHERE "flame_sfc"=\'{6}\' and "flame_sfci"=\'{7}\' and "flame_sfp"=\'{8}\' and time>={9} and time<{10} GROUP BY "flame_sfe", "flame_location", "flame_sf“
```

The placeholders are filled from the JSON configuration described in the previous slides. Depending on the results we can identify:

- Service function endpoint nodes (from the ***flame_sfe*** tag)
- Service function nodes (from the ***flame_sf*** tag)
- Service function package nodes (from the ***flame_sfp*** tag)
- Service function chain nodes (from the ***flame_sfc*** and ***flame_sfci*** tag)
- Cluster nodes (from the ***flame_location*** tag)



Endpoint **<id>:** 29 **name:** storage.demo-sfc.ict-flame.eu-172.90.12.52
ServiceFunction **<id>:** 7 **name:** storage
ServiceFunctionPackage **<id>:** 86 **name:** sandstorage
ServiceFunctionChain **<id>:** 24 **name:** demo-sfc
Cluster **<id>:** 71 **name:** 20-sr1-cluster1-cluster

Thus, the full media service graph is built on top of the network topology graph – cluster nodes being the intersection point between the two graphs.

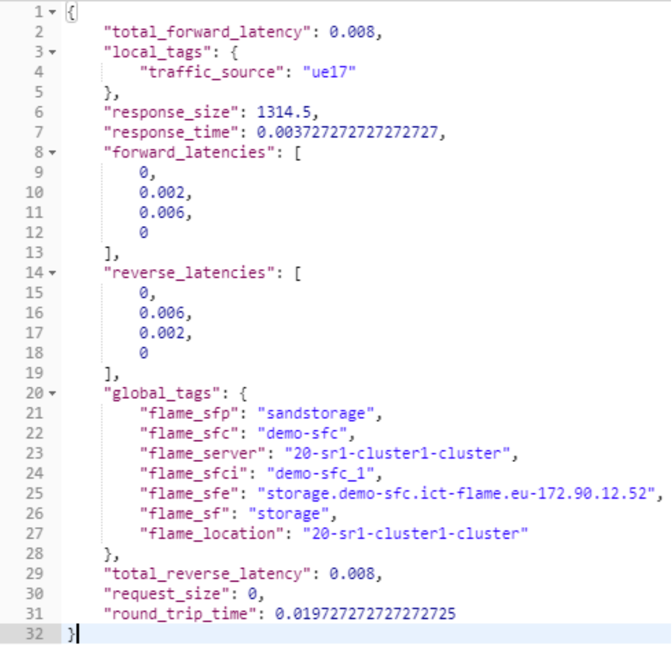
Query for round-trip-time estimation

Once the full graph has been built, CLMC offers an API endpoint to query for round-trip-time from a start point (Cluster, Switch or a User Equipment node) to an end point (Service Function Endpoint). The response is a breakdown of the round-trip-time measurement:

- ***GET http://platform/clmc/clmc-service/graph/temporal/<uuid>/round-trip-time?startpoint=<cluster, switch or ue>&endpoint=<SF endpoint>***

The API endpoint is basically a query to the temporal graph for round-trip-time; the UUID parameter uniquely identifies the subgraph.

The UUID of the temporal graph can be retrieved from the response of the build request described in the previous slides.



```json
{
    "total_forward_latency": 0.008,
    "local_tags": {
        "traffic_source": "ue17"
    },
    "response_size": 1314.5,
    "response_time": 0.003727272727272727,
    "forward_latencies": [
        0,
        0.002,
        0.006,
        0
    ],
    "reverse_latencies": [
        0,
        0.006,
        0.002,
        0
    ],
    "global_tags": {
        "flame_sfp": "sandstorage",
        "flame_sfc": "demo-sfc",
        "flame_server": "20-sr1-cluster1-cluster",
        "flame_sfci": "demo-sfc_1",
        "flame_sfe": "storage.demo-sfc.ict-flame.eu-172.90.12.52",
        "flame_sf": "storage",
        "flame_location": "20-sr1-cluster1-cluster"
    },
    "total_reverse_latency": 0.008,
    "request_size": 0,
    "round_trip_time": 0.019727272727272725
}
```

Combining network and service measurements

In the background, a Neo4j Cypher query for shortest-path is executed (based on number of hops) to retrieve the network path between the start point and the end point defined in the URL of the round-trip time query request:

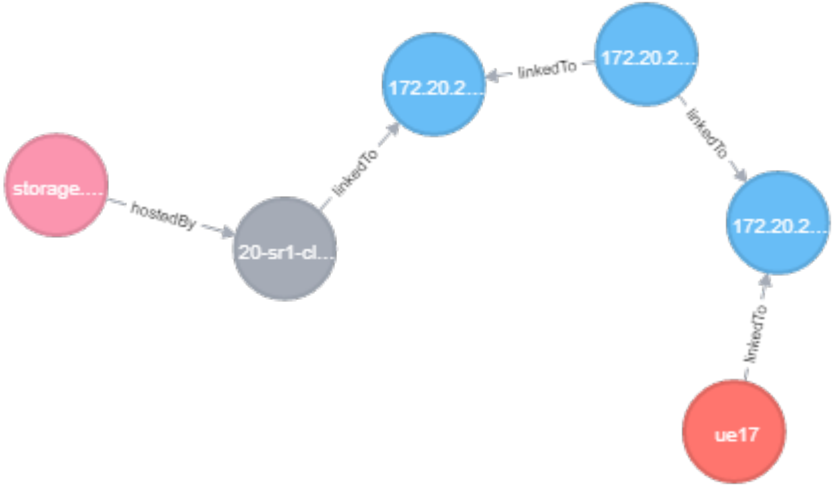
```
MATCH (startpoint:{0} {{ name: '{1}' }}),(endpoint:Endpoint {{ name: '{2}', uuid: '{3}'}}),
path = shortestPath((startpoint)-[*]-(endpoint))
WHERE ALL(r IN relationships(path) WHERE type(r)='linkedTo' or type(r)='hostedBy' )
WITH extract(y in filter(x in relationships(path) WHERE type(x) = 'linkedTo') | y.latency) as latencies,
endpoint.response_time as response_time, endpoint.request_size as request_size,
endpoint.response_size as response_size
RETURN latencies  as forward_latencies, reverse(latencies) as reverse_latencies, response_time,
request_size, response_size
```

Placeholders are filled with the request URL query parameters.

Combining network and service measurements

Through the result of the previous query, measurements from the network topology layer are combined with measurements from the application layer, i.e. the Service Function Endpoint level.

- Network measurements coming from the network path to the service function endpoint
- Application measurements coming from the temporal service function endpoint node



Converting round-trip time results to time-series data

Once the graph has been built and a round-trip time query is executed, the results can be written back as a new measurement in InfluxDB.

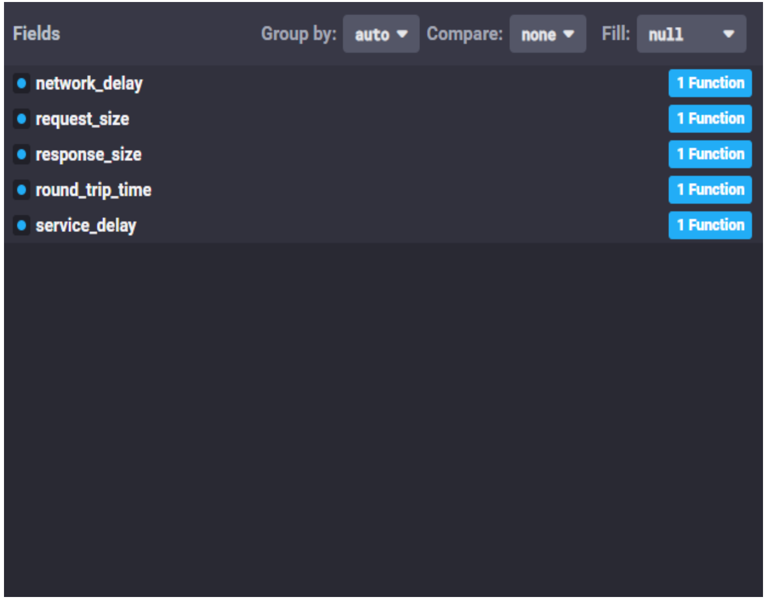
The following steps need to be followed to achieve this:

- convert the JSON response from the Graph API to Influx line protocol format

Example:

graph_measurements,flame_sfp=sandstorage,flame_sfc=demo-sfc,flame_server=17-sr1-cluster1-cluster,flame_sfci=demo-sfc_1,flame_sfe=storage.demo-sfc.ict-flame.eu-172.90.4.52,flame_sf=storage,flame_location=17-sr1-cluster1-cluster,traffic_source=ue18 round_trip_time=0.002,service_delay=0.002,network_delay=0,request_size=0,response_size=1422.5 1560675146000000000

- send a POST request to InfluxDB including the measurement line above



Fields

Group by: auto

Compare: none

Fill: null

network_delay	1 Function
request_size	1 Function
response_size	1 Function
round_trip_time	1 Function
service_delay	1 Function

Clean up the media service graph

Since the endpoints layer of the full graph is *temporal* and only valid for the defined time window, it needs to be deleted and created again if a more recent round-trip-time measurement is needed.

- **DELETE** http://platform/clmc/clmc-service/graph/temporal/<uuid>

This finishes the lifecycle of a graph monitoring activity – build temporal graph, query it , delete it.

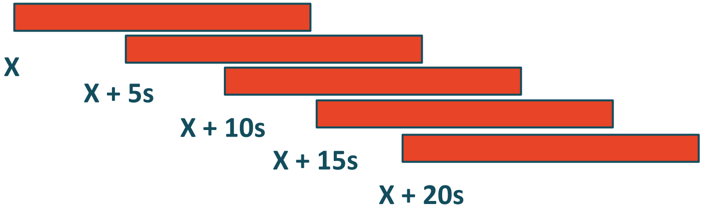
Alternatively, if the full graph of a service function chain must be deleted there is a separate API endpoint to use:

- **DELETE** http://platform/clmc/clmc-service/graph/static/<SFC identifier>

Graph monitoring

Ultimately, the media service provider can choose how to manage it. Two main strategies:

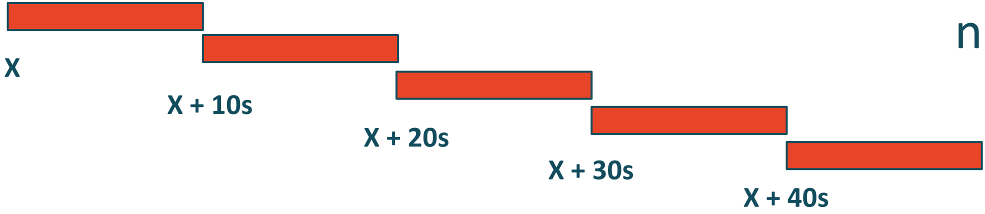
Sliding window:



overlapping time periods

Out-of-the-box support from CLMC

Tumbling window:



non-overlapping time periods

Automating the end-to-end delay monitoring process

CLMC offers the full graph-based pipeline as a service. An API endpoint allows the activation of a graph monitoring process, running in the background on CLMC constantly executing the pipeline described in the previous slides.

A JSON configuration, similar to the one used in the build request for a temporal graph, is sent to CLMC to start a graph monitoring process

- **POST** http://platform/clmc/clmc-service/graph/monitor

The difference is that instead of defining a time window, we define a query period (e.g. 30 seconds, that is how often the pipeline script will execute) and the name of the measurement where results will be written in.

Automating the end-to-end delay monitoring process

Example JSON description:

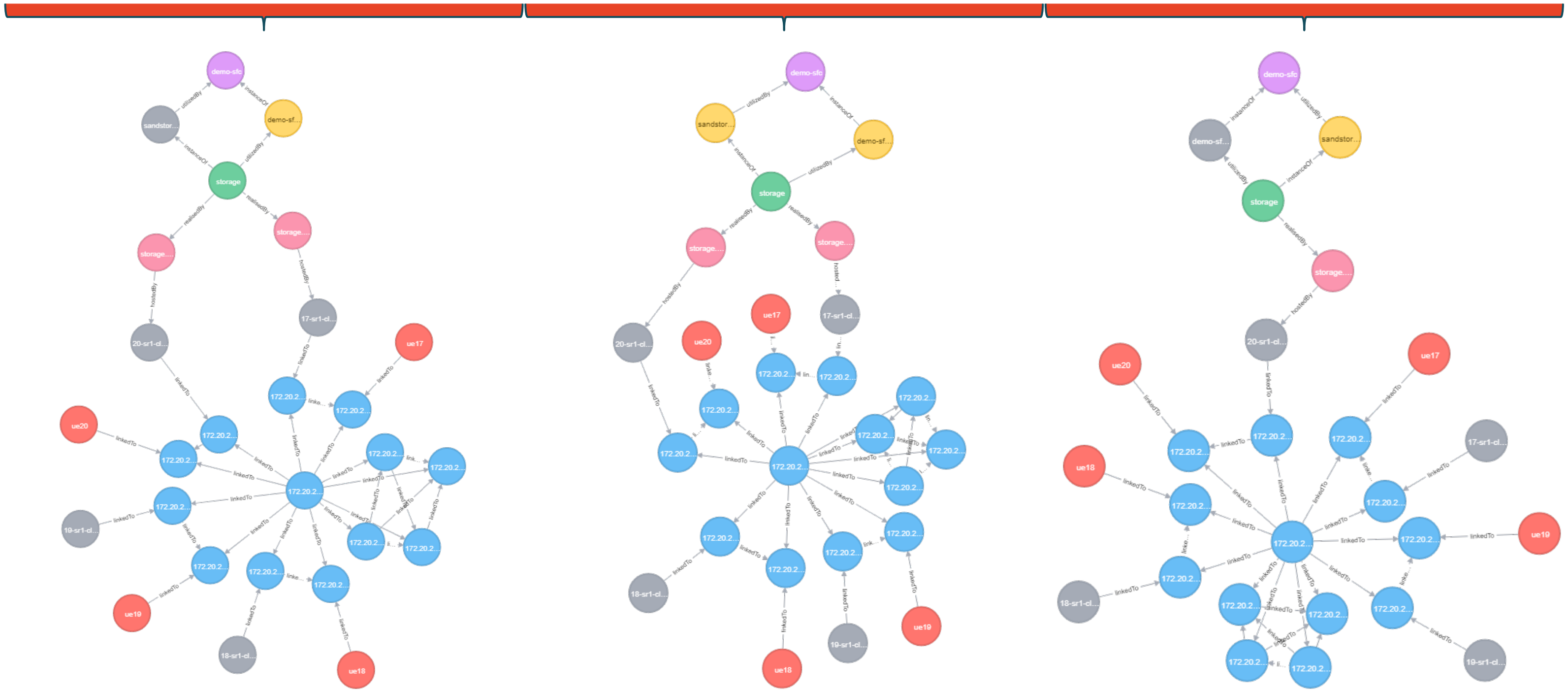
```
{
    “query_period": 30,
    "results_measurement_name":  “graph_measurements",
    "service_function_chain": “demo-sfc",
    "service_function_chain_instance": “demo-sfc_1",
    "service_functions": {
        “sandstorage": {
            "response_time_field": "(last(processing_time) - first(processing_time)) / ((last(request_count) – first(request_count)) * 1000)",
            "request_size_field": "(max(bytes_received) - min(bytes_received)) / (last(request_count) – first(request_count))",
            "response_size_field": "(max(bytes_sent) - min(bytes_sent)) / (last(request_count) – first(request_count))"
            "measurement_name": "tomcat_connector"
        }
    }

}
```

- the pipeline executes every 30 seconds building a temporal graph for the time window between *now() – 30s* and *now()*
- end-to-end delay metrics will be written into measurement named *graph_measurements*

Automating the end-to-end delay monitoring process

Runtime execution of the graph monitoring process:

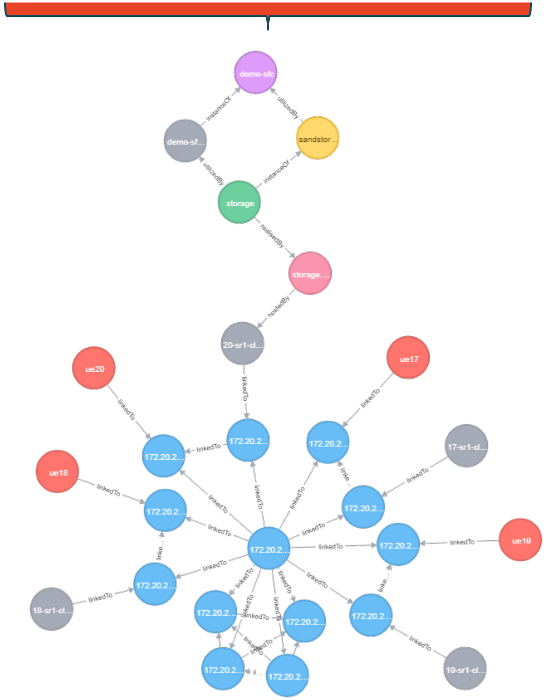
Timestamp X ***X + 30s*** ***X + 60s*** ***X + 90s***

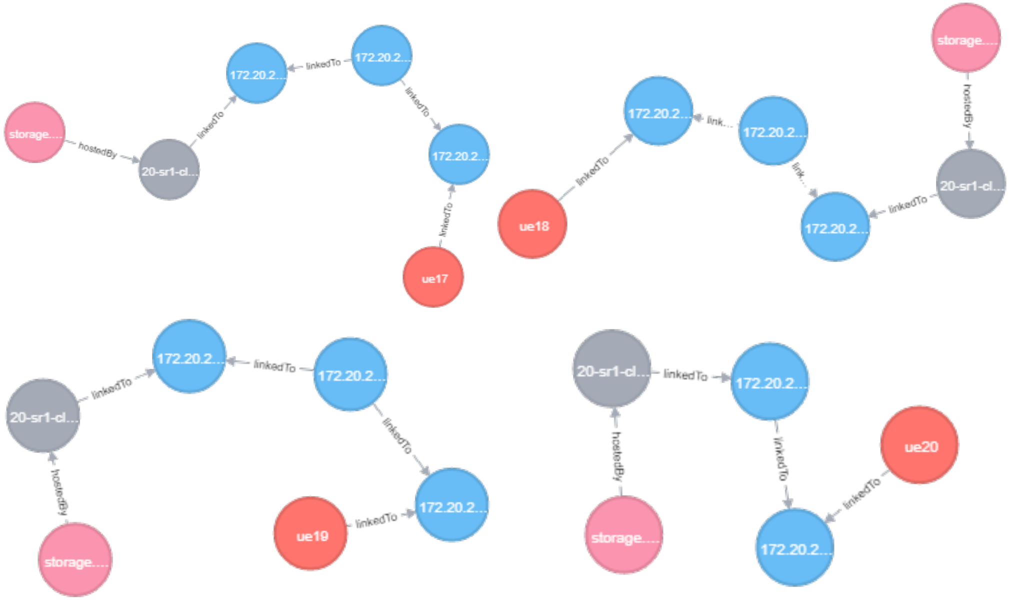


Automating the end-to-end delay monitoring process

Runtime execution of the graph monitoring process – querying for round-trip time from all UE nodes:

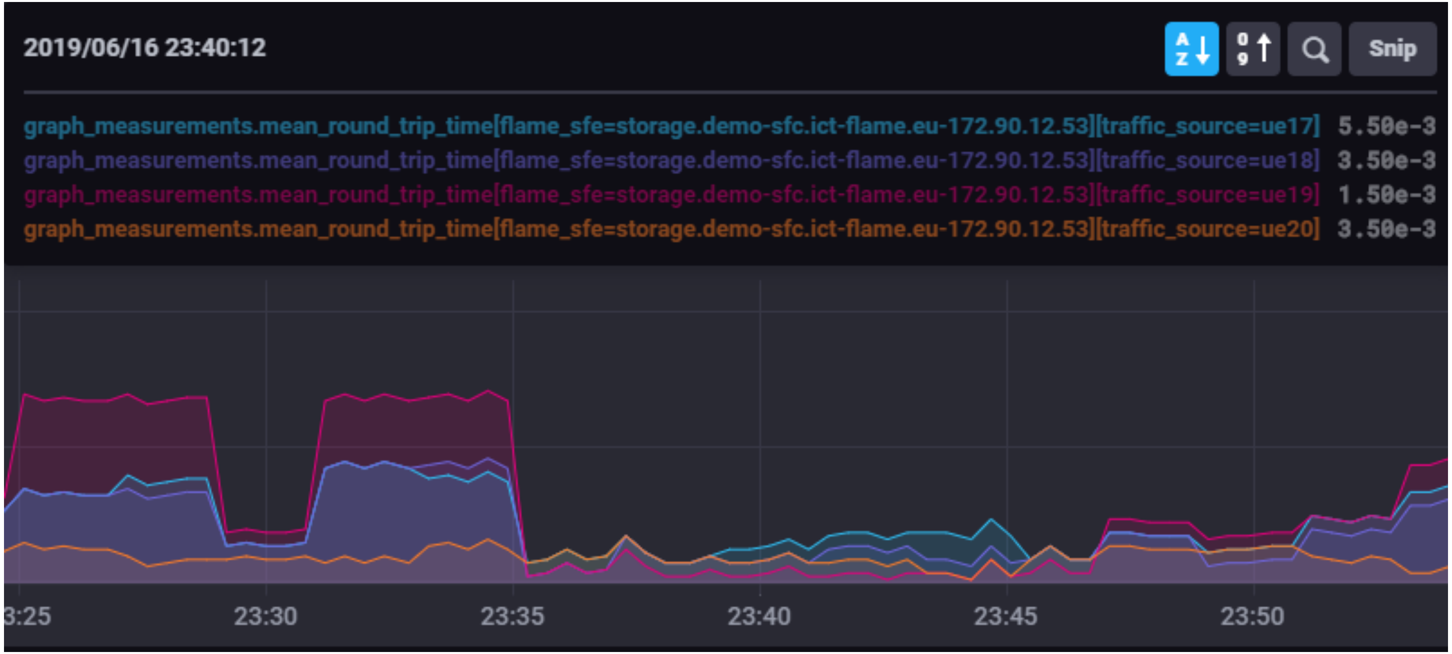
Timestamp X ***X + 30s***





Automating the end-to-end delay monitoring process

Runtime execution of the graph monitoring process – writing back the results from all round-trip time queries into InfluxDB and generating a new measurement with contextualised data:



2019/06/16 23:40:12

graph_measurements.mean_round_trip_time[flame_sfe=storage.demo-sfc.ict-flame.eu-172.90.12.53][traffic_source=ue17] 5.50e-3
graph_measurements.mean_round_trip_time[flame_sfe=storage.demo-sfc.ict-flame.eu-172.90.12.53][traffic_source=ue18] 3.50e-3
graph_measurements.mean_round_trip_time[flame_sfe=storage.demo-sfc.ict-flame.eu-172.90.12.53][traffic_source=ue19] 1.50e-3
graph_measurements.mean_round_trip_time[flame_sfe=storage.demo-sfc.ict-flame.eu-172.90.12.53][traffic_source=ue20] 3.50e-3

Managing the graph monitoring process

As with any other monitoring agents or measurement plugins, we need to be able to manage this graph pipeline.

A graph monitoring process can be stopped:

- **DELETE** http://platform/clmc/clmc-service/graph/monitor/<uuid>

or we can check its status:

- **GET** http://platform/clmc/clmc-service/graph/monitor/<uuid>

The UUID identifying a graph monitoring process is retrieved from the CLMC response for the request that started it.

Creating an alert policy for the round-trip time metric

Now that we have a new metric, we can create a simple threshold alert policy which will boot a second service function endpoint once the average round-trip time performance exceeds the given value:

```
scale_out:
    event_type: threshold
        metric: graph_measurements.round_trip_time
        condition:
            threshold: 2
            granularity: 30
            comparison_operator: gte
            resource_type:
                flame_sf: storage
                flame_location: 20-sr1-cluster1-cluster
    action:
        implementation:
            - flame_sfemc
```

Usually, it is the *flame_sfe* tag used to identifier an endpoint, runtime generation though – therefore, we use the combination of the two tags *flame_sf* and *flame_location* to identifies the first service function endpoint.

Creating an alert policy for the round-trip time metric

In addition, we also create a trigger to stop the second service function endpoint when it is not doing any work:

```
scale_in:
    event_type: threshold
        metric: graph_measurements.round_trip_time
        condition:
            threshold: 0.5
            granularity: 65
            comparison_operator: lt
            resource_type:
                flame_sf: storage
                flame_location: 17-sr1-cluster1-cluster
    action:
        implementation:
            - flame_sfemc
```

Again, the combination of the two tags *flame_sf* and *flame_location* is used to identify the second service function endpoint.

FLAME

This project received funding from the European Union's Horizon2020 research and innovation programme under grant agreement No 731677

THANK YOU FOR YOUR ATTENTION

ICT-FLAME.EU

@ICT_FLAME