Enhancing End-to-End Tracing Systems for Automated Performance Debugging in Distributed Systems

Jethro S. Sun
January 23, 2018
MassOpenCloud Research Group
Introduction
A Sad Story
A Sad Story ...

A distributed system is one in which the failure of a computer you didn’t even know existed can render your own computer unusable.

– Leslie Lamport
What developers and operators really need is a way to understand and troubleshoot a distributed system as a whole.
OpenStack Bug # 1587777 was filed against Horizon.

OpenStack Identity (keystone)

Mitaka: dashboard performance

Bug #1587777 reported by eblock@ndn.ai on 2016-06-01

This bug affects 1 person. Does this bug affect you?

<table>
<thead>
<tr>
<th>Affects</th>
<th>Status</th>
<th>Importance</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenStack Identity (keystone)</td>
<td>Fix Released</td>
<td>Medium</td>
</tr>
</tbody>
</table>

Also affects project

Also affects distribution/package

Bug Description

Environment: Openstack Mitaka on top of Leap 42.1, 1 control node, 2 compute nodes, 3-node-Ceph-cluster.

Issue: Since switching to Mitaka, we're experiencing severe delays when accessing the dashboard - i.e. switching between "Compute - Overview" and "Compute - Instances" takes 15+ seconds, even after multiple invocations.
And only took **10 Month** to figure out it was something wrong in **Keystone**.

<table>
<thead>
<tr>
<th><a href="mailto:eblock@nde.ag">eblock@nde.ag</a> (eblock) on 2017-04-13</th>
</tr>
</thead>
<tbody>
<tr>
<td>Changed in horizon:</td>
</tr>
<tr>
<td><strong>status:</strong> Expired → New</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Akihiro Motoki (amotoki) wrote on 2017-04-18:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is it a keystone issue now?</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Akihiro Motoki (amotoki) wrote on 2017-04-18:</th>
</tr>
</thead>
<tbody>
<tr>
<td>After following the comments above, it looks related to keystone not horizon now. Changing the project.</td>
</tr>
<tr>
<td><strong>affects:</strong> horizon → keystone</td>
</tr>
</tbody>
</table>
Question: Is there a way to make developers’ and operators’ life less miserable?
Question:

Is there a way to make developers’ and operators’ life less miserable?

YES. End-to-end tracing
End-to-End Tracing, what is it and where we are today?
End-to-End Tracing

Definition (End-to-End Tracing)

End-to-end tracing captures the workflow of causally-related activity (e.g., work done to process a request) within and among every component of a distributed system.¹

¹So, you want to trace your distributed system? Key design insights from years of practical experience. Raja Sambasivan et al.
A Typical End-to-End Tracing Infrastructure

**Definition (Trace Metadata)**

Fields propagated with causally-related event to identify their workflows. They are usually unique IDs or in a format of logical clock stored thread-locally or context-locally.

**Definition (Trace Points)**

Instrumentation points in the system used to identify individual work done, and also propagate necessary metadata.

**Definition (Backend)**

Central collector that gathers pieces of trace data and reconstruct them into full feature-riched trace.
End-to-end Tracing gains its popularity gradually...

### Table 1: Timeline

<table>
<thead>
<tr>
<th>Year</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>2002</td>
<td>Pinpoint</td>
</tr>
<tr>
<td>2004</td>
<td>Magpie, SDI</td>
</tr>
<tr>
<td>2005</td>
<td>Causeway</td>
</tr>
<tr>
<td>2006</td>
<td>Pip, Stardust</td>
</tr>
<tr>
<td>2007</td>
<td>X-Trace</td>
</tr>
<tr>
<td>2010</td>
<td>Google Dapper</td>
</tr>
<tr>
<td>2012</td>
<td>Zipkin, HTrace</td>
</tr>
<tr>
<td>2013</td>
<td>Node.js CLS</td>
</tr>
<tr>
<td>2014</td>
<td>Apple Activity Tracing, <strong>Blkin</strong></td>
</tr>
<tr>
<td>2015</td>
<td>AppNeta, AppDynamics, NewRelic, <strong>OSProfiler</strong></td>
</tr>
<tr>
<td>2017</td>
<td></td>
</tr>
</tbody>
</table>
End-to-end Tracing gains its popularity gradually...

<table>
<thead>
<tr>
<th>Year</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>2002</td>
<td>Pinpoint</td>
</tr>
<tr>
<td>2004</td>
<td>Magpie, SDI</td>
</tr>
<tr>
<td>2005</td>
<td>Causeway</td>
</tr>
<tr>
<td>2006</td>
<td>Pip, Stardust</td>
</tr>
<tr>
<td>2007</td>
<td>X-Trace</td>
</tr>
<tr>
<td>2010</td>
<td>Google Dapper</td>
</tr>
<tr>
<td>2012</td>
<td>Zipkin, HTrace</td>
</tr>
<tr>
<td>2013</td>
<td>Node.js CLS</td>
</tr>
<tr>
<td>2014</td>
<td>Apple Activity Tracing, Blkin</td>
</tr>
<tr>
<td>2015</td>
<td>AppNeta, AppDynamics, NewRelic, OSPProfiler</td>
</tr>
<tr>
<td>2017</td>
<td>..., Twitter, Prezi, SoundCloud, HDFS, HBase, Accumulo, Phoenix, Baidu, Neflit, Pivotal, Coursera, Census (Google), Canopy (Facebook), Jaeger (Uber), ...</td>
</tr>
</tbody>
</table>
To distinguish tracing systems:
To distinguish tracing systems:

- On-demand (Rudimentary)
To distinguish tracing systems:

- On-demand (Rudimentary)
- Be always on (Smart Sampling)
To distinguish tracing systems:

- On-demand (Rudimentary)
- Be **always on** (Smart Sampling)
- Collect trace data asynchronously
To distinguish tracing systems:

- On-demand (Rudimentary)
- Be always on (Smart Sampling)
- Collect trace data asynchronously
- DAG-based model to represent events
End-to-End Tracing Systems Service Model

To distinguish tracing systems:

- On-demand (Rudimentary)
- Be **always on** (Smart Sampling)
- Collect trace data asynchronously
- DAG-based model to represent events
- Logical clock support
### Table 2: Comparing end-to-end tracing systems features between Jaeger, Zipkin, Pivot Tracing, Dapper, Canopy, OSProfiler and Blkin.

<table>
<thead>
<tr>
<th>Systems Can Be Applied to</th>
<th>Rudimentary</th>
<th>Features Needed to Be Always on</th>
<th>Advanced Features</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>On-demand</td>
<td>Sampling</td>
<td>Async. Collect.</td>
</tr>
<tr>
<td><strong>Jaeger Tracing</strong></td>
<td>Broadly (K8s, OpenShift)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Zipkin Tracing</strong></td>
<td>Broadly</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Pivot Tracing</strong></td>
<td>Hadoop/Java based systems</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Dapper</strong></td>
<td>N/A</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Canopy</strong></td>
<td>N/A</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>OSProfiler</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Blkin</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table 2: Comparing end-to-end tracing systems features between Jaeger, Zipkin, Pivot Tracing, Dapper, Canopy, OSPProfiler and Blkin.

<table>
<thead>
<tr>
<th>Systems Can Be Applied to</th>
<th>Rudimentary Features</th>
<th>Advanced Features</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>On-demand</td>
<td>Sampling</td>
</tr>
<tr>
<td>Jaeger Tracing</td>
<td>Broadly (K8s, OpenShift)</td>
<td>✓</td>
</tr>
<tr>
<td>Zipkin Tracing</td>
<td>Broadly</td>
<td>✓</td>
</tr>
<tr>
<td>Pivot Tracing</td>
<td>Hadoop/Java based systems</td>
<td>✓</td>
</tr>
<tr>
<td>Dapper</td>
<td>N/A</td>
<td>✓</td>
</tr>
<tr>
<td>Canopy</td>
<td>N/A</td>
<td>✓</td>
</tr>
<tr>
<td>OSPProfiler</td>
<td>OpenStack</td>
<td>✓</td>
</tr>
<tr>
<td>Blkin</td>
<td>Ceph</td>
<td>✓</td>
</tr>
</tbody>
</table>
Approaches for Enabling Sophisticated Tracing in OpenStack
Jaeger Tracing

**Advantages**

- Support smart sampling
- Support collecting trace data async.

**Disadvantages**

- Doesn’t support DAG-based model
- Doesn’t use advanced logical clock as the metadata
Jaeger vs OSProfiler

OSProfiler

Advantages
• Rudimentary on-demand tracing
• Already adopt by OpenStack and have instrumentation

Disadvantages
• Doesn’t have sampling
• Doesn’t collect trace data asynchronously
• Doesn’t support DAG-based model
• Doesn’t use advanced logical clock as the metadata
Jaeger vs OSProfiler

**OSProfiler**

**ADVANTAGES**

- Rudimentary on-demand tracing
- Already adopt by OpenStack and have instrumentation

**DISADVANTAGES**

- Doesn’t have sampling
- Doesn’t collect trace data asynchronously
- Doesn’t support DAG-based model
- Doesn’t use advanced logical clock as the metadata
OSProfiler with Jaeger Tracing

**ADVANTAGES**
- Rudimentary on-demand tracing
- Already adopt by OpenStack and have instrumentation

**DISADVANTAGES**
- Doesn’t have sampling
- Doesn’t collect trace data asynchronously
- Doesn’t support DAG-based model
- Doesn’t use advanced logical clock as the metadata
Jaeger vs OProfiler

OSPProfiler with Jaeger Tracing

Advantages

• Rudimentary on-demand tracing
• Already adopt by OpenStack and have instrumentation
• Modifications we done can be directly other Jaeger instrumented systems

Disadvantages

• Doesn’t have sampling
• Doesn’t collect trace data asynchronously
• Doesn’t support DAG-based model
• Doesn’t use advanced logical clock as the metadata
Feasibility

Key Challenges:

**Trace Metadata/OSProfiler library change**
- Implement CONTEXT generation using Jaeger
- Implement CONTEXT propagation using Jaeger

**Trace Points/OpenStack instrumentation**
- All of the instrumentation will be able to be reused\(^2\)

**Backend side**
- Need to deploy Backend/Collector for Jaeger Tracing

---
\(^2\)Modifying instrumentation for the purpose of our research is orthogonal.
Feasibility

Key Challenges:

**Trace Metadata/OSProfiler library change**
- Implement CONTEXT generation using Jaeger
- Implement CONTEXT propagation using Jaeger

**Trace Points/OpenStack instrumentation**
- All of the instrumentation will be able to be reused² ✓

**Backend side**
- Need to deploy Backend/Collector for Jaeger Tracing ✓

² Modifying instrumentation for the purpose of our research is orthogonal.
Feasibility

Definition (Context)

Context is an abstraction of the metadata so that it is easier to interact with (injecting/extracting a trace to/from).

Example Implementation

```go
// Context holds the basic metadata.
type Context struct {
    TraceID uint64
    SpanID uint64
    Sampled bool
    Baggage map[string]string // initialized on first use
}
```
Feasibility: Context Generation

**CONTEXT generation:**

*All of the modification will be done in OSProfiler library*\(^3\)

- The span context generation will be done using Jaeger to substitute the OSProfiler implementation.

---

\(^3\)In OpenStack developers instrument their codebase using functionalities implemented in OSProfiler library.
Feasibility: Context Propagation

CONTEXT propagation:

**OpenStack Instrumentation side**
- **REST API**
  Transform the metadata propagation in OpenStack clients to propagate Jaeger metadata. We might only need to change OSProfiler library.

- **RPC API**
  Need to implement helper functions for metadata propagation RPC. We might need to modify component codebase depends on the RCP is handled in different components.

**OSProfiler Library side**
- Need to deploy Backend/Collector for Jaeger Tracing
**CONTEXT generation:**

- A talk during 2017 OpenStack Sydney Summit demonstrates how easy to plainly record all the OSProfiler tracing information in Jaeger. *(i.e. Context generation is done in OSProfiler)*
- Additionally we need to generate context using Jaeger tracing.

**CONTEXT propagation:**

- Will begin to look at ways to enforce metadata propagation in OpenStack **RPC API** and **REST API**
Jaeger Tracing Approach
OSProfiler with Jaeger

Two key challenges to address:

• Doesn’t support DAG-based model
• Doesn’t use advanced logical clock as the metadata
Two key challenges to address:

- Doesn’t support DAG-based model
- Doesn’t use advanced logical clock as the metadata
Definition (Span)

A Span represents a logical unit of work in the system that has an operation name, the start time of the operation, and the duration. Spans may be nested and ordered to model causal relationships. An RPC call is an example of a span.
Definition (DAG-based Model)

Modeling traces as directed, acyclic graphs (DAGs), with nodes representing events in time, and edges representing causality.
Pattern #1

`func bar` and `func grunt` are issued by `func foo` **concurrently**, and `func foo` only ends after both of the individual work are done in `func bar` and `func grunt`. 
Pattern #1

func bar and func grunt are issued by func foo concurrently, and func foo only ends after both of the individual work are done in func bar and func grunt.

This pattern we referred to fan-in-and-fan-out in our group.
Pattern #2

`func bar` and `func grunt` are also both issued by `func foo`, but `func grunt` can start only after the work in `func bar` is done.
**Pattern #2**

`func bar` and `func grunt` are also both issued by `func foo`, but `func grunt` can start only after the work in `func bar` is done.

*func bar* and *func grunt* are executed **in sequential** instead of in parallel.
Since span model doesn’t really capture concurrency and synchronization, Pattern #1 and Pattern #2 are both recognized and documented as the same.
To be able to adopt the DAG-based model, start and stop of a span must be treated as separate events, and get captured.
• Implemented a Proof-of-Concept in OSProfiler before we are considering move to Jaeger Tracing.
• Now need to re-implement in Jaeger and evaluate it
Logical Clock Support for Metadata Propagation
At the heart of end-to-end tracing is metadata propagation to identify causally-related events across nodes.
• At the heart of end-to-end tracing is metadata propagation to identify causally-related events across nodes.

• Usually the metadata are stored in thread-local or context-local storage.
Example Implementation

```
Span(
    Tracer tracer,
    String operationName,
    SpanContext context,
    long startTimeMicroseconds,
    long startTimeNanoTicks,
    ...
)

// SpanContext holds the basic Span metadata.

type SpanContext struct {
    TraceID uint64
    SpanID uint64
    Sampled bool
    Baggage map[string]string // initialized on first use
}
```
Limitations:

- Simple timestamp are not resilient to failures
- Extremely tricky to deal with “fan-in and fan-out”
- Usually need a static view of the distributed system for generating the globally unique identifier
Interval Tree Clock

Interval Tree Clock:

• Can create, retire and reuse identifiers autonomously.

• Works in dynamically setting (stamps grow and shrink adapting to the system)

Interval Tree Clock models causality tracking by operations:

• **Fork**
  Branch a stamp into a pair.

• **Event**
  Add a new event to the component.

• **Join**
  Merge two stamps to create a new one.
Our Plan:

Use Interval Tree Clock as the logical clock to avoid dealing with the branching and rejoining using random identifiers.
Additional Changes If without Jaeger
To control the cost of the metadata propagation, **Tracing Agents** are deployed to:

- collection trace data asynchronously
- enforce smart sampling methods
- control the usage of local resources
Jaegr Tracing:

The agent abstracts the routing and discovery of the collectors away from the client.
• We think adopting Jaeger in OSProfiler can avoid unnecessary effort for performance diagnosis in OpenStack.
• We identify implementing DAG-based model and advanced logical clock in the tracing infrastructure to be the important part in a novel and efficient end-to-end tracing system.