DiTrace

Release 1.0

January 13, 2017

Contents

CHAPTER 1

Contents

1.1 Overview

DiTrace is a distributed tracing system.

Like the others distributed tracing system as Zipkin or Dapper, DiTrace is an instrument to find the problem zones in distributed systems.

But with different and more simple architecture.

1.1.1 How it works

Every time, when one of distributed system's microservice made a call to another microservice, the data object, called "span" should be created.

Span has an arbitrary number of annotations such as request and response parameters (timestamps, url, response code, etc).

Spans are belong to one trace in hierarchic order with one root span.

Microservices are responsible for:

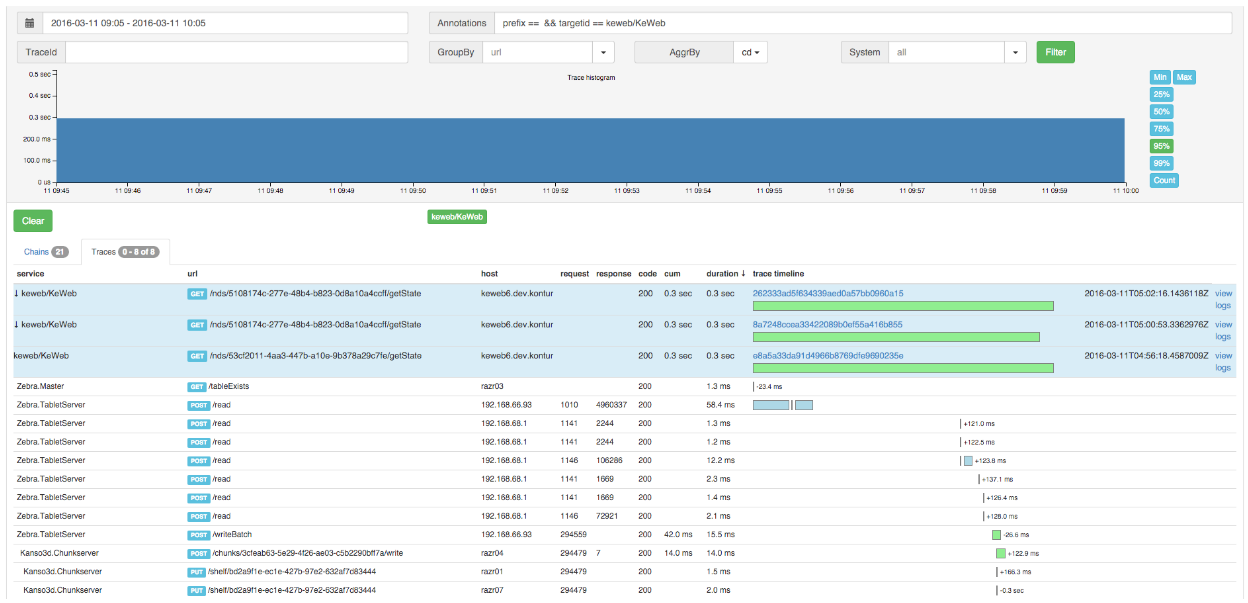
1. Creating traces, spans and sending traceid and spanid within requests between each other.
2. Collecting spans and sending it to the DiTrace gate.

Look DiTrace gate API for more details.

DiTrace gate are responsible for:

1. Collecting and grouping spans from multiple microservices
2. Saving traces to elasticsearch

Elasticsearch has a various stats aggregations for data analysis. UI is an visualization tool for this aggregations and data.



2016-03-11 09:05 - 2016-03-11 10:05

Annotations: prefix == && targetid == keweb/KeWeb

TraceId:

GroupBy: url

AggrBy: cd

System all

service	url	host	request	response	code	cum	duration ↓	trace timeline		
↓ keweb/KeWeb	GET /nds/5108174c-277e-48b4-b823-0d8a10a4ccff/getState	keweb6.dev.kontur			200	0.3 sec	0.3 sec	262333ad5f634339aed0a57bb0960a15	2016-03-11T05:02:16.1436118Z	view logs
↓ keweb/KeWeb	GET /nds/5108174c-277e-48b4-b823-0d8a10a4ccff/getState	keweb6.dev.kontur			200	0.3 sec	0.3 sec	8a7248ccea33422089b0ef55a416b855	2016-03-11T05:00:53.3362976Z	view logs
keweb/KeWeb	GET /nds/53cf2011-4aa3-447b-a10e-9b378a29c7fe/getState	keweb6.dev.kontur			200	0.3 sec	0.3 sec	e8a5a33da91d4966b8769dfe9690235e	2016-03-11T04:56:18.4587009Z	view logs
Zebra.Master	GET /tableExists	razr03			200		1.3 ms	-23.4 ms		
Zebra.TabletServer	POST /read	192.168.66.93	1010	4960337	200		58.4 ms			
Zebra.TabletServer	POST /read	192.168.68.1	1141	2244	200		1.3 ms	+121.0 ms		
Zebra.TabletServer	POST /read	192.168.68.1	1141	2244	200		1.2 ms	+122.5 ms		
Zebra.TabletServer	POST /read	192.168.68.1	1146	106286	200		12.2 ms	+123.8 ms		
Zebra.TabletServer	POST /read	192.168.68.1	1141	1669	200		2.3 ms	+137.1 ms		
Zebra.TabletServer	POST /read	192.168.68.1	1141	1669	200		1.4 ms	+126.4 ms		
Zebra.TabletServer	POST /read	192.168.68.1	1146	72921	200		2.1 ms	+128.0 ms		
Zebra.TabletServer	POST /writeBatch	192.168.66.93	294559		200	42.0 ms	15.5 ms	-26.6 ms		
Kanso3d.Chunkserver	POST /chunks/3cfeab63-5e29-4f26-ae03-c5b2290bff7a/write	razr04	294479	7	200	14.0 ms	14.0 ms	+122.9 ms		
Kanso3d.Chunkserver	PUT /shelf/bd2a9f1e-ec1e-427b-97e2-632af7d83444	razr01	294479		200		1.5 ms	+166.3 ms		
Kanso3d.Chunkserver	PUT /shelf/bd2a9f1e-ec1e-427b-97e2-632af7d83444	razr07	294479		200		2.0 ms	-0.3 sec		

1.2 Installation

1.2.1 Manual Installation

There are following components you need to install before running DiTrace gate and UI:

1. golang version 1.5 or higher 3. elasticsearch version 2.2 4. web server e.g. nginx

Install DiTrace gate

```
export GOPATH=<your gopath>
go get github.com/ditrace/ditrace
```

Download Web UI Application

https://github.com/ditrace/web/releases/latest

Configure

1. Place configuration file to the default location, `/etc/ditrace/config.yml`

You can dive into Configuration syntax on a separate page.

2. Place nginx configuration file to `/etc/nginx/conf.d/ditrace.conf`

```
# elasticsearch cluster for traces
upstream elastic {
    server vm-ditrace1:9200;
    server vm-ditrace2:9200;
    server vm-ditrace3:9200;
}
```

```
server {
    listen       0.0.0.0:80;
    server_name  vm-ditrace1;

    location / {
        root /var/local/www/ditrace/web/static/;
        index  index.html;
    }

    location /elasticsearch/ {
        rewrite      /elasticsearch/(.*)  /$1  break;
        proxy_pass http://elastic;
    }
}
```

3. Place UI config.json file to `/var/local/www/ditrace/config.json`

Run

1. Run nginx
2. Run elasticsearch
3. Setup indices template

```
curl -XPUT http://elasticsearch:9200/_template/traces --data-binary @template.json
```

`template.json`

4. Run ditrace gate

```
$GOPATH/bin/ditrace --config=/etc/ditrace/config.yml
```

1.2.2 Configuration

By default, DiTrace gate will look for `./config.yml`, but you can change this by command-line parameter

```
log_dir: stdout
log_level: debug

# Send statistics to graphite
stats:
  enable: true
  graphite_host: "vm-graphite"
  graphite_port: 2003
  graphite_prefix: DevOps

http:
  enable: true
  address: ":8080"
  # List replicas of elasticsearch cluster
  elasticsearch:
    - "http://vm-elastic:9200"
  # Sampling can be used to limit part of incoming traces.
  # Value of N means that only one of N traces will be written to elastic.
  sampling: 1
  replicas:
    - "http://localhost:8080"
  # Number of seconds to wait if trace is completed before write it to elastic,
  min_ttl: 10
  # Number of seconds to wait if trace is not completed before it will be cleaned out.
  max_ttl: 120

profiling:
  # https://golang.org/pkg/net/http/pprof/
  enable: true
```

1.3 DiTrace gate API

Gate accept only the following http request.

POST /spans?system=(*string*)

System parameters is used if no system annotation in span json

Example request:

```
POST /spans?system=mysystem HTTP/1.1
Content-Type: application/x-ldjson

{
  "traceId":"c38efe4edb2d4a008af2805ee4e061c1",
  "spanId":"8256",
  "timeline":{
      "sr":"2015-04-24T09:53:49.5595869Z",
      "ss":"2015-04-24T09:53:50.5595869Z"
  },
  "annotations":{
      "url":"/url?arg1=arg1&arg2=arg2",
      "host":"hostname",
      "rqbl":"42",
      "rsbl":"4200",
      "targetId":"service-0"
  }
 }\r\n
 {
  "traceId":"c38efe4edb2d4a008af2805ee4e061c1",
  "parentSpanId":"8256",
  "spanId":"904a",
  "timeline":{
      "sr":"2015-04-24T09:53:49.5595869Z",
      "ss":"2015-04-24T09:53:50.5595869Z"
  },
  "annotations":{
      "url":"/url",
      "host":"hostname",
      "rqbl":"42",
      "rsbl":"4200",
      "targetId":"service-1"
  }
 }
```

Example response:

```
HTTP/1.1 200 OK
```

Query Parameters

- **`system`** – spans source system name

Status Codes

- 200 OK – no error
- 400 Bad Request – server can't parse request content or any of required field is missing (system, traceid, spanid)

Important: For better visibility example's jsons are formatted with additional new lines. Only the new lines that separate jsons should be in real requests.

1.3.1 Span format

Field, preferrable format, description

- (required) **TraceId**, UUID, unique identifier to group multiple spans into one trace
- (required) **SpanId**, part of UUID, unique identifier of span within one trace
- (required) **Annotations**, object of arbitrary annotations
- (required) **Timeline**, object of timestamps annotations
- (optional) **ParentSpanId**, identifier of parent span
- (optional) **ProfileId**, UUID, unique identifier to group multiple traces
- (optional) **System**, string, unique identifier of distributed system

1.3.2 Known annotations

UI counts on following annotations

- (required) **url** string (/path)
- (optional) **url_method** string (POST, GET, DELETE, etc)
- (required) **host** string (hostname of target host, taken from URL)
- (required) **targetId** string (target service unique name)
- (optional) **targetHost** string (human-readable name of target host, overrides **host** annotation)
- (optional) **srcId** string (source service unique name)
- (optional) **srcHost** string (human-readable name of source host)
- (optional) **rc** int (response code)
- (optional) **rqbl** long (request body length)
- (optional) **rsbl** long (response body length)
- (optional) **wrapper** empty string (marks span as a wrapper of child spans to override targetid)
- (optional) **root** empty string (marks span as a root span)

- (optional) **revision** int (revision number to overwrite old annotations value)

1.3.3 Timeline annotations

DiTrace gate and UI are using this timeline annotations to calculate trace and spans durations. All timeline annotations should have RFC3389 datetime format.

- (optional) **cs** client has sent request
- (optional) **cr** client has recived response
- (optional) **sr** server has received request
- (optional) **ss** server has sent response

1.4 Integration

Distributed system's services should have an DiTrace gate API client implementation.

The main requirement for clients is low load impact.

There are following techniques for achieve low impact:

- Sampling
- Ring buffering
- Async sending

1.4.1 Sampling

There is no need to trace 100% of requests to get correct statistical results.

You can set sampling to 10% or even lower.

1.4.2 Ring buffering

Client should handle unavailability of "DiTrace" gate. In the other hand, collecting of tracing data should not consume too much memory. Using ring buffer with certain limit is good practice to achieve that.

1.4.3 Async sending

Sending tracing data to the gate should be performed in async way, e.g. in separate thread.

1.4.4 C# client

CSharp client utilize "Logical Call Context" to flow tracing data.

How to use

1. Implement configuration provider

```
using Kontur.Tracing.Core.Config;

public interface IConfigurationProvider
{
    [NotNull]
    ITracingConfig GetConfig();
}
```

2. Init tracing with your configuration provider

```
using Kontur.Tracing.Core.Config;

Trace.Initialize(configProvider);
```

3. Create traces

```
using Kontur.Tracing.Core;


using (var rootContext = Trace.CreateRootContext("Processing client request"))
{
     rootContext.RecordTimepoint(Timepoint.Start);
     rootContext.RecordAnnotation(Annotation.RequestUrl, requestUrl);

     // ... somewhere deep in your code

     using (var childContext = Trace.CreateChildContext("Fetching data from database"))
     {
         childContext.RecordTimepoint(Timepoint.Start);
         data = db.Fetch();
         childContext.RecordTimepoint(Timepoint.Finish);
     }

     // ...
     rootContext.RecordTimepoint(Timepoint.Finish);
}
```

4. Continue traces

   Assume that service A make a call to service B, so service B should continue tracing.

```
using Kontur.Tracing.Core;

HttpListenerContext context;

RequestExtensions.ExtractFromHttpHeaders(context.Request.Headers, out traceId, out contextIc
using (var serverContext = Trace.ContinueContext(traceId, contextId, isActive ?? false, isRc
{
     serverContext.RecordTimepoint(Timepoint.ServerReceive);
     // ...
     // Handle service A request
     // ...
     serverContext.RecordTimepoint(Timepoint.ServerSend);
}
```

1.5 Contact DiTrace Developers

The best way to contact us is to visit our Gitter chat. We usually reply within a day, but sometimes immediately :)

CHAPTER 2

Overview

DiTrace is a distributed tracing system.

Like the others distributed tracing system as Zipkin or Dapper, DiTrace is an instrument to find the problem zones in distributed systems.

But with different and more simple architecture.

2.1 How it works

Every time, when one of distributed system's microservice made a call to another microservice, the data object, called "span" should be created.

Span has an arbitrary number of annotations such as request and response parameters (timestamps, url, response code, etc).

Spans are belong to one trace in hierarchic order with one root span.

Microservices are responsible for:

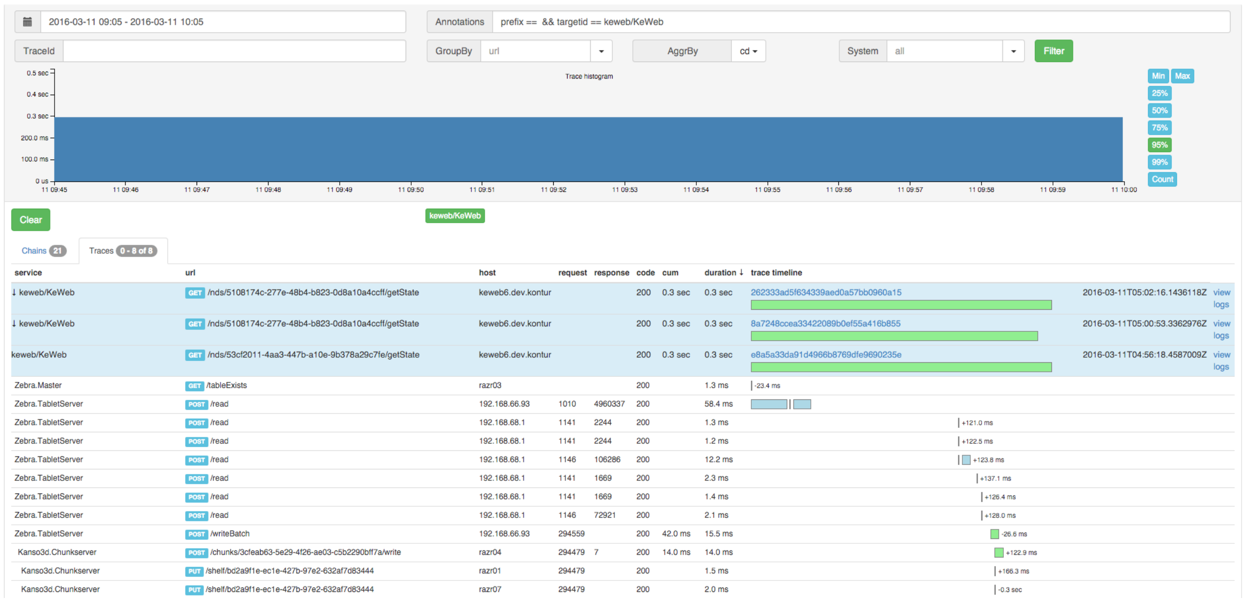
1. Creating traces, spans and sending traceid and spanid within requests between each other.
2. Collecting spans and sending it to the DiTrace gate.

Look DiTrace gate API for more details.

DiTrace gate are responsible for:

1. Collecting and grouping spans from multiple microservices
2. Saving traces to elasticsearch

Elasticsearch has a various stats aggregations for data analysis. UI is an visualization tool for this aggregations and data.



service	url	host	request	response	code	cum	duration ↓	trace timeline		
↓ keweb/KeWeb	GET /nds/5108174c-277e-48b4-b823-0d8a10a4ccff/getState	keweb6.dev.kontur			200	0.3 sec	0.3 sec	262333ad5f634339aed0a57bb0960a15	2016-03-11T05:02:16.1436118Z	view logs
↓ keweb/KeWeb	GET /nds/5108174c-277e-48b4-b823-0d8a10a4ccff/getState	keweb6.dev.kontur			200	0.3 sec	0.3 sec	8a7248ccea33422089b0ef55a416b855	2016-03-11T05:00:53.3362976Z	view logs
keweb/KeWeb	GET /nds/53cf2011-4aa3-447b-a10e-9b378a29c7fe/getState	keweb6.dev.kontur			200	0.3 sec	0.3 sec	e8a5a33da91d4966b8769dfe9690235e	2016-03-11T04:56:18.4587009Z	view logs
Zebra.Master	GET /tableExists	razr03			200		1.3 ms	-23.4 ms		
Zebra.TabletServer	POST /read	192.168.66.93	1010	4960337	200		58.4 ms			
Zebra.TabletServer	POST /read	192.168.68.1	1141	2244	200		1.3 ms	+121.0 ms		
Zebra.TabletServer	POST /read	192.168.68.1	1141	2244	200		1.2 ms	+122.5 ms		
Zebra.TabletServer	POST /read	192.168.68.1	1146	106286	200		12.2 ms	+123.8 ms		
Zebra.TabletServer	POST /read	192.168.68.1	1141	1669	200		2.3 ms	+137.1 ms		
Zebra.TabletServer	POST /read	192.168.68.1	1141	1669	200		1.4 ms	+126.4 ms		
Zebra.TabletServer	POST /read	192.168.68.1	1146	72921	200		2.1 ms	+128.0 ms		
Zebra.TabletServer	POST /writeBatch	192.168.66.93	294559		200	42.0 ms	15.5 ms	-26.6 ms		
Kanso3d.Chunkserver	POST /chunks/3cfeab63-5e29-4f26-ae03-c5b2290bff7a/write	razr04	294479	7	200	14.0 ms	14.0 ms	+122.9 ms		
Kanso3d.Chunkserver	PUT /shelf/bd2a9f1e-ec1e-427b-97e2-632af7d83444	razr01	294479		200		1.5 ms	+166.3 ms		
Kanso3d.Chunkserver	PUT /shelf/bd2a9f1e-ec1e-427b-97e2-632af7d83444	razr07	294479		200		2.0 ms	-0.3 sec		

/spans?system=(string)