Trace by RisingStack

Monitoring microservices architectures the easier way

Gergely Nemeth

@nthgergo

$ whoami

Work - RisingStack

Twitter - @nthgergo

GitHub - gergelyke

Stuff I write - https://blog.risingstack.com

Agenda

- monitoring microservices
- how Trace works
- transaction tracking
- getting started

RisingStack

A microservice monitoring and debugging tool that empowers you to get all the metrics you need when operating microservices.

Get Trace

monitoring microservices

Monitoring microservices

The microservice pattern is not the silver bullet for designing systems - it helps a lot, but also comes with new challenges.

RisingStack

Monitoring microservices

- ***Debugging and monitoring microservices can be really challenging:***
  - ***no stack trace, hard to debug***
  - ***easy to lose track of services when dealing with a lot***
  - ***bottleneck detection***

RisingStack

Monitoring microservices

- *Trace solves these problems by adding the ability to*
  - *do distributed stack traces,*
  - *topology view for your services,*
  - *and alerting for overwhelmed services,*
  - *third-party service monitoring (coming soon),*
  - *trace heterogeneous infrastructures with languages like Java, PHP or Ruby (coming soon).*

RisingStack

Monitoring microservices

- *Trace is mostly based on the Google Dapper white paper - so we implemented the ServerReceive, ServerSend, ClientSend, ClientReceive events for monitoring the lifetime of a request.*

- *In the example above, we want to catch the very first incoming request: SR (A): Server Receive. The http.Server will emit a request event, with http.IncomingMessage and a http.ServerResponse with the signature of* `function (request, response) { }`

how Trace works

How Trace works

With wrapping the http core module's request function and the Server.prototype object, we can sniff all the information we need.

How Trace works

Trace is mostly based on the Google Dapper white paper -

so we implemented the *ServerReceive, ServerSend,*

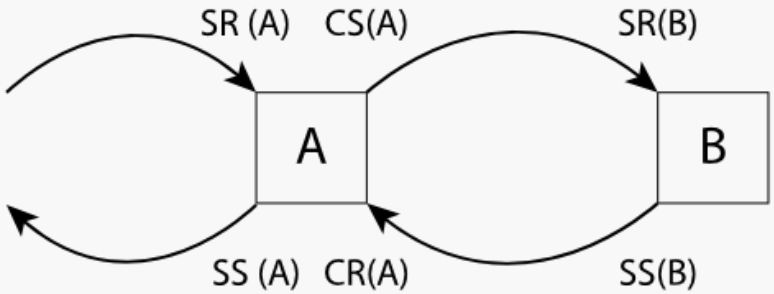
ClientSend, ClientReceive ***events for monitoring the lifetime***

of a request.

RisingStack

How Trace works

- *Trace is mostly based on the Google Dapper white paper - so we implemented the ServerReceive, ServerSend, ClientSend, ClientReceive events for monitoring the lifetime of a request.*



- *In the example above, we want to catch the very first incoming request: SR (A): Server Receive. The http.Server will emit a request event, with http.IncomingMessage and a http.ServerResponse with the signature of* `function (request, response) { }`

RisingStack

How Trace works

In the wrapper, we can record every information we want, like timing, the source, the requested path, or even the whole HTTP header for further investigation.

RisingStack

transaction tracking

Transaction tracking

We do it by setting a** request-id **header on the outgoing requests.

Transaction tracking

- ***If our service has to call another service before it can send the response to its caller, we have to track this kind of request-response pairs, spans as well. A span always comes from*** *http.request* ***by calling an endpoint. By wrapping the*** *http.request* ***function, we can do the same as in the*** *http.Server.prototype* ***with one minor difference: here we want to pair the corresponding request and response, and assign a*** *span-id* ***to it.***

Transaction tracking

- ***However, the** request-id **will just pass through the span. In order to store the generated** request-id**, we use** Continuation-Local Storage**: after a request arrived and we generated the** request-id**, we store it in CLS, so when we try to call another service we can just get it back.***

getting started

Getting started

- *Trace is completely open source - feel free to play with it, open issues, send pull requests. We will soon start to implement Trace to other languages as well - if you'd like to participate, say hi, and let's get working on them!*

RisingStack

Getting started

- *Trace is also available as a hosted service. If you don't want run your own infrastructure we provide microservice monitoring as a service as well. This is what it looks like:*

Marketing Site service

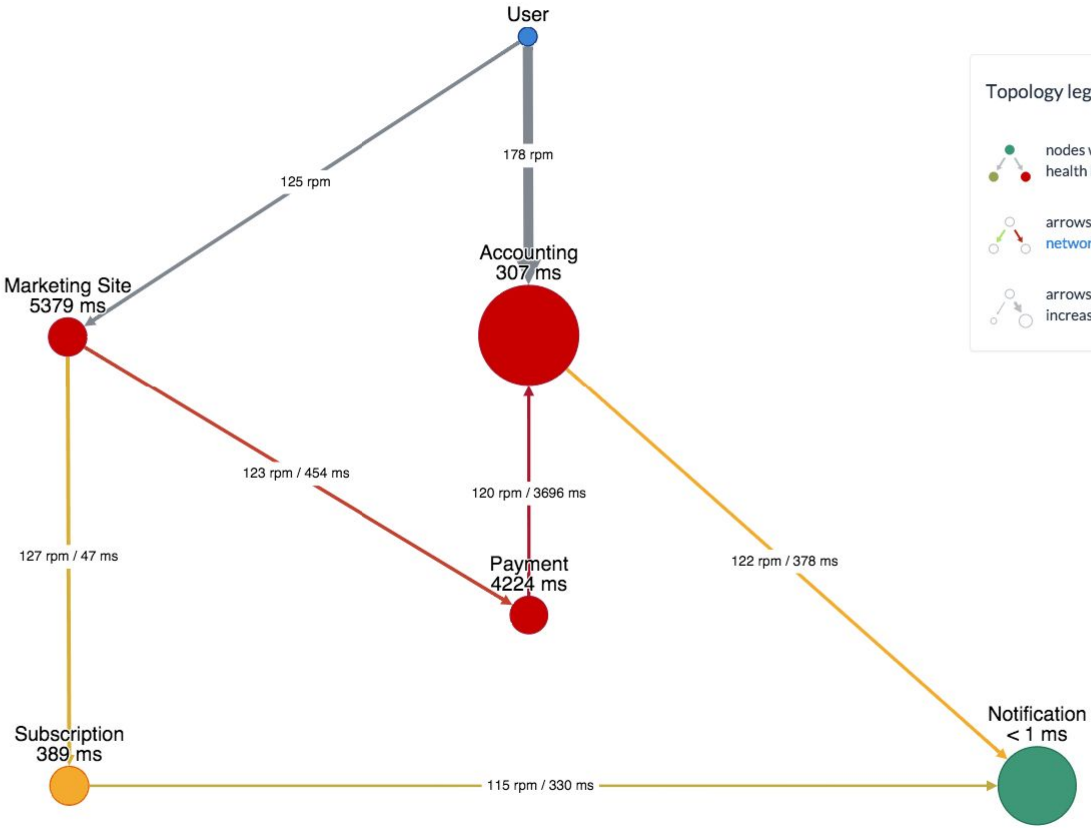
Requests received (avg. in requests/min)
Total: 125 rpm
Response time (in milliseconds)
current: 5 419 ms - average: 4 158 ms

Accounting service

Requests received (avg. in requests/min)
Total: 300 rpm
Response time (in milliseconds)
current: 313 ms - average: 250 ms

Payment → Accounting

Requests handled (avg. in requests/min)
Total: 120 rpm
Network delay (in milliseconds)
current: 3 696 ms - average: 2 780 ms



Topology legend

nodes with green to red color
health level of metric like response time

arrows color from green to red
network latency from better to worse

arrows width and nodes size
increase with more requests count

Thanks!

Gergely Nemeth

@nthgergo