Dissecting Open Source Cloud Evolution: An OpenStack Case Study

Salman A. Baset, Chunqiang Tang, Byung Chul Tak, Long Wang
IBM T.J. Watson Research Center, Yorktown Heights, NY, USA

Abstract

Open source cloud platforms are playing an increasingly significant role in cloud computing. These systems have been undergoing rapid development cycles. As an example, OpenStack has grown approximately 10 times in code size since its inception two and a half years ago. Confronting such fast-pace changes, cloud providers are challenged to understand OpenStack's up-to-date behaviors and adapt and optimize their provisioned services and configurations to the platform changes quickly. In this work, we use a black-box technique for conducting a deep analysis of four versions of OpenStack. This is the first study in the literature that tracks the evolution of a popular open source cloud platform. Our analysis results reveal important trends of SQL queries in OpenStack, help identify precise points for targeted error injection, and point out potential ways to improve performance (e.g. by changing authentication to PKI). The OpenStack case study in this work effectively demonstrates that our automated black-box methodology aids quick understanding of platform evolution and is critical for effective and rapid consumption of an open-source cloud platform.

1 Introduction

Open source cloud management systems are being developed at a rapid pace. In one such infrastructure-as-a-cloud (IaaS) system, namely, OpenStack [8], a new version is released every six months and incorporates numerous features and bug fixes from community developers. As shown in Table 1, Grizzly, the latest version of OpenStack, which was released in April 2013 comprises of approximately 330 K lines of code. This is a ten fold increase in term of lines of code from the first release of OpenStack in October 2010, and an order of magnitude increase from the previous release Folsom.

Cloud providers looking to adopt latest changes from the source trunk or a release of a rapidly evolving cloud platforms such as OpenStack are faced with a unique dilemma. On one hand, they want to adopt features and bug fixes as soon as they become available in the source trunk. On the other hand, they are wary of the lack of understanding of component interaction, and how the upgrade will impact the system's stability under normal conditions as well as failure scenarios. The plethora of options available for configuring an open source cloud system further complicates the dilemma of a cloud provider. Without effectively understanding how OpenStack evolves from one version to the next and across versions, and the potential impact of configuration options and failures, a cloud provider may not be able to quickly consume the rapidly evolving system.

The distributed nature of cloud management systems complicates their understanding and analysis. These systems consist of many distributed components and leverage many existing tools, which sometimes interact with each other in non-intuitive ways. Currently, the choice of readily available tools for tracing and analyzing heterogeneous distributed systems in a black-box manner is limited. Mostly, black-box tracing tools offer little help when it comes to system diagnosis based on precise tracing of messages across distributed nodes since they are mostly based on statistical techniques [4, 5]. Instrumentation-based approaches [6,7] are also not easy to use because source code availability and comprehension is required.

In this paper, we present a deep analysis of how OpenStack has evolved over the years. By significantly enhancing vPath [10], we analyze complete message interaction among all OpenStack components for logical operations such as creating or deleting a virtual machine under multiple configuration options. We perform this analysis without modifying any source code in OpenStack, and analyze four releases. Using our analysis, we precisely reason about the data flow, understand the impact of various configuration options, and perform what-if analysis. As part of ongoing work, we are building a

Release	Time	Nova	Glance	Keyst.	Cinder	Quant.	Swift	Total
Austin	Oct'10	17,288					12,979	30,627
Bextar	Feb'11	27,734	3,629				16,014	47,377
Cactus	Apr'11	43,947	4,927				16,665	65,539
Diablo	Sep'11	66,395	9,961	12,451			15,591	91,947
Essex	Apr'12	87,750	15,698	11,555			17,646	149,596
Folsom	Sep'12	103,637	20,271	13,939	20,271	42,118	19,114	230,320
Grizzly	Apr'13	120,968	21,261	20,071	49,797	60,485	23,035	321,081

Table 1: OpenStack evolution in terms of lines of code written in python. An empty box indicates that the component was not part of OpenStack at the time of release. Keyst. and Quant. are abbreviations for Keystone and Quantum, respectively.

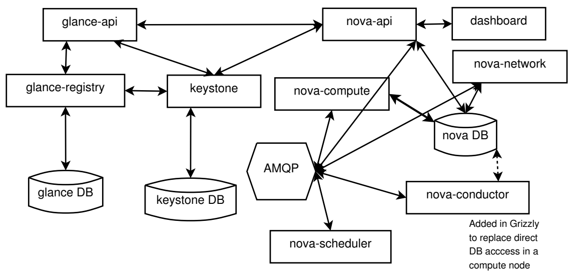

Figure 1: OpenStack Grizzly logical architecture without Quantum. `nova-conductor` service has been added in the Grizzly release. The service runs on a controller and removes direct database operations from `nova-compute` which runs on a compute node.

tool that will leverage the message interaction gathered from our analysis to inject faults at precise points and explore the robustness of OpenStack system. As such, this analysis is not a performance study, but can serve as a useful input to any performance evaluation.

The rest of the paper is organized as follows. In Section 2, we give an overview of OpenStack. In Section 3, we describe challenges and experiences in designing a tool for performing dynamic analysis of OpenStack releases. In Section 4, we discuss our results across OpenStack releases. Section 5, we discuss ongoing work.

2 OpenStack Background

OpenStack is a fully distributed infrastructure-as-a-service software system. Its components communicate via REST, SQL, and AMQP (Advanced Message Queuing Protocol) to perform cloud operations. In this paper, we restrict our analysis to three logical operations in OpenStack, namely, creating, deleting, and listing virtual machines.

OpenStack has six main components: compute (Nova), image repository (Glance), authentication (Keystone), networking-as-a-service (Quantum), volume storage (Cinder), and object storage (Swift). Figure 1 and Figure 2 shows the logical architecture of Open-

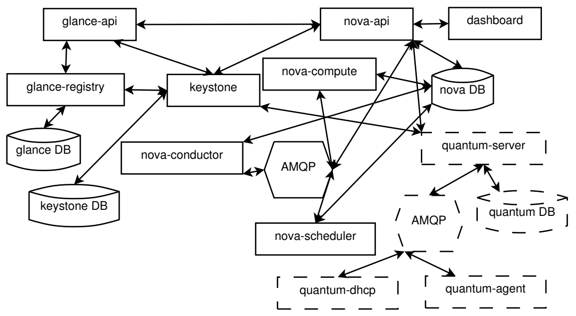

Figure 2: OpenStack Grizzly logical architecture with Quantum.

Stack with and without Quantum, respectively. (Cinder and Swift are not discussed in this paper due to space limitation). Quantum implements a subset of features for software defined networking. In Grizzly release, Quantum has been renamed as OpenStack Networking. Each component comprises of one or more processes (or services). Except for `nova-compute` and `nova-network`, all services shown in Figure 1 run on one or more controller nodes. For OpenStack deployment leveraging Quantum (Figure 2), `nova-compute` and `quantum-agent`[1] run on a compute node. Up to Folsom release, (`nova-compute`) directly connected to nova database and performed database operations. In Grizzly release, the direct database operations have been factored out of the compute node and moved into a service `nova-conductor` which runs on a controller node.

Nova services comprise of an API server (`nova-api`), a scheduler (`nova-scheduler`), and a conductor (`nova-conductor`) that run on the controller, and compute and network workers (`nova-compute`, `nova-network`, `quantum-agent`) that run on each physical server that is part of the OpenStack compute cloud. The nova services interact with each other through a asynchronous message passing server (RabbitMQ [3]) which runs the advanced message queuing protocol (AMQP) [1]. The use of AMQP prevents nova-* components from storing state and facilitate scalability. In addition, nova-* components use a database for storing persistent state. The use of central database and AMQP is potentially a limiting factor for scalability.

Any user or services of OpenStack must be authenticated using a token issued by Keystone (Keystone queries the database during authentication). Moreover, any message from the cloud controller node containing the token must be authenticated and verified with Keystone. Thus, Keystone (and its querying of the database

[1] In OpenStack terminology, all Quantum components except for Quantum server are referred to as agents. We refer `quantum-agent` to the component that runs on the compute node.

during authentication) is a potential bottleneck in OpenStack (as shown in Figure 1 and Figure 2). As a result, Grizzly changes the default authentication mechanism to PKI for reducing database interaction for token verification.

Further details about OpenStack architecture are available at [2].

3 Discovering Message Flow Among Components: Challenges and Experience

Understanding application behavior and its evolution can be achieved in several ways. One approach is to manually read the source code. Another approach is to read the easily-available information such as application logs or system monitoring utilities to infer application behavior. These approaches usually take a significant amount of time and effort, and moreover, capture only bits-and-pieces of application execution. For a complex system comprising of multiple components such as OpenStack, significant inferencing efforts are required to get a clear understanding of individual system execution flows out of the source code and application logs.

In order to avoid these difficulties, we have chosen the vPath [10] approach that traces end-to-end message flows across application components by capturing and analyzing system and library call traces. Particularly, captures of `receive()/send()` and `read()/write()` calls (and their variations) allow for observing causal relationships among components; capturing other calls helps understand what steps are taken to complete user-issued operations. Such dynamic interactions of components are difficult to ascertain otherwise, through methods such as tcpdump.

Path tracing consists of two steps, namely, monitoring and analyzing system and library calls. Source code of the target application (OpenStack in our case) is not required in either step. The target application is handled as a black box.

Monitoring of system and library calls. We use vPath [10] tool to intercept system and library calls. vPath tool makes use of the LD_PRELOAD technique. By setting this environment variable, it can attach a wrapper to most of the glibc functions. Upon intercepting these glibc functions, vPath logs the various parameters in the invocation, e.g. process name/ID, thread ID, port number, parameters passed, timestamp, socket number, and connection ID. Then the original glibc functions are called to serve the application requests. The noted event sequence and parameters are processed in the analysis step to construct the causal event chain.

Analysis of collected data and construction of traces. In vPath, tracking of message causality across application processes is made possible based on the following assumption about the threading behavior: *once a message is received by a thread, that thread is the one to engage in processing of the data; it continues to be the processing thread until it makes another send or receive operation.* This implies that any activities of a certain thread following the *receive* of a message can be considered to belong to the processing of the same user request. This assumption allows establishing a linkage between the initial incoming *receive* with subsequent activities and outgoing *send* messages. The linkage between *send* and *receive* across processes can be formed by pairing the TCP/IP connection's socket tuple information which is unique per connection.

Any scenarios violating this assumption lead to failure of path tracing. Such scenarios are observed in OpenStack messages that are exchanged among nova components of OpenStack using RabbitMQ running AMQP protocol. The presence of message passing queues allows a thread to continue execution, thus violating vPath's original assumption. Without using time stamps or application level knowledge, it is not possible to construct end-to-end path tracing. In this paper, we use time stamp information of queued messages for constructing the path trace. As using time stamp information across distributed nodes requires clock synchronization, we conduct our experiments by placing all OpenStack processes within a single machine. As part of ongoing work, we are instrumenting the analyzer with application specific knowledge (i.e., message identifiers) to correlate messages that are pushed and pulled from AMQP queues.

We use analyzer to (i) gather aggregate statistics about operations (ii) ask what-if questions (iii) analyze a subset of actions from a message flow that meet certain conditions. In the next section, we present a subset of results.

4 Experimental Setup and Results

We ran all components of OpenStack in a single machine, since our focus is on message flow/path and what-if analysis, and not performance. We disabled all OpenStack timers, and started logging of system and library calls on all OpenStack components using vPath. We initially choose the command-utility, `python-novaclient`, to create, delete, and list VMs but quickly realized that it performs a multitude of other operations to support the logical operations. Consequently, we discontinued the use of this utility and instead crafted REST call messages that were initiated using `curl`. Further, we ran CLI of each OpenStack component so that each service is forced to authenticate with Keystone and cache the authenticated token. The authentication for user tokens is in addition to service tokens. In

	Diablo	Essex	Folsom nova-net	Folsom quantum	Grizzly nova-net	Grizzly quantum
SELECT (create)	16 (450)	17 (95)	21 (409)	26 (560)	20 (139)	37 (343)
SELECT (delete)	8 (37)	10 (36)	17 (63)	23 (241)	13 (36)	31 (192)
SELECT (list)	5 (31)	4 (12)	6 (24)	7 (25)	1 (1)	1(1)
INSERT (create)	4 (4)	4 (4)	8 (23)	9 (24)	10 (37)	13 (40)
INSERT (delete)	0 (0)	0 (0)	1 (3)	1 (3)	3 (6)	4 (6)
INSERT (list)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
UPDATE (create)	2 (9)-5	3 (12)-5	7 (60)-11	7 (58)-12	8 (74)-13	8 (70)-16
UPDATE (delete)	4 (6)-4	6 (10)-7	8 (22)-9	8 (25)-9	10 (31)-11	10(26)-12
UPDATE (list)	0 (0)-0	0 (0)-0	0 (0)-0	0 (0)-0	0 (0)-0	0(0)-0
DELETE (create)						
DELETE (delete)	1 (1)	1 (1)	1 (1)	1 (1)	1 (1)	1 (1)
DELETE (list)						
Total tables	53	63	67	83	136	160
glance	4	4	5	5	6	6
keystone	9	10	10	10	19	19
nova	39	49	52	52	111	111
quantum	n/a	n/a	n/a	16	n/a	24

Table 2: SQL statistics for create, delete, list VMs. The number before parenthesis is the total number of tables touched, and the number within parenthesis is the actual number of queries. The number after '-' in the UPDATE rows shows the total number of tables touched across INSERT and UPDATE queries. For Grizzly, there are 55 shadow tables for nova which archive operation information.

	Diablo	Essex	Folsom nova-net	Folsom quantum	Grizzly nova-net	Grizzly quantum
keystone	422 30 GET	54 9 GET	358 17 GET	484 23 GET	82 3 GET	243 6 GET 2 POST
nova-api	4 1 POST	11 1 POST	11 1 POST	9 1 POST	10 1 POST	10 1 POST
nova-compute	4	5	13	14	0	0
nova-conductor	n/a	n/a	n/a	n/a	15	16
nova-network	13	19	17	n/a	20	n/a
nova-scheduler	1	2	1	1	4	4
glance-api	0 2 GET 5 HEAD	0 4 HEAD	0 8 HEAD	0 8 HEAD	0 8 HEAD	0 8 HEAD
glance-registry	6 7 GET	4 4 GET	8 8 GET	8 8 GET	8 8 GET	8 8 GET
quantum-server	n/a	n/a	n/a	44 5 GET 1 POST	n/a	62 9 GET 1 POST

Table 3: Creating a VM: SELECT queries issued by components and HTTP requests received by different components.

steady-state, OpenStack services are expected to cache tokens, baring any bugs or configuration options.

For each experiment, we collected logs generated by vPath, analyzed them offline to construct message path, and performed detailed analysis. For Folsom and Grizzly release, we analyzed OpenStack operation with and without Quantum.

4.1 Database and REST Call Analysis

Table 2 shows the total number of distinct SQL queries for VM creation, VM deletion, and listing all VMs. There are several interesting observations to be made from this table that shed light on the evolution of OpenStack. First, the number of SQL SELECT queries increases from Essex to Folsom, and then it decreases in the Grizzly release. The decrease in the number of SQL SELECT queries in Grizzly is attributed to significant refactoring of code in Keystone, the authentication component. A large number of SELECT queries does imply a processing overhead, especially, if a SELECT query includes complicated joins. A detailed performance study is needed to evaluate the impact, which is not the focus of this paper. In Diablo, the large number of SELECT queries is related to HTTP GET requests that Keystone receives (30 GET requests in total, see Table 3). In Diablo, Keystone was just integrated with other OpenStack components. The developers did not optimize Keystone interaction with other OpenStack components. The increase in SELECT queries for Grizzly (quantum) is attributed to Quantum service token authentication. We were expecting the Quantum token to be cached (as described earlier), but we did not observe this to be the case over multiple runs. We are investigating the cause of service token verification in Quantum.

Second, the number of INSERT and UPDATE queries continue to grow across releases. For a single VM creation in Grizzly (quantum), 40 INSERT and 70 UPDATE queries are issued that altogether touch 16 tables. Since OpenStack is a distributed system, a request can fail when traversing from one component to the other. If a VM creation or deletion request fails midway, only a subset of INSERT and UPDATE queries would have been issued which will need to be subsequently processed to effectively recover from a failed request. Moreover, any other configuration state will also need to be removed. Our path analysis technique can precisely identify which tables and configuration state were touched by a particular component, which makes it easier for developers to write failure recovery code and for operations personnel to zoom in on a problem. Another implication of these

	Diablo	Essex	Folsom nova-net	Folsom quantum	Grizzly nova-net	Grizzly quantum
SELECT	371	52	292	290	100	140
INSERT	3	3	10	10	22	22
UPDATE	1	2	10	10	8	8

Table 4: Creating a VM: SQL queries before a request is sent to compute. Compare results with Table 2.

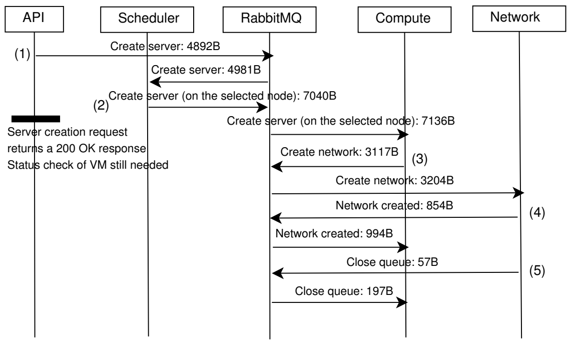

Figure 3: Creating a VM: Message flow among nova components over AMQP up to Folsom (nova-network).

results is for database sizing. Since these results are per logical operation (e.g., create VM), a system architect can easily obtain a bound on the number of newly created records as a function of VM provisioning rate.

Table 4 shows an aggregate of INSERT and UPDATE queries issued by different components before `nova-scheduler` sends a request to `nova-compute` over AMQP to create a VM. From Table 2 and Table 4, it can be easily deduced that bulk of UPDATE operations occur after a request is sent to `nova-compute`. Though not shown in the paper, we discovered from our analysis that bulk of UPDATE queries (44 in total for Grizzly (quantum)) were for updating statistics for the compute node on which VM was being created, while the remaining queries were for instance related updates.

Third, when deprovisioning a VM, only one or two records are deleted from the tables while all other VM-related records are archived. Our analysis indicates that the deleted record corresponds to virtual interface and ports (for Folsom/Grizzly-quantum) of a VM being deprovisioned. In addition, in Folsom and Grizzly release, records are inserted in the 'reservations' table to indicate the release of VM resource. For instance, if a VM is provisioned with 512 MB of memory, a record with -512 MB will be inserted into the 'reservations' table to indicate the release of memory resource. From a logging and audit perspective, such a mechanism is better than a simple decrementing of provisioned resource.

While SQL query count shown in Table 2 could also have been obtained through SQL server logs, the logs do not tell us which component is issuing the queries, and the message flow which triggered these logs. For create VM operation, we want to understand which components are issuing SELECT queries. Table 3 gives a break down of SQL SELECT queries sent by different components. It becomes very clear from looking at this table that token-based authentication in Keystone is responsible for issuing a bulk of SQL SELECT queries. If we configure PKI-based token verification, the number of SELECT queries in Grizzly (nova-network) drops to seven. Our tracing tool allows us to easily understand the impact of configuration options such as the one mentioned above.

Single byte `recv()`. We observed a strange behavior where components receive HTTP responses one byte at a time by issuing a (`recv()`) call for one byte. From studying the source code, we have found that the cause was a python library (webob) used for HTTP message communications. Although, Openstack is not the direct reason for this single byte `recv()`, the use of this library may degrade performance.

4.2 AMQP and aggregate traffic analysis

Figure 3 shows the message flow over AMQP among nova-* components of OpenStack which we were able to deduce using our message path when using `nova-network`. This flow is applicable till Folsom. In Grizzly, the introduction of `nova-conductor` service has increased interaction with AMQP server by an order of magnitude. In total, five messages are exchanged among nova-* components over AMQP. After scheduler sends the VM creation request to compute node, a successful (200 OK) HTTP response is returned to the caller. As shown in Table 4, our path analysis allows us to precisely determine what happens in terms of database queries before scheduler sends the VM creation request to nova-compute over AMQP.

4.3 Flow Evolution

How does OpenStack flow evolve from one version to the next and how much similarity is there in the "shape" of paths? We believe that analyzing flow evolution can best be performed using an interactive tool (similar to those available for checking source code differences). Confined to text, we offer some insights. First, `nova-api`'s interaction with databases over releases has significantly increased. Up until Essex, only four records were inserted by `nova-api` in total, three of which were inserted before a request was sent to `nova-compute` (Table 4). For Folsom and Grizzly, the number of inserted records for a similar path are 10 and 22, respectively. The increase in these inserted records is due to introduction of a new table 'instance_system_metadata' which contains the bulk of new records (13 for Grizzly).

Second, the code path for interacting with Glance has been relatively stable since Folsom. This is also evident from the limited change in Glance code base from Folsom to Grizzly.

Third, while the introduction of `nova-conductor` service has removed direct database interaction from a compute node, the interaction between `nova-compute` and database through `nova-conductor` has actually increased which manifests itself in SQL UPDATE operations. As OpenStack code evolves, the developers are focusing on storing intermediate state from compute node on the persistent storage, which results into an increase in UPDATE queries.

5 Discussion and Ongoing work

Tracing OpenStack message flow for logical operations is the first step in understanding evolution of OpenStack. The tracing, by design, is performed before a system is actually put into production, and is meant to uncover any changes in overall code. We are currently enhancing the tool for automatically correlating log writes with the message flow and for systematic error injection to analyze OpenStack robustness against failures. The idea is to automatically insert an error (e.g., modifying the return value of a system call) at potentially every point in the trace message flow. As part of systematic error injection, the errors inserted, message flow resulting from the error, and the logs will be collected in a central repository which will be indexable. The goal is to create this repository in a small time-frame (e.g., one day). System administrator can potentially consult this repository when searching for a problem diagnosis. Such an automated mechanism for inserting faults can significantly increase the test coverage of a rapidly evolving code base such as OpenStack.

The tracing results alone shed light on the working of the system and allow an architect to perform a rough sizing of the components (e.g., DB disk or AMQP message flow as a function of logical operations such as number of VMs created.). However, tracing is performed prior to putting the system in production without instrumenting any source code. It may also be useful to follow a Dapper [9] and Zipkin [11] like approach for collecting request progress in a distributed system through code instrumentation.

A particular challenge in tracing, which we also identified in earlier sections, is the use of AMQP queues which cause path information to break. As a design principle, the distributed systems making use of a similar queueing mechanism must augment messages with a unique message identifier that traverses different components of the stack to enable offline or online tracing.

6 Conclusion

OpenStack, an open-source cloud management system, is being widely adopted as an IaaS platform. It's rapid development and complexity growth complicates its understanding. This paper presents a deep analysis of how OpenStack has evolved over the past few years. This is the first study in the literature that tracks the evolution of a popular open source cloud platform. We leverage and enhance a path-tracing tool to automate analysis of four versions of OpenStack by capturing and analyzing interactions among distributed components for creating and deleting a virtual machine.

Our analysis reveals important trends of SQL queries, help understand impact of configuration options, analyze subset of messages to answer what-if questions, and identify precise points for error injection. As part of ongoing work, we are leveraging the message flow to develop a framework for error injection. The techniques presented in this paper are useful for understanding other rapidly evolving open source systems for effectively consuming them within a provider.

References

[1] AMQP. `https://en.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol`.

[2] Openstack architecture. `http://docs.openstack.org/grizzly/openstack-compute/admin/content/logical-architecture.html`.

[3] Rabbitmq. `http://www.rabbitmq.com/`.

[4] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP*, pages 74–89, New York, NY, USA, 2003. ACM.

[5] A. Anandkumar, C. Bisdikian, and D. Agrawal. Tracking in a spaghetti bowl: monitoring transactions using footprints. In *SIGMETRICS*, pages 133–144, New York, NY, USA, 2008. ACM.

[6] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *OSDI*, Berkeley, CA, USA, 2004. USENIX Association.

[7] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *DSN*, pages 595–604, Washington, DC, USA, 2002. IEEE Computer Society.

[8] Openstack. `http://www.openstack.org/`.

[9] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical Report dapper-2010-1, Google, April 2010.

[10] B. C. Tak, C. Tang, C. Zhang, S. Govindan, B. Urgaonkar, and R. N. Chang. vpath: precise discovery of request processing paths from black-box observations of thread and network activities. In *USENIX*, pages 19–19, Berkeley, CA, USA, 2009. USENIX Association.

[11] Twitter. Zipkin. `http://twitter.github.io/zipkin/`, 2013. [Online; accessed May 2013].