Innovative R&D by NTT

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

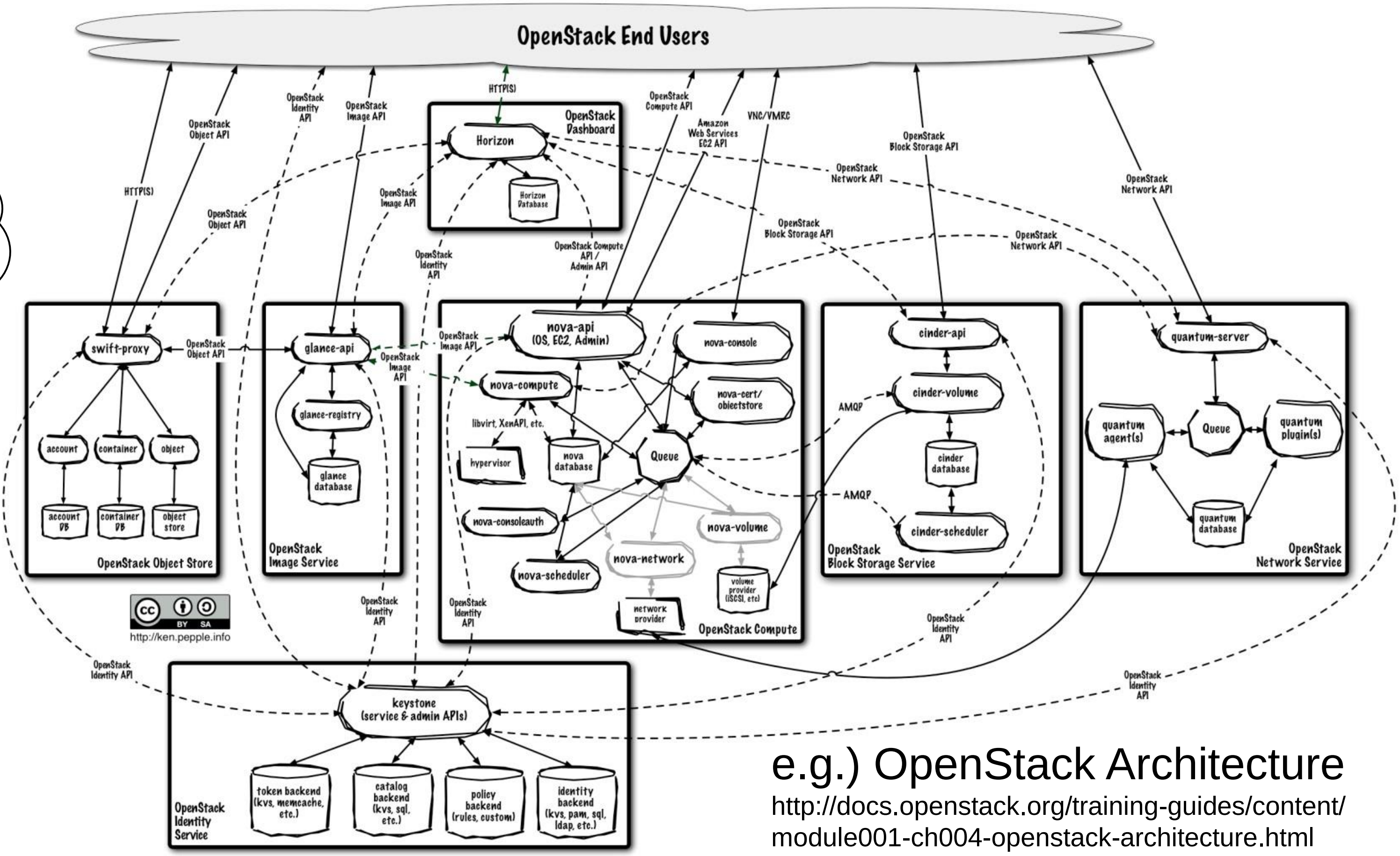
- **working on techniques for improving reliability of distributed systems such as**
  - Sheepdog (scale out storage system)
  - OpenStack Swift (object storage system)

Agenda

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

Background

- **Finding performance bottlenecks in modern large-scale distributed systems is difficult**



e.g.) OpenStack Architecture

http://docs.openstack.org/training-guides/content/module001-ch004-openstack-architecture.html

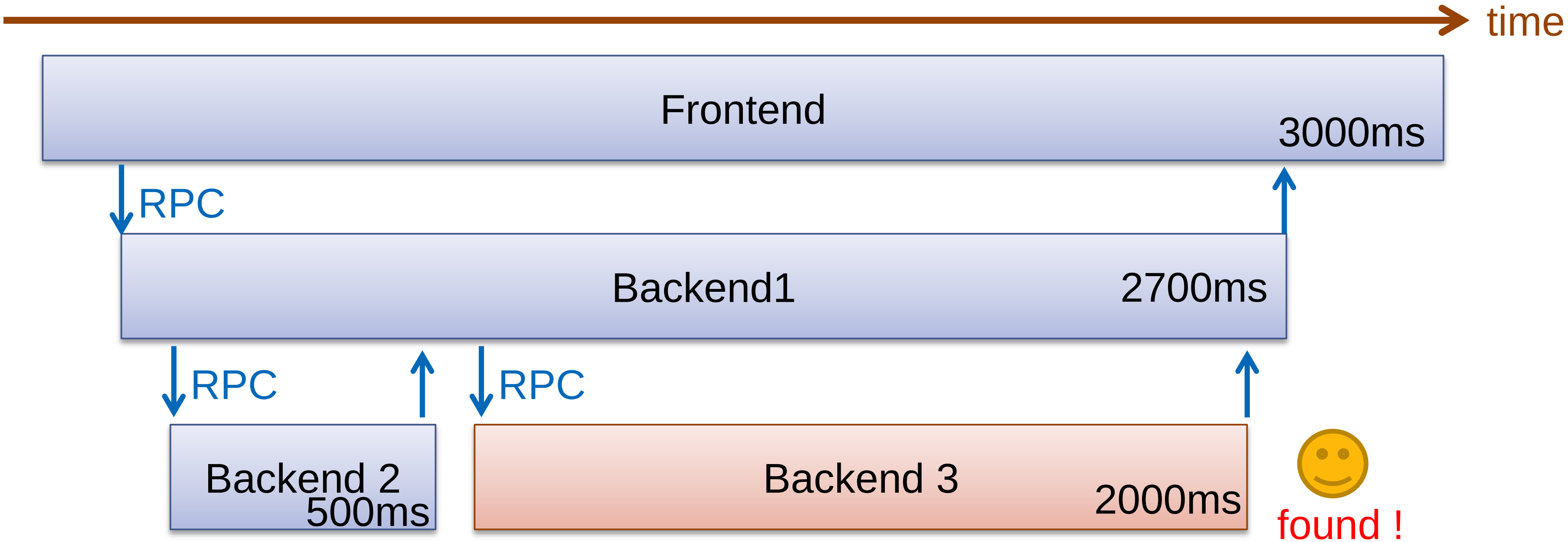
How should we find bottlenecks?

- **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** Today's topic
  - 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

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

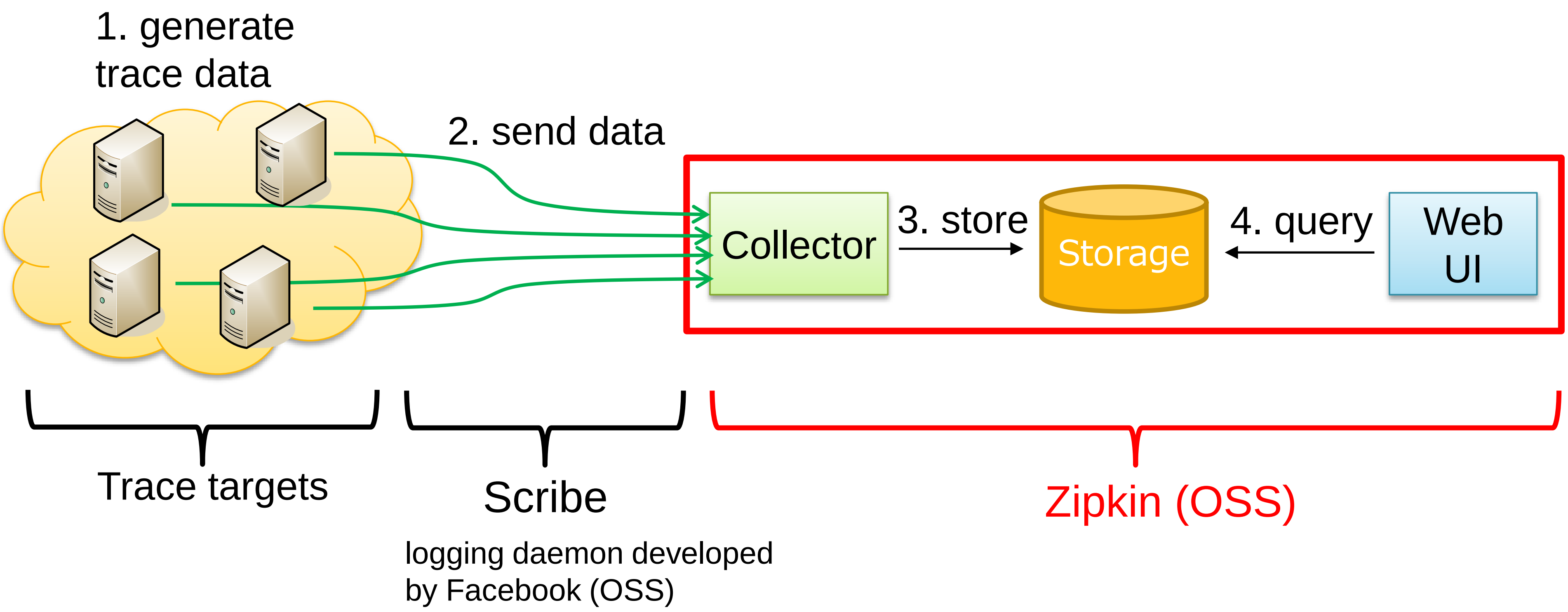
Approaches of distributed tracing

<table>
<tr><td>Black-box based approach</td><td colspan="2">Project5 [1], WAP5 [2]
✓ higher degree of app-level transparency
x some amount of imprecision and possibly larger overheads</td></tr>
<tr><td rowspan="4">Explicit annotation-based approach</td><td colspan="2">✓ deeper understanding of process flow
x need for trace targets to be modified</td></tr>
<tr><td>X-Trace [3]</td><td>comprehensive modifications (client, server, NW devices)</td></tr>
<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>
</table>

[1] Aguilera et al. SOSP ’03
[2] http://googleblog.blogspot.com/2008/04/developersstart-your-engines.html
[3] Fonseca et al. NSDI ’07
[4] http://research.google.com/pubs/pub36356.html
[5] https://github.com/twitter/zipkin

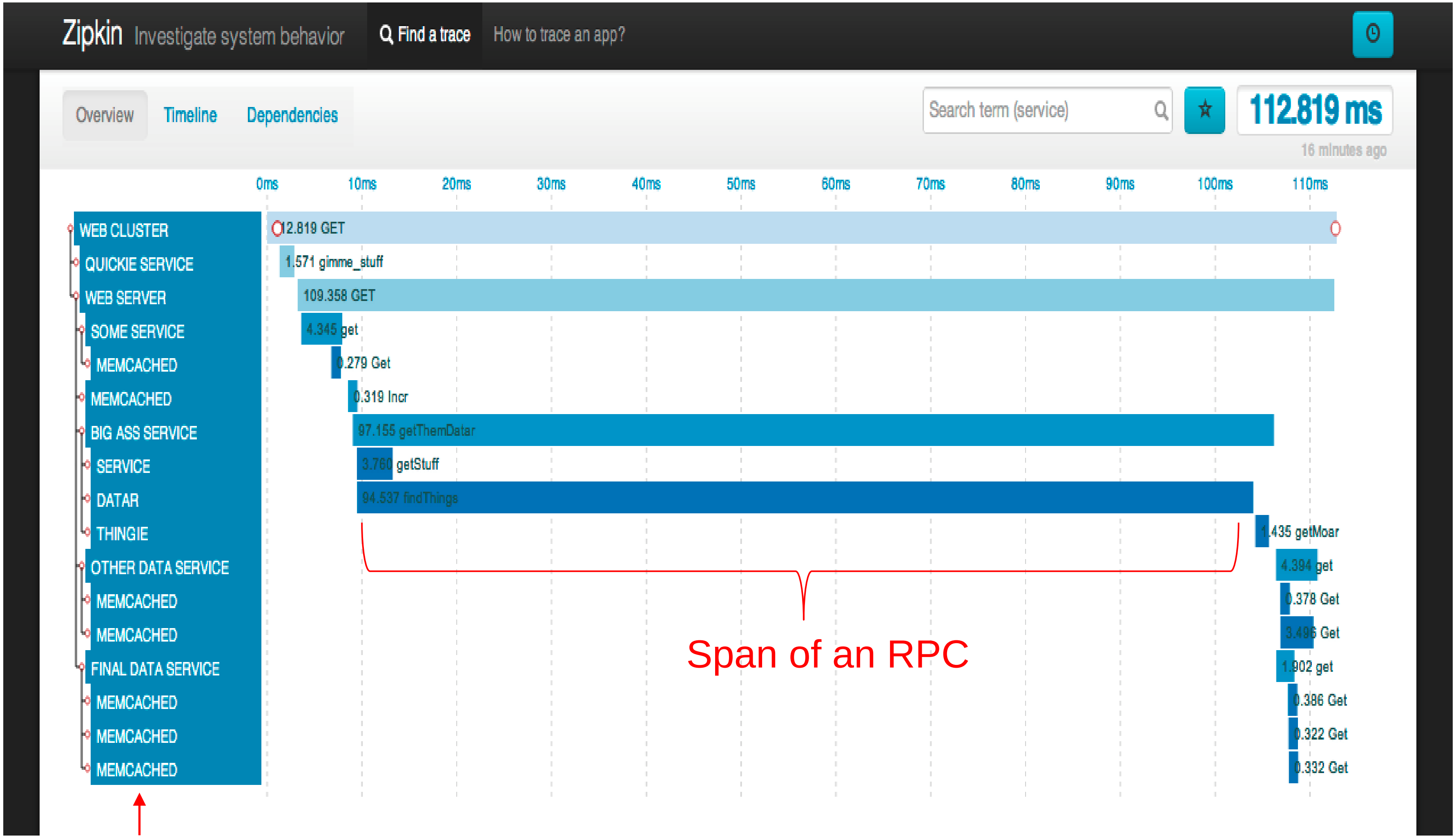
What's Zipkin ?

- **Zipkin is a distributed tracing framework which helps us collect and visualize trace data**



Architecture of Zipkin tracing

What’s Zipkin ?



Span of an RPC

Services

Web UI of Zipkin

Trace data for Zipkin

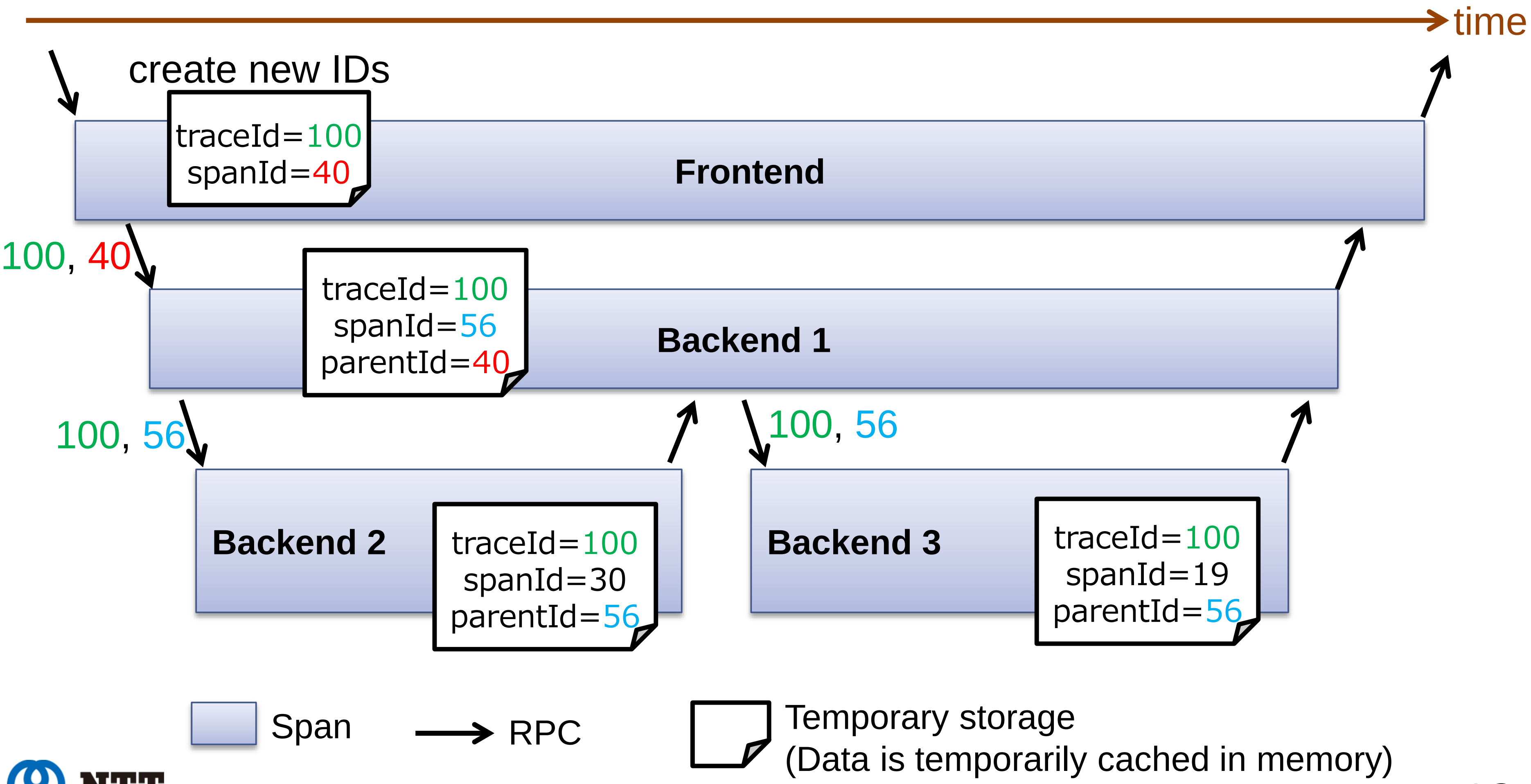
- **RPC timing info of every task**
  - Timestamp of when a service sends a request or receives a response

- **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

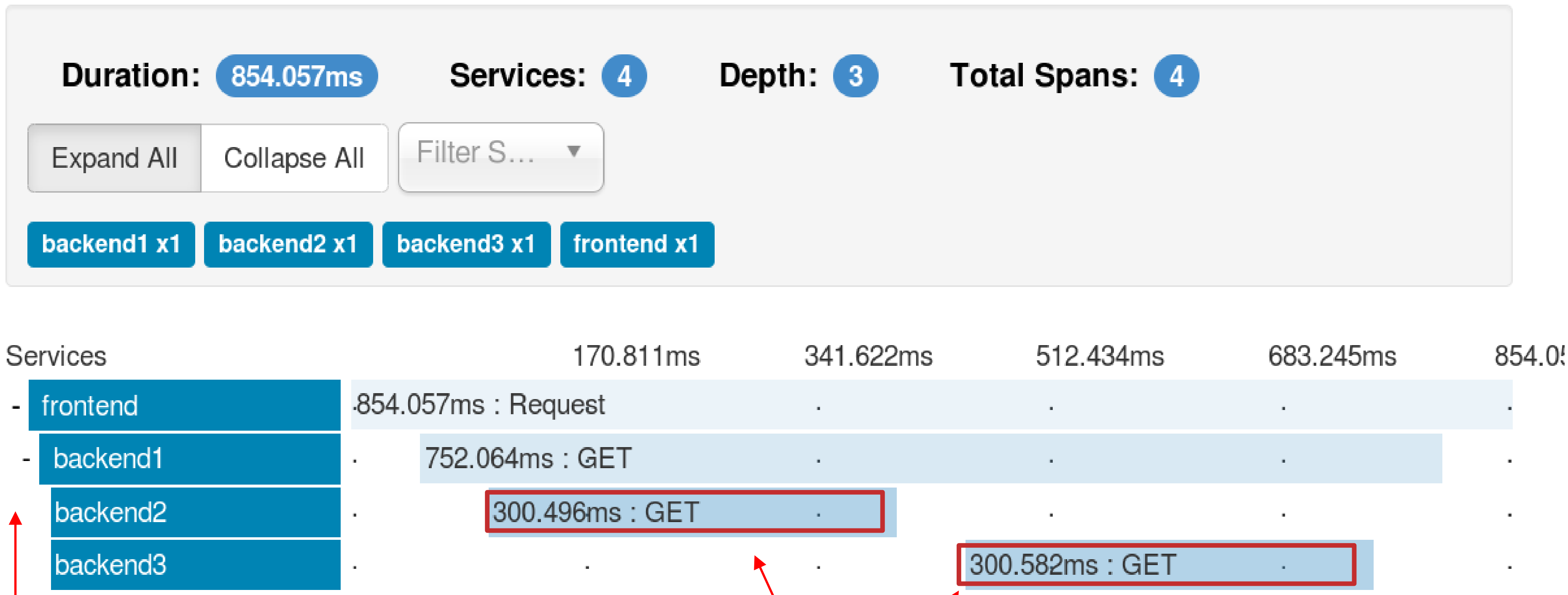
Note: Zipkin does NOT require high-precision timestamp since pairs of spanId and parentId give causal relationships among RPCs

Example: propagation of IDs

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



Web UI of Zipkin



Duration: 854.057ms **Services:** 4 **Depth:** 3 **Total Spans:** 4

backend1 x1 backend2 x1 backend3 x1 frontend x1

Services	170.811ms	341.622ms	512.434ms	683.245ms	854.0
- frontend	854.057ms : Request				
- backend1	752.064ms : GET				
backend2	300.496ms : GET				
backend3			300.582ms : GET		

Levels of nesting represent hierarchical relationships among RPCs

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"

[1] https://github.com/twitter/finagle/tree/master/finagle-zipkin
[2] https://github.com/racker/tryfer
[3] https://github.com/prezi/django-zipkin

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…

[1] http://eventlet.net/

Tracing WSGI applications using Eventlet

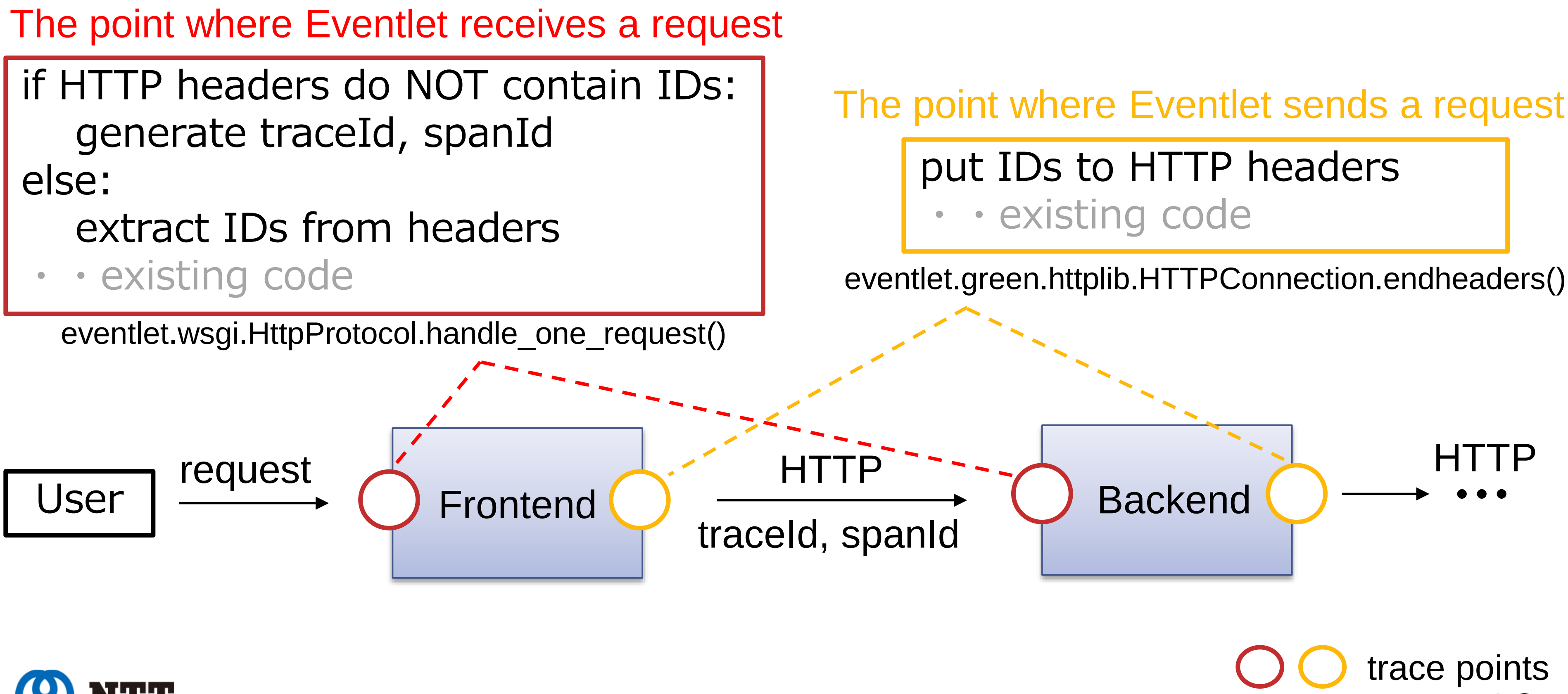
- **We implemented trace feature to Eventlet**
  - Scope
    - Eventlet/WSGI applications which use HTTP for internal communications
    - OpenStack Swift is an example

  - 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
```

eventlet.wsgi.HttpProtocol.handle_one_request()

The point where Eventlet sends a request

```
put IDs to HTTP headers
・・existing code
```

eventlet.green.httplib.HTTPConnection.endheaders()

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)

```
from eventlet.green.httplib 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

module which we added

```
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)

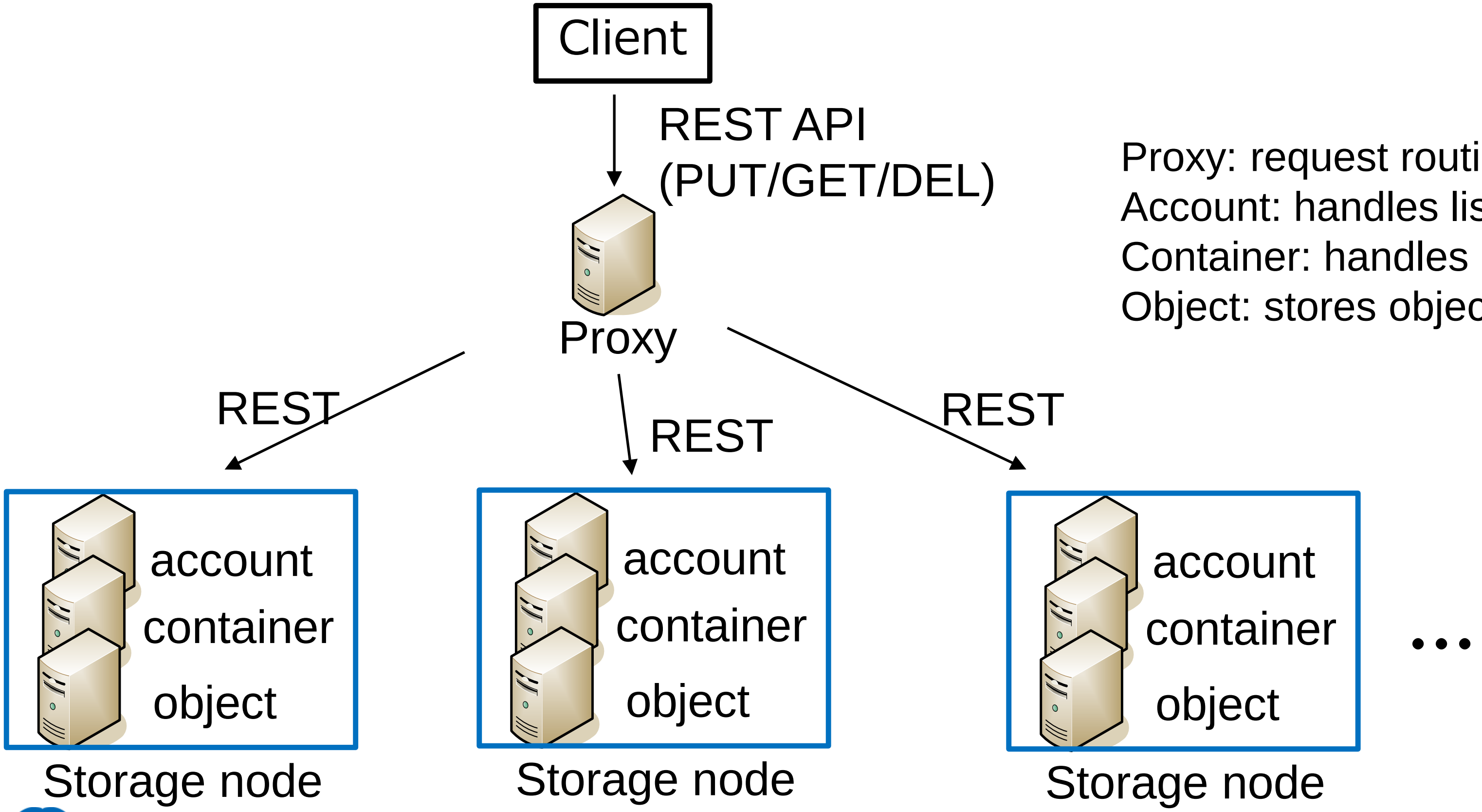
[1] https://lists.secondlife.com/pipermail/eventletdev/2015-February/001205.html
[2] https://github.com/eventlet/eventlet/pull/218

Agenda

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

What's Swift?

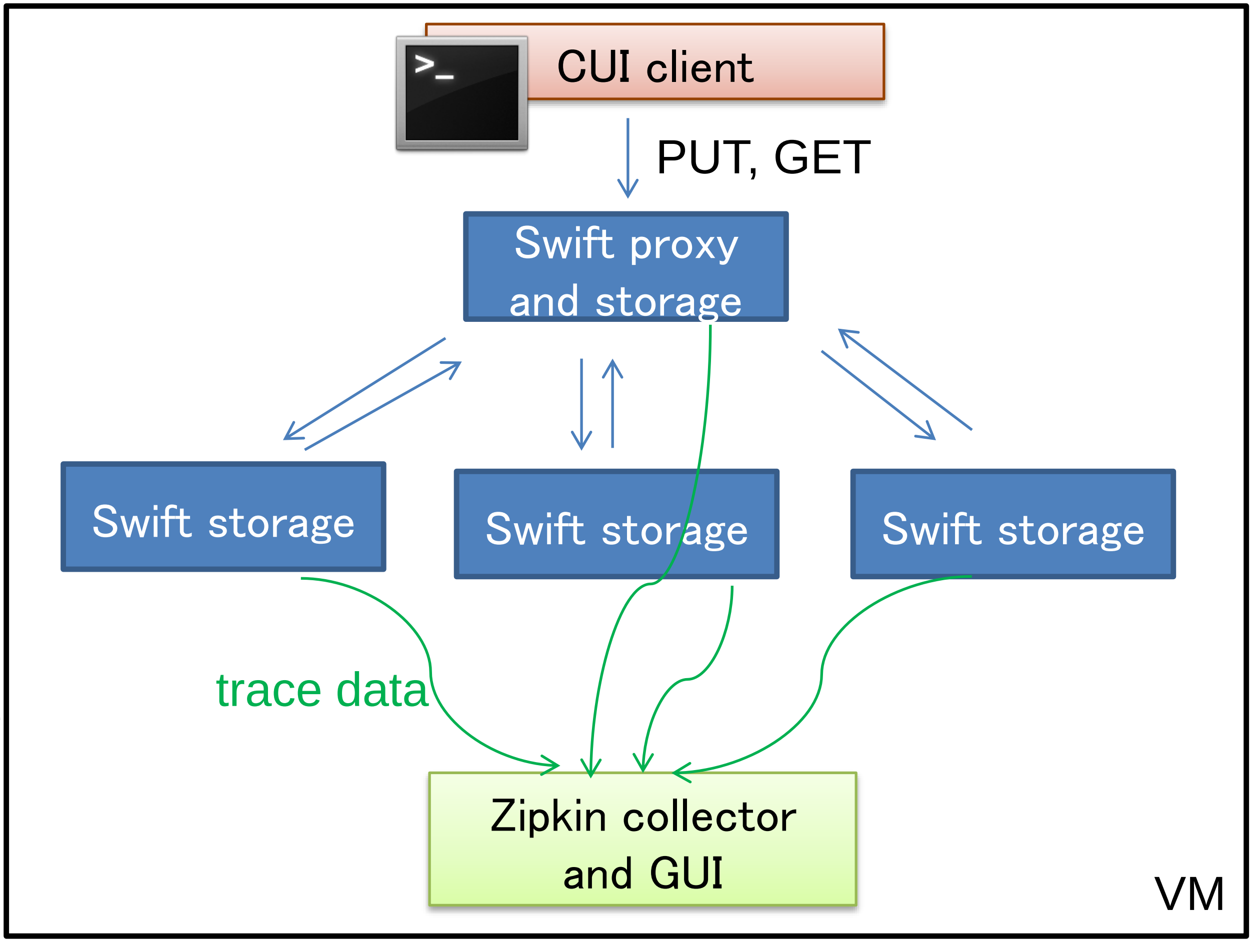
- **A distributed object storage system**
  - implemented as Eventlet/WSGI application
  - uses HTTP for internal communications



Proxy: request routing
Account: handles listing of containers
Container: handles listing of objects
Object: stores objects (has 3 replicas)

Demo

- **Tracing Swift with patched Eventlet**



CUI client

PUT, GET

Swift proxy and storage

Swift storage

Swift storage

Swift storage

trace data

Zipkin collector and GUI

VM

VM on my laptop emulates a four node Swift cluster

Agenda

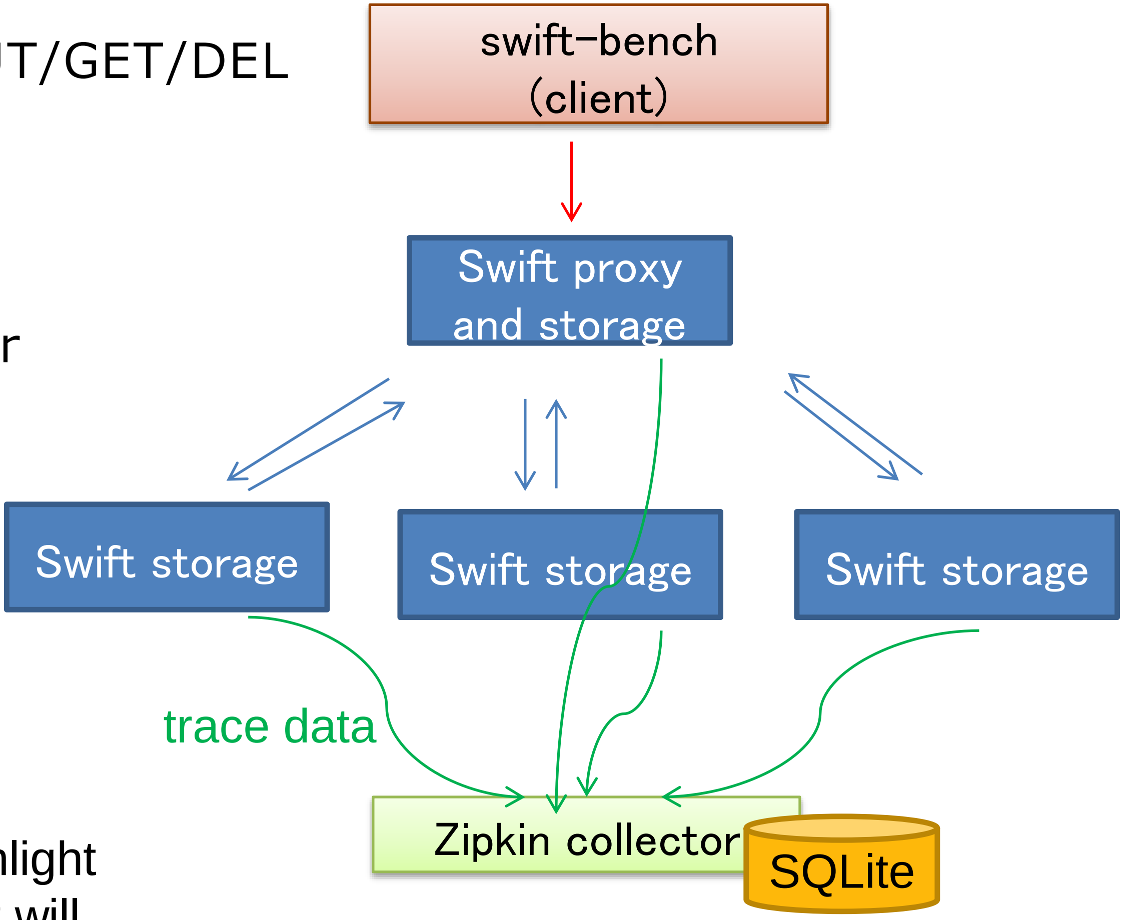
1. Background
2. Introduction to distributed tracing
3. Trace feature enhancement to Eventlet/WSGI
4. Demo with OpenStack Swift
5. **Evaluation**

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

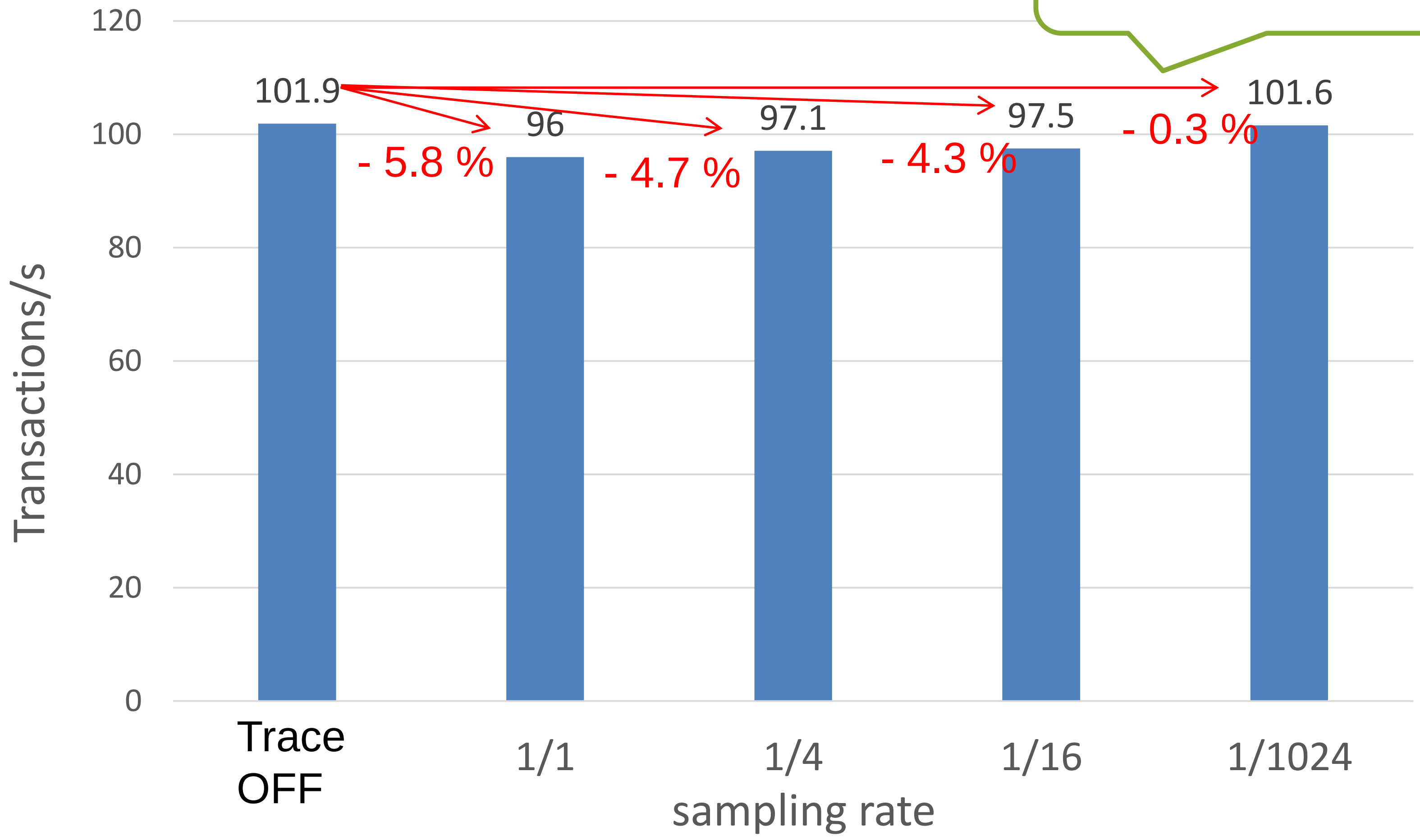


* Setting small object size will highlight the overhead since each request will be lightweight

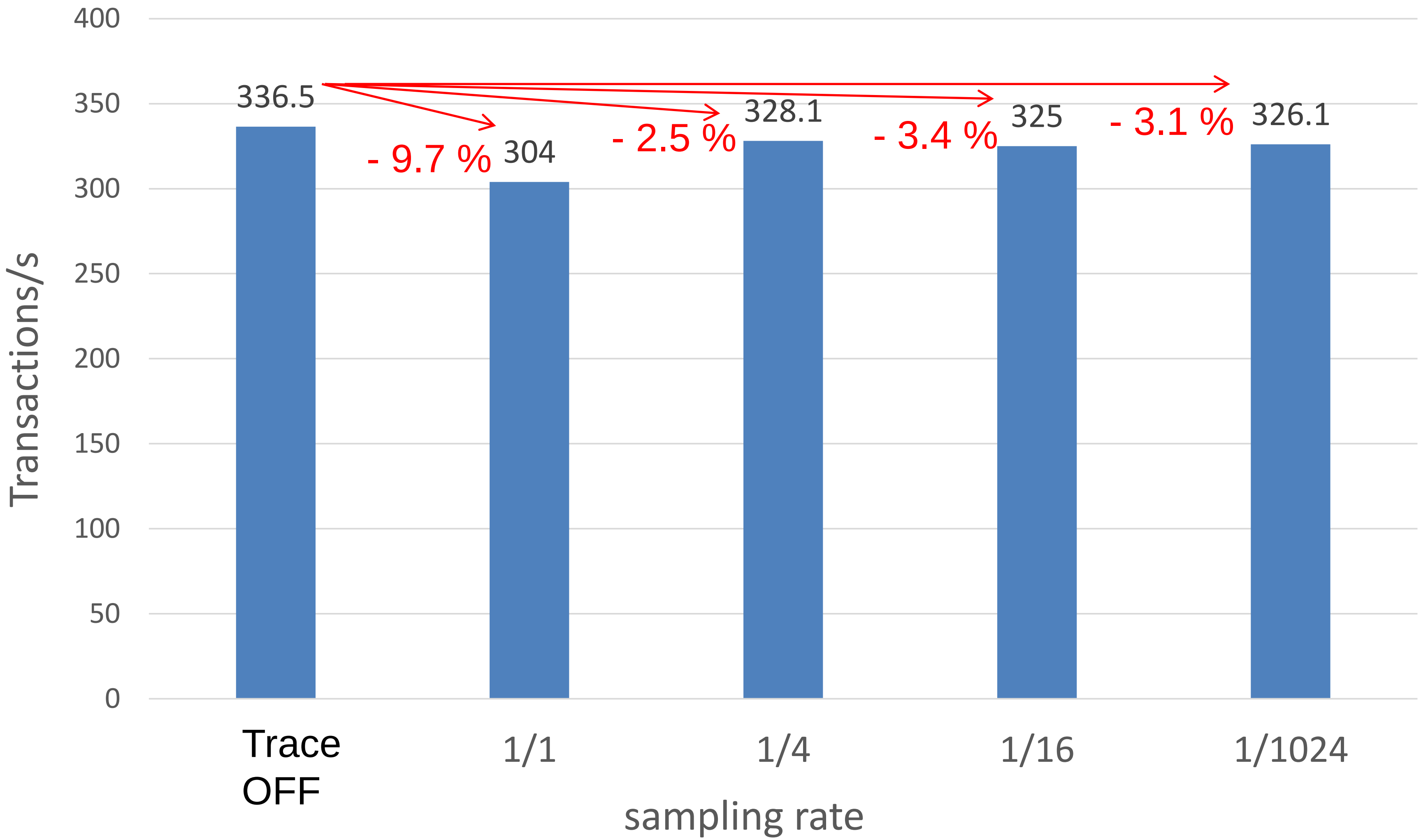
Each component ran on separated physical machine

Impact on Swift throughput (PUT)

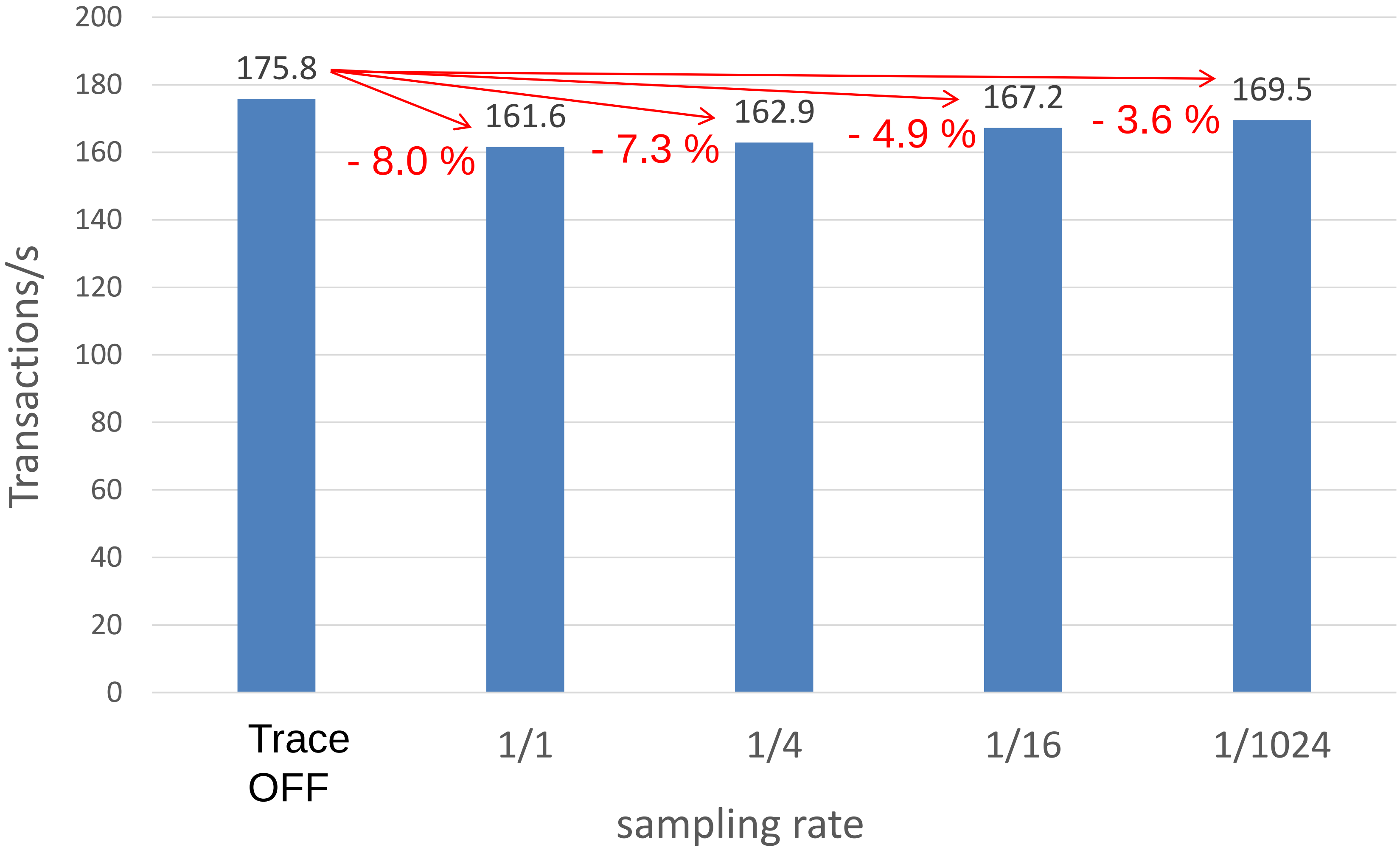
Lowering sampling rate reduces overhead



Impact on Swift throughput (GET)



Impact on Swift throughput **(DEL)**



Impact on resource usage of Swift cluster

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

* 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"*

http://research.google.com/pubs/archive/36356.pdf

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 ?

APPENDIX

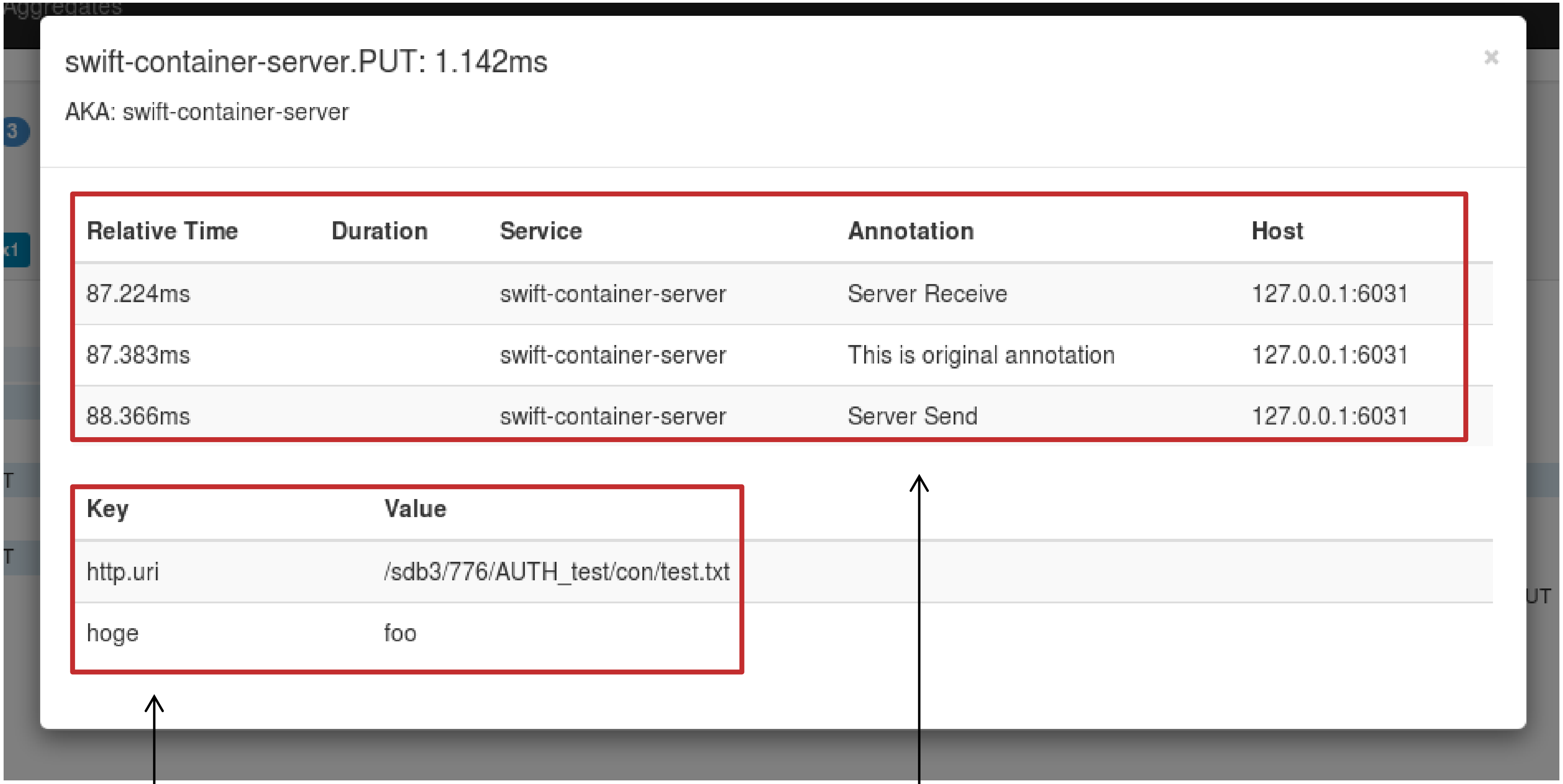
Out patch: other option 1

- **Annotation API**
  - Add your own additional info for deeper understanding
  - from anywhere in your code

```
from eventlet.zipkin import api

api.put_annotation(‘Your own message')
api.put_key_value('key', 'value')
```

Out patch: other option 1



swift-container-server.PUT: 1.142ms

AKA: swift-container-server

Relative Time	Duration	Service	Annotation	Host
87.224ms		swift-container-server	Server Receive	127.0.0.1:6031
87.383ms		swift-container-server	This is original annotation	127.0.0.1:6031
88.366ms		swift-container-server	Server Send	127.0.0.1:6031

Key	Value
http.uri	/sdb3/776/AUTH_test/con/test.txt
hoge	foo

api.put_key_value()

Key-value has no time component

api.put_annotation()

Annotation is recorded with timestamp

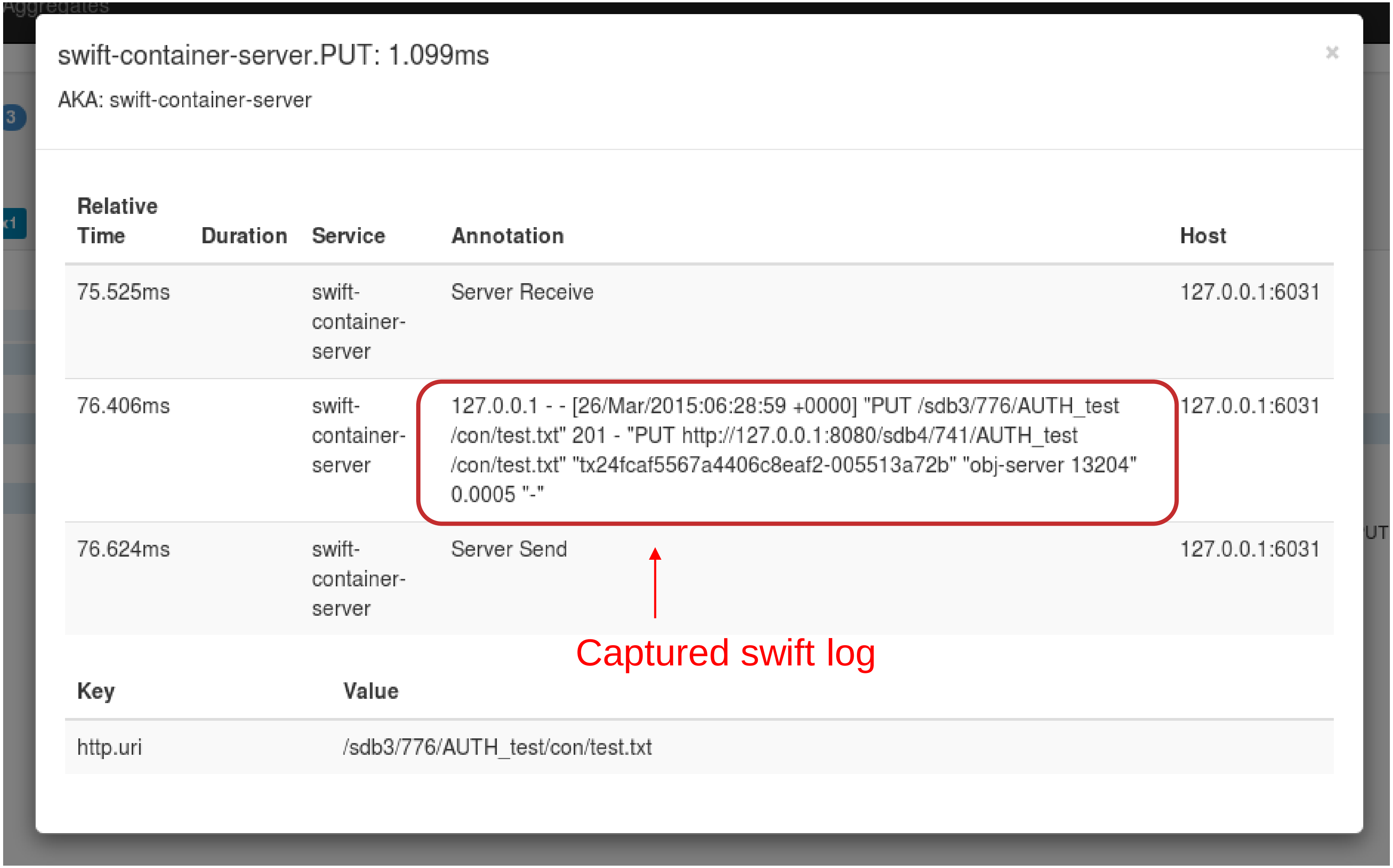
Out patch: other option 2

- **Application Log Tracing**
  - Add application log as annotations for deeper understanding

```
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



swift-container-server.PUT: 1.099ms

AKA: swift-container-server

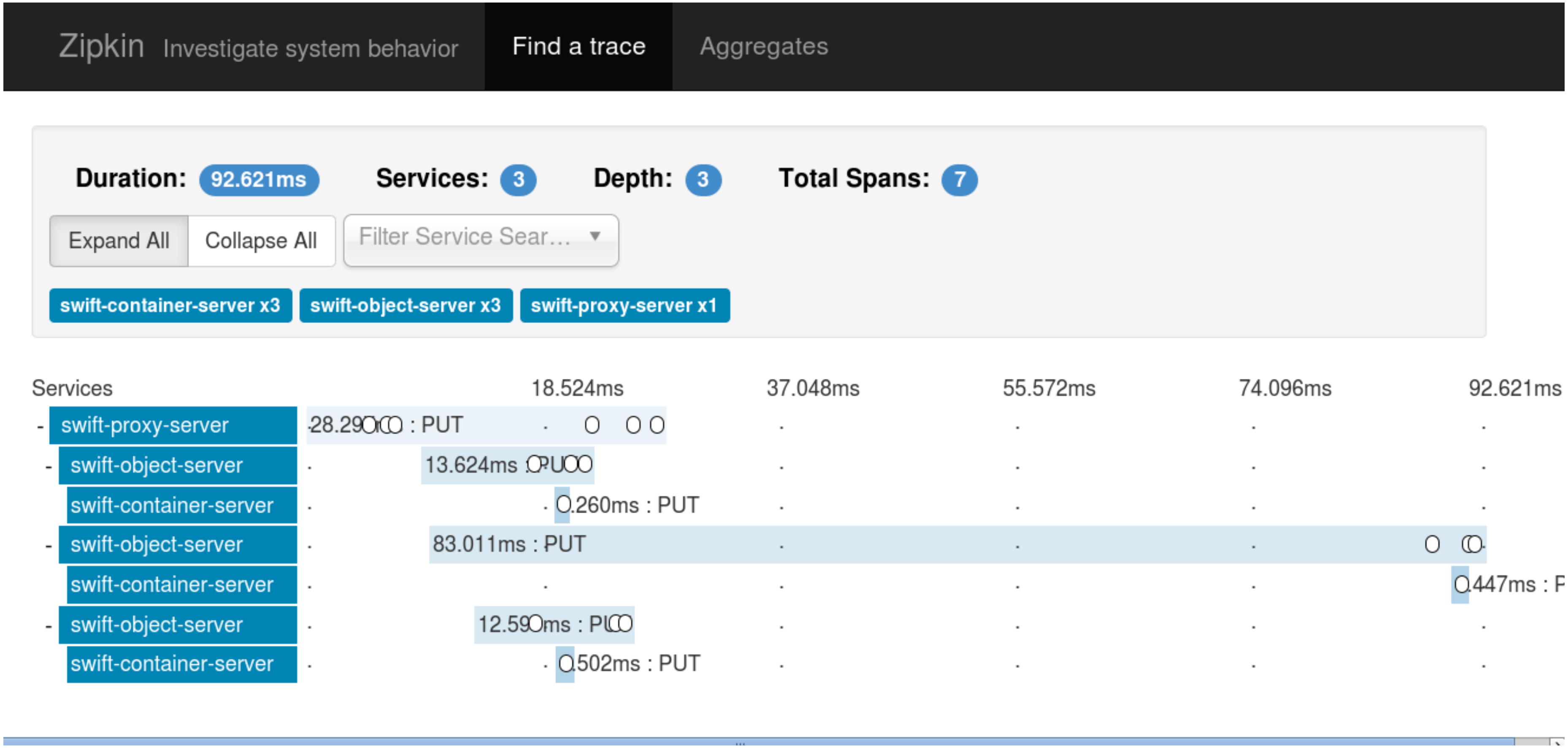
Relative Time	Duration	Service	Annotation	Host
75.525ms		swift-container-server	Server Receive	127.0.0.1:6031
76.406ms		swift-container-server	127.0.0.1 - - [26/Mar/2015:06:28:59 +0000] "PUT /sdb3/776/AUTH_test /con/test.txt" 201 - "PUT http://127.0.0.1:8080/sdb4/741/AUTH_test /con/test.txt" "tx24fcaf5567a4406c8eaf2-005513a72b" "obj-server 13204" 0.0005 "-"	127.0.0.1:6031
76.624ms		swift-container-server	Server Send	127.0.0.1:6031

Captured swift log

Key	Value
http.uri	/sdb3/776/AUTH_test/con/test.txt

DEMO: screen shot

Trace Swift PUT request



Duration: 92.621ms **Services:** 3 **Depth:** 3 **Total Spans:** 7

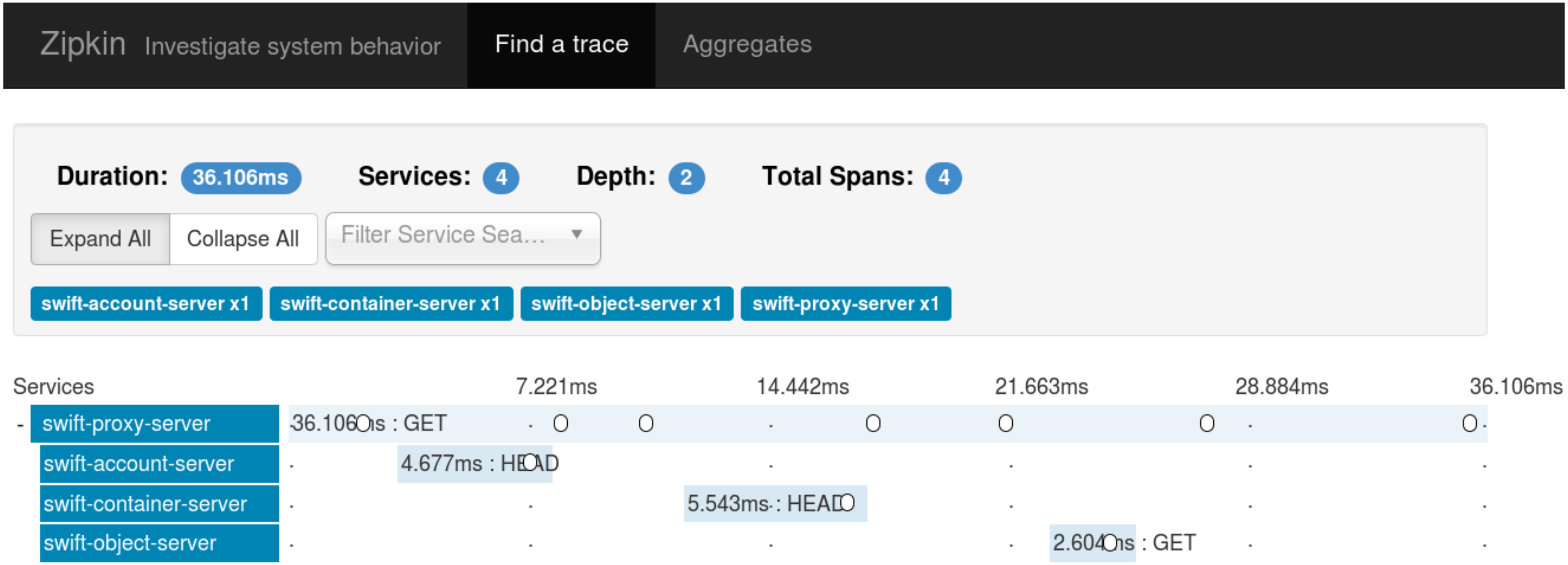
swift-container-server x3 swift-object-server x3 swift-proxy-server x1

Services 18.524ms 37.048ms 55.572ms 74.096ms 92.621ms

- swift-proxy-server 28.29[illegible] : PUT
- swift-object-server 13.624ms : PUT
  - swift-container-server 0.260ms : PUT
- swift-object-server 83.011ms : PUT
  - swift-container-server 0.447ms : P
- swift-object-server 12.59[illegible]ms : PUT
  - swift-container-server 0.502ms : PUT

DEMO: screen shot

Trace Swift GET request



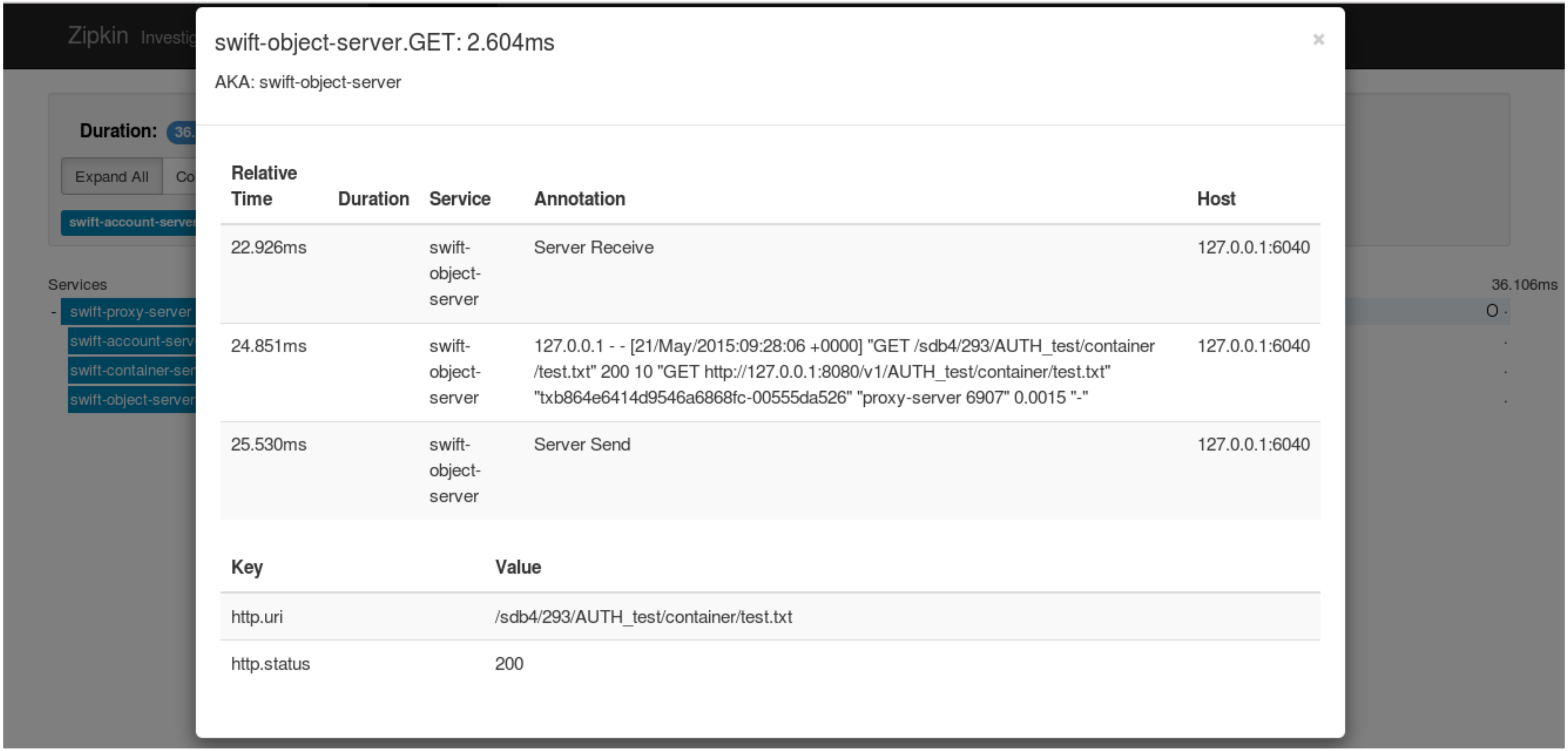
Duration: 36.106ms **Services:** 4 **Depth:** 2 **Total Spans:** 4

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

Services	7.221ms	14.442ms	21.663ms	28.884ms	36.106ms
- swift-proxy-server	36.106ms : GET				
swift-account-server	4.677ms : HEAD				
swift-container-server		5.543ms : HEAD			
swift-object-server				2.604ms : GET	

DEMO: screen shot

Detailed information view



swift-object-server.GET: 2.604ms

AKA: swift-object-server

Relative Time	Duration	Service	Annotation	Host
22.926ms		swift-object-server	Server Receive	127.0.0.1:6040
24.851ms		swift-object-server	127.0.0.1 - - [21/May/2015:09:28:06 +0000] "GET /sdb4/293/AUTH_test/container/test.txt" 200 10 "GET http://127.0.0.1:8080/v1/AUTH_test/container/test.txt" "txb864e6414d9546a6868fc-00555da526" "proxy-server 6907" 0.0015 "-"	127.0.0.1:6040
25.530ms		swift-object-server	Server Send	127.0.0.1:6040

Key	Value
http.uri	/sdb4/293/AUTH_test/container/test.txt
http.status	200

Evaluation: Software version

Swift	2.0.0
Swift-bench	1.0
Eventlet	0.17.1
Fluentd	0.10.61
Zipkin	1.1.0

Evaluation: swift-bench.conf

```
[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

	1 PUT	1 GET	1 DEL
Size of trace data (Bytes)	4096	1024	4096

* 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 ?