blkin: An end-to-end tracing tool for software defined storage systems

Abstract

Distributed storage systems require special treatment concerning their monitoring and tracing because of their low-latency and high-throughput demands. In this paper we describe the development of a low-impact, live tracing infrastructure used in monitoring of large scale storage systems. *blkin* is a system based on LTTng[3] implementing the tracing scheme proposed in the Dapper paper[6]. Our contribution to the problem includes a tracing library based on the Dapper semantics targeting low-latency applications as well as a proposed architecture for live tracing of such kind applications. As a use-case of our system, we decouple and analyze Archipelago[2] and RADOS[1] measure read/write requests initiated with the VMs that end up being served by RADOS.

Keywords: tracing, monitoring, low-latency, LTTng, RADOS

1 Introduction

The recent burst of cloud computing has oriented researchers' attention towards distributed file systems in an attempt to manipulate the huge load of data being produced. Consequently, the need to monitor and trace this kind of distributed systems has aroused. A monitoring tool that could provided information about each request's interaction with the storage system would be of vital importance since vulnerabilities and latencies of the system would become obvious. However, the nature and the purpose of storage systems disables us to use conventional monitoring tools. High-throughput and low-latency are key characteristics of these systems and monitoring should not affect them. Previous approaches to the problem either separated the monitoring and tuning phase from the operation phase (first trace the system, tune it and then put it in production), or concentrated mostly on per module tracing and did not provide a full stack visualization of the storage request. We present the design and implementation of *blkin*. *blkin* is a system that enables us to trace a storage system in real time, with a very low overhead and visualize the aggregated information. We finally provide the results from using *blkin* in tracing a software defined storage system *Archipelago*[2] backed by RADOS[1].

2 Tracing logic

One of the most important aspects of system tracing is the aggregated information interpretation. There are two dominant schools concerning trace information. *Black box* monitoring scheme assumes there is no additional information other than the message record. So statistical regression techniques should be used to infer any existent association. *Annotation-based* schemes, on the other hand, rely on applications or middleware to explicitly tag every record with a global identifier that links these message records back to the originating request. In order to have an overall review of the system per specific request, we have to implement an annotation-based monitoring scheme.

After examining the nature of distributed storage systems, we found some common characteristics that should be taken into consideration when designing an end-to-end tracing infrastructure.

- Every request in order to be completed passes through **different phases** in order to get served. Each phase implements a different functionality. Between these distinct phases there is communication overhead crucial to the system's performance that needs to be measured.
- Throughout the serving of the request, **different nodes** of the infrastructure are activated. These nodes may vary depending on the request. Information about the exact time and place of a trace record should also be kept.

Google proposed a complete annotation-based monitoring scheme in the Dapper paper[6]. This proposed scheme enables us to depict casual relationships between the different processing phases. In short Dapper describes the following semantics for tracing:

annotation The actual information being logged. There are two kinds of annotations. Either *timestamp*, where the specific timestamp of an event is being logged or *key-value*, where a specific key-value pair is being logged.

span The basic unit of the process tree. Each specific phase of processing can be depicted as a different span. Each span should have a specific name and a distinct span id.

trace Every span is associated with a specific trace. A different trace id is used to group data so that all spans associated with a specific trace also share the common trace id. For our case, information concerning a specific request to the storage system share the same trace id and each distinct request initiates a new trace id.

parent span In order to depict the causal relationships between different spans in a single trace, parent span id is used. Spans without a parent span ids are known as *root spans*.

So, by creating tracing data according to these semantics we can have an end-to-end sense of our system's performance, behavior and internal latencies that may vary depending on the nature and size of each request.

Based on these primitives, Twitter created Zipkin[7], a distributed tracing system used to gather timing data for all the disparate services running at Twitter. Zipkin consists of a data collector, a database service and a Web interface to show the aggregated data created according to the principles described above.

Zipkin uses Scribe[8] to transport all the traces from the different services. Scribe is a logging server created by Facebook, aiming to be scalable and reliable. Scribe servers are arranged in a directed graph, with each server knowing only about the next server in the graph. This network topology allows for adding extra layers of fan-in as a system grows, and batching messages before sending them between datacenters as well as providing reliability in case of intermittent connectivity or node failure.

Although Zipkin offers various libraries (Python, Ruby, Scala) in order to instrument applications, there was no providence for low-latency applications written in C/C++. Our contribution was to create a C/C++ library that encapsulates the Dapper semantics and can be used in C/C++ projects to create trace information in accordance with the formentioned logic. Although this library's backend is independent, which means that each one can implement a log aggregation backend for the library, we offered a specific solution according to our initial prerequisites that is being thoroughly examined in the next chapter.

3 Low overhead tracing

A storage system performance is a crucial matter. Especially recently, that block storage systems are used in provisioning virtual volumes for IAAS providers, every single latency would result in performance degradation and user dissatisfaction. So every approach to monitor or trace this kind of systems should have the least possible added overhead to the instrumented application. This is the reason why traditional logging systems would fail to tackle the problem.

3.1 Tracing backend

blkin was designed driven by a strict low-overhead prerequisite. So the first important decision to make concerned the system that would implement the system's backend, namely the system responsible for aggregating tracing data from the instrumented applications. So we chose to use *LTTng* (Linux Trace Toolkit - next generation)[3] in our system backend.

LTTng is a toolchain that allows integrated kernel and user-space tracing from a single user interface. *LTTng* was initially designed and implemented to reproduce, under tracing, problems occurring in normal conditions. It encapsulates synchronization primitives that meet the low-impact requirements by using a linearly scalable and wait-free RCU(Read-Copy Update) synchronization mechanism. This mechanism was also ported from the kernel to userspace. In addition, *LTTng* supports a variety of operational systems (Debian, Fedora, Arch) and since version 2.x kernel tracer modules are built against a vanilla or distribution kernel, without need for additional patches. So, *LTTng* enabled us to use the same toolkit for both user and kernel tracing and it could be easily installed on almost every instrumented system.

However, unlike other similar tracing systems (eg. SystemTap[4]) that follow a different approach by enabling to probe kernel-space events without having to resort to instrument, recompile, install and reboot the kernel, *LTTng* supports static tracepoint instrumentation. This means to manually insert tracepoints in the application source code and rebuild the application. Although it may seem counterproductive, static instrumentation abilities are limitless. Consequently, we can instrument and trace every part of the application that might be problematic and result with more meaningful conclusions since we can locate the root of the problem causing a possible latency for example.

3.2 Live tracing

The second prerequisite mentioned in the introduction, was the need for live tracing. This would mean to trace a production system and in real time have access and process the aggregated information. Although this may sound natural for traditional logging systems like syslog, in combination with the demand for low-impact tracing several difficulties arise. Older approaches required separation between the tracing and operation phases. For LTTng especially, since tracing data were in a binary form, this form could be decoded only after the end of the tracing session. However, since version 2.4 of *LTTng* live tracing is possible. Tracing data can be streamed and processed in real time in different nodes than in those being aggregated using a daemon process called *relayd* that collects the tracing data over the network.

However, *LTTng* live tracing supports only its native CTF format. Consequently, in order to be able to process and visualize tracing data in real time with Zipkin, we needed to implement a live-tracing plugin that would transform real time CTF data to scribe messages sent to either a local or the central Zipkin Scribe server. So based on Babeltrace[5] which is a CTF converter and trace viewer, we developed a plugin that reads and decodes the CTF-formatted information, it creates Thrift[9] encoded messages recognizable by the Zipkin collector and sends these messages to the Scribe server. So we end up with the architecture presented in figure 1.

The instrumented application produces tracing information through the instrumentation points in its source code. These information are aggregated by *LTTng* and sent to the *relayd* daemon. This daemon run either on the same or on a different node. After that, our Babelrtace plugin communicates with the *relayd*, gets the tracing data, processes them as mentioned and finally sends the data to the Scribe server. In figure 1 each arrow represents a TCP communication. So, the deployment can vary a lot. Depending on the deployment, the mentioned Scriber server can be either the central Zipkin collector or a local Scribe daemon that will eventually send the data to the central collector, thus exploiting the asynchronous, buffered communication provided by Scribe.

So in a cluster deployment with many nodes, we end up having the architecture presented in figure 2. In this deployment, there is a whole *blkin* stack running on every node of the cluster. All the communication is over localhost and there is also a local Scribe daemon. This daemon will store data locally in case of a connection loss with the central server or if the central zipkin collector is busy. So we can make sure that no data will be lost, because any data loss would mess up the tracing semantics and end up with an inconsistent UI and database state.

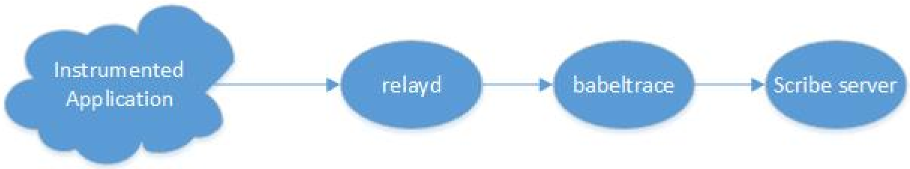

Figure 1: *blkin* architecture

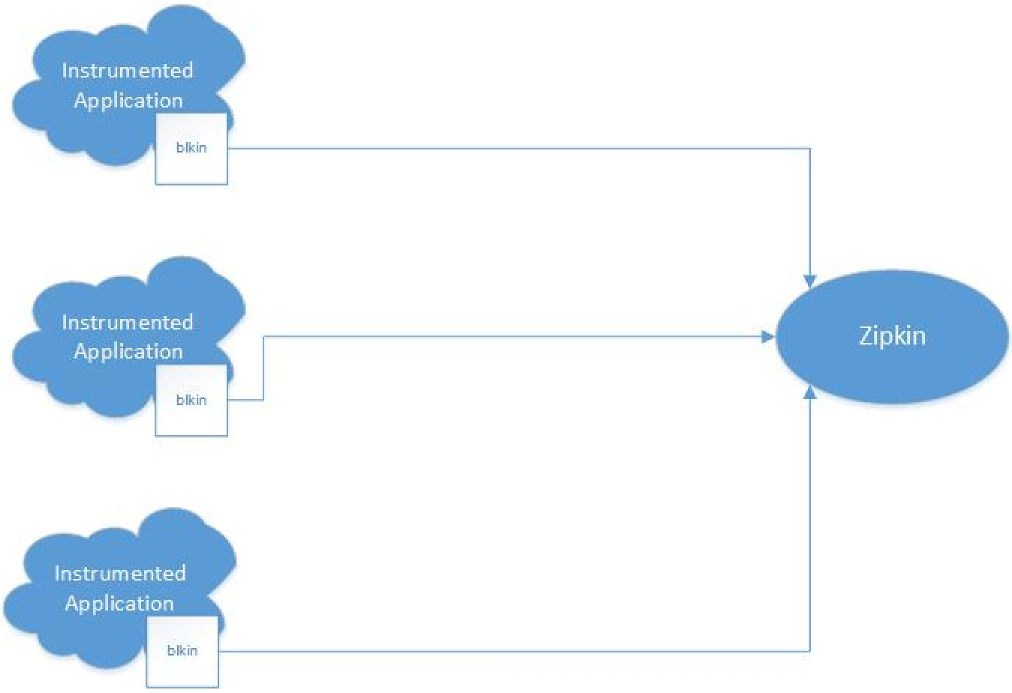

Figure 2: *blkin* deployment

4 Evaluation - Usecase

In order to evaluate our system and our tracing model's expressiveness, we instrumented *Archipelago*[2]. Archipelago is a distributed storage layer that decouples Volume and File operations/logic from the actual underlying storage technology, used to store data and is part of the Synnefo project[10]. Although Archipelago supports several storage backends, we chose to use and instrument RADOS[1].

Through this instrumentation we expect to identify and measure the different logic layers through which a user read or write request passes from the time of its creation till the answer reaches back to the user, as well as the enclosed latencies between them.

References

[1] RADOS: A Scalable, Reliable Storage Service for Petabyte-scale Storage Clusters

[2] Archipelago, info about

[3] LTTng, info about

[4] SystemTap, info about

[5] Babeltracep, info about

[6] Dapper, info about

[7] Zipkin, info about

[8] scribe, info about

[9] scribe, info about

[10] scribe, info about