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*

A Sad Story ...

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)

Overview Code **Bugs** Blueprints Translations Answers

Mitaka: dashboard performance

Bug #1587777 reported by eblock@nde.ag on 2016-06-01

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

Affects	Status	Importance
OpenStack Identity (keystone)	Fix Released	Medium

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.

Changed in horizon:
status:Expired → New

Is it a keystone issue now?

After following the comments above, it looks related to keystone not horizon now. Changing the project.

affects:horizon → keystone

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?

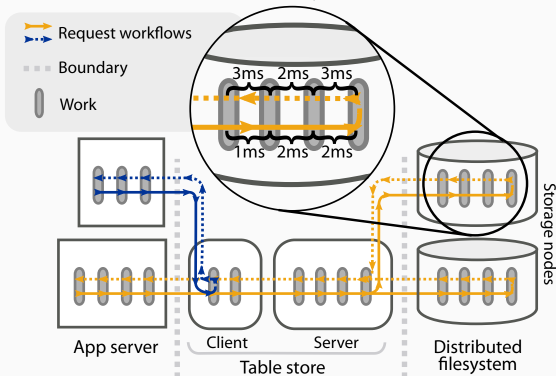
YES. End-to-end tracing

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]



[1] **So, 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

Year	Systems
2002	Pinpoint
2004	Magpie, SDI
2005	Causeway
2006	Pip, Stardust
2007	X-Trace
2010	Google Dapper
2012	Zipkin, HTrace
2013	Node.js CLS
2014	Apple Activity Tracing, **Blkin**
2015	AppNeta, AppDynamics, NewRelic, **OSProfiler**
2017	

End-to-end Tracing gains its popularity gradually...

Table 1 Timeline

Year	Systems
2002	Pinpoint
2004	Magpie, SDI
2005	Causeway
2006	Pip, Stardust
2007	X-Trace
2010	Google Dapper
2012	Zipkin, HTrace
2013	Node.js CLS
2014	Apple Activity Tracing, **Blkin**
2015	AppNeta, AppDynamics, NewRelic, **OSProfiler**
2017	..., Twitter, Prezi, SoundCloud, HDFS, HBase, Accumulo, Phoenix, Baidu, Neflit, Pivotal, Coursera, Census (Google), Canopy (Facebook), Jaeger (Uber), ...

End-to-End Tracing Systems Service Model

To distinguish tracing systems:

End-to-End Tracing Systems Service Model

To distinguish tracing systems:

- On-demand (Rudimentary)

End-to-End Tracing Systems Service Model

To distinguish tracing systems:

- On-demand (Rudimentary)
- Be **always on** (Smart Sampling)

End-to-End Tracing Systems Service Model

To distinguish tracing systems:

- On-demand (Rudimentary)
- Be **always on** (Smart Sampling)
- Collect trace data asynchronously

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

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

Comparing End-to-End Tracing Systems

Table 2: Comparing end-to-end tracing systems features between Jaeger, Zipkin, Pivot Tracing, Dapper, Canopy, OSProfiler and Blkin.

	Systems Can Be Applied to	**Rudimentary**	**Features Needed to Be Always on**		**Advanced Features**	
		On-demand	Sampling	Async. Collect.	DAG-based Model	Interval Tree Clock
Jaeger Tracing	Broadly (K8s, OpenShift)	✗	✓	✓	✗	✗
Zipkin Tracing	Broadly	✗	✓	✓	✗	✗
Pivot Tracing	Hadoop/Java based systems	✗	✓	✓	✓	✓
Dapper	N/A	✗	✓	✓	✗	✗
Canopy	N/A	✗	✓	✓	✓	✗
OSProfiler						
Blkin						

Comparing End-to-End Tracing Systems

Table 2: Comparing end-to-end tracing systems features between Jaeger, Zipkin, Pivot Tracing, Dapper, Canopy, OSProfiler and Blkin.

	Systems Can Be Applied to	**Rudimentary**	**Features Needed to Be Always on**		**Advanced Features**	
		On-demand	Sampling	Async. Collect.	DAG-based Model	Interval Tree Clock
Jaeger Tracing	Broadly (K8s, OpenShift)	✗	✓	✓	✗	✗
Zipkin Tracing	Broadly	✗	✓	✓	✗	✗
Pivot Tracing	Hadoop/Java based systems	✗	✓	✓	✓	✓
Dapper	N/A	✗	✓	✓	✗	✗
Canopy	N/A	✗	✓	✓	✓	✗
OSProfiler	OpenStack	✓	✗	✗	✗	✗
Blkin	Ceph	✓	✗	✗	✗	✗

Approaches for Enabling Sophisticated Tracing in OpenStack

Jaeger vs OSProfiler

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

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

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

OSProfiler **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

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.

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

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

Status Update

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

Two key challenges to address:

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

OSProfiler with Jaeger

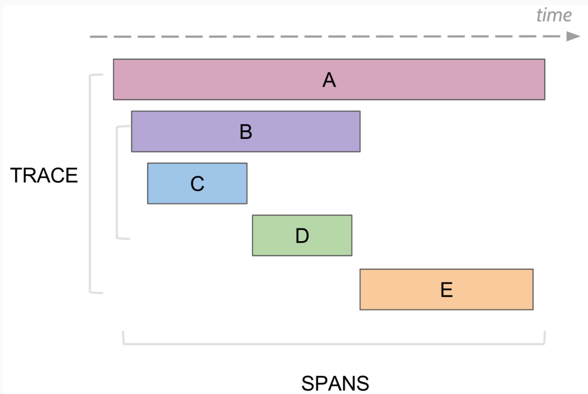
Two key challenges to address:

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

DAG-based Model vs Span Model

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.

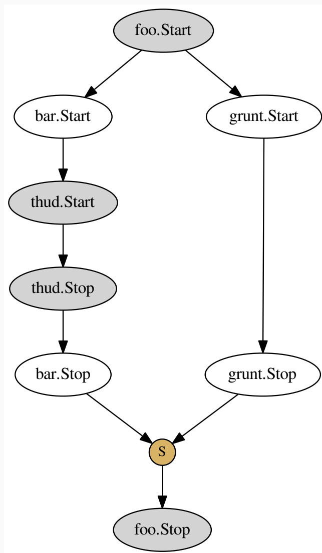


DAG-based Model vs Span Model

Definition (DAG-based Model)

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

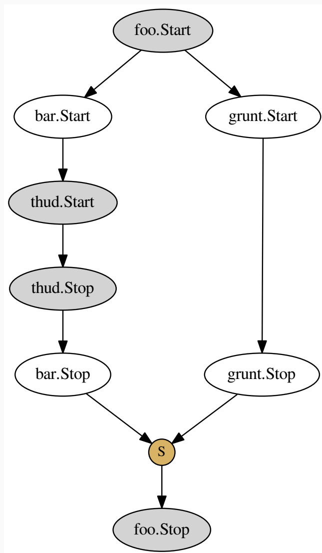
DAG-based Model vs Span Model



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.

DAG-based Model vs Span Model

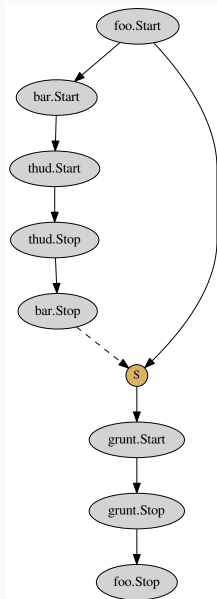


Pattern #1

`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.

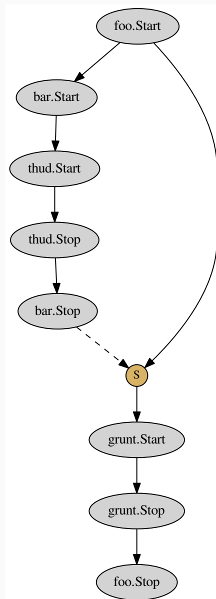
This pattern we referred to ***fan-in-and-fan-out*** *in our group.*

DAG-based Model vs Span Model



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.

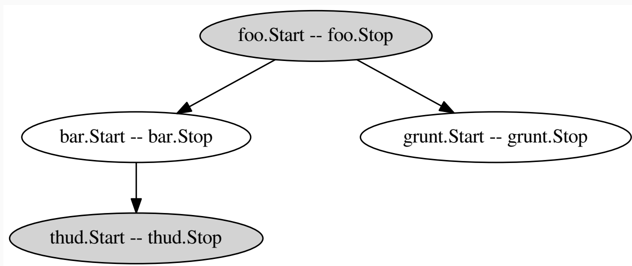


Pattern #2

`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.*

DAG-based Model vs Span Model



Since span model doesn't really capture **concurrency** and **synchronization**, PATTERN #1 and PATTERN #2 are both recognized and documented as the same.

Applying DAG-based Model

To be able to adopt the DAG-based model, start and stop of a span must be treated as separate events, and get captured.

Status Update

- 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

Metadata Propagation

Metadata Propagation

- At the heart of end-to-end tracing is metadata propagation to identify causally-related events across nodes.

Metadata Propagation

- 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
}
```

Logical Clock Support for Metadata Propagation

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:

- 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.

Status Update

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

Requirements for Always-on

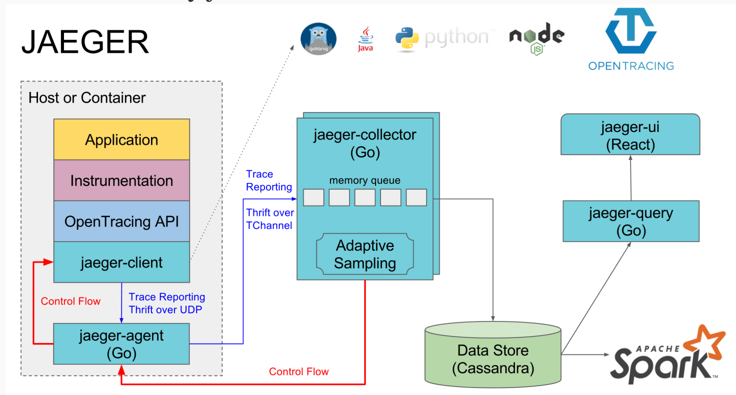
To control the cost of the metadata propagation, Tracing Agents are deployed to:

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

Requirements for Always-on

Jaegr Tracing:

The agent abstracts the routing and discovery of the collectors away from the client.



Summary

- 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.