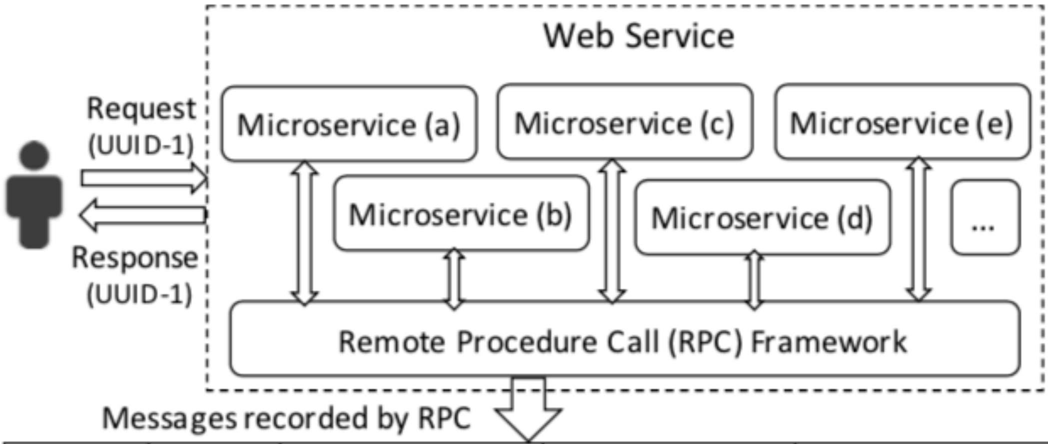
ANM 2020 project：MicroService System Troubleshooting

introduction

Troubleshooting of a microservice-based large software system is very challenging due to the large number of underlying microservices and the complex call relationships between them. In this project, you will design an online algorithm to do the anomaly detection and root cause troubleshooting. And you will deploy the program that is able to consume real-time data to do the test part in your Tencent Cloud Virtual Machine(CVM).

MicroService System



Message order	UUID	Sending time of (m → n) at m (msec)	Receiving time of (m → n) at n (msec)	Message (m→n)
①	UUID-1	-	1519747202138	call(start→a)
②	UUID-1	1519747202144	1519747202146	call(a→b)
③	UUID-1	1519747202149	1519747202151	call(b→c)
④	UUID-1	1519747202149	1519747202150	call(b→d)
⑤	UUID-1	1519747202155	1519747202156	*response*(c→b)
⑥	UUID-1	1519747202159	1519747202160	call(d→e)
⑦	UUID-1	1519747202188	1519747202190	*response*(e→d)
⑧	UUID-1	1519747202194	1519747202196	*response*(d→b)
⑨	UUID-1	1519747202253	1519747202256	call(b→e)
⑩	UUID-1	1519747202323	1519747202324	*response*(e→b)
⑪	UUID-1	1519747202355	1519747202356	*response*(b→a)
⑫	UUID-1	1519747202360	-	*response* (a→end)

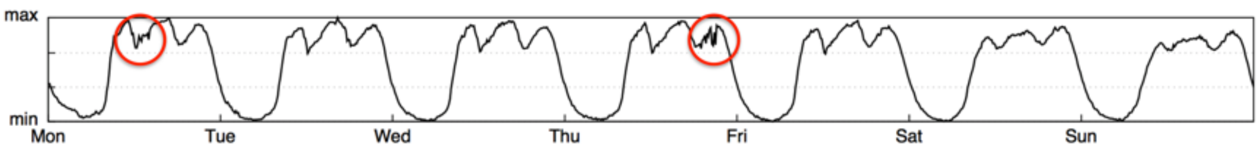
Recently, microservice architecture has become more and more popular for large-scale software systems in web-based services. This architecture decouples a web service into multiple microservices, each with well-defined APIs. There are complex call relationships between different microservices. Each microservice can be individually upgraded and maintained in microservice system. The above figure shows a specific user request to a microservice-based web service, in which the request is completed through several calls between microservices.

In this project, microservices are deployed in virutal machines(host) and you need to deal with 3 kinds of data sources. All these data are time series data and will be introduced in the subsequent part.

Anomaly Detection

Time series data can present several unexpected patterns (e.g., jitters, slow ramp-ups, sudden spikes and dips) in different severity levels, such as a sudden drop by 20% or 50%.

Anomaly Detection of time series data with the format of (timestamp, value) can be formulated as follows: for anytime t, given historical observations $x_{t-T+1}, \cdots, x_t$, determine whether an anomaly occurs (denoted by $y_t = 1$). An anomaly detection algorithm typically computes a real-valued score indicating the certainty of having $y_t = 1$, *e.g.*, $p(y_t = 1|x_{t-T+1}, \ldots, x_t)$, instead of directly computing y_t. Human operators can then affect whether to declare an anomaly by choosing a threshold, where a data point with a score exceeding this threshold indicates an anomaly.



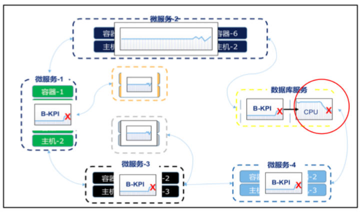
TroubleShooting

A failure in the microservice system can lead to many anomalous behaviors of different KPIs on different components due to the system's complex interactions between microservices. The root cause of a failure needs to be located as soon as possible, otherwise the failure will cause huge loss.

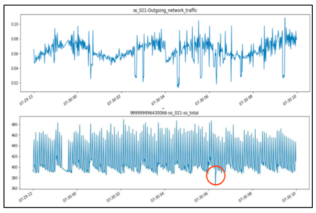
In this project, the task is to find out system nodes (vm or docker) and KPIs where the root cause occurs when a failure happens. More concretely, a feasible process of troubleshooting in this project is shown in the figure below:

serviceName	startTime	avg_time	num	succee_num	succee_rate
osb_001	1.594599e+12	0.0	552.0	0.0	0.0
osb_001	1.594599e+12	0.0	588.0	0.0	0.0
osb_001	1.594599e+12	0.0	575.0	0.0	0.0
osb_001	1.594599e+12	0.0	576.0	0.0	0.0

Downgrade of success rate in ESB business indicator



Find anomalous component in trace around the downgrade time



Find the anomalous KPI in the abnormal node

The throubleshooting in the figure has 3 steps:

1. Find the time point t when the business success rate was significantly lower than 1.
2. Around the time point t, look into the anomalous behaviors of microservices and record containers or hosts where the microservices are deployed. For example, microservice

`csf_001` in docker node `docker_003` had very long response time arount the time point t. Typically, this step can be finished by analysing the trace data.
3. In step2, the abnormal nodes, hosts or containers, are found. And then, you can detect which KPIs of the nodes perform anomalously.

Data

In this project, you will get 3 types of KPI data sources. In this section, the format of all these data will be introduced.

ESB business indicator(ESB)

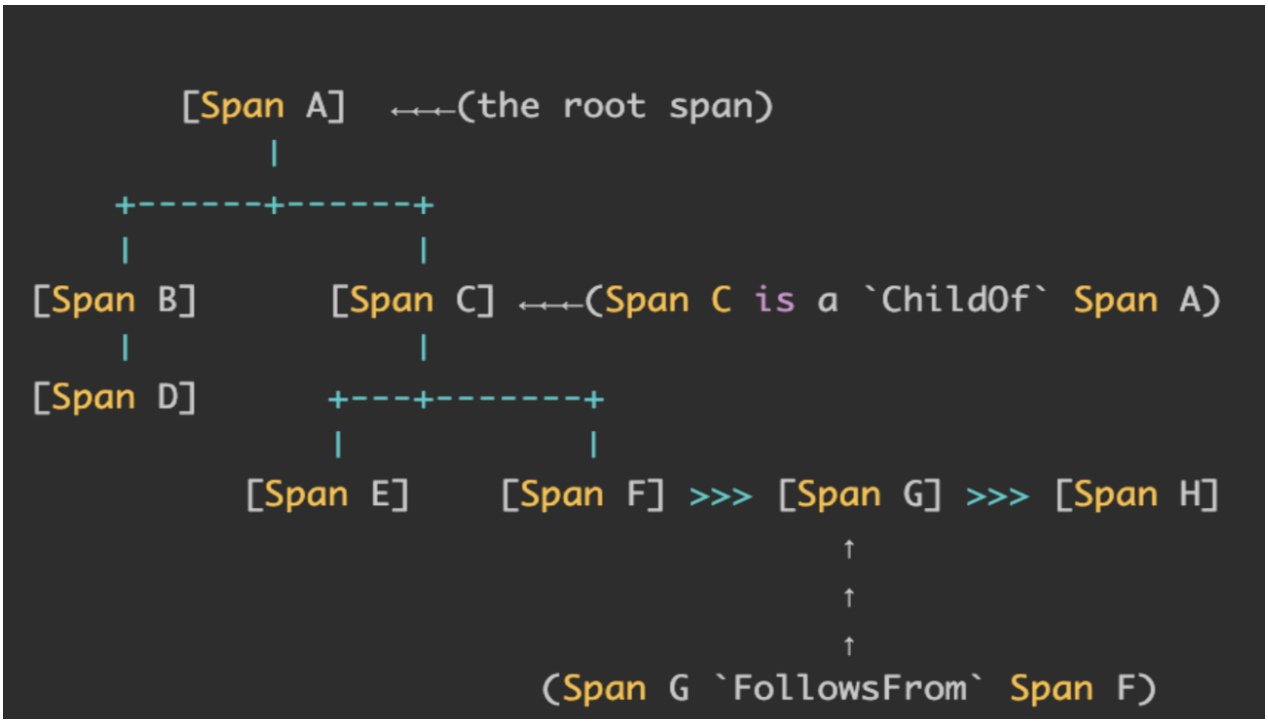
serviceName	startTime	avg_time	num	succee_num	succee_rate
osb_001	1588262400000	0.4718	361	361	1.0
osb_001	1588262460000	0.4915	343	343	1.0
osb_001	1588262520000	0.4901	359	359	1.0
osb_001	1588262580000	0.5824	359	359	1.0
osb_001	1588262640000	0.4923	385	385	1.0

In this project, the microservice system deals with ***only 1*** kind of service request `osb_001`. The ESB data records the request information **every minute**:

- serviceName: service name, always `osb_001` in this project.
- startTime: information of request [startTime, startTime + 1min) is recorded in this row.
- avg_time: average time spent processing a request.
- num: the number of submitted requests.
- succee_num: the number of submitted requests which are successfully completed.
- succee_rate: $\frac{\#\{succee\_num\}}{\#\{num\}}$

Trace

A trace corresponds to a user request and has a unique traceID. A trace consists of several microservice call records, called as `span`. A trace example is shown as the below figure:



As shown in the picture, the spans forms a tree structure, which means that every span except the root has a parent span. The parent relationship represents the call relationship between two microservices.

Span mainly records 4 useful attributes about a microservice call:

- start time
- elapsed time
- host
- microservice name

It is important to notice that span is divided into 2 categories(**inside span** and **outside span**) according to where the span is recorded. A Example will be given to explain how spans make up a traces.

```
def foo():
    print('foo begin time is: ', time.now) # t1
    ...
    print('call bar begin time is:', time.now)    # t3
    bar()
    print('call bar end time is:', time.now) # t4
    ...
    print('foo end time is :', time.now)    # t2


def bar():
    print('bar begin time is: ', time.now) # t5
    ...
    print('bar end time is: ', time.now) # t6

foo() # user request
```

In the above example, the user calls the function `foo` and there is a call statement in `foo` to call the function `bar`. Intuitively, the user request (calling `foo`) can be considered as a trace: $foo \rightarrow bar$ consisting of 3 spans:

id	parent id	start_time	elapsed_time	service	host	category
span1	None	t1	t2 - t1	foo	host of foo	inside
span2	span1	t3	t4 - t3	bar	host of foo	outside
span3	span2	t5	t6 - t5	bar	host of bar	inside

As shown in the code block and table, a span corresponds to 2 logs (print statements) recording start time and end time respectively. The inside span(span1 and span3) records the service being processed while the outside span(span2) records the service that will be called. It is important to notice that the **host** column is **where the span is generated**.

Specifically, a real span has the following attributes:

- id: unique id of this span.
- pid: id of its parent span.
- traceId: id of trace the span belongs to. spans with the same `traceId` make up a trace.
- startTime: `start_time` in table.
- elapsedTime: `elapsed_time` in table.
- serviceName: `service` in table.
- cmdb_id: `host` in table.
- callType: there is six calltypes in the trace data: `osb`, `remoteprocess`, `flyremote`, `csf`, `local` and `jdbc`. spans in `osb`, `remoteprocess` and `flyremote` are inside span and the others are outside span.
- success: `True` or `False` representing whether the service is processed successfully.
- dsName: There is a column named `dsName` in `trace_local` and `trace_jdbc`, which is the database accessed by microservice. And in `jdbc`, you can regard accessing databases as the microservice.

Host KPIs data

Host KPIs data are the time series data with the format of (timestamp, value). There are multiple KPIs in each host (db, linux vm, docker...) and the host name is consistent with the `cmdb_id` column in the trace data and KPIs data.

itemid	name	bomc_id	timestamp	value	cmdb_id
999999998651280	CPU_free_pct	ZJ-002-056	1588521600000	98.119746	db_008
999999998650980	CPU_free_pct	ZJ-002-056	1588521600000	98.837793	db_003
999999998650680	CPU_free_pct	ZJ-002-056	1588521600000	98.994754	db_001
999999998651100	MEM_real_util	ZJ-002-053	1588521600000	81.740000	db_007
999999996381601	CPU_free_pct	ZJ-002-056	1588521601000	95.593747	db_009

There are 6 attributes in the KPIs data:

- (itemid, name, bomc_id): **the type of KPIs**. For example, `itemid` values are different between the 1st line and the 2nd line in the above table, thus, these 2 lines describe different KPIs (you can see that the `cmdb_id` values are different in the first 2 lines).
- timestamp, value: the KPI is the `value` at the time of `timestamp`.
- cmdb_id: the host name of KPI. It is the same as `cmdb_id` in the trace data.

Evaluation

Your program need to give a list of predicted root causes for every failure in the microservice system in real time.

For a failure that occurs at time T, your program should give the troubleshooting result within 10 minutes, that is, before the time $T + 10m$. The format of result is a list with elements of length 2: [`cmdb_id` of KPI, `name` of KPI]. If you find that the root cause is on a host, but you cannot locate the specific KPIs, which will happen with some network failures, you can give only the `cmdb_id`.

An example of a specific result is as follows:

```
1970-01-01T00:01:00.000000000Z [["docker_003","container_cpu_used"]]
1970-01-01T00:11:00.000000000Z [["docker_003","container_cpu_used"],
["docker_004",null]]
```

You shouldn't print the time information `1970-01-01T00:01:00.000000000Z`. The evaluatioin program will get time information automatically.

Let N be the total number of failures during the test. T_i(*$i \le N$*)**, the time when the *ith* failure occurs, will not given to your program.** The evaluation will be based on the last answer submitted by your program in the time window of $[T_i, T_i + 10m]$. Meanwhile, **results submitted after the $2N$ times are invalid.**

There will be 2 results, ground truth and list generated by your program, after each failure. $\lceil \frac{T_i' - T_i}{F_\beta \times M_i \times 10} \rceil$ will be used as your score for the *ith* failure, where T_i' is the time when your program submits the result, F_β is the F-beta-score between 2 lists, M_i is the number of correct elements in your list and β is 0.5 . And then, in each failure, groups are sorted in ascending order according to the scores, the first one gets the highest points, and the last one gets the lowest points. Finally, the total ranking of each group is determined according to the total points;

More details can be found in: https://github.com/NetManAIOps/aiops2020-judge/tree/master/final.

Other information

data

https://cloud.tsinghua.edu.cn/f/7eece510dc784e70a083/?dl=1

References

https://cloud.tsinghua.edu.cn/f/e06aaab7135c44e8beec/?dl=1; There are 6 slides **in Chinese** from the teams who have finished the competition.

you can also find more on: https://workshop.aiops.org

Consumer

There is a example program named `consumer.py`, which shows how to read data from `Kafka` and how to `submit` (line 74) your answer to the server.

Test part

We provide 12 hours of test data and they will be produced repeatly in the Kafka queue. From 0:00 to 12:00 and 12:00 to 24:00, the test data will be output completely from the beginning to the end, meanwhile, the `timestamp` in the test data will be adjusted to keep same as real time. To get your score, you need to `submit` your answer to the server (`submit` function in `consumer.py`).

If your program is deployed and works normally from 0:00 to 12:00, **your answer for a test** will be submited to the server and the server will calculate your score and show on the board.

And the rank board will be on:

http://81.70.98.179:8000/standings/show/