sre-with-java-microservices

Chapter 1. Application platform

To manage microservices, an organization needs to standardize specific communication protocols and supporting frameworks.

Monitoring

Metrics, logs, and distributed tracing are three forms of observability that enable the measure of service availability and aid in debugging complex distributed systems problems. Before going further in the workings of any of these, it is useful to understand what capabilities each enables.

Availability

Availability signals measure the overall state of the system and whether that system is functioning as intended in the large. It is quantified by service level indicators (SLIs). These indicators include signals for the health of the system (e.g., resource consumption) and business metrics like number of sandwiches sold or streaming video starts per second.

SLIs are tested against a threshold called a service level objective (SLO) that sets an upper or lower bound on the range of an SLI. SLOs in turn are a somewhat more restrictive or conservative estimate than a threshold you agree upon with your business partners about a level of service you are expected to provide, or whats known as a service level agreement (SLA). The idea is that an SLO should provide some amount of advance warning of an impending violation of an SLA so that you dont actually get to the point where you violate that SLA.

L-USE METHOD

- Latency
- Usage
- Saturation
- Errors

Utilization and saturation may seem similar at first, and internalizing the difference will have an impact on how you think about charting and alerting on resources that can be measured both ways. A great example is JVM memory. You can measure JVM memory as a utilization metric by reporting on the amount of bytes consumed in each memory space. You can also measure JVM memory in terms of the proportion of time spent garbage collecting it relative to doing anything else, which is a measure of saturation. In most cases, when both utilization and saturation measurements are possible, the saturation metric leads to better-defined alert thresholds.

Chapter 2. Application metrics

Dimensional metrics

For example, an application-wide counter metric named http.server.requests that contains a tag for an HTTP method of which only GET and POST are ever observed.

If you are starting with real-time application monitoring now, you should be using a dimensional monitoring system.

Naming metrics

To get the most out of metrics, they need to be structured in such a way that selecting just the name and aggregating over all tags yields a meaningful (if not always useful) value. For example, if a metric is named http.server.requests, then tags may identify application, region (in the public cloud sense), API endpoint, HTTP method, response status code, etc.

An aggregate measure like throughput for this metric of all unique combinations of tags yields a measure of throughput for every interaction with many applications across your application stack.

Assuming many applications are instrumented with some metric such as http.server.requests, when building a visualization on http.server.requests the monitoring system will display an aggregate of the performance of all of the http.server.requests across all applications, regions, etc. until you decide to dimensionally drill down on something.

Common tags

Meter filters allow you to add common tags to accomplish exactly this, enriching every metric published from an application with additional tags. Pick common tags that help turn your metrics data into action. Following are common tags that are always useful:

- application name
- cluster and server group name
- instance name
- stack: dev, pre, pro, test

Classes of meters

Gauges

Gauges are a measure of an instantaneous value that may increase and decrease over time.

The speedometer and fuel level on a vehicle are classic examples of gauges.

Think of a gauge as a meter that only changes when it is observed.

Counters

Counters report a single metric, a count. The Counter interface allows you to increment by a fixed amount, which must be positive.

When building graphs and alerts off of counters, generally you should be most interested in measuring the rate at which some event is occurring over a given time interval. Consider a simple queue. Counters could be used to measure things like the rate at which items are being inserted and removed.

WHEN A COUNTER MEASURES OCCURRENCES, IT MEASURES THROUGHPUT

When we talk about the rate of occurrences of things happening, conceptually we are talking about throughput. When displaying counters as a rate on a chart, we are displaying a measure of throughput, the rapidity with which this counter is being incremented. When you have the opportunity to instrument an operation with metrics for each individual occurrence, you should almost always use timers instead, which provide a measure of throughput in addition to other useful statistics.

SHOULD I USE A COUNTER OR A TIMER (OR A DISTRIBUTION SUMMARY)?

Never count something you can time. And if the base unit is not a unit of time, then the corollary is, awkwardly: never count something you can record with a distribution summary.

Timers

Timers are for measuring short-duration latencies and the frequency of such events. All implementations of Timer report at least a few individual statistics.

- Count: A measure of the number of individual recordings of this timer. For a Timer measuring an API endpoint, this count is the number of requests to the API. Count is a measure of throughput.

The count statistic of a timer is individually useful. It is a measure of throughput, the rate at which the timed operation is happening. When timing an API endpoint, its the number of requests to that endpoint. When measuring messages on a queue, its the number of messages being placed on the queue.

- Sum: The sum of the time it took to satisfy all requests. So if there are three requests to an API endpoint that took 5 ms, 10 ms, and 15 ms, then the sum is 5 + 10 + 15 = 30 ms. The sum may be shipped literally as 30 ms, or as a rate, depending on the monitoring system.

Count and Sum Together Mean Aggregable Average

A sum of 1 second may be bad for an individual request to a user-facing API endpoint, but 1,000 requests of 1 ms each yielding a sum of 1 second sounds quite good!

Sum and count together can be used to create an aggregable average. If we instead published the average of a timer directly, it couldnt be combined with the average data from other dimensions (such as other instances) to reason about the overall average.

Note that average should not be used if possible at all. Use distributions.

- max: The individual timing that took the longest, but decaying over an interval.
- SLO boundaries, i.e total number of request observed that are less than X. Percentiles. Histograms

Standard deviation

Ignore it. Standard deviation is not a meaningful statistic for real-world timmings.

Percentiles/Quantiles

Percentiles are a special type of quantile, described relative to 100%. In a list of 100 samples ordered from least to greatest, the 99th percentile (P99) is the 99th sample in order. In a list of 1,000 samples, the 99.9th percentile is the 999th sample.

By this definition, percentiles, particularly high percentiles, are useful for determining what most users are experiencing (i.e., P99 is the worst latency that 99 out of 100 users experienced). For timings, percentiles usefully cut out the spiky behavior of VM or garbage collection pauses while still preserving majority user experience.

It is tempting to monitor a high-percentile value like the 99th and feel at ease that your users are experiencing good response times. Unfortunately, our intuition leads us astray with these statistics. The top 1% typically hides latencies that are one or two orders of magnitude larger than P99.

Any single request will avoid the top 1% exactly 99% of the time. When considering N requests, the chance that at least one of these requests is in the top 1% is (1 - 0 . 99 N) 100 % (assuming these probabilities are independent, of course). It takes surprisingly few requests for there to be a a greater than majority change that one request will hit the top 1%. For 100 individual requests, the chance is (1 0 . 99 100) 100 % = 63 . 3 % !

Consider the fact that a user interaction with your system likely involves many resource interactions (UI, API gateway, multiple microservice calls, some database interactions, etc.). The chance that any individual end-to-end user interaction experiences a top 1% latency on some resource in the chain of events satisfying their request is actually much higher than 1%. We can approximate this chance as (1 - 0 . 99 N) 100 % . If a single request in a chain of microservices experiences a top 1% latency, then the whole user experience suffers, especially given the fact that the top 1% tends to be an order of magnitude or more worse in performance than requests under the 99th percentile.

The best we can do with precomputed percentiles is to simply plot all of the values and look for outliers.

Because of the limitations of precomputed percentiles, if you are working with a monitoring system that supports histograms, always use them instead.

Histograms

Metrics are always presented in aggregated form to the monitoring system. The individual timings that together we represent as the latency of a block of code are not shipped to the monitoring system. If they were, metrics would no longer have a fixed cost irrespective of throughput.

or example, for an API endpoint latency histogram, we care more about the distinction between 1, 10, and 100 ms latencies than we do about 40 s and 41 s latencies. The latency buckets will be more granular around the expected value than well outside the expected value.

Importantly, by accumulating all the individual timings into buckets, and controlling the number of buckets, we can retain the shape of the distribution while maintaining a fixed cost.

SLOs

I.e imagine the following scenario

- 90% better than 20 milliseconds
- 99.99% better than 100 milliseconds
- 100% better than 2 seconds

We will configure Micrometer then to publish SLO counts for 20 milliseconds, 100 milliseconds, and 2 seconds. And we can simply compare, for example, the ratio of requests less than 20 milliseconds to the total number of requests; and if this ratio is less than 90%, then alert.

Shipping SLO boundaries does have an effect on total storage in the monitoring system and memory consumption in your application. However, because typically only a small set of boundaries is published, the cost is relatively low compared to percentile histograms.

Percentile histograms and SLOs can be used together. Adding SLO boundaries simply adds more buckets than would be shipped with just a percentile histogram.

Publishing SLO boundaries is a far cheaper (and accurate) way of testing whether the Nth percentile exceeds a certain value. For example, if you determine that an SLO is that 99% of requests are below 100 ms, then publish a 100 ms SLO boundary. To set an alert on violations of the SLO boundary, simply determine whether the ratio of requests below the boundary to total requests is less than 99%.

Distribution

It is similar to a timer structurally, but it records values that do not represent a unit of time. For example, a distribution summary could be used to measure the payload sizes of requests hitting a server.

Long Task Timers

The long task timer is a special type of timer that lets you measure time while an event being measured is still running. A timer does not record the duration until the task is complete. Long task timers ship several statistics:

- active: number of executions in progress
- total duration: sum of all in-progress exectuion times
- max: longest in-progress timing
- histograms, percentiles

Choosing the right meter type

Never gauge something you can count, never count something you can time

Chapter 3. Debugging with observability

While logs, distributed traces, and metrics are three distinct forms of telemetry with unique characteristics, they roughly serve two purposes: proving availability and debugging for root cause identification.

Logs

The context of a log is scoped to an event. Log data provides context into the execution behavior of a particular interaction.

Distributed tracing

A distributed tracing system can reason about a user interaction end to end across the whole system. So for a given request that was known to exhibit some degradation, this end-to-end view of the satisfaction of a user request shows which part of the distributed system was degraded.

Metrics

Metrics are presented in aggregate and are used to understand some service level indicator (SLI) as a whole, rather than providing detail about the individual interactions that, taken together, are measured as an SLI.

WHICH OBSERVABILITY TOOL SHOULD YOU CHOOSE?

Tracing is preferable to logging whenever possible because it can contain the same information but with richer context.

Where tracing and metrics overlap, start with metrics because the first task should be knowing when some system is unavailable.

Adding additional telemetry to help remediate problems can come later.

When you do add tracing, start at places where timed metrics instrumentation exists, because it is likely also worth tracing with a superset of the same tags.

Components of a distributed trace

A full distributed trace is a collection of individual spans, which contain information about the performance of each touchpoint in the satisfaction of an end-user request.

Spans contain a name and set of key-value tag pairs, much like metrics instrumentation does.

Sampling

No matter how smart the sampling strategy, it is important to remember that data is being discarded. Whatever collection of traces you get as a result are going to be skewed in some way. This is perfectly fine when you are pairing distributed tracing data with metrics data. Metrics should alert you to anomalous conditions and traces used to do in-depth debugging when necessary.

Sampling strategies fall into a few basic categories, ranging from not sampling at all to propagating sampling decisions down from the edge.

- No sampling
- Rate limiting samplers
- Probabilistic samplers

Summary on debugging

In this chapter, weve shown the difference between monitoring for availability and monitoring for debugging. The event-based nature of debugging signals means they tend to want to grow proportionally with increased throughput through a system, a cost-limiting measure is necessary. Different methods of sampling to control cost were discussed. The fact that debugging signals are typically sampled should give us pause about trying to build aggregations around them, since every form of sampling discards some part of the distribution and thus skews the aggregation in one form or another.

Chapter 4. Charting and alerting

Styles

REMEMBER TO DISABLE INTERPOLATION WHEN REQUIRED: This can give an invalid visual reprsentation and be misinterpreted.

WHEN PLOTTING ERRORS/SUCESS/OTHER USE COLOR AND LINE STYLE: This will improve accessibility.

GAUGES: A time series representation of a gauge presents more information about as compactly as an instantaneous gauge.

Gauges have a tendency to be spiky. Thread pools can appear to be temporarily near exhaustion and then recover. Queues get full and then empty. Memory utilization in Java is especially tricky to alert on since short-term allocations can quickly appear to fill up a significant portion of allocated space only for garbage collection to sweep away much of the consumption.

One of the most effective methods to limit alert chattiness is to use a rolling count function. In this way we can define an alert that only fires if a threshold is exceeded more than three times in the last five intervals, or some other combination of frequency and number of lookback intervals.

COUNTERS

Counters are often tested against a maximum (or less frequently, a minimum) threshold. The need to test against a threshold reinforces the idea that counters should be observed as rates rather than a cumulative statistic, regardless of how the statistic is stored in the monitoring system.

I.e plot an HTTP endpoints request throughput as a rate (yellow solid line) and also the cumulative count (green dots) of all requests to this endpoint since the application process started.

LATENCY: On dashboards, latency is the most important to view, because it is most directly tied to user experience. After all, users care mostly about the performance of their individual requests.

One recognized approach to limiting the effect of the top 1% is a client-side load-balancing strategy called hedge requests.

Setting an alert on max latency is key. But once an engineer has been alerted to a problem, the dashboard that they use to start understanding the problem doesnt necessarily need to have this indicator on it. It would be far more useful to see the distribution of latencies as a heatmap.

Are most requests failing close to the max value, or are there just one or a few stray outliers? The answer to this question likely affects how quickly an alerted engineer escalates the issue and brings others in to help.

Indicators

Now that we have a sense of how to visually present SLIs on charts, we will turn our focus to the indicators you can add. They are presented in approximately the order of importance. So if you are following the incrementalist approach to adding charts and alerts, implement these in sequence.

Errors

When timing a block of code its useful to differentiate between successful and unsuccessful operations for two reasons.

First, we can directly use the ratio of unsuccessful to total timings as a measure of the frequency of errors occurring in the system.

Status tags should be added to timing instrumentation in most cases on two levels.

- status: error code, exception name
- outcome: a more course-grained error category that separates success, user-caused error, service-caused error

For HTTP request metrics, for example, Spring Boot automatically tags http.server.requests with a status tag indicating the HTTP status code and an outcome tag that is one of SUCCESS, CLIENT_ERROR, or SERVER_ERROR.

If every 10th request fails, and 100 requests/second are coming through the system, then the error rate is 10 failures/second. If 1,000 requests/second are coming through the system, the error rate climbs to 100 failures/second! In both cases, the error ratio relative to throughput is 10%. **This error ratio normalizes the rate and is easy to set a fixed threshold for.**

ERROR RATE IS BETTER THAN ERROR RATIO FOR LOW-THROUGHPUT SERVICES

In general, prefer error ratio to error rate, unless the endpoint has a very low throughput. In this case, even a small difference in errors can lead to wild shifts in the error ratio. It is more appropriate in these situations to pick a fixed error rate threshold.

Latency

Alert on maximum latency (in this case meaning maximum observed for each interval), and use high-percentile approximations like the 99th percentile for comparative analysis.

It is also useful to monitor latency from the perspective of the caller (client). In this case, by client I generally mean service-to-service callers and not human consumers to your API gateway or first service interaction. A services view of its own latency doesnt include the effects of network delays or thread pool contention.

Garbage Collection Pause Times

Garbage collection (GC) pauses often delay the delivery of a response to a user request, and they can be a bellwether of an impending out of memory application failure. There are a few ways we can look at this indicator.

- max pause time: Set a fixed alert threshold on the maximum GC pause time you find acceptable
- proportion of time speint in gc: Since jvm.gc.pause is a timer, we can look at its sum independently. Specifically, we can add the increases in this sum over an interval of time and divide it by the interval to determine what proportion of the time the CPU is engaged in doing garbage collection.
- presence of any humongous allocation: it indicates that somewhere you are allocation an object >50% of the total size of the Eden space

  We mentioned that saturation metrics are usually preferable to utilization metrics when you have a choice between the two. This is certainly true of memory consumption. The views of time spent in garbage collection as a measure of memory resource problems are easier to get right.

Heap utilization

- eden space (young generation), old generation, rolloing count occurrences, low pool memory, low total memory

CPU utilization

File descriptors

Suspicious traffic

A rapid succession of HTTP 403 Forbidden (and similar) or HTTP 404 Not Found may indicate an intrusion attempt.

Unlike plotting errors, monitor total occurrences of a suspicious status code as a rate and not a ratio relative to total throughput. Its probably safe to say that 10,000 HTTP 403s per second is equally suspicious if the system normally processes 15,000 requests per second or 15 million requests per second, so dont let overall throughput hide the anomaly.

Batch runs or long running tasks

To monitor long-running tasks, it is better to look at the running time of in-flight or active tasks.

Long task timers ship several distribution statistics: active task count, the maximum in-flight request duration, the sum of all in-flight request durations, and optionally percentile and histogram information about in-flight requests.

Building alerts using forecasting methods

Chapter 5. Safe, multicloud continous delivery

Resource types

- Instance: a running copy of some microservice. in k8s an instance is a pod.
- Server group: a collection of instances, managed together.
- Cluster: A set of server groups that may span multiple regions.
- Load balancer: A component that allocates requests to instances in one or more server groups.

Deployment strategies

- delete + none
- highlander
- green/blue

Canary

When deploying a canary analysis is recommended to have 3 server groups: prod, base and canary

Think of the L-USE acronym when considering canary metrics that are useful to start with. In fact, many of the same SLIs that you should alert on for most microservices are also good canary metrics, with a twist.

Chapter 6. Source code observability

What we need to set up your software delivery life cycle in such a way that you can map deployed assets back to the source code that was included inside of them.

Chapter 7. Traffic management

Cloud native applications expect failure and low availability from the other services and resources they interact with. In this chapter, we introduce important mitigation strategies involving load balancing (platform, gateway, and client-side) and call resilience patterns (retrying, rate limiters, bulkheads, and circuit breakers) that work together to ensure your microservices continue to perform.

Concurrency systems

There is a natural bound to concurrency in any system, usually driven by a resource like CPU or memory or the performance of a downstream service when requests are satisfied in a blocking manner.

Services fail when during prolonged periods of time the request rate exceeds the response rate. As the queue grows, so will the latency (since requests dont even begin getting serviced until they are removed from the queue). Eventually queued requests will start timing out.

Platform load balancing

These load balancers serve to distribute traffic across the instances in a cluster one way or another (often round-robin), but also have a wide range of other responsibilities. For example, AWS Elastic Load Balancers also serve the interests of TLS termination, content-based routing, sticky sessions, etc.

Gateway load balancing

From the perspective of reliability, the goal of load balancing is to direct traffic away from servers that have high error rates. The goal should not be to optimize for the fastest response time. Avoiding instances with high error rates still allows traffic to be distributed to instances that are not optimally performant, but available enough.

User-facing traffic comes in through a platform load balancer, which distributes the traffic in a round-robin fashion to a cluster of gateway instances.

WHY SERVICE DISCOVERY INSTEAD OF A CLOUD LOAD BALANCER? Perhaps if VPC was around when Netflix first migrated to AWS, Eureka would never have come about. Nevertheless, its use has extended beyond just load balancing to available instances in a cluster. I.e it showed how it is also used in blue/green deployments of event-driven microservices to take the instances in a disabled cluster out of service. Not every enterprise will take advantage of this kind of tooling, and if not, private cloud load balancers are probably simpler to manage.

Hedge requests

One well-tested strategy to mitigate the effects of the top 1% latency when calling a downstream service or resource is to simply ship multiple requests downstream and accept whichever response comes back first, discarding the others.

Call resilliency patterns

Retries

- wether retries are appropiate
- number of retry attempts, durations (backoffs)
- which responses

Rate limiters

Bulkheads

The bulkhead pattern isolates downstream services from one another, specifying different concurrency limits for each downstream service. In this way, only requests that require a service call to B become unresponsive, and the rest of the A service continues to be responsive.

- concurrency limit by the bulkhead
- timeout for a blocked thread
- service specific configurations

Circuit breaker

In the closed and half-open states, executions are allowed. In the open state, a fallback defined by the application is executed instead.

Summary

Failure and degradation of performance should be expected and planned for in any production microservice architecture. In this chapter, we introduced a number of strategies for dealing with these conditions, from load balancing to call resilience patterns.

- Rate limiter, limits rate per interval. This does not limit instantaneous concurrency (spike) unless that spike exceeds the limit for the interval
- Bulkhead, limits instantaneous concurrency level at the time a request is attempted. Does not limit the number of requests to the downstream except inderictly since the downstream service responds in a nonzero amout of time
- CB, responds to error, this does not limit the instantaneos or per interval rate except inderectly when the downstream is saturated