A Tracing Technique for Understanding the Behavior of Large-Scale Distributed Systems

Yuichi Bando
NTT Software Innovation Center
Who am I?

• Research engineer at NTT Software Innovation Center (SIC)
  • SIC is developing open source cloud platforms and promoting collaborative service development with NTT operating companies

• working on techniques for improving reliability of distributed systems such as
  • Sheepdog (scale out storage system)
  • OpenStack Swift (object storage system)
Agenda

1. Background
2. Introduction to distributed tracing
3. Adding trace feature to Eventlet
4. Demo with OpenStack Swift
5. Evaluation
Finding performance bottlenecks in modern large-scale distributed systems is difficult.
How should we find bottlenecks?

• There are several useful tracing tools for stand-alone systems
  • ftrace: tracing tool for the Linux Kernel
  • LTTng: tracing tool for the Linux Kernel and applications

• However, such tools are not enough for distributed systems
  • cannot trace actions and interactions of hundreds of components located on many different machines
How should we find bottlenecks?

- **Distributed Tracing**
  - performance profiling method for finding bottlenecks of complex distributed systems
  - gather cluster-wide timing data
  - extract the causal relationships among RPCs

---

Example of distributed tracing

<table>
<thead>
<tr>
<th>Time (ms)</th>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>3000</td>
<td>Frontend</td>
</tr>
<tr>
<td>2700</td>
<td>Backend1</td>
</tr>
<tr>
<td>2000</td>
<td>Backend3</td>
</tr>
<tr>
<td>500</td>
<td>Backend2</td>
</tr>
</tbody>
</table>

RPC found!
Agenda

1. Background
2. Introduction to distributed tracing
3. Adding trace feature to Eventlet
4. Demo with OpenStack Swift
5. Evaluation
Focus in this talk

Approaches of distributed tracing

<table>
<thead>
<tr>
<th>Black-box based approach</th>
<th>Project5 [1], WAP5 [2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔ higher degree of app-level transparency</td>
<td>x some amount of imprecision and possibly larger overheads</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Explicit annotation-based approach</th>
<th>✔ deeper understanding of process flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>x need for trace targets to be modified</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X-Trace [3]</th>
<th>comprehensive modifications (client, server, NW devices)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Google Dapper [4]</td>
<td>only limited modification (common RPC library)</td>
</tr>
<tr>
<td>Twitter Zipkin [5]</td>
<td>only limited modification (common RPC library) OSS implementation based on Dapper</td>
</tr>
</tbody>
</table>

[1] Aguilera et al. SOSP ’03
[3] Fonseca et al. NSDI ’07
What’s Zipkin?

• Zipkin is a distributed tracing framework which helps us collect and visualize trace data.

1. generate trace data
2. send data
3. store
4. query

Architecture of Zipkin tracing

Trace targets
Scribe
logging daemon developed by Facebook (OSS)

Collector
Storage
Web UI

Zipkin (OSS)
What’s Zipkin?

Web UI of Zipkin

Span of an RPC

Services
Trace data for Zipkin

• RPC timing info of every task
  • Timestamp of when a service sends a request or receives a response

• A few unique IDs
  • traceId: identifies a request
  • spanId: identifies a span of the request
    • A span represents one specific RPC call
  • parentId: identifies the parent span

Note: Zipkin does NOT require high-precision timestamp since pairs of spanId and parentId give causal relationships among RPCs
Example: propagation of IDs

- traceId and spanId are passed to downstream servers along with RPC

```
create new IDs

traceId=100
spanId=40

100, 40

traceId=100
spanId=56
parentId=40

100, 56

100, 56
```

Frontend

Backend 1

Backend 2

Backend 3

Span → RPC

Temporary storage
(Data is temporarily cached in memory)
# Web UI of Zipkin

## Service Breakdown

<table>
<thead>
<tr>
<th>Service</th>
<th>Duration</th>
<th>170.811ms</th>
<th>341.622ms</th>
<th>512.434ms</th>
<th>683.245ms</th>
<th>854.057ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>frontend</td>
<td>854.057ms</td>
<td>Request</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>backend1</td>
<td>752.064ms</td>
<td>GET</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>backend2</td>
<td>300.496ms</td>
<td>GET</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>backend3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>300.582ms</td>
</tr>
</tbody>
</table>

### Levels of Nesting

Levels of nesting represent hierarchical relationships among RPCs.

### Latency Breakdown

Latency breakdown of upper level service.
How can we start Zipkin tracing?

- Middleware such as RPC Library needs to generate trace data
  - Some libraries already support Zipkin tracing
    - Finagle: Asynchronous network stack for JVM [1]
    - Twisted: Python event-driven networking engine [2]
    - Django: Python web framework [3]

- Libraries that support Zipkin are, however, still limited
  - Not available for popular cloud platforms such as OpenStack
  - Need to expand its support to key OSS libraries toward wide adoption of "tracing"

Agenda

1. Background
2. Introduction to distributed tracing
3. Adding trace feature to Eventlet
4. Demo with OpenStack Swift
5. Evaluation
What’s Eventlet?

• A popular Python networking library [1]
  • over 2.5M downloads from PyPI
  • widely used in OpenStack project
    • Compute (Nova)
    • Identity (Keystone)
    • Image Service (Glance)
    • Networking (Neutron)
    • Block Storage (Cinder)
    • Object Storage (Swift) etc…

We implemented trace feature to Eventlet

Scope
- Eventlet/WSGI applications which use HTTP for internal communications
  - OpenStack Swift is an example

Some OpenStack components also use AMQP, but it's not supported
- Hybrid protocol support is a future work

WSGI: Web Server Gateway Interface
AMQP: Advanced Message Queuing Protocol
Implementation to Eventlet

• To capture causal relationships of spans, our patch propagates IDs via HTTP headers

The point where Eventlet receives a request

if HTTP headers do NOT contain IDs:
  generate traceId, spanId
else:
  extract IDs from headers
  • • existing code

The point where Eventlet sends a request

put IDs to HTTP headers
  • • existing code

```
eventlet.wsgi.HttpProtocol.handle_one_request()
eventlet.green.httplib.HTTPConnection.endheaders()
```

User → request → Frontend
HTTP traceId, spanId → Backend
→ HTTP
Implementation to Eventlet

• We used **monkey patching** technique to insert code for tracing
  
  • No modification to original code
  • We override two methods (listed in previous page)

```python
from eventlet.green.http import HTTPConnection

org_endheaders = HTTPConnection.endheaders

def my_endheaders(self):
    put IDs to HTTP headers #code for tracing
    org_endheaders(self)   #original one

HTTPConnection.endheaders = my_endheaders #override

e.g.) Monkey patch to endheaders()
```
How to use

• Add two lines to your application to start tracing

• Optionally set sampling rate for reducing overhead
  • if sampling_rate=1.0, all requests will be traced
  • if sampling_rate=0.1, only 1/10 requests will be traced

```
from eventlet.zipkin import patcher
patcher.enable_trace_patch(sampling_rate=0.1)
```
Current status

• We first proposed this distributed tracing idea and Eventlet maintainer agreed with it [1]

• We proposed the patch [2], and it is planned to be merged in Eventlet v0.18
  • May 9, 2015: v0.17.4 (latest release)

Agenda

1. Background
2. Introduction to distributed tracing
3. Adding trace feature to Eventlet
4. Demo with OpenStack Swift
5. Evaluation
What’s Swift?

- A distributed object storage system
  - implemented as Eventlet/WSGI application
  - uses HTTP for internal communications

![Diagram of Swift architecture]

- Client
- REST API (PUT/GET/DEL)
- Proxy: request routing
- Account: handles listing of containers
- Container: handles listing of objects
- Object: stores objects (has 3 replicas)

Storage node

account

container

object

...
Demo

- Tracing Swift with patched Eventlet

VM on my laptop emulates a four node Swift cluster
1. Background
2. Introduction to distributed tracing
3. Trace feature enhancement to Eventlet/WSGI
4. Demo with OpenStack Swift
5. Evaluation
What we measure

• **Tracing overhead**
  - Impact on Swift throughputs (PUT/GET/DEL)
  - Impact on resource usage (CPU, MEM, NW)
Environment

• 1 swift-bench
  • # of request: 10000 PUT/GET/DEL
  • object size: 4 KB*
  • concurrency: 10

• 4 node Swift cluster
  • Fluend is used as logger

• 1 Zipkin collector
  • with SQLite

* Setting small object size will highlight the overhead since each request will be lightweight
Impact on Swift throughput (PUT)

Lowering sampling rate reduces overhead

Transactions/s

Trace OFF 1/1 1/4 1/16 1/1024

- 5.8 % - 4.7 % - 4.3 % - 0.3 %
Impact on Swift throughput (GET)

Transactions/s vs. sampling rate

Trace OFF: 336.5
304: -9.7%
1/4: 328.1
3.4%
1/16: 325
-3.1%
1/1024: 326.1

Copyright©2015 NTT Corp. All Rights Reserved.
Impact on Swift throughput (DEL)

- Trace OFF: 175.8
- 1/1: 161.6 (-8.0%)
- 1/4: 162.9 (-7.3%)
- 1/16: 167.2 (-4.9%)
- 1/1024: 169.5 (-3.6%)
## Impact on resource usage of Swift cluster

<table>
<thead>
<tr>
<th>Sampling rate</th>
<th>Avg. CPU Usage (% change)</th>
<th>Avg. MEM Usage (% change)</th>
<th>Avg. NW write rate (% change)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace OFF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1/1</td>
<td>0.95 %</td>
<td>1.2 % (+ 27 MB)</td>
<td>16.8 % (+ 303 KB/s)</td>
</tr>
<tr>
<td>1/4</td>
<td>0.39 %</td>
<td>- 0.038 %</td>
<td>4.1 %</td>
</tr>
<tr>
<td>1/16</td>
<td>0.23 %</td>
<td>- 0.31 %</td>
<td>0.34 %</td>
</tr>
<tr>
<td>1/1024</td>
<td>0.11 %</td>
<td>- 0.11 %</td>
<td>- 1.3 %</td>
</tr>
</tbody>
</table>

* some negative numbers due to experimental error
Discussion

• Even in the worst case (rate=1/1), decrease in application throughput is less than 10%
  • Though tracking all requests consumes some amount of NW bandwidth, it is acceptable for debugging or lower traffic services

• In addition, low sampling rate is enough for analyzing the tendency of performance
  • In Dapper paper, Google reported
    • “In practice, we have found that there is still an adequate amount of trace data for high-volume services when using a sampling rate as low as 1/1024”

Conclusion

• Distributed tracing gives a practical way to find bottlenecks in distributed systems

• Our patch to Eventlet will help you understand WSGI-based distributed systems (e.g. Swift) even if you are not familiar with the interior
  • low overhead
  • useful for both debugging and monitoring

If you have a similar issue with a distributed system, try Zipkin! Even if your networking library is not Zipkin compliant, our patch will be a useful reference to modify it.
Thanks a lot for your kind attention!
Any questions?
• Annotation API
  • Add your own additional info for deeper understanding
  • from anywhere in your code

```python
from eventlet.zipkin import api
api.put_annotation('Your own message')
api.put_key_value('key', 'value')
```
Out patch: other option 1

```python
# Key-value has no time component
api.put_key_value()
```

```python
# Annotation is recorded with timestamp
api.put_annotation()
```
Out patch: other option 2

• Application Log Tracing
  • Add application log as annotations for deeper understanding

```python
from eventlet.zipkin import patcher
patcher.enable_trace_patch(trace_app_log=True)
```

* Assume that target application uses python standard logging library
Out patch: other option 2

Captured swift log
## DEMO: screen shot

### Trace Swift PUT request

<table>
<thead>
<tr>
<th>Duration: 92.621ms</th>
<th>Services: 3</th>
<th>Depth: 3</th>
<th>Total Spans: 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expand All</td>
<td>Collapse All</td>
<td>Filter Service Search</td>
<td></td>
</tr>
</tbody>
</table>

**swift-container-server x3**  | **swift-object-server x3**  | **swift-proxy-server x1**

<table>
<thead>
<tr>
<th>Services</th>
<th>Duration</th>
<th>18.524ms</th>
<th>37.048ms</th>
<th>55.572ms</th>
<th>74.096ms</th>
<th>92.621ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>swift-proxy-server</td>
<td>28.290ms</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>swift-object-server</td>
<td>13.624ms</td>
<td>CPU 0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>swift-container-server</td>
<td>0.260ms</td>
<td>PUT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>swift-object-server</td>
<td>83.011ms</td>
<td>PUT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>swift-container-server</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>swift-object-server</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.447ms</td>
</tr>
<tr>
<td>swift-container-server</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.447ms</td>
</tr>
<tr>
<td>swift-object-server</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.447ms</td>
</tr>
<tr>
<td>swift-container-server</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.447ms</td>
</tr>
</tbody>
</table>
DEMOn: screen shot

Trace Swift GET request

<table>
<thead>
<tr>
<th>Duration: 36.106ms</th>
<th>Services: 4</th>
<th>Depth: 2</th>
<th>Total Spans: 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expand All</td>
<td>Collapse All</td>
<td>Filter Service Search...</td>
<td></td>
</tr>
</tbody>
</table>

- swift-account-server x1
- swift-container-server x1
- swift-object-server x1
- swift-proxy-server x1

<table>
<thead>
<tr>
<th>Services</th>
<th>7.221ms</th>
<th>14.442ms</th>
<th>21.663ms</th>
<th>28.884ms</th>
<th>36.106ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>swift-proxy-server</td>
<td>36.106ms: GET</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>swift-account-server</td>
<td>-</td>
<td>4.677ms: HEAD</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>swift-container-server</td>
<td>-</td>
<td>-</td>
<td>5.543ms: HEAD</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>swift-object-server</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2.604ms: GET</td>
<td>-</td>
</tr>
</tbody>
</table>
### DEMO: screen shot

**Detailed information view**

<table>
<thead>
<tr>
<th>Relative Time</th>
<th>Duration</th>
<th>Service</th>
<th>Annotation</th>
<th>Host</th>
</tr>
</thead>
<tbody>
<tr>
<td>22.926ms</td>
<td>swift-object-server</td>
<td>Server Receive</td>
<td></td>
<td>127.0.0.1:6040</td>
</tr>
<tr>
<td>24.851ms</td>
<td>swift-object-server</td>
<td>127.0.0.1 - - [21/May/2015:09:28:06 +0000] &quot;GET /sd4/293/AUTH_test/container/test.txt&quot; 200 10 &quot;GET <a href="https://127.0.0.1:8080/v1/AUTH_test/container/test.txt">https://127.0.0.1:8080/v1/AUTH_test/container/test.txt</a>&quot; &quot;bxb864e6414d9546a8868fc-00555da526&quot; &quot;proxy-server 6907&quot; 0.0015 &quot;-&quot;</td>
<td>127.0.0.1:6040</td>
<td></td>
</tr>
<tr>
<td>25.530ms</td>
<td>swift-object-server</td>
<td>Server Send</td>
<td></td>
<td>127.0.0.1:6040</td>
</tr>
</tbody>
</table>

**Key** | **Value**
---|---
http.uri | /sd4/293/AUTH_test/container/test.txt
http.status | 200
## Evaluation: Software version

<table>
<thead>
<tr>
<th>Software</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Swift</td>
<td>2.0.0</td>
</tr>
<tr>
<td>Swift-bench</td>
<td>1.0</td>
</tr>
<tr>
<td>Eventlet</td>
<td>0.17.1</td>
</tr>
<tr>
<td>Fluentd</td>
<td>0.10.61</td>
</tr>
<tr>
<td>Zipkin</td>
<td>1.1.0</td>
</tr>
</tbody>
</table>
[bench]
auth = http://swift_proxy_ip:8080/auth/v1.0
user = test:tester
key = testing

concurrency = 10
object_size = 4096

#Number of objects to PUT
num_objects = 10000

#Number of GET operations to perform
num_gets = 10000

#Number of containers to distribute objects among
num_containers = 20
Evaluation: td-agent.conf (Fluentd)

# in_scribe
<source>
  type scribe
  port 9999
</source>

# out_scribe
<match zipkin.**>
  type scribe
  host zipkin_collector_ip
  port 9410
  flush_interval 60s
</match>
Evaluation: Zipkin configuration

$ git clone https://github.com/twitter/zipkin.git
$ cd zipkin

# Open 3 terminals
(terminal1) $ bin/collector
(terminal2) $ bin/query
(terminal3) $ bin/web
### Evaluation: Size of trace data per request

<table>
<thead>
<tr>
<th></th>
<th>1 PUT</th>
<th>1 GET</th>
<th>1 DEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of trace data (Bytes)</td>
<td>4096</td>
<td>1024</td>
<td>4096</td>
</tr>
</tbody>
</table>

* The size is measured from `zipkin/zipkin.db`
* Core annotations and `http.uri` annotation are traced

• **Note: This result is an example since data size is dependent on each service**
  • How many RPCs does your service issue?
  • How many annotations do you add?