Tuan Nguyen
Manish Pandey
CS244B

Dprof - Distributed System Profiling and Tracing

Abstract

Performance troubleshooting of distributed systems, as they scale up in size and complexity, is an increasingly critical problem. These systems are constructed from collections of multiple software modules, with complex interaction between different tiers in the system, and are often deployed on hundreds of machines. Debugging the system for performance bottlenecks, capacity planning, and problem diagnosis is difficult as traditional machine-centric monitoring approaches that collect and aggregate time series metrics and system logs do not lend themselves to analysis of distributed operation, where the execution of a single operation can span hundreds of machines. For the class project, we have built dprof, a distributed light-weight system profiling and tracing system that continuously monitors performance of distributed system components and RPC operations, and is capable of tracing system events and detecting system bottlenecks. In this paper, we describe the design and the implementation of the monitoring system. We have run the system on the Dynomite distributed system replicating Redis across multiple nodes to identify system bottlenecks.

1. Introduction

Modern distributed services are large complex systems, built from hundreds to thousands of serves, and often layered on top of other distributed systems. Even simple web applications have a multi-tier architecture with application services, memcache, globally replicated databases, etc. that span multiple machines.

Traditional approaches to monitor and manage these systems have been limited to monitoring individual machines, where individual machines memory/cpu/IO traffic is monitored, but provide little insight into how the system behaves as an aggregate. Consequently, system performance debugging, identification of bottlenecks in the system becomes a challenge for such systems. Monitoring tools such as VMware Vcenter Operations[1], AWS Cloudwatch[2], Openstack Ceilometer[3], all approach the problem by collecting time-series metrics and system logs. In a complex environment with multiple services which are continuously changing, the user or administrator may not even know the services that are invoked for a fulfilling an end-user request. Each service is typically built and managed by different teams, and same machines may have multiple services running on them. Therefore, the user or the cloud administrator is typically unable to determine how system requests propagate through the system and where the bottlenecks lie. In some cases [4], users have written scripts that process system logs to determine the causality of requests and events in the system, but these approaches are brittle and do not scale out to other applications.

What is needed is a system that can tracing how requests flow through the system and gather timing data for all the different services that build up a distributed system. One needs the ability to observing related activities across many different programs and machines, and have the ability to causally relate how one system request is triggered by another.

The objectives for dprof, our distributed monitoring system include the following:

1. Ability to monitor heterogeneous collection of systems - any application, process, machine configuration
2. Application independent - should not require applications to be rewritten
3. Low overhead - monitoring should have minimal overhead on the system being monitored, in terms of latency and throughput.

In this paper, Section 2, covers related work on distributed systems monitoring, and Section 3 describes the RPC tracing approach, and the system design and tradeoffs. Section 4 has some additional implementation details, and sections 5 and 6 contain experimental results and concluding remarks.

2. Related Work

Tradition system infrastructure monitoring tools (VMware Vcenter Operations[1], AWS Cloudwatch[2], Openstack Ceilometer[3]) are *machine centric* [9] . These collect detailed system statistics, and even aggregate the data for average/median/upper or lower bound utilization across virtual machines, hosts, racks and the datacenter, but provide little insight into the distributed service's nodes and dependencies which run across multiple machines, and possibly even multiple datacenters.

Our project is most closely related to the *workflow centric* approaches in Google Dapper [5], Cloudera HTrace[6] or Twitter Zipkin [7]. These approaches perform end-to-end tracing to capture detailed work□ow of causally-related activity of the the components of a distributed system. X-Trace [10] is another workflow-based system that has been successfully used for network monitoring and applications, but requires instrumentation at multiple points in the underlying system. In contrast, our approach just requires instrumentation of the the RPC libraries. Magpie [11] is another system tracing tool, that processes events generated by operating system, and application instrumentation, but infers causal relations from the events themselves. Application performance monitoring tools such as NewRelic APM [8] trace transactions across the software stack, but are limited to monitoring specific web application development frameworks such as Ruby On Rails, or Java Spring/Tomcat. They do not support instrumentation and tracing of general purpose distributed systems.

3. System Tracing and Profiling Approach

Spans and Traces are two fundamental tracing constructs used in Dprof. This terminology closes follows that in Dapper [5].

- Span: The basic unit of work in the system is a RPC, and corresponding to every RPC call we have one span.
- Trace: A trace consists of a set of spans that form a tree-like structure, typically arising from one request.

These are illustrated in Figure 1, where P1 could correspond to a request made at the front-end of a system, which then cascades into RPC calls to P2, P3 and P4. Each of P1, P2, P3 and P4 have a corresponding span. Together, these 4 spans constitute a trace, which is the tree of nested RPCs for performing the task requested at P1, possibly by an end-user at the front-end.

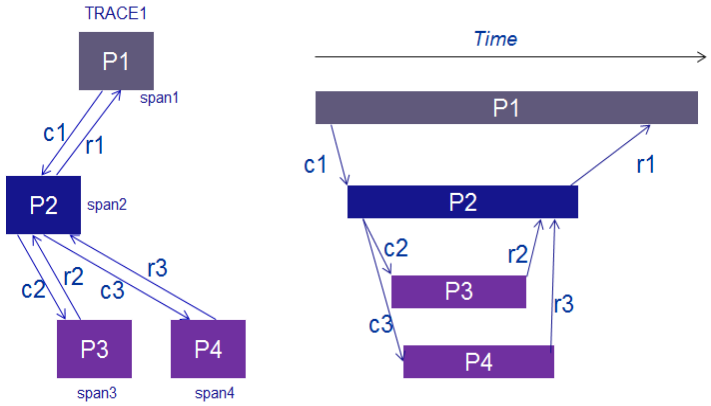

Figure 1: Spans and traces in a end-user system call

So, our approach then consists of instrumenting the RPCs, collecting the traces, transporting them, and aggregating them at a centralized distributed store, and then performing analysis on the collected data for work

done in the traces and the spans. We describe ahead the different components that make up the system that we implemented to perform this task.

3.1 System Overview

The overall system, shown in Figure 2, consists of the following components:

1. Instrumented RPC Library - We modified the RPC libraries in the Dynomite [12,13] system to generate traces for the system. By project presentation date we expect to instrument at least one other library. Log entries containing RPCs are written on to local files in the host.
2. Dprof Daemon - The dprof daemon reads the log files on local hosts and transports the trace information to a distributed data store deployed as a Cassandra [14] ring.
3. Cassandra Data Store - We use a multi-node Cassandra store to hold the trace information. The Trace ID serves as the the key for the store. The data organization is described in section 3.2.3.
4. Query Front-End - Users can type common queries to the system to determine interesting stats on the traces, such as finding the average/min/max latency of a trace, 99% of the slowest spans in a trace, etc.
5. Configuration management - We can use this to configure various system parameters, including trace sampling frequency (to minimize overhead).
6. UI - We expect to have a UI in place by final demo/presentation time for displaying the results of our system.

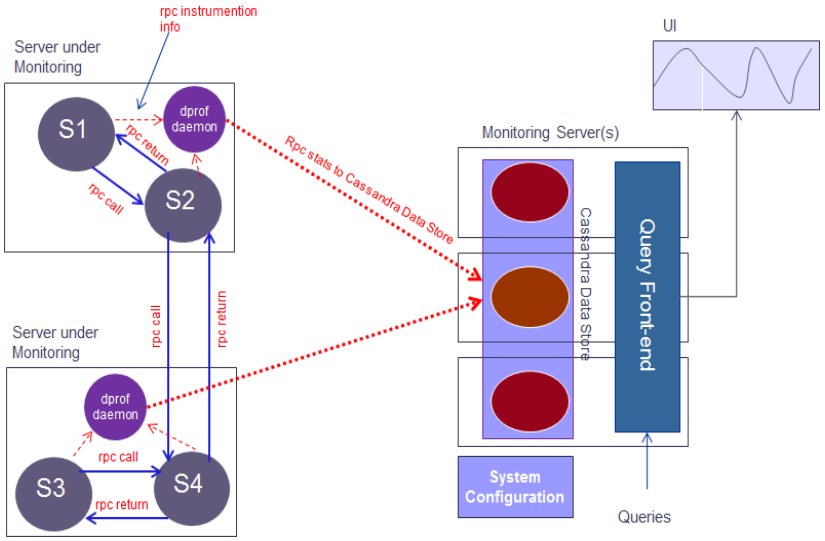

Figure 2: Dprof System Overview

3.2 Details of individual components

We have described below some of these components in additional detail.

3.2.1 Log Generation and Collectors

The instrumented RPC traces are collected and written into log files in individual machines. Spans and traces have unique 64-bit IDs, and in addition to other information such as tags, Key-value annotations, etc. described in section 4. The log files are written in batch mode, with a few hundred entries at a time to minimize disk IO overhead. The dtrace daemon periodically reads the log file, extracts the traces and writes the trace information to the Cassandra data store. System configuration setting for *trace sampling factor* allows the user to control the fraction of the traces to be logged and transported. Sampling, even 0.1% of the traces allows robust monitoring with minimal impact to application latency and throughput [5,9].

3.2.2 Cassandra Distributed Store

The Cassandra distributed store is used to collect and aggregate the traces. The schema includes column families for traces, span names, service names, service span-name index, and duration index. A front-end process receives the requests and populates the different tables to allow fast access by different criteria such as trace ID, time of request, duration etc. Multiple spans, indexed by the trace ID are used to construct the table entries corresponding to a trace.

3.2.5 Time synchronization

We do not depend on time values to create traces from individual spans. The causal ordering between the spans is captured by span ID and parent span ID. Additionally, the start and end times for each span are measure on the same machine.We use NTP to synchronize time across multiple machines. We configure the log aggregator node as the reference server and any additional nodes to get their time from the system configuration node.

3.3 Design choices and tradeoffs

The dprof system is designed for diagnosing steady-state problems which manifest as latencies, or resource usages, and these problems are generally performance related. Problems that affect a large fraction of the system traces, e.g., over 10% of traces, can be efficiently detected and diagnosed in dtrace. Sampling is an important aspect of keeping system overhead low. However, sampling has the downside that identifying and debugging problems that occur in rare work□ows becomes more difficult. Therefore, dprof is not suitable for anomaly detection.

Another aspect of the dtrace system is that it follows the “trigger-preserving” [9, Section 3.1] approach of accounting for time within spans. Trigger preserving essentially means that if the RPC callee returns back control to the caller immediately, while leaving unfinished a large amount of work, e.g., cache eviction, that the callee would need to perform in a subsequent RPC call to it, the actual amount of time spent in the system could be incorrectly attributed to the second call rather than the first RPC call. This can make diagnosis of specific anomalies in dtrace difficult, as the excess time can be incorrectly attributed to the wrong RPC call.

4. Monitoring use case with Dynomite.

Dynomite is a distributed system that enables the construction of peer-to-peer, linearly scalable, clustered systems from single-server datastore solutions such as Redis or Memcached [12,13]. We selected this system to trace and profile using dprof. Below, we describe the Dynomite system and some salient aspects of the profiling.

4.1 Dynomite system and Instrumentation

Figure 3 illustrates the Dynomite system and its logging. The Dynomite cluster consists of multiple data centers having racks joined by different nodes in a ring. Each rack has a data store, which is partitioned across multiple nodes whose unique tokens identify their datasets. Each node has a Dynomite process connecting to the datastore server acting as a proxy and gossiper. Compared with Dynamo [15], Dynomite is the Dynamo layer with the support for pluggable datastore proxy. We instrument Dynomite RPC calling for monitoring the performance for dynomite system In the context of this class, we focus on the intercommunication between dynomite nodes, as well as between dynomite nodes and data store servers. For every message sent and received, the following information is logged. We have deployed Dynomite on a cluster of AWS EC2 instances, with Redis as the underlying datastore.

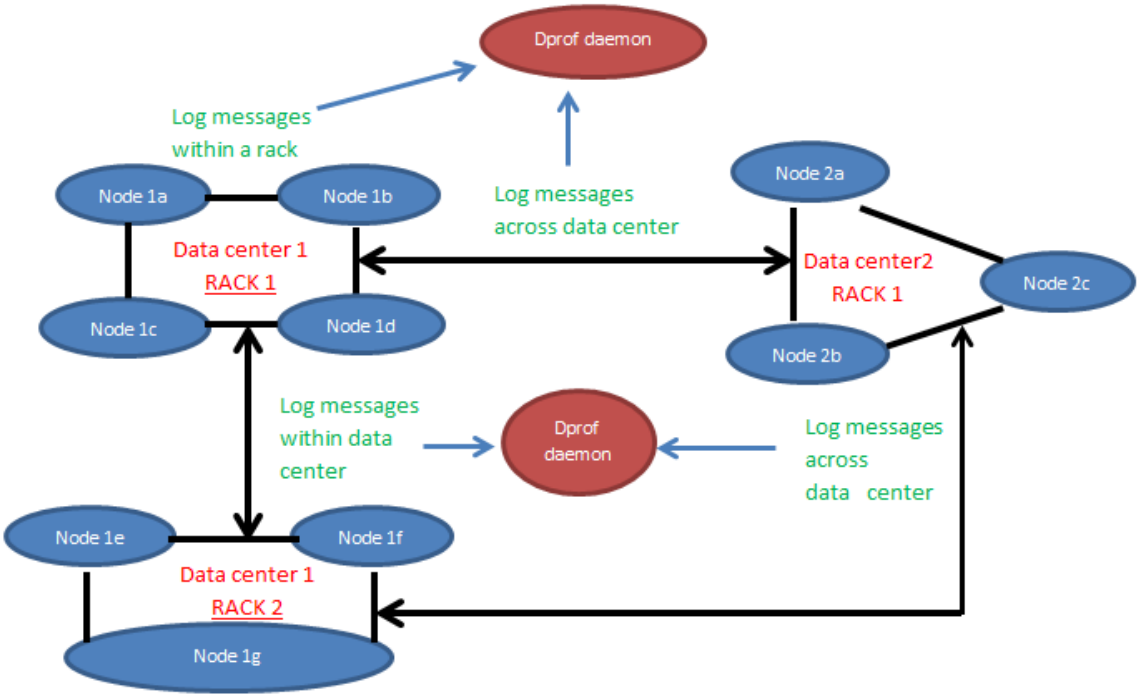

Figure 3: Logging systems in Dynomite architecture

Trace ID	Span ID	Message tag, other IDs ...	Timestamp	Caller metadata	Operation metadata

Figure 4: Dynomite message logging

The message tag in the dynomite message is used for every event related to sending and receiving (Figure 4). When dynomite processes receive/send message, message tag increases. Time-stamp is the time that event occurs. Caller meta-data includes data center, rack, and node ID and port number. Operation metadata includes operation types such as GET/SET and number of bytes sent and received. Trace ID is unique across dynomite nodes and a 64 bit encoding of {message tag, timestamp, call meta data, and operation meta data}.

5. Experimental Results

We instrumented dynamite-redis and ran it on a two node cluster. We ran a script to generate commands to redis and noted replication occurring across cluster from one redis instance to another. These were all run on EC2 m3.medium instances, with data aggregation on a 3-node Cassandra cluster. Table 1 shows a sampling of some of the messages generated from Dynomite from our instrumentation.

Our tracing provided us insights into the operation of the Dynomite replication protocol – analyzing the traces allowed us to observe the system showing asynchronous replication, e.g., Figure 5 shows an instance of tracing, where we can see that response to write requests at node Dyno.86 is made before data is replicated at Redis with Dyno.87.

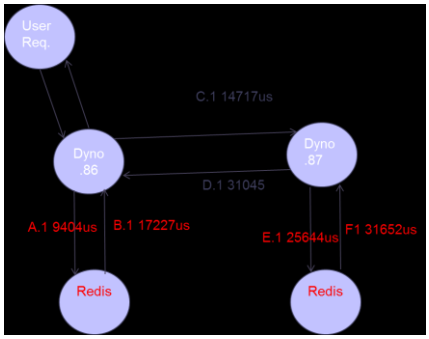

Figure 5: Tracing a request call

Trace ID	Span ID	Message ID	Timestamp	Caller ID	Operation
6417710171003740	1	1	1417710173316301	171.67.216.86	sent 77bytes
6417710171003740	1	3	1417710170991210	171.67.216.87	received 77bytes
7417710170992190	1	5	1417710181270820	171.67.216.87	sent 195bytes
7417710170992190	1	8	1417710181267940	171.67.216.86	received 195bytes
4417710175472954	2	2	1417710173338739	171.67.216.86	sent "set a b"
4417710175472954	2	1	1417710173382460	171.67.216.88	received "set a b"

Table 1: Sample trace Collection from Dynomite

Using cql Cassandra queries, we determined the longest and shortedst traces, and observe replication delays when network delay (using tc qdisk/class) between Dyno.86 and Dyno.87 nodes – the user requests made to Dyno.86 remained unchanged. We measured response delays to requests, with sampling of 10% of the traces – within the margin of error , we were unable to measure any significant monitoring overhead. Redis is a highly capable K-V store. We believe that had our setup pumped at a higher throughput, we could have observed the monitoring overhead. [5] reports latency change of 2% and throughput change or -0.08% for 1/16 sampling.

6. Conclusions

Dprof, working at the RPC layer, transparently allows the monitoring of heterogeneous collection of systems, independent of applications, processes, machine configurations, and datacenters. In addition, by sampling traces, dprof ensures that there is minimal overhead on the system being monitored. The system is primarily targeted at diagnosing steady-state or performance problems in distributed systems. Since dtrace samples traces, and it follows "trigger-preserving" time attribution, it is not directly applicable for diagnosing anomalies in rare workflows or one-off problems.

7. References

[1] VMWare Vcenter Operations: http://www.vmware.com/products/vrealize-operations
[2] AWS Cloudwatch: http://aws.amazon.com/cloudwatch/
[3] OpenStack Telemetry (Ceilometer), https://wiki.openstack.org/wiki/Ceilometer
[4] Monitoring and Alerting for OpenStack, http://www.subbu.org/blog/2013/10/monitoring-and-alerting-for-openstack
[5] Sigelman, B., Barroso, L., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., and Shanbhag, C., Dapper, a large-scale distributed systems tracing infrastructure. Technical Report dapper-2010-1, Google, April 2010.
[6] Cloudera HTrace. http://github.com/cloudera/htrace.
[7] Twitter Zipkin. https://github.com/twitter/zipkin.
[8] New Relic Application Performance Monitoring - http://newrelic.com/application-monitoring
[9] Sambasivanm R.□, Fonseca, R., Shafer, I., and Ganger, G. - So, you want to trace your distributed system? Key design insights from years of practical experience, CMU PDL Technical Report, CMU-PDL-14-102, April 2014.
[10] Fonseca, R., Porter, G., Katz, R., Shenker, S., and Stoica., I., X-Trace: a pervasive network tracing framework. In NSDI 07: Proceedings of the |th USENIX Symposium on Networked Systems Design and Implementation, 2007.
[11] Barham, P., Donnelly, A., Isaacs, R., and Mortimer,, R. Using Magpie for Request Extraction and Workload Modeling. In Proc. USENIX OSDI (2004).
[12] Dynomite - http://techblog.netflix.com/2014/11/introducing-dynomite.html
[13] Dynomite GIT Repository - https://github.com/Netflix/dynomite

[14] Apache Cassandra - https://github.com/apache/cassandra
[15] Hastorun, D., Jampani, M., Kakulapati, G., Pilchin, A., Sivasubramanian, S., Vosshall, P., & Vogels, W. Dynamo: Amazon's highly available key-value store. In *In Proc. SOSP (2007)*.