Multi-Layer Observability for Fault Localization in Microservices Based Systems

Rupashree Rangaiyengar
Indian Institute of Science, India
rupashreeh@iisc.ac.in

Raghavan Komondoor
Indian Institute of Science, India
raghavan@iisc.ac.in

Raveendra Kumar Medicherla
TCS Research, India
raveendra.kumar@tcs.com

***Abstract*—For cloud native microservice monitoring and incident detection, companies and developers tend to largely focus only on generating logs, metrics, and traces at the application layer. However, in order to enable precise fault localization, it is necessary to access and correlate logs pertaining to a single end-user request across non-application layers as well, such as the load balancer at the front and the database at the back end. In this paper, we propose an observability library and an observability platform that addresses this problem and generates alerts that precisely point to fault locations. Logs at multiple layers are tagged with a common request identifier that helps in performing correlation. The observability platform is architected such that it lends itself to extensions to catch multiple types of errors and issues. The proposed observability platform has been tested on five open source benchmarks. The results confirm that our tool can be used deterministically and precisely to detect elusive issues.**

***Index Terms*— Logs, Metrics, HTTP 504 Gateway Timeout, Deterministic Fault Localization, Alerts.**

I. Introduction

In the microservices architectural style, a software application is organized as a suite of loosely coupled services. Each microservice handles a single business concern and can be built, maintained, and deployed independently of the other microservices in the application. This approach allows developers to test and release the software faster. This is the most preferred architectural paradigm for service scaling when the incoming request load is a critical factor [1].

The different services typically invoke each other and hence have inter-dependencies. If a service becomes unavailable or becomes too slow in responding to requests due to any sort of fault or issue, the observable symptom could be a timeout of the outermost load balancer (aka *Gateway*) that is waiting for the faulty service. This makes it challenging for deployment engineers to localize the service that is the root cause of the observed loss of availability (i.e., rejected requests).

A. A motivating example

Figure 1 depicts an interaction (or *trace*) in a simple food delivery microservices application that we had developed as an exemplar [2]. (A) A user with customer ID 101 issues a HTTP GET request to the system to fetch the balance in their wallet, which is accepted by the *load balancer*. (B) The "Delivery" service is the root or outermost layer that receives all the requests. The load balancer forwards the request to one of the instances of the delivery microservice. (C) The "Delivery" service makes a HTTP GET call to the "Wallet" service to fetch the balance for the customer. (D) The "Wallet" microservice makes a blocking database call to the MySQL database service. (E) The query is extremely slow due to some reason and does not return a result quickly. (F) The load balancer hits a pre-configured timeout for the request and returns a HTTP 504 response "Gateway timeout exception" to the end-user. There is no indication in this response as to what was the root cause of the timeout.

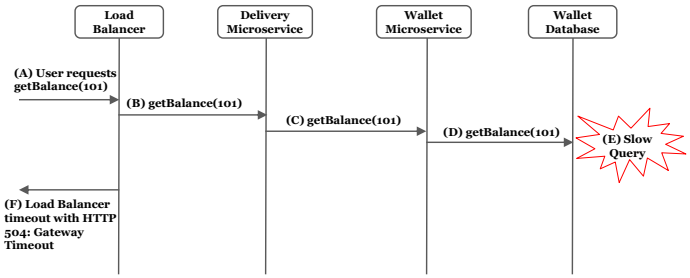

Fig. 1: Wallet Get Balance throwing Gateway Timeout

B. Challenges and issues in fault localization

The inability to quickly address 504 responses could impact the quality of the service leading to customer dissatisfaction. Deployment engineers would like to identify the root cause of a timed-out request and resolve the root cause in a timely manner to prevent subsequent requests from meeting the same fate. A simple investigation into load balancer logs cannot reveal which layer, service, or component is the root cause of the timed-out request. The returned error message is entirely misleading as it portrays that the root cause for the timeout is the gateway (i.e., load balancer), whereas, in reality, the root cause is somewhere deeper down. In the absence of any efficient aids to find the root cause, the one investigating why a request timed out has to manually go through the logs of the load balancer, application services, and database and then correlate the timestamps of the key events in these logs with the timeout timestamp at the gateway. However, this task can get very complicated due to other concurrent requests also

potentially reaching the slow service or database and hence also getting timed out. Therefore, it is non-trivial to localize the faulty layer quickly.

There are real instances where such an incorrect diagnosis has been reported. For example, Fowler [3] describes a real example where a Redis data store ran out of memory and subsequent requests showed timeout responses, masking the original cause of failure. In another instance, an Amazon Dynamo DB outage in US-EAST-1 in 2015 [4] was triggered by a transient network problem that overwhelmed the metadata servers. This caused higher latency to serve requests, which caused timeouts and subsequent retries of the same requests by clients, which eventually compromised the end-to-end availability of the entire system.

C. Our contribution

We present a novel approach, which is an *observability library* and *observability platform*, which enables investigators to *quickly* find out the *correct* layer that ultimately caused a 504 timeout error at the gateway for a request. The incident remediation can then be focused on rectifying or restarting this problematic faulty layer. The crux of our approach and novelty compared to known existing solutions is to insert a unique *request ID* into each request when it arrives, and propagate this ID through all layers as the request is processed, and hence pin down precisely the layer that was the root cause of a timeout of the request if the request got timed out.

D. Critique on existing approaches

Both practitioners and researchers have proposed different observability platforms and monitoring tools for microservices. These tools instrument and monitor specific service layers and provide various reports. Picoreti et al. [5] utilize infrastructure logs with application logs to enhance observability. This approach does not extend observability to database logs and load balancer logs. Therefore, the utility of this approach is limited. *Pharos* [6] is an observability platform built in the industry using open-source technology and the public cloud. The paper does not indicate whether a layer that specifically causes a request to timeout can be pinned down automatically. Experimental evaluation results with precision and recall are not available in the paper.

Garcia et al. [7] provides a tracing tool for distributed heterogeneous applications, where application-specific event IDs re-construct traces that cross thread boundaries, thus reducing the need for structural source-code modification. Mace et al. provided a tool *Pivot* [8], which uses Lamport's happens-before relation to filter and group events based on properties of any events that causally precede them in execution. They describe a case study on how *Pivot* detected the HDFS Replica Selection bug. Both of these papers do not address multi-layer and end-to-end observability that enables accurate alerting.

II. Our Approach

The main goal of our approach and tool is to provide multi-layer observability for detecting which layer (e.g., microservice or database) was the root cause of each timed-out request. Our tool's output is in the form of alerts. Our approach issues an alert for an end-user request under the following two scenarios as of now:

- A slow database query issued by some microservice invoked as part of processing the request causes the gateway (i.e., end-user facing load balancer) to timeout and return the 504 response code.
- A slow microservice that is invoked during the request-processing does not respond in time to its invoking microservice (or to the invoking gateway, if it was the first microservice involved in the trace), which causes the gateway to timeout and respond with 504.

Note, these are known to be among the common causes of outer-level gateway timeouts. Each alert contains a request ID that uniquely identifies the timeout request and the name of the microservice or database instance that caused the timeout.

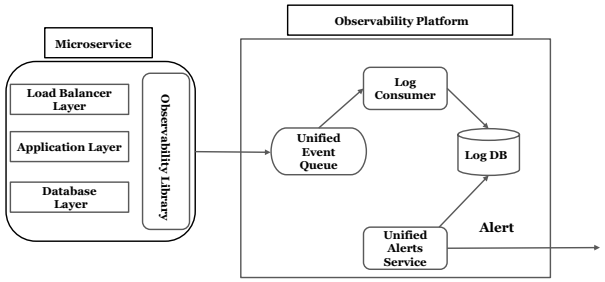

Fig. 2: Architecture of observability library and platform

The overall architecture of our approach is summarized in Figure 2. Our approach has two main components – logging and alerting. The log records are emitted by an "Observability library" that we provide, which needs to be linked to the application, and which logs certain key events occurring at all layers of the system (as depicted in the left part of the figure). The alerts are generated by our "Observability platform", which runs concurrently with the application as a separate service, and processes the log records to raise alerts.

A. Observability library

The observability library emits all log records in a standardized format to make it easier to implement the subsequent observability platform. The log records have several fields, but a log record emitted on behalf of a specific layer may use only a subset of the fields. In the interest of space, we list below some of the key fields in the log record:

- **Request ID** (*rid*): A unique ID identifying the outermost request (to the gateway), which while being processed at some layer, triggered this log record.
- **Layer name** (*layer*): The layer that emits this log record. It could be Load Balancer, Application, or Database.

- **Layer event** (*event*): The kind of event being logged. The different event kinds that we log are described subsequently in this section.
- **Current Method** (*cm*): The controller method that is in execution and causing this log record. Valid only in the application layer.
- **Time stamp** (*ts*): The time at which this log record was generated. We make an assumption in our work that the clocks of all microservices are synchronized.

Controllers are basically programmatic methods in microservices that receive requests from gateways or from other microservices. To use our system, the application needs to be linked to our library, and the annotations "@Log" and "@DBLog", which are defined by our library, need to be inserted in the source code by the application developer just before each controller method definition (these annotations need no parameters). The above-mentioned annotations are implemented in our library using aspect-oriented programming and cause certain events occurring at each layer to be logged, as discussed below.

a) Gateway layer: This layer, also known as the outer-level load balancer, receives the original end-user request. We assume that this layer assigns a unique ID to each incoming request. Cloud providers generally have this facility, e.g., *X-Amzn-Trace-Id* by Amazon Web Services. Our key idea, which underpins our entire approach, is to percolate this unique ID through the entire trace (through all layers) that executes to fulfill this request. It is this ID that is placed in the *rid* fields of all log records emitted throughout this trace. The following events that occur at the gateway layer are logged by our approach.

- **Load Balancer Entry**: We configure the cloud-proivider's gateway to generate and pass the *rid* in the HTTP header to the initial application controller that receives this request. (In Figure 1 this controller would be in the 'Delivery' microservice.) The gateway also typically logs this event using its own mechanism (not our logging mechanism). The '@Log' annotation on the receiving controller invokes our observability library, which looks up the gateway's log to find the time-stamp of creation of this *rid*, and then emits a log record of type 'load balancer entry' on behalf of the gateway to our observability platform.
- **Load Balancer Exit**: This represents the event of the gateway responding back to the end-user when the request processing finally completes or times out. This log is emitted directly by our alerting service, and hence we discuss it in Section II-B.

b) Application layer: The *Application Layer* consists of the microservices, which in turn contain controllers that receive requests. Outer-level microservices may invoke controllers in inner-level microservices; e.g., the Delivery microservice contacts the Wallet microservice in Figure 1. The '@Log' annotation on each controller causes the following events to be logged during the execution of the controller.

if "Database Entry" log for r exists and the next log record for r is a Load Balancer Exit by timeout **then** alert("timeout due to DB layer", r).
if "Application Continue" log for r exists and the next log record for r is a Load Balancer Exit by timeout **then** alert("timeout due to slow service", r).

Fig. 3: Generating alerts for a request ID r

- **Application Entry**: The beginning of execution of the controller. The *rid* is obtained from the header of the request coming into this controller.
- **Application Continue**: The currently executing controller is about to place a (blocking) request to another microservice. The *rid* is present in the header of the incoming request to the executing controller and is passed along to the other microservice via the new request's header.
- **Application Exit**: The currently executing controller is about to respond to its requester normally (no exception thrown).
- **Application Exception**: The currently executing controller throws an application exception, and the requester gets notified of this.

c) Database layer: Whenever a controller makes a database request, the '@DBLog' annotation in the controller ensures the logging of the following events.

- **Database Entry**: Sending the query to the database.
- **Database Exit**: Receipt of the response to the controller from the database.

B. Observability Platform

The Observability Platform is a separate service that runs concurrently with the application and generates alerts. The log events generated as described above are processed by this service. It organizes the log records as a map, keyed by the unique request identifier (*rid*), with the value against each *rid* being the set of all logs corresponding to this *rid* emitted so far. The alerts service identifies when any request with *rid* r has exited the system with a normal response to the end-user by the load balancer or has been timed out by the load balancer, by querying the load balancer's internal logs periodically. When such an exit is detected, it generates the "Load Balancer Exit" log for this request with *rid* r, sorts all the log records for r by their timestamp (*ts*) field, and then invokes the alert generation algorithm described in Figure 3.

III. Implementation

The "Observability library" is built as a Spring Boot, Java 11, and Maven compiled library. The annotations are aspects built using the AspectJ library. Once any log record is created within the library, it is pushed to the Observability Platform's "Unified Event Queue" (UEQ). This UEQ is an event streaming platform built using Kafka version 3.3.1 [9].

Benchmark	Description	Github stars	# micro-services
SJWT [10]	Sample app for JWT token based authentication	1200	1
RBAC [11]	User Management through Role Based Access Control	55	1
PayBill [12]	Online bill payment system	5	4
Comic [13]	Comic bookstore	1	1
Bank [14]	Online banking	2	3

TABLE I: Benchmarks

Benchmark	Ground Truth			Number of Alerts	Precision	Recall
	Service	DB	Total			
SJWT	0	7	7	7	100.0%	100.0%
RBAC	3	9	12	12	100.0%	100.0%
PayBill	5	9	14	14	100.0%	100.0%
Comic	6	5	11	11	100.0%	100.0%
Bank	5	3	8	8	100.0%	100.0%

TABLE II: Performance of our approach

The Kafka Consumer for the log events fetches the log records and inserts them into a *LogDB*. The LogDB is an RDS MySQL database instance. The "Observability Platform" is built as a Spring Boot, Java 11 microservice. This service uses a configurable Cron job scheduler pattern that periodically fetches the log records from the database and runs the alert detection algorithm specified in Figure 3. We can get alert notifications in three distinct ways: by email, SMS, and by writing to the local file system. The platform is easily extendable and supports the addition of extra notification channels.

IV. Tool Evaluation

In this section, we provide the details of the tool evaluation.

A. Benchmarks

We searched for open-source benchmarks on Github to evaluate our approach. The search terms we used were "Spring Boot", "Java", "Microservices", and "MySql". Our current tooling works only on applications that use REST APIs (without any front-end specific logic in the backend code), use the MySQL database, and do not use any asynchrony or internal load balancers in their internal architecture. Extensions would certainly be possible in our tool to overcome these limitations. We identified the top five benchmarks (by Github stars) that matched our criteria and used them in our subsequent evaluations. We have provided some information about the selected benchmarks in Table I. We manually added the observability annotations @Log and @DBLog to the controller methods in the benchmarks.

B. Workload

We then proceeded to create a *workload* (set of end-user HTTP requests) for each benchmark, so that we could test whether our platform catches timeouts that occur when processing these requests. We started off by fixing two specific requests – a POST request followed by a related GET request – for each benchmark. We created these requests suitably after understanding the benchmarks. For e.g., the workload for SJWT [10] comprises a POST request for signing up a new user followed by a GET request to search for the same user.

We use the popular load testing tool Gatling [15] to send the requests. We parameterized Gatling such that it would add a total of one hundred new (virtual) users every minute over a 10-minute duration, using 50 parallel threads. Each virtual user sends the two requests mentioned earlier, and therefore, there will be a total of 2000 distinct requests over ten minutes in a run for a benchmark. The requests of different users differ in their request parameters or payload, but not in the URLs or request paths. In order to account for non-determinism, we do three runs for each benchmark; the same 2000 requests are used in each run, but the schedule and concurrency of these requests may differ due to non-determinism.

We ran our Gatling script from an Ubuntu machine, which has an Intel Core i7-6700 CPU with 32 GB memory. We hosted the benchmarks on Amazon AWS EC2 "t2.large" instances, with 8 GB memory and a 3.0 GHz Intel Scalable Processor.

C. Experiment Setup

We did not observe any naturally occurring 504 timeouts for our benchmarks using the workload mentioned above. Increasing the workload intensity could have helped, but we still might not have been able to ensure that timeouts occurred without facing network bottlenecks first from our local Ubuntu machine. Therefore, to evaluate if our system catches timeouts, we injected a slow service or a slow database query fault during the processing of randomly chosen requests. The actual slowdown was effected using *sleep* statements manually placed at suitable points in the benchmark sources, guarded by conditionals that would become true with very low probability. The sleep duration was set to be longer than the load balancer timeout duration. Both service and database slowdown occurrences were injected using sleep statements.

We also record an entry into a *Ground Truth File* (GTF) for every request for which a slowdown was injected. Each entry contains the timestamp at which the sleeping started, the *rid* for the request, and the layer where slowdown occurred (database or service). We use this GTF subsequently to compute the precision and recall of our approach. Note that there can be at most one entry per *rid* in the GTF.

D. Evaluation

Table II summarizes the results from our approach. Columns 2-5 are cumulative numbers over the three runs for the benchmark. The three "Ground Truth" columns summarize the total number of slowed-down requests as per the GTF, segregated by layer. Column 5 depicts the number of alerts raised by our platform.

We say that an alert *matches* with an entry in the GTF when both have the same *rid* and the same layer (i.e., database or service layer). The *precision* of our approach is defined as the percentage of all alerts that match some entry in the GTF, while *recall* is defined as the percentage of GTF entries that

Benchmark	Ground Truth			Alerts	Alerts
	Service	DB	Total	W=10	W=50
SJWT	0	7	7	9 (3)	21 (4)
RBAC	3	9	12	16 (4)	31 (5)
PayBill	5	9	14	20 (4)	39 (6)
Comic	6	5	11	15 (4)	35 (5)
Bank	5	3	8	12 (3)	28 (4)

TABLE III: Alerts raised by a baseline approach

have matching alerts. In our runs, we observed 100% precision and recall for all our benchmarks.

E. A baseline

To serve as a baseline, we simulated a naive approach similar to what practitioners would employ today when they don't have access to sophisticated observability tooling. For each entry in the Ground Truth File (GTF), say the timestamp in the entry is T_1 and the unique Request ID is rid_1. A script we built searches for "Application Entry" log records whose timestamp is up to W milliseconds before T_1, and emits an alert for each such log record, where W is a configurable parameter. The *rid*'s in the Application Entry logs are not used in this process, since it is our approach (and not a naive approach) that propagates the unique *rid* of each request across all layers. Table III indicates the number of alerts emitted by the naive approach for different values of W (in milliseconds). The number within brackets indicates, of the alerts emitted, how many were due to Application Entry records that actually contained the request ID rid_1 (i.e., are true alerts).

For $W = 0$, no alerts were reported by the naive baseline for any of the benchmarks. It is notable that the number of alerts is somewhat close to the number of ground truth sleep events for $W = 10$ and too few or too many for other values of W. In other words, the precision and recall of the approach are highly sensitive to this parameter.

Consider the first benchmark SJWT given in Table III. The naive baseline produced 9 alerts with $W = 10$, and the GTF recorded 7 slowed-down requests. Only 3 alerts, though, correspond to actually slowed-down requests. As a result, the recall is 43% and the precision is 33% for $W = 10$, which is substantially below the recall and precision obtained by our approach.

F. Artifacts

We have provided the source code for our observability library and platform in a repository [2]. We have also provided a script that will run a simple exemplar application that we had developed that we had introduced in Section I. The script will also run our platform and will generate alerts from the run of the exemplar application. We have also uploaded a video recording of this run to the repository.

V. Discussion and future work

Our approach reveals very encouraging results from our experimentation and evaluation so far. It is the first approach that can deterministically link the layer that is the root cause of a timed-out request and has the potential to help deployment engineers quickly remediate issues as timeout requests arise.

One item of future work in our plans is to experiment with our approach on a greater variety of benchmarks. We would also like to evaluate the tool under heavier workloads, so that timeouts may occur naturally without us having to inject them. An automatic log annotation addition strategy can be used to minimize developer involvement. We would also like to in the future measure the resource usage of the queues, alert service, etc., as well as any performance loss in the program caused by the injection of logging via the library.

Currently, our tool can identify two very common types of root causes for load balancer timeouts. We would like to extend the approach to identify other root causes such as servers or VM instances reaching very high CPU or memory utilization. The challenge here would be to link infrastructure-related slowdown occurrences with specific requests that happen to get timed out due to these occurrences.

References

[1] W. Hasselbring, "Microservices for scalability: Keynote talk abstract," in *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ser. ICPE '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 133–134. [Online]. Available: https://doi.org/10.1145/2851553.2858659

[2] "Observability Tool Source Code," https://doi.org/10.6084/m9.figshare.21563730, Nov. 2022.

[3] S. J. Fowler, *Production-Ready Microservices: Building Standardized Systems Across an Engineering Organization*, 1st ed. O'Reilly Media, Inc., 2016.

[4] L. Nolan, "How to avoid cascading failures in distributed systems," Feb. 2020. [Online]. Available: https://www.infoq.com/articles/anatomy-cascading-failure/

[5] R. Picoreti, A. Pereira do Carmo, F. Mendonça de Queiroz, A. Salles Garcia, R. Frizera Vassallo, and D. Simeonidou, "Multilevel observability in cloud orchestration," in *IEEE DASC/PiCom/DataCom/-CyberSciTech*, 2018, pp. 776–784.

[6] S. Vippagunta, K. Finnigan, and K. Pusukuri, "Pharos: The observability platform at workday," *SIGOPS Oper. Syst. Rev.*, vol. 56, no. 1, p. 51–54, jun 2022. [Online]. Available: https://doi.org/10.1145/3544497.3544505

[7] G. Garcia, "Melange: A hybrid approach to tracing heterogeneous distributed systems," 2019.

[8] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems," *Commun. ACM*, vol. 63, no. 3, p. 94–102, Feb 2020. [Online]. Available: https://doi.org/10.1145/3378933

[9] "Kafka documentation," Jun. 2022. [Online]. Available: https://kafka.apache.org/

[10] M. Urraco, "Spring boot jwt," 2022. [Online]. Available: https://github.com/murraco/spring-boot-jwt

[11] A. Giassi, "Role based access control - user management microservice," 2022. [Online]. Available: https://github.com/andreagiassi/microservice-rbac-user-management

[12] K. L. Wickraasinghe, "Online electricity bill payment system," 2022. [Online]. Available: https://github.com/Kavindulakmal/MicroServices-SpringBoot

[13] J. Akademie, "Comic book store application," 2017. [Online]. Available: https://github.com/javaakademie/Spring-Boot-Microservice-Java8

[14] A. Garde, "Simple bank application," 2022. [Online]. Available: https://github.com/adityagarde/spring-bank-microservices-app

[15] "Gatling reference documentation - injection," 2022. [Online]. Available: https://gatling.io/docs/gatling/reference/current/