Are We All on the Same Page?
Let's Fix That

LUIS MINEIRO

Luis's broad background in software engineering includes experience in DevOps, system administration, networking, and more. Luis has been with Zalando since 2013, working with approximately two hundred engineering teams increasing the observability and reliability of the Zalando e-commerce platform, currently heading Site Reliability Engineering.
luis@zalando.de

Industry has defined as good practice to have as few alerts as possible, by alerting on symptoms that are associated with end-user pain rather than trying to catch every possible way that pain could be caused. Organizations with complex distributed systems that span dozens of teams can have a hard time following such practice without burning out the teams owning the client-facing services. A typical solution is to have alerts on all the layers of their distributed systems. This approach almost always leads to an excessive number of alerts and results in alert fatigue. I propose a solution to this problem by paging only the team closest to the problem.

The Age of the Monolith

Many organizations became successful running a monolith. In the age of the monolith we had single, large boxes that did everything—they handled every request. There were some minor evolutions of this basic model, namely for redundancy and availability, but that's not so relevant. What's important—monoliths were simple. They were easy to reason about and easy to monitor.

This was the time of the Ops and Dev silos. The Ops people monitored the hardware and checked whether the monolith process was up. The Devs monitored the requests and the responses.

This approach had its own share of problems, particularly as businesses grew and the approach didn't allow the business to scale further. Microservices have become the solution for those problems.

Modern Microservices

The diagram in Figure 1 is a possible representation of a typical business operation in e-commerce websites—placing an order.

Founded in 2008 in Berlin, Zalando is Europe's leading online fashion platform and connects customers, brands, and partners. It has more than 200 software delivery teams. Organizations such as Zalando can have north of 60 microservices involved in such a business operation, including some so-called legacy ones. Other organizations can actually be simpler or more complex, so mileage may vary. The relevant question is, how do we monitor and alert on this?

The industry came up with new job roles, some call them DevOps, some call them SRE, but the name is not important. We could call them Cupcake Fairies; it doesn't matter. What matters is how we monitor didn't change much, and the new roles didn't change anything. We still check whether boxes are alive, processes are responsive, and individual microservices succeed. Most times, we also check whether responses are fast enough.

When it comes to monitoring, I'd say that we're just monitoring distributed monoliths.

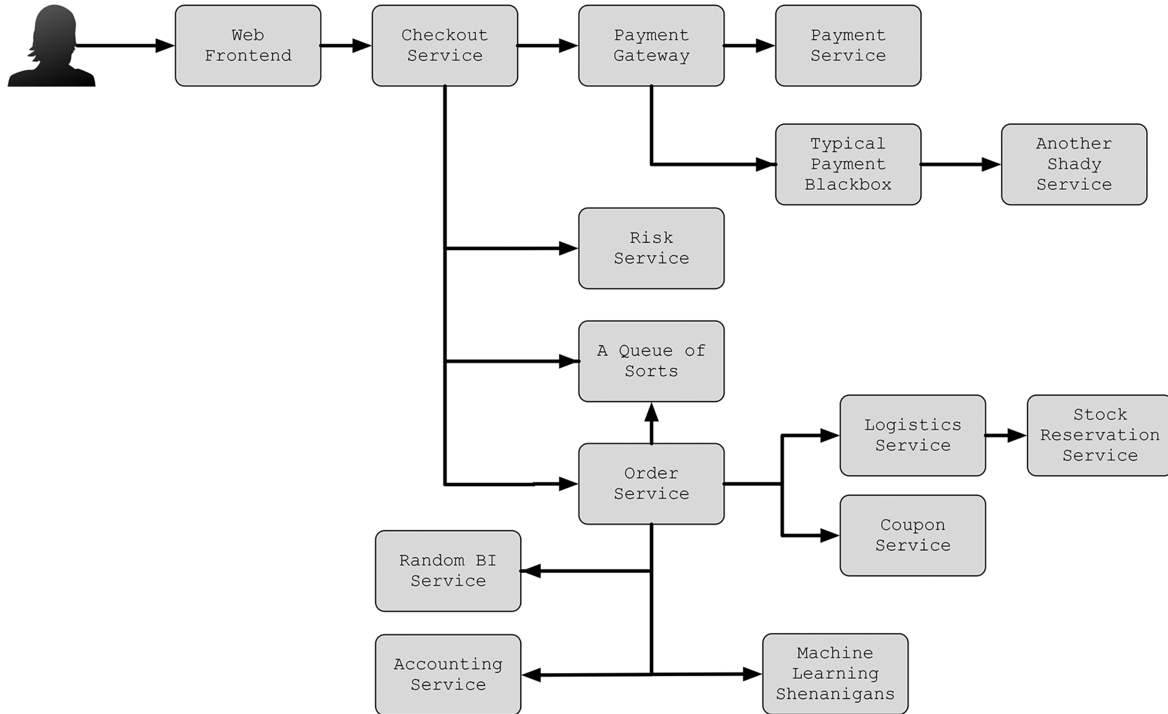

Figure 1: An example set of the microservices involved in fulfilling a customer request. Arrows show the flow through the services and also indicate dependencies.

Problem Statement

What about alerting? What happens when the Accounting Service from the example diagram in Figure 1 has an outage? What almost always happens is that dozens (or hundreds) of alerts come up, making it look like all services failed.

I call this the Christmas Tree effect. Lots of blinking lights, almost the same as Christmas except the happiness level is different, and definitely no one is getting any presents!

This approach almost always leads to an excessive number of alerts and results in alert fatigue. Only one of those teams can actually do something about it—the one operating the Accounting Service.

The alternative to this is to alert on symptoms instead. That's something the industry already accepted—in theory. How would it look if we were alerting on symptoms?

We can measure signals like latency and errors where the Web front end calls the Checkout Service. This is a good place to measure such service level indicators, where the signal-to-noise ratio is optimal and as close as possible to the customer pain.

What happens when alerting on the symptom if the Accounting Service has an outage?

The alert created based on the symptom will be triggered. This looks better. Is there anything wrong with the approach? What happens with this approach if the Payment Service has an outage? The same alert will be up. The team owning the client-facing service, and typically the owner of the alert rule, gets the paging alert *for each and every possible failure in the distributed system!*

This sort of pivoting is a serious problem that hasn't been addressed properly as far as I know. Alerting on all the layers of the distributed system is not healthy, and the alternative, alerting on symptoms, can result in bombing the team owning the client-facing service.

In a Twitter thread [1] early this year, Jacob Scott (@jhscott) brought up the question—"In a 'microservices organization' where teams own specific components/services of a distributed production system, who is responsible for triage/debugging/routing of issues that don't present with a clear owner? And how do they not hate their lives?" Charity Majors' (@mipsytipsy) reply, that I totally agree with, was "alright, this is a damn good question. and tbh i am surprised it doesn't come up more often, because it gets right to the beating heart of what makes any microservices architecture good or bad." This captures the essence of the problem. The so-called "microservices organizations" struggle to figure this out.

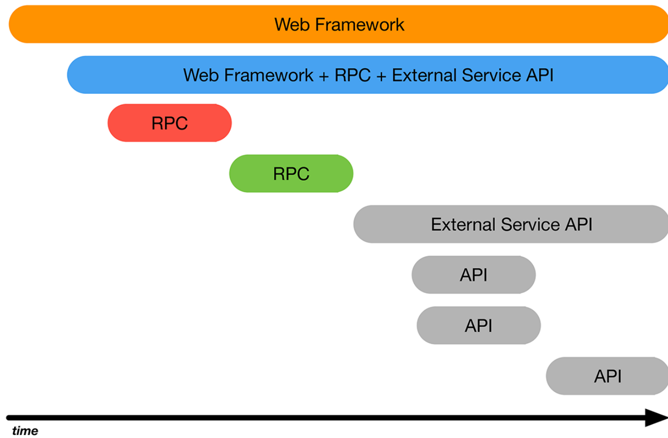

Figure 2: An example trace containing many spans from different microservices

Adaptive Paging

At Zalando we started addressing this problem with a custom alert handler that leverages the causality from tracing and OpenTracing's semantic conventions to page the team closest to the problem. We called it Adaptive Paging.

Five-Minute Introduction to OpenTracing

OpenTracing is a set of vendor-neutral APIs and a code instrumentation standard for distributed tracing. A trace tells the story of a transaction or workflow as it propagates through a distributed system. It's basically a directed acyclic graph (DAG), with a clear start and a clear end—no loops. A trace is made up of spans representing contiguous segments of work in that trace.

You can find a lot more details by checking distributed tracing's origins, namely the Dapper paper [2].

It's worth mentioning that OpenTracing has merged with another instrumentation standard—OpenCensus—resulting in OpenTelemetry. OpenTelemetry will offer backwards compatibility with existing OpenTracing integrations. The concepts and strategy for Adaptive Paging are still valid.

Spans

A Span is a named operation which records the duration, usually a remote procedure call, with optional Tags and Logs. This is probably the most important element of OpenTracing. A trace is a collection of spans.

Operations can trigger other operations and depend on their outcome. For example, *place_order* triggers and depends on all the other operations, including *update_account* in the *accounting-service*. This causality is important.

Tags

The other most relevant element from OpenTracing is Tags. A tag is a "mostly" arbitrary key-value pair, where the value can be a string, number, or bool. Every operation can have its own set of tags.

We can consider Tags as metadata that enrich the operation abstraction (the span) with additional context.

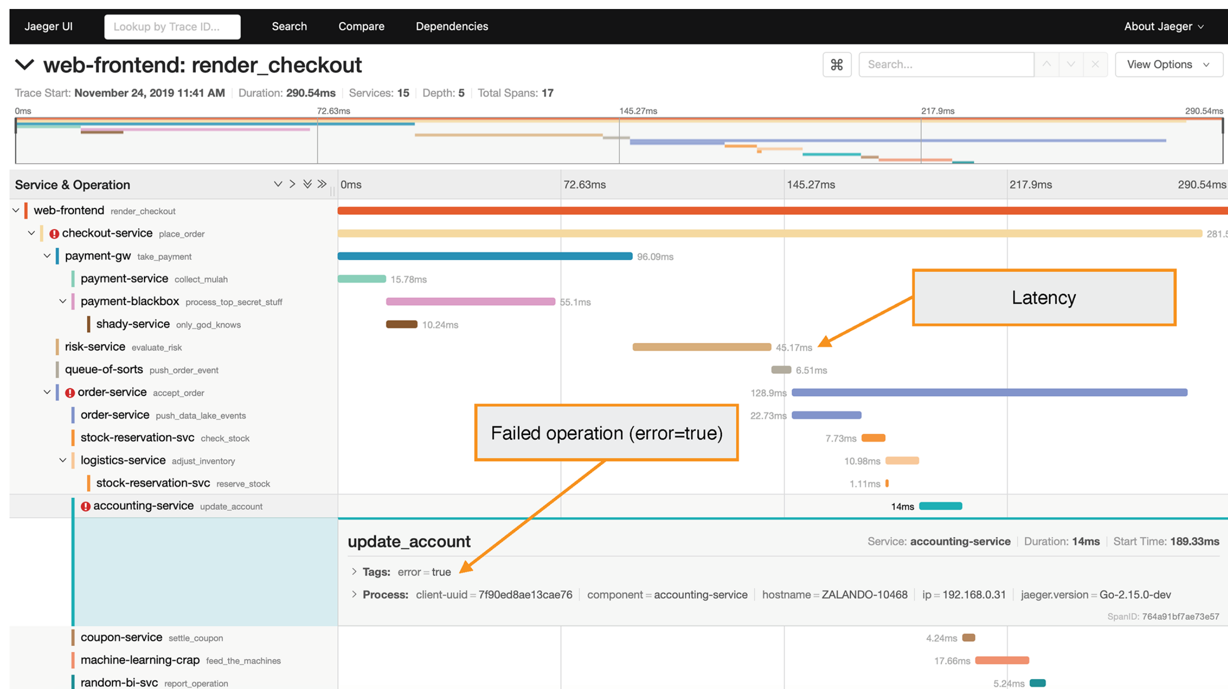

Figure 3: Screen capture of a trace in the open source tracing tool Jaeger [3]

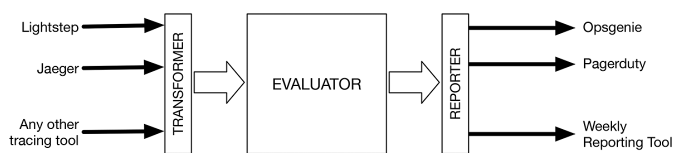

Figure 4: Adaptive Paging components and data flow

Semantic Conventions

OpenTracing's semantic conventions establish certain tag names and their meanings. The existing conventions are strong enough to set certain expectations and enable tools to apply different behaviors when analyzing the tracing data.

OpenTracing Monitoring Signals

OpenTracing can provide, implicitly, measurements for latency and throughput (number of operations over a certain time period). Through the semantic conventions it's also possible to measure errors, by checking the spans with the *error* tag set to the Boolean value *true*.

Latency, traffic, saturation, and errors are the Four Golden Signals [4]. If you can only measure four metrics of your user-facing system, focus on these four. They are great for alerting.

In this article we'll focus on one concrete signal—errors.

Alert Handler

Let's assume that an alert was configured for the *place_order* operation which has a service level objective (SLO) of 99.9 success rate. A typical way to measure this would be to query the tracing back end for spans that match a certain criteria. The keys *operation* and *component* are implicit on most tracing systems and represent the named span and the microservice itself, respectively. An expression such as *component: checkout_service && operation: place_order* represents the symptom and is where we want to measure customer pain. Different tools, open source and commercial, will usually provide different means to configure the alert itself. That's not in the scope of this article.

Adaptive Paging is an alert handler, and its architecture is broken down into three main components. The *transformer* is the actual alert handler, typically a webhook, and it's vendor specific. It's possible to have multiple alert handlers. The webhook receives alerts and converts them into symptoms. Then the symptom is passed to an evaluator, which implements the actual root-cause identification algorithm. The evaluator tries to determine the most probable root cause and generates a report. After the report is created it is made available to any reporter(s) which can deliver the page via different vendor-specific implementations or store debugging data to troubleshoot the alert handler itself.

Transformer

The transformer receives or collects vendor-specific exemplars and converts them into a vendor-agnostic data model that we called Symptoms. Exemplars are traces that should be representative of the symptoms that led to the alert being triggered. Some vendors can include exemplars as part of the alert payload. If they're not part of the payload, the transformer can query the tracing back end for exemplars that match the same criteria of the alert rule during the time of the incident.

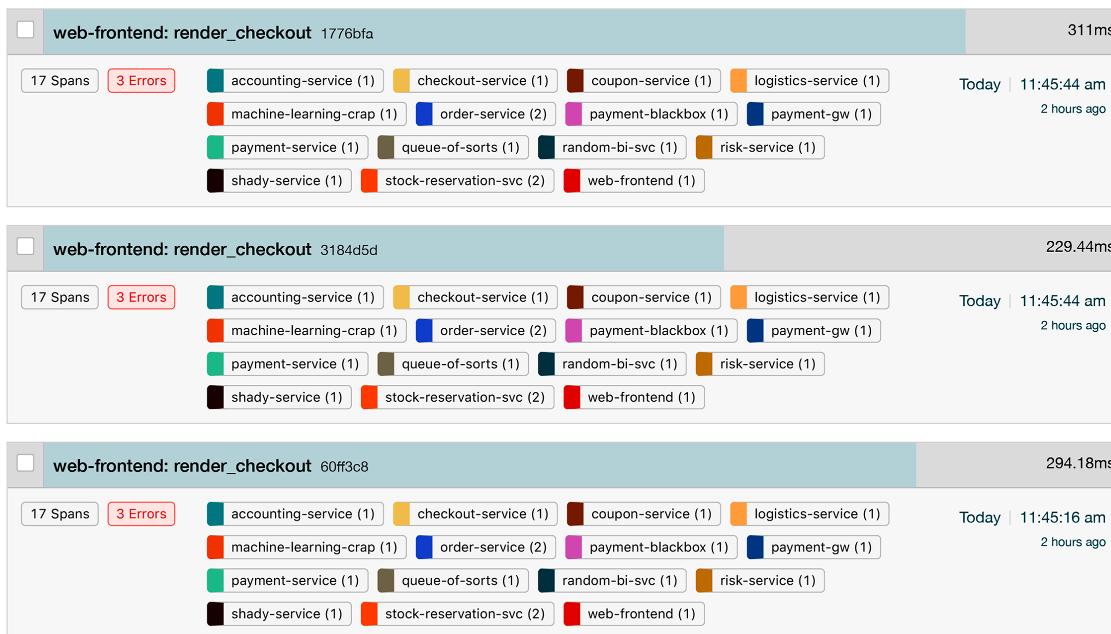

Figure 5: Collection of traces (exemplars) that contain the failed operations (*error=true*)

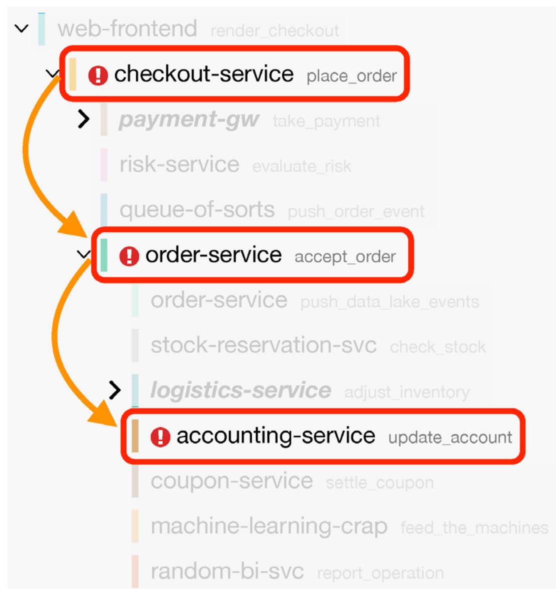

Figure 6: Probable root cause algorithm inspecting failed operations

Evaluator

The evaluation algorithm can have many different implementations. There can be different implementations for different signals—latency or errors, for example, or for any other known criteria for which a certain root-cause-identification algorithm performs better.

Example Errors Algorithm

The following example is one possible implementation to identify the probable root cause for errors. All exemplars (traces) are analyzed. Starting at the span that was defined as the signal source, each trace is inspected in a recursive way. For every child span, its tags and respective values are checked to decide which path to take.

In the example from Figure 6, none of the operations *take-payment*, *evaluate_risk,* or *push_order_event* were tagged as failed (**error=true**).

The *accept_order* operation in the order-service was tagged. The algorithm follows the path where **error=true**.

The same process is repeated. None of the operations of the *order-service, stock-reservation-svc, logistics-svc,* or the others which were triggered by *accept_order* were tagged with errors.

Only the *update_account* operation in the *accounting-service* was tagged as failed.

Without any child spans to continue the traversal, the *update_account* operation in the *accounting-service* is selected as the most probable cause of the errors.

After all exemplars are analyzed, a Report is generated.

Reporting

The Report generated by the evaluation algorithm contains information about the operation and microservice that is considered the most probable root cause. For reporters that page on-call engineers, the implementation needs to map the operation and/or service to the respective team or on-call escalation.

Putting It All Together

Going back to the original example, what happens if the Accounting Service has an outage and we're using Adaptive Paging? As you can guess, the team that operates the Accounting Service will get the single page triggered.

A similar situation would happen if any of the services involved in the "Place Order" operation breached its SLO, but the team that operates the probable root cause is the only one getting the paging alert—the one that will be able to actually do something about it—that is, no more page bombing.

Challenges

As mentioned before, the detection algorithm can adopt many different strategies. Zalando's current implementation uses a couple of heuristics that are easy to reason about.

Some of the things we had to work around when creating Adaptive Paging were:

- Multiple child spans tagged as errors: follow each path, attribute the probable cause a score. Analyze more exemplars and adjust the scores. Worst case scenario, page multiple probable causes. Paging two teams is still better than paging everyone.
- Missing instrumentation or circuit breaker open: either of these situations results in a premature evaluation of the probable root cause. We leveraged the semantic conventions to allow the caller to identify the callee, suggesting to the evaluator algorithm who to page, using the *peer.service=foo* and *span.kind=client* tag to suggest which service would be the target. This has the side effect of being a good incentive for teams to instrument their services.
- Mapping services to escalation: the service identified as probable root cause may not have a mapping to an on-call escalation. The evaluator keeps a stack of the probable causes and uses the one that is available and hopefully closest.

Finding probable causes due to latency is a challenge of its own. The strategy that we considered requires us to query the baselines for each operation and service combination, using that information to select which combination has a bigger variation at the time of the incident. This strategy can be a bit expensive, increasing the time to dispatch the paging alert.

Next Steps

Adaptive Paging was created with a multi-vendor reality in mind. Observability still has a ways to go, and some vendors are pushing the boundaries as we speak. Distributed tracing is still not a commodity, just like unit testing wasn't when it was initially introduced. No one would challenge the benefits of unit testing, and I believe no one will challenge the benefits of proper observability of distributed systems.

We've also started looking at some excellent work from LinkedIn—MonitorRank [5] from 2013, which fits nicely into Adaptive Paging; it's something we're considering as a possible improvement to the evaluator.

With Adaptive Paging we hope to contribute to improve the alerting situation, in particular paging alerts that burn out humans.

References

[1] Twitter thread on page bombing: https://twitter.com/mipsytipsy/status/1120911207903268864.

[2] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, C. Shan-bhag, "Dapper, a Large-Scale Distributed Systems Tracing Infrastructure": https://ai.google/research/pubs/pub36356.

[3] Jaeger: https://www.jaegertracing.io/.

[4] Four Golden Signals: https://landing.google.com/sre/sre-book/chapters/monitoring-distributed-systems/.

[5] M. Kim, R. Sumbaly, S. Shah, "Root Cause Detection in a Service-Oriented Architecture," SIGMETRICS '13: http://infolab.stanford.edu/~mykim/pub/SIGMETRICS13-Monitoring.pdf.