Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής
και Υπολογιστών

Τίτλος

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

MARIOS

Επιβλέπων : test
test

Αθήνα, Ιανουάριος 1111

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής
και Υπολογιστών

Τίτλος

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

MARIOS

Επιβλέπων : test
test

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 1η Ιανουαρίου 1111.

....	....	....
test	test	test
test	test	test

Αθήνα, Ιανουάριος 1111

....

marios

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Λέξεις κλειδιά

Abstract

Key words

Ευχαριστίες

marios,

Αθήνα, 1η Ιανουαρίου 1111

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

When back in April 1965 Gordon E. Moore stated the following

> "The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000. I believe that such a large circuit can be built on a single wafer."[25]

had no idea that he had actually started a race among the academia and the industry to overcome or at least abide the this law.

At first, since the technology was premature, the evolution in VLSI technology went hand in hand with the evolution in computer architecture. The more and faster transistors resulted in achievements in instruction level parallelism (ILP). From 1975 to 2005 the endeavour put in computer architecture resulted in technological advances varying from deeper pipelines and faster clock speeds to superscalar architectures. But in around 2005 the ILP wall was hit. Transistors could not be utilized to increase serial performance, logic became too complex and performance attained was very low compared to power consumption. This lead to the creation of multicore systems and entered the programmers to the jungle of parallel software. So far the evolution was almost in accordance with the famous law. However, in around 2009 to 2011, it was the power wall's time to be hit. The famous power equation $P = cV^2 f$ along with the CPU to memory gap (eikona) led to the technological burst of distributed and cloud computing.

In 2009 Amazon.com introduced the Elastic Compute Cloud and since then the term 'cloud' is one of the hottest buzzwords not only among the industry and academia but also among everyday people that take advantage of the 'power of cloud'. Although the term may be vague, the definition of cloud computing, according to NIST (National Institute of Standards and Technology), is the following:

> "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics ,three service models, and four deployment models."[26]

In the previous brief computer chronology, I kept describing bottlenecks and walls to be overcome. However, it not clear how these bottlenecks become obvious and how scientists can be sure that they have reached one's technology's limits before moving on to the next one. The answer to the previous questions has always been given through tracing. Tracing is a process recording information about a program's execution, while it is being executed. These information may be low level metrics like performance counters or time specific metrics in order to evaluate system's latencies and throughput. Tracing data are mostly useful for developers and can be used for debugging, performance tuning and performance evaluation. From the single-cpu, integrated computer to the hundreds-node cloud infrastructure, trace and performance engineers face challenging problems that vary from platform to platform, but in any case play a vital role the system's design and implementation.

Cloud and distributed computing provided trace engineers with more challenging problems. The system scale is now much greater and program execution is far from deterministic and can take place in any cluster node. So each program execution is not bounded to a specific context. Other problems that needed solving was data and time correlation between the different computing nodes. Also, unlike single chip platforms that can be individually traced and evaluated, cloud infrastructures need to be traced with full-load under production conditions. This set more restrictions concerning the overhead that tracing adds to the application. Finally, tracing is notorious about the amount of data that produces. So distributed and cloud tracing demands the use of distributed data storage systems and processing methods like distributed NOSQL databases and Map-Reduce frameworks.

So to sum up, as described by any design model, the system verification consists a major part of a system's implementation and working process. Verification is achieved through monitoring and tracing. Depending on the system's nature tracing and monitoring process and the tools used may vary. Picking the right tracing tools that will reveal the system's vulnerabilities and faults can be very demanding and the performance engineer for bringing them to light, respecting all the prerequisites set by the system.

1.1 Thesis motivation

The motivation behind this thesis emerged from concerns about the storage performance of the Synnefo [1] cloud software, which powers the ~**okeanos** [2] public cloud service [20]. I will briefly explain what ~**okeanos** and Synnefo are in the following paragraphs.

~**okeanos** is an IaaS (Infrastructure as a Service) that provides Virtual Machines, Virtual Networks and Storage services to the Greek Academic and Research community. It is an open-source service that has been running in production servers since 2011 by GRNET S.A. [3]

Synnefo [21] is a cloud software stack, also created by GRNET S.A., that implements the following services which are used by ~**okeanos** :

- *Compute Service*, which is the service that enables the creation and management of Virtual Machines.
- *Network Service*, which is the service that provides network management, creation and transparent support of various network configurations.
- *Storage Service*, which is the service responsible for provisioning the VM volumes and storing user data.
- *Image Service*, which is the service that handles the customization and the deployment of OS images.

[1] www.synnefo.org/

[2] https://okeanos.grnet.gr/

[3] Greek Research and Technology Network, https://www.grnet.gr/

- *Identity Service*, which is the service that is responsible for user authentication and management, as well as for managing the various quota and projects of the users.

Synnefo provides each virtual machine with at least one virtual volume provisioned by the Volume Service called Archipelago[16] and will be furthered detailed in Chapter . This thesis' purpose is to provide the developer or the system administrations with a cross-layer representation accompanied with the equivalent metrics and time information of an I/O request's route within the infrastructure from the time it is created inside the virtual machine till it is finally served by the storage backend. The design and implementation has to be done respecting the following two prerequisites:

- The tracing information should be gathered and processed in real-time from every node participating in the request serving.
- The tracing infrastructure should add the least possible overhead to the instrumented system, which should continued working properly production-wise

After the end of the tracing infrastructure implementation, the developer should be able to identify the distinct phases and the duration of each that an IO request passes through, measure communication latencies between the different layers and collect all the necessary information (chosen by him) that would help him understand the full context under which this specific request was served. All these information can be used for software faults detection and performance tuning as well as hardware malfunctions and faults like disk or network failures that would be difficult to detect otherwise.

The novelty of this thesis consists in combining live cross-layer, multi-node data aggregation, which is typical for monitoring but not for tracing, with the precision and accuracy of tracing, respecting a hard prerequisite of low overhead. Previous tracing infrastructures offered only partial solutions. Some of them would separate the tracing from the working phase because of the great added overhead, others provided no mechanism for data correlation, while the traditional monitoring systems did not meet our low-level tracing needs.

The proposed system is called *B*lkKin. It is designed respected the aforementioned prerequisites and make use of the latest tracing semantics and infrastructures employed by great tech companies like Google and Twitter.

1.2 Thesis structure

This thesis is structured as follows:

Chapter 2

Theoretical Background

In this chapter we provide the necessary background to familiarize the reader with the main concepts and mechanism used later in the document. For every subsystem employed in BlkKin we briefly describe some counterparts justifying our choice. The approach made is rudimentary, intended to introduce a reader with elementary knowledge on distributed systems.

Specifically, Section 2.1 covers the concepts around distributed storage systems and the difficulties concerning their monitoring. In Section 2.2 we describe Archipelago, Synnefo's Volume Service, and how IO requests initiated within the virtual machine end up being served by a distributed storage system. In Section 2.3 we explain the need for tracing and cite various open-source tracing systems with their advantages and disadvantages. Finally, in Section 2.4 we describe the different needs covered by logging and cite some popular logging systems.

2.1 Distributed storage systems

Providing reliable, high-performance storage that scales has been an ongoing challenge for system designers. High-throughput and low-latency storage for file systems, databases, and related abstractions are critical to the performance of a broad range of applications. Historically, data centers first created 'islands' of SCSI disk arrays as direct-attached storage (DAS), each dedicated to an application, and visible as a number of 'virtual hard drives' (i.e. LUNs). Initally, a SAN (Storage-Area-Network) consolidates such storage islands together using a high-speed network. However, a SAN does not provide file abstraction, only block-level operations. Also, the cost of scaling a SAN infrastructure scales exponentially. These boosted the development of more service-oriented-architectures. Emerging clustered storage architectures constructed from storage bricks or object storage devices (OSDs) seek to distribute low-level block allocation decisions and security enforcement to intelligent storage devices, simplifying data layout and eliminating I/O bottlenecks by facilitating direct client access to data. OSDs constructed from commodity components combine a CPU, network interface, and local cache with an underlying disk or RAID, and replace the convention block-based storage interface with one based on named, variable-length objects. As storage clusters grow to thousands of devices or more, consistent management of data placement, failure detection, and failure recovery places an increasingly large burden on client, controller, or metadata directory nodes, limiting scalability.

One of the design principles of object storage is to abstract some of the lower layers of storage away from the administrators and applications. Thus, data is exposed and managed as objects instead of files or blocks. Objects contain additional descriptive properties which can be used for better indexing or management. Administrators do not have to perform lower level storage functions like constructing and managing logical volumes to utilize disk capacity or setting RAID levels to deal with disk failure. File metadata are explicitly separate from data and data manipulation is allowed through programmatic interfaces. These interfaces include CRUD functions for basic read, write and delete operations, while some object storage implementations go further, supporting additional functionality like object versioning, object replication, and movement of objects between different tiers and types of storage. Most API implementations are ReST-based, allowing the use of many standard HTTP calls. This results in the abstraction shown in Figure 2.1.

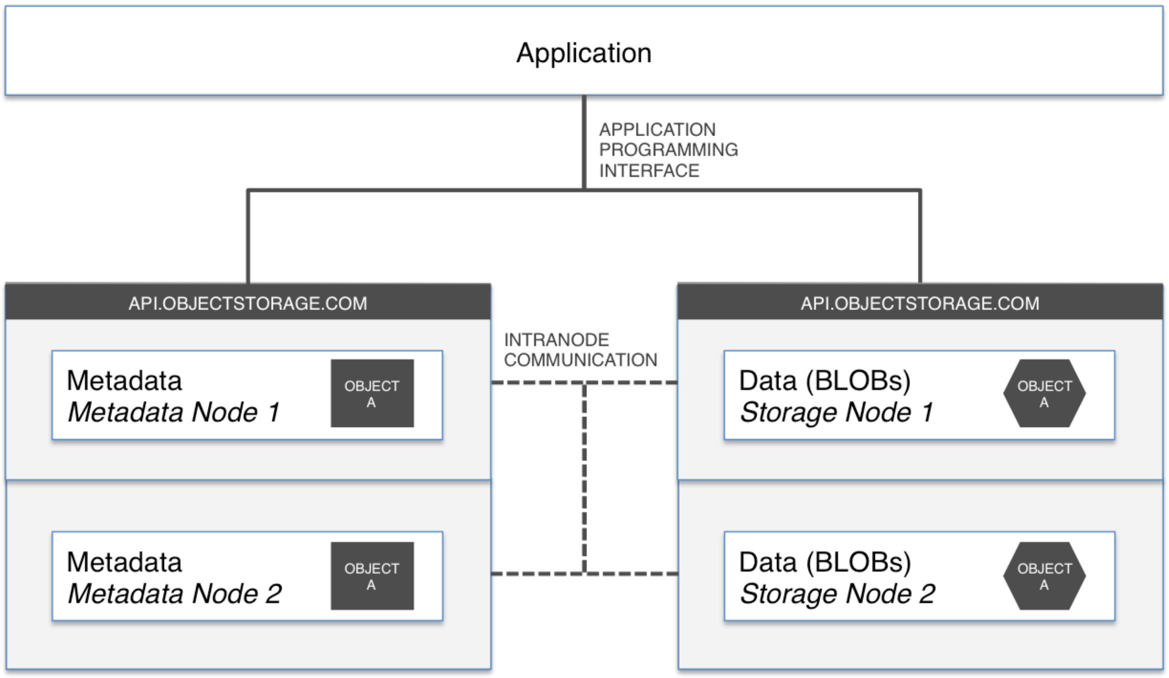


Figure 2.1: Storage Abstraction

Alhtough they differ substantially concerning their implementation, some of the most popular examples of such systems are: Amazon S3, OpenStack Swift and RADOS.

However, one common characteristic of all these systems, that led to the development of this thesis, is that they provide an architecture that easily scales out, based on APIs, but which is difficult to monitor and find out what really went wrong in case of a problem. This leads to a di-centralized data collection and a centralized data processing architecture for tracing information which is further explained in Chapter 1.1.

2.1.1 RADOS

RADOS stands for Reliable, Autonomic Distributed Object Store. It is the object store component of Ceph[1].Ceph is a free distributed object store and file system that has been created by Sage Weil for his doctoral dissertationi[30] and has been supported by his company, Inktank, ever since. RADOS seeks to leverage device intelligence to distribute the complexity surrounding consistent data access, redundant storage, failure detection, and failure recovery in clusters consisting of many thousands of storage devices.

RADOS basic characteristics are:

- *Replication*, which means that there can be many copies of the same object so that the object is always accessible, even when a node experiences a failure.

[1] http://ceph.com/

- *Fault tolerance*, which is achieved by not having a single point of failure. Instead, RADOS uses elected servers called **monitors**, each of which have mappings of the storage nodes where the objects and their replicas are stored.

- *Self-management*, which is possible since monitors know at any time the status of the storage nodes and, for example, can command to create new object replicas if a node experiences a failure.

- *Scalability*, which is aided by the fact that there is no point of failure, which means that adding new nodes theoretically does not add any communication overhead.

Ceph's building blocks can be seen in Figure 2.2

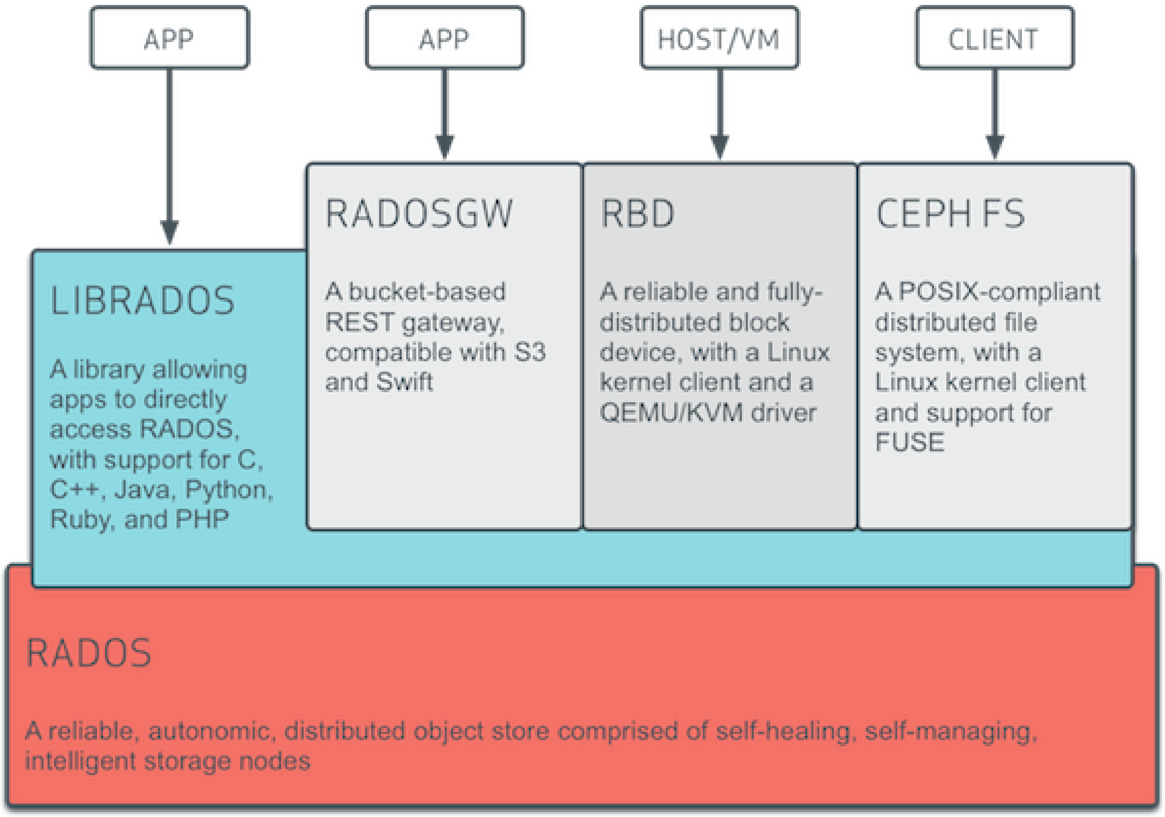

APP
APP
HOST/VM
CLIENT
RADOSGW
A bucket-based REST gateway, compatible with S3 and Swift
RBD
A reliable and fully-distributed block device, with a Linux kernel client and a QEMU/KVM driver
CEPH FS
A POSIX-compliant distributed file system, with a Linux kernel client and support for FUSE
LIBRADOS
A library allowing apps to directly access RADOS, with support for C, C++, Java, Python, Ruby, and PHP
RADOS
A reliable, autonomic, distributed object store comprised of self-healing, self-managing, intelligent storage nodes

Figure 2.2: Ceph abstraction

RADOS operations are based on the following components:

- *object store daemons*, which are userspace processes that run in the storage backend and are responsible for storing the data.

- *monitor daemons*, which are monitoring userspace processes that run in an odd number of servers that form a Paxos part-time parliament[22]. Their main responsibility is holding and reliably updating the mapping of objects to object store daemons, as well as self-healing when an object store daemon or monitor daemon has crashed.

Ceph's logic is based on CRUSH algorithm. According to this algorithm a map is created, called CRUSH map, which maps objects to store daemons. A fundamental idea in RADOS is the *placement group* (pg). Placement groups are used for load balancing. The number of placement groups is predefined. Then, when we want to create a new object, its name is hashed and assigned to a specific group.

Each placement group makes IO requests to the same OSDs. So, objects belonging to the same pg, will be replicated across the same OSDs. The relationship between placement groups and object store daemons is stored in CRUSH maps that each monitor daemon holds.

Since we would like to instrument RADOS code and measure its performance, apart from the theoretical background, we should also explain some of its operating internals, so that further analysis is consolidated. So, in brief, we will try to explain an IO request's route within a RADOS infrastructure.

Although, as seen in Figure 2.2, RADOS has multiple entry points (RBD, CephFS, RADOSGW), we are interested in the interaction with librados. Librados provides a well defined API for data manipulation and control, namely an API that enables to modify (CRUD) objects and interact with the Ceph monitors. There are binding for various languages like C, Python and Java.

Hypothetically, we have an application using librados, which can also run remotely from the RADOS cluster. The application want to write an objects. So, a nIO request is initiated from librados. RADOS employs an asynchronous, ordered point to point message passing library for communication. So, this request is serialized and a TCP message is created and sent to the RADOS cluster. After receive, this packet is handled by the equivalent RADOS Messenger classes, decoded and based on its kind, is placed in a *dispatch queue* to be served. This specific object belongs to a specific placement group. So, when the request reaches the top of the queue, based on this pg, the equivalent OSD undertakes its serving. Based on the replication factor, the equivalent number of replication requests is sent to other OSDs responsible for the same pg. During request handling per OSD, based on the request type, there are phases like *Journal Access* and finally the *Filestore Access*.

From the above analysis, we understood that request processing in RADOS is a perplexed procedure including multiple remote nodes collaborating. The only way to understand the internals and debug possible latencies and bottlenecks is through tracing and this is what we are going to examine further in this thesis.

2.2 Archipelago

2.2.1 Overview

Archipelago is a distributed storage layer and is part of the Synnefo cloud software. It decouples Volume and File operations/logic from the actual underlying storage technology, used to store data. It provides a unified way to provision, handle and present Volumes and Files independently of the storage backend. It also implements thin clones, snapshots, and deduplication, and has pluggable drivers for different backend storage technologies. It was primarily designed to solve problems that arise on large scale cloud environments. Archipelago's end goal is to:

- Decouple storage logic from the actual data store
- Provide logic for thin cloning and snapshotting
- Provide logic for deduplication
- Provide different endpoint drivers to access Volumes and Files
- Provide backend drivers for different storage technologies

As it is show in Figure 2.3, Archipelago lies between the VM's block device and the underlying storage level.

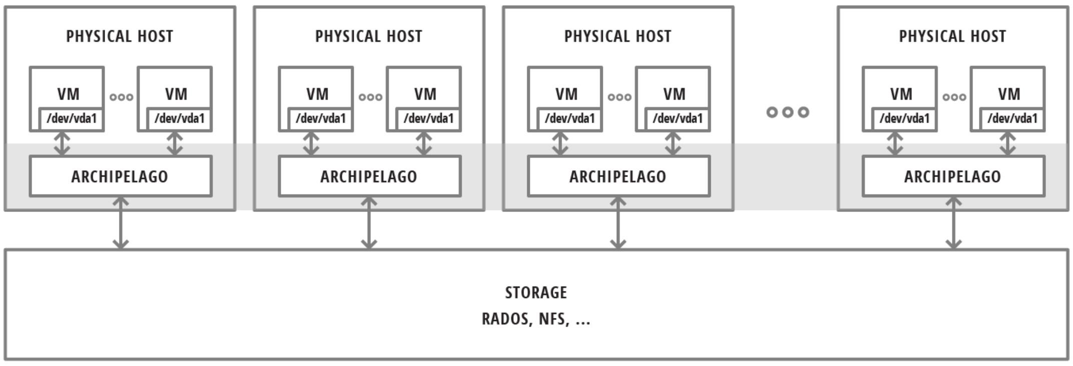


Figure 2.3: Archipelago Overview

2.2.2 Archipelago Internals

Archipelago has a modular internal architecture consisting of multiple components communicating over a custom made IPC mechanism called *X*SEG. Each component communicating over XSEG is called *p*eer.

XSEG is a custom mechanism that defines a common communication protocol for all peers, regardless of their type (userspace/kernspace, singlethreared/multithreaded). It builds a shared-memory segment, where peers can share data using zero-copy techniques.

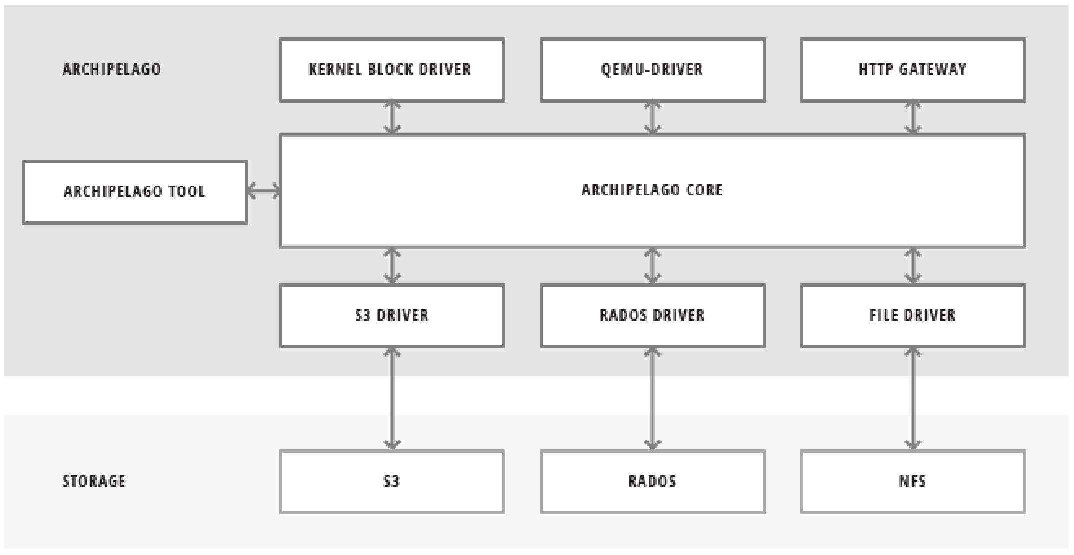


Figure 2.4: Archipelago APIs

Peers are considered either the Archipelago endpoints (Figure 2.4):

- block device driver
- qemu driver
- user provided process

- command line tool
- http gateway for files

or the Archipelago internal components:

VoLuMeComposerDaemon(vlmcd) vlmcd accepts requests from the endpoints and translates them to object requests, with the help of mapperd.

MapperDaemon(mapperd) mapperd is responsible for the mapping of volumes to objects. This means that it must tackle a broad set of tasks such as knowing the objects that a volume consists of, cloning and snapshotting volumes and creating new ones.

BlockerDaemon(blockerd) blockerd is not a specific entity but a family of drivers, each of which is written for a specific storage type (as seen at the down part of Figure 2.4).Blockers have a single purpose, to read/write objects from/to the storage.Currently, there are blockers for NFS and the RADOS object storage.

Figure 2.5 shows the interaction between the different peers over XSEG for a VM to perform an IO operation.

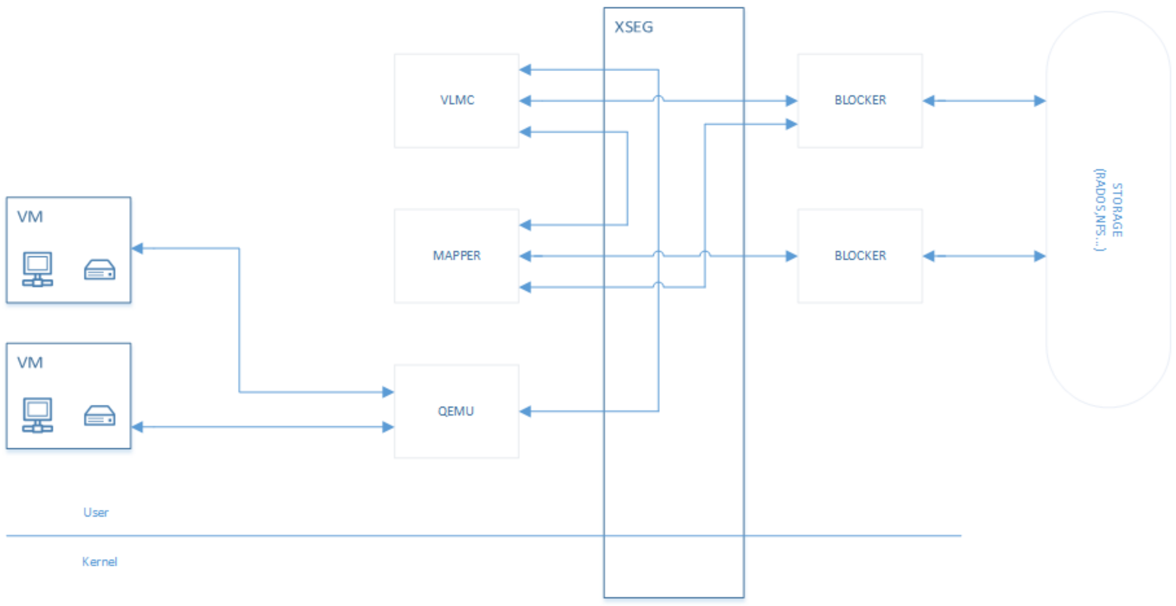


Figure 2.5: XSEG communication

As it is obvious from the above analysis, Archipelago's modular design is based on XSEG. XSEG affects significantly the overall Archipelago performance. So, we would like to have a mechanism that enables Archipelago monitoring without degrading Archipelago performance. This mechanism would reveal the latencies and bottlenecks between the several peers and enable Archipelago engineers to improve its performance. This mechanism is the proposed BlkKin system. Specifically, we will use BlkKin to monitor and measure the path from the QEMU driver until the storage layer.

2.3 Tracing Systems

Understanding where time has been spent in performing a computation or servicing a request is at the forefront of the performance analyst's mind. Measurements are available from every layer of a computing system, from the lowest level of the hardware up to the top of the distributed application stack.

In recent years we have seen the emergence of tools which can be used to directly trace events relevant to performance. This is augmenting the traditional event count and system state instrumentation, and together they can provide a very detailed view of activity in the complex computing systems prevalent today.

Event tracing has the advantage of keeping the performance data tied to the individual requests, allowing deep inspection of a request which is useful when performance problems arise. The technique is also exceptionally well suited to exposing transient latency problems. The downsides are increased overheads (sometimes significantly) in terms of instrumentation costs as well as volumes of information produced. To address this, every effort is taken to reduce the cost of tracing - it is common for tracing to be enabled only conditionally, or even dynamically inserted into the instrumented software and removed when no longer being used.

In early 1994, a technique called dynamic instrumentation or Dyninst API [19] was proposed to provide efficient, scalable and detailed data collection for large-scale parallel applications (Hollingsworth et al., 1994). Being one of the first tracing systems, the infrastructure built for data extraction was limited. The operating systems at hand were not able to provide efficient services for data extraction. They had to build a data transport component to read the tracing data, using the ptrace function, that was based on a time slice to read data. A time slice handler was called at the end of each time slice, i.e when the program was scheduled out, and the data would be read by the data transport program built on top.

This framework made possible new tools like DynaProf [10] and graphical user interface for data analysis. DynaProf is a dynamic profiling tool that provides a command line interface, similar to gdb, used to interact with the DPCL API and to basically control tracing all over your system.

Kernel tracing brought a new dimension to infrastructure design, having the problem of extracting data out of the kernel memory space to make it available in user-space for analysis. The K42 project [3] used shared buffers between kernel and user space memory, which had obvious security issues. A provided daemon waked up periodically and emptied out the buffers where all client trace control had to go through. This project was a research prototype aimed at improving tracing performance. Usability and security was simply sacrificed for the proof of concept. For example, a traced application could write to these shared buffers and read or corrupt the tracing data for another application, belonging to another user.

In the next sections, recent tracers and how they built their tracing infrastructure will be examined.

2.3.1 Magpie

One of the earliest and most comprehensive event tracing frameworks is Magpie [5]. This project builds on the Event Tracing for Windows infrastructure which underlies all event tracing on the Microsoft Windows platform. Magpie is aimed primarily at workload modelling and focuses on tracking the paths taken by application level requests right through a system. This is implemented through an instrumentation framework with accurate and coordinated timestamp generation between user and kernel space, and with the ability to associate resource utilisation information with individual events.

The Magpie literature demonstrates not only the ability to construct high-level models of a distributed system resource utilisation driven via Magpie event tracking, but also provides case studies of low-level performance analysis, such as diagnosing anomalies in individual device driver performance. Magpie utilises a novel concept in behavioural clustering, where requests with similar behaviour (in terms of temporal alignment and resource consumption) are grouped. This clustering underlies the workload modelling capability, with each cluster containing a group of requests, a measure of “cluster diameter”, and one selected “representative request” or “centroid”. The calculation of cluster diameter indicates deep event knowledge and inspection capabilities, and although not expanded on it implies detailed knowledge of individual types of events and their parameters. This indicates a need for significant user intervention to extend the system beyond standard operating system level events.

As an aside, it is worth noting here that, for the first time, we see in Magpie the use of a binary tree graph to represent the flow of control between events and sub-events across distinct client/server processes and/or hosts.

2.3.2 DTrace

Then, Sun Microsystemsa released, in 2005, DTrace [7] which offers the ability to dynamically instrument both user-level and kernel-level software. As part of a mass effort by Sun, a lot of tracepoints were added to the Solaris 10 kernel and user space applications. Projects like FreeBSD and NetBSD also ported dtrace to their platform, as later did Mac OS X. The goal was to help developers find serious performance problems. The intent was to deploy it across all Solaris servers and to use it in production. If we look at the DTrace architecture, it uses multiple data providers, which are basically probes used to gather tracing data and write it to memory buffers. The framework provides a user space library (libdtrace) which interacts with the tracer through ioctl system calls. Through those calls, the DTrace kernel framework returns specific crafted data for immediate analysis by the dtrace command line tool. Thus, every interaction with the DTrace tracer is made through the kernel, even user space tracing. On a security aspect, groups were made available for different level of user privileges. You have to be in the dtrace proc group to trace your own applications and in the dtrace kernel group to trace the kernel. A third group, dtrace user, permits only system call tracing and profiling of the user own processes. This work was an important step forward in managing tracing in current operating systems in production environment. The choice of going through the kernel, even for user space tracing, is a performance trade-off between security and usability.

2.3.3 SystemTap

In early 2005, Red Hat released SystemTap [27] which also offers dynamic instrumentation of the Linux kernel and user applications. In order to trace, the user needs to write scripts which are loaded in a tapset library. SystemTap then translates these in C code to create a kernel module. Once loaded, the module provides tracing data to user space for analysis. Two system groups namely stapdev and stapusr are available to separate possible trac- ing actions. The stapdev group can do any action over Systemtap facilities, which makes it the administrative group for all tracing control (Don Domingo, 2010) and module creation. The second group, stapusr, can only load already compiled modules located in specific protected directories which only contain certified modules. The project also provides a compile-server which listens for secure TCP/IP connections using SSL and handles module compilation requests from any certified client. This acts as a SystemTap central module registry to authenticate and validate kernel modules before loading them. This has a very limited security scheme for two reasons. First, privileged rights are still needed for specific task like running the compilation server and loading the modules, since the tool provided by Systemtap is set with the setuid bit. Secondly, for user space tracing, only users in SystemTap's group are able to trace their own application, which implies that a privileged user has to add individual users to at least the stapusr group at some point in time, creating important user management overhead. It is worth noting that the compilation server acts mostly as a security barrier for kernel module control. However, like DTrace, the problem remains that it still relies on the kernel for all tracing actions. Therefore, there is still a bottleneck on performance if we consider that a production system could have hundreds of instrumented applications tracing simultaneously. This back and forth in the kernel, for tracing control and data retrieval, cannot possibly scale well.

2.4 Logging Systems

Logs are a critical part of any system, they give you insight into what a system is doing as well what happened. Unlike tracing, log data are not low-level and do not refer to the system's performance and there is no special care about the overhead that logging add to the system. Virtually every process running on a system generates logs in some form or another. Usually, these logs are written to files on local disks. When your system grows to multiple hosts, managing the logs and accessing them can get complicated. Searching for a particular error across hundreds of log files on hundreds of servers is difficult without good tools. A common approach to this problem is to setup a centralized logging solution so that multiple logs can be aggregated in a central location.

There are various options for log data aggregation as well as for visualizing the aggregated data. Some of them are cited here:

2.4.1 Syslog

Syslog is a standard for computer message logging. It permits separation of the software that generates messages from the system that stores them and the software that reports and analyzes them.

Syslog can be used for computer system management and security auditing as well as generalized informational, analysis, and debugging messages. It is supported by a wide variety of devices (like printers and routers) and receivers across multiple platforms. Because of this, syslog can be used to integrate log data from many different types of systems into a central repository.

Messages are labeled with a facility code (one of: auth, authpriv, daemon, cron, ftp, lpr, kern, mail, news, syslog, user, uucp, local0 ... local7) indicating the type of software that generated the messages, and are assigned a severity (one of: Emergency, Alert, Critical, Error, Warning, Notice, Info, Debug).

Implementations are available for many operating systems. Specific configuration may permit directing messages to various devices (console), files (/var/log/) or remote syslog servers. Most implementations also provide a command line utility, often called logger, that can send messages to the syslog. Some implementations permit the filtering and display of syslog messages.

Syslog is standardized by the IETF in RFC 5424. This standardization specifies a very important characteristic of Syslog that we would like to have available in our tracing infrastructure and this is severity levels. Every event to be traced is associated with a severity level varying from Emergency when the system is unusable to informational or debug level messages. From the syslog side the administrator can define which events he is interested about. So, for testing environments more events should be traced, while for production environments the events to be traced should be restricted to the absolutely needed.

2.4.2 Scribe

A new class of solutions that have come about have been designed for high-volume and high-throughput log and event collection. Most of these solutions are more general purpose event streaming and processing systems and logging is just one use case that can be solved using them. They generally consist of logging clients and/or agents on each specific host. The agents forward logs to a cluster of collectors which in turn forward the messages to a scalable storage tier. The idea is that the collection tier is horizontally scalable to grow with the increase number of logging hosts and messages. Similarly, the storage tier is also intended to scale horizontally to grow with increased volume. This is gross simplification of all of these tools but they are a step beyond traditional syslog options.

One popular solution is Scribe[2]. Scribe is a server for aggregating log data that's streamed in real time from clients. It is designed to be scalable and reliable. It was used and released by Facebook as open source. Scribe is written in C++ and it worths mentioning its transport layer and how Scribe logging data are processed and finally stored.

Concerning its **transport** layer, Scribe uses Thrift[3]. The Apache Thrift software framework, for scalable cross-language services development, combines a software stack with a code generation engine to build services that work efficiently and seamlessly between different programming languages. After describing the service in a specific file (thrift file), the framework is responsible for generating the code to be used to easily build RPC clients and servers that communicate seamlessly across programming languages. For Scribe especially the thrift file is the following:

```
enum ResultCode
{
OK,
TRY_LATER
}
struct LogEntry
{
1: string category,
2: string message
}
service scribe extends fb303.FacebookService
{
ResultCode Log(1: list<LogEntry> messages);
}
```

Listing 2.1: Scribe thrift definition file

In the above file a Log method is defined, which takes a list of LogEntry items as parameter. Every LogEntry consists of two strings, a category and a message. This specific Log method can return two different results codes, either 'OK' or 'TRY_LATER'. Based on this file, using Thrift we can create Scribe clients for every programming language.

Concerning **data manipulation** Scribe provides the following options. Based on the message's category, it can store the log entries in different files, one per category. Also Scribe has Hadoop support and can store the tracing information to an HDFS so that they can be processed later using Map-Reduce jobs.

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. So a Scribe server can operate either as a terminal server where data are finally stored, or as an intermediate server that forwards data to the next Scribe server. In case of congestion or of network problems, data are stored locally and forwarded when the problem is restored.

2.4.3 Graphite

Graphite is an enterprise-scale monitoring tool that runs well on cheap hardware. It is released under the open source Apache 2.0 license and it is used by many big companies like Google and Canonical.

[2] https://github.com/facebookarchive/scribe
[3] https://thrift.apache.org/

Although Graphite is not responsible for collecting data, it can store efficiently numeric time-series data and render graphs of this data on demand. Graphite can cooperate with other tools like collectd[4] for data aggregation.

From an architectural aspect, Graphite consists of 3 software components:

carbon - a Twisted daemon that listens for time-series data

whisper - a simple database library for storing time-series data (similar in design to RRD)

graphite webapp - A Django webapp that renders graphs on-demand using Cairo[5]

2.4.4 Ganglia

Ganglia[23] is a scalable distributed monitoring system for high performance computing systems such as clusters and Grids and grew out of the University of California, Berkeley. It is based on a hierarchical design targeted at federations of clusters. It relies on a multicast-based listen/announce protocol to monitor state within clusters and uses a tree of point-to-point connections amongst representative cluster nodes to federate clusters and aggregate their state. It leverages widely used technologies such as XMLfor data representation, XDR for compact, portable data transport, and RRDtool for data storage and visualization. It uses carefully engineered data structures and algorithms to achieve very low per-node overheads and high concurrency. The implementation is robust, has been ported to an extensive set of operating systems and processor architectures, and is currently in use on over 500 clusters around the world.

Ganglia architecture is made up of the following components.

gmond The Ganglia MONitor Daemon is a data-collecting agent that you must install on every node in a cluster. Gmond gathers metrics about the local node and sends information to other nodes via XML to a browser window. Gmond is portable and collects system metrics, such as CPU, memory, disk, network and process data. The Gmond configuration file /etc/gmond.conf controls the Gmond daemon and resides on each node where Gmond is installed.

gmetad The Ganglia METAdata Daemon is a data-consolidating agent that provides a query mechanism for collecting historical information about groups of machines. Gmetad is typically installed on a single, task-oriented server (the monitoring node), though very large clusters could require more than one Gmetad daemon. Gmetad collects data from other Gmetad and Gmond sources and stores their state in indexed RRDtool (round-robin) databases, where a Web interface reads and returns information about the cluster. The Gmetad configuration file /etc/gmetad.conf controls the Gmetad daemon and resides on the monitoring node.

RRDtool RRDTool is an open-source data logging and graphing system that Ganglia uses to store the collected data and to render the graphs for Web-based reports. Cron jobs that run in the background to collect information from the HP Vertica monitoring system tables are stored in the RRD database.

PHP-based Web interface — The PHP-based Web interface contains a collection of scripts that both the Ganglia Web reporting front end and the HP Vertica extensions use. The Web server starts these scripts, which then collect HP Vertica□specific metrics from the RRD database and generate the XML graphs. These scripts provide access to HP Vertica health across the cluster, as well as on each host.

[4] https://collectd.org/
[5] http://www.cairographics.org/

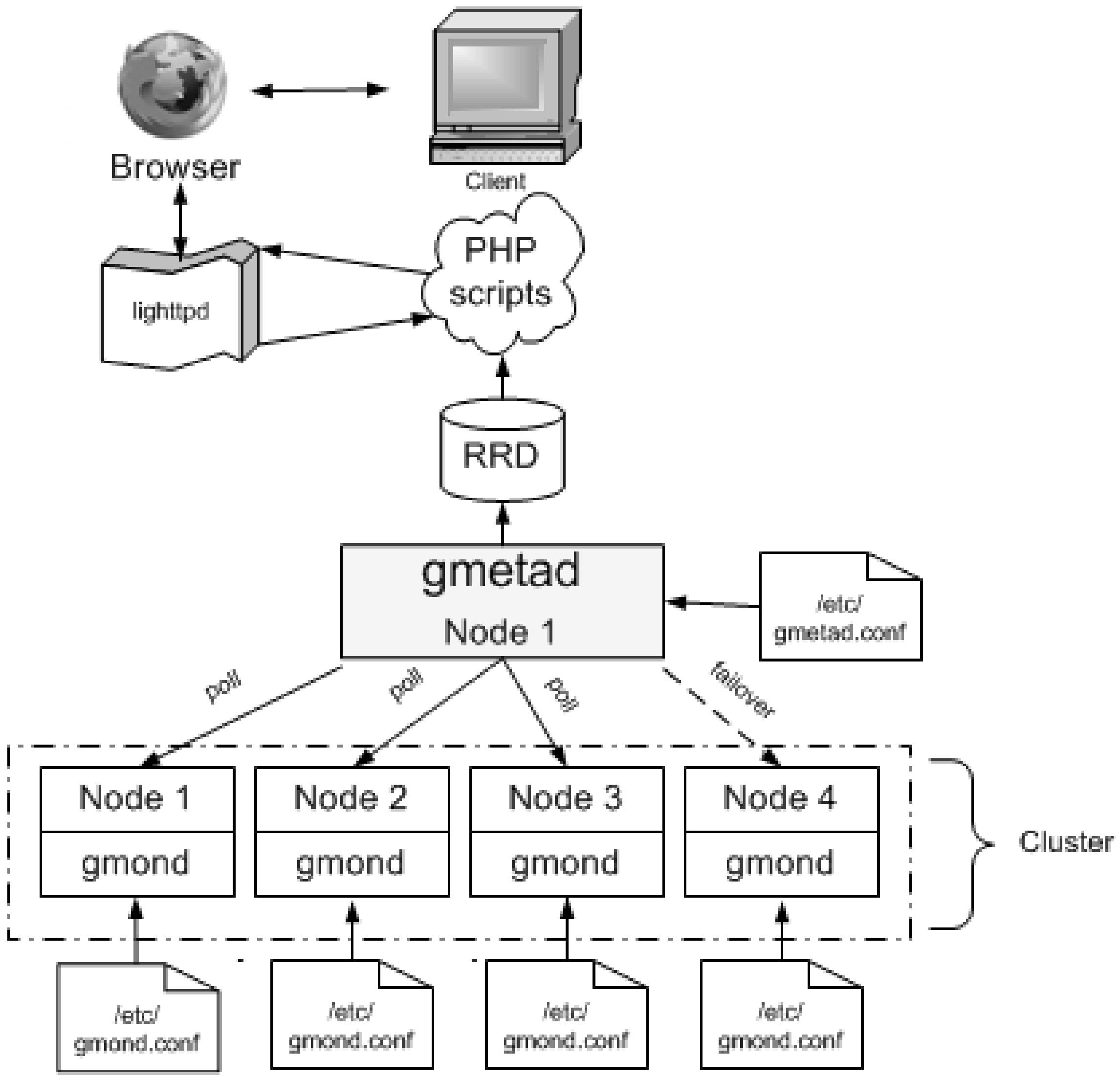


Figure 2.6: Ganglia architecture

Web server The Web server uses lighttpd, a lightweight http server that can be any Web server that supports PHP, SSL, and XML. The Ganglia web front end displays the data stored by Gmetad in a graphical web interface using PHP.

Advanced tools Gmetric, an executable, is added during Ganglia installation. Gmetric provides additional statistics and is used to store user-defined metrics, such as numbers or strings with units.

2.5 Conclusion

To sum up, it is obvious from the previous analysis that the tracing systems mentioned do not fit in our demands concerning the added overhead to the instrumented application since their solutions pass through the kernel space. This extra overhead makes them unsuitable for live tracing. The solution for for the BlkKin tracing backend was given from the Linux Trace Toolkit - next generation (LTTng) because it provides separate mechanisms for kernel and user space tracing. LTTng is furthered examined in Chapter 1.1.

Concerning the logging systems, we need to imitate their architecture for BlkKin's architecture, since we need a central trace aggragation point and a UI that visualizes the information. We can conclude that we need:

tracing daemon that runs on every cluster node and collects data with a low-overhead

central data collector where all tracing information are stored

Web UI where tracing information are rendered in a way that extracts the necessary information revealing problems and performance issues in the first place. For more elaborate information extraction, trace information can be furthered processed apart from the UI.

For data collection, we are going to use LTTng, while for the data aggregation and the visualization we are going to use Zipkin, a distributed tracing system created by Twitter. Zipkin as well as the reasons for our choice are furthered examined in Chapter 1.1.

Chapter 3

Linux Trace Toolkit - next generation (LTTng)

In this chapter we analyze Linux Trace Toolkin - next generation (LTTng), which was our choice for BlkKin's tracing backend, and we describe its internal characteristics that led us to using it. Specifically, we give an overall outline of its architecture and basic notions in Section 3.1. Then, we describe the buffering scheme used both for kernel and user space (Section 3.2) and we continue by citing kernel and use space implementation mechanism in Sections 3.3, 3.4. Finally we cite the tracing format used by LTTng (Section 3.5) and the mechanism for live tracing in Section .

3.1 Overview

Linux Trace Toolkin - next generation is the successor of Linux Trace Toolkit. It started as the Mathew Desnoyer's PhD dissertation [11] in École Polytechnique de Montréal. Since then, it is maintained by EfficiOS Inc[1] and the DORSAL lab in École Polytechnique de Montréal.

The LTTng project aims at providing highly efficient tracing tools for Linux. Its tracers help tracking down performance issues and debugging problems involving multiple concurrent processes and threads. Tracing across multiple systems is also possible. This toolchain allows integrated kernel and user-space tracing from a single user interface. It was initially designed and implemented to reproduce, under tracing, problems occurring in normal conditions. It uses a linearly scalable and wait-free RCU (Read-Copy Update) synchronization mechanism and provides zero-copy data extraction. These mechanisms were implemented in kernel and then ported to user-space as well.

Apart from LTTng's kernel tracer and userspace tracer, viewing and analysis tools are part of the project. In this thesis, we worked with and extended *Babeltrace* [2].

Apart from the fact LTTng is a complete toolchan that can be easily installed in almost any Linux distribution and the integrated kernel and user space tracing offered, we chose LTTng because of its minimal performance overhead. Since it was initially designed to 'reproduce, under tracing, problems occurring in normal conditions', LTTng was the ideal tool to use for real-time low-overhead, block-storage tracing with BlkKin.

In order to understand how LTTng manages to have such a good performance, we have to go through its internals. But first, we give an overview outline of its architecture and basic components. According to D. Goulet's Master thesis ([17]), LTTng's architecture can be summarized as shown in Figure 3.1.

The `lttng` command line interface is a small program used to interact with the session daemon. Possible interaction are creating sessions, enabling events, starting tracing and so on. The use of this command line tool is further explained in Chapter 1.1 about how to use BlkKin.

[1] http://www.efficios.com/
[2] http://lttng.org/babeltrace

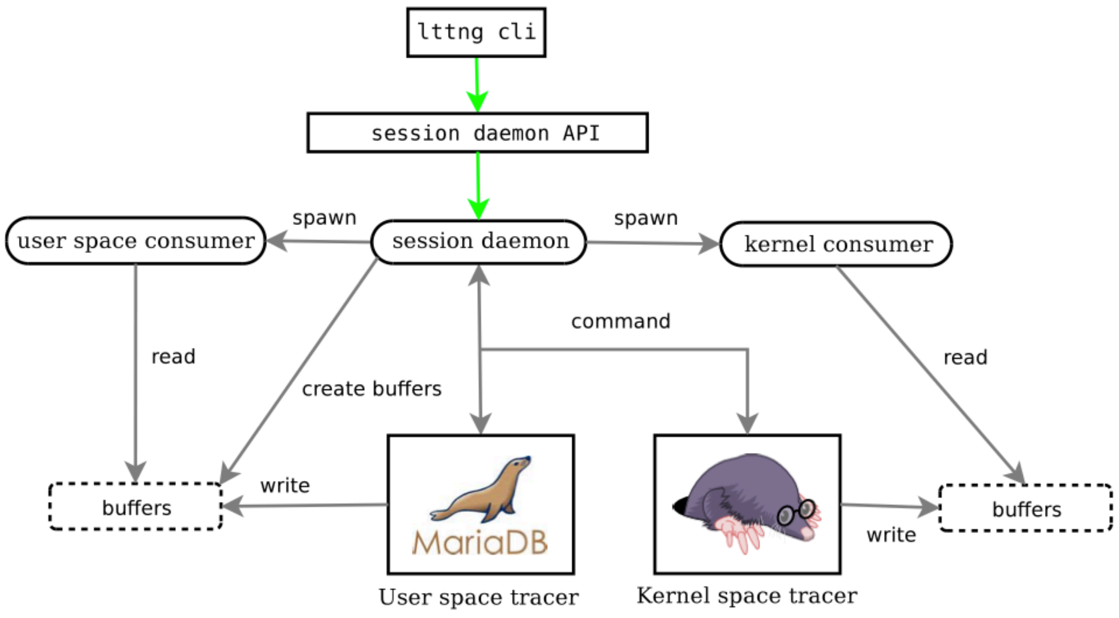


Figure 3.1: LTTng Architecture

Tracing sessions are used to isolate users from each other and create coherent tracing data between all tracing sources (Ex: MariaDB vs Kernel). This *session daemon* routes user commands to the tracers and keeps an internal state of the requested actions. The daemon makes sure that this internal state is in complete synchronization with the tracers, and therefore no direct communication with the tracers is allowed other than via the session daemon. This daemon is self-contained between users. Each user can run its own session daemon but only one is allowed per user. No communication happens between daemons.

Consumer daemons extract data from buffers containing recorded data and write it to disk for later analysis. There are two separate consumer daemons, one handling user space and the second one the kernel. A single consumer daemon handles all the user space (and similarly for kernel space) tracing sessions for a given session daemon. It is the session daemon that initiates the execution of the user space and kernel consumer daemons and feeds them with tracing commands.

LTTng internals define and make use of the following concepts in order to create an abstraction layer between the user and the tracers.

Domains are essentially a type of tracer or tracer/feature tuple. Currently, there are two domains in lttng-tools. The first one is `UST` which is the global user space domain. Channels and events registered in that domain are enabled on all current and future registered user space applications. The other domain is `KERNEL`. Three more domains are not yet implemented but are good examples of the tracer/feature concept. They are UST PID for specific PID tracing, UST EXEC NAME based on application name and UST PID FOLLOW CHILDREN which is the same as tracing a PID but follows spawned children.

Session is an isolated container used to separate tracing sources and users from each other. It takes advantage of the session feature offered by the tracer. Each tracing session has a human readable name (Ex.: myapps) and a directory path where all trace data is written. It also contains the user UID/GID, in order to handle permissions on the trace data and also determine who can interact with it. Credentials are passed through UNIX socket for that purpose.

Event relates to a TRACE EVENT statement in your application code or in the Linux kernel instrumentation. Using the command line tool `lttng`, you can enable and disable events for a specific tracing session on a per domain basis. An event is always bound to a channel and associated tracing context.

Channel is a pipe between an information producer and consumer. They existed in the earlier LTTng tracers but were hardcoded and specified by the tracer. In the new LTTng 2.0 version, channels are now definable by the user and completely customizable (size of buffers, number of subbuffer, read timer, etc.). A channel contains a list of user specified events (e.g. system calls and scheduling switches) and context information (e.g. process id and priority). Channels are created on a per domain basis, thus each domain contains a list of channels that the user creates. Each event type in a session can belong to a single channel. For example, if event A is enabled in channel 1, it cannot be enabled in channel 2. However, event A can be enabled in channel 2 (or channel 1 but not both) of another session.

3.2 Buffering scheme

In this part we analyze the buffering scheme employed by LTTng for efficient tracing.

As mentioned, a channel is a pipe between an information producer and consumer. It serves as a buffer to move data efficiently. It consists of one buffer per CPU to ensure cache locality and eliminate false-sharing. Each buffer is made of many sub-buffers where slots are reserved sequentially. A slot is a sub-buffer region reserved for exclusive write access by a probe. This space is reserved to write either a sub-buffer header or an event header and payload. Figure 3.2 shows space being reserved. On CPU 0, space is reserved in sub-buffer 0 following event 0.

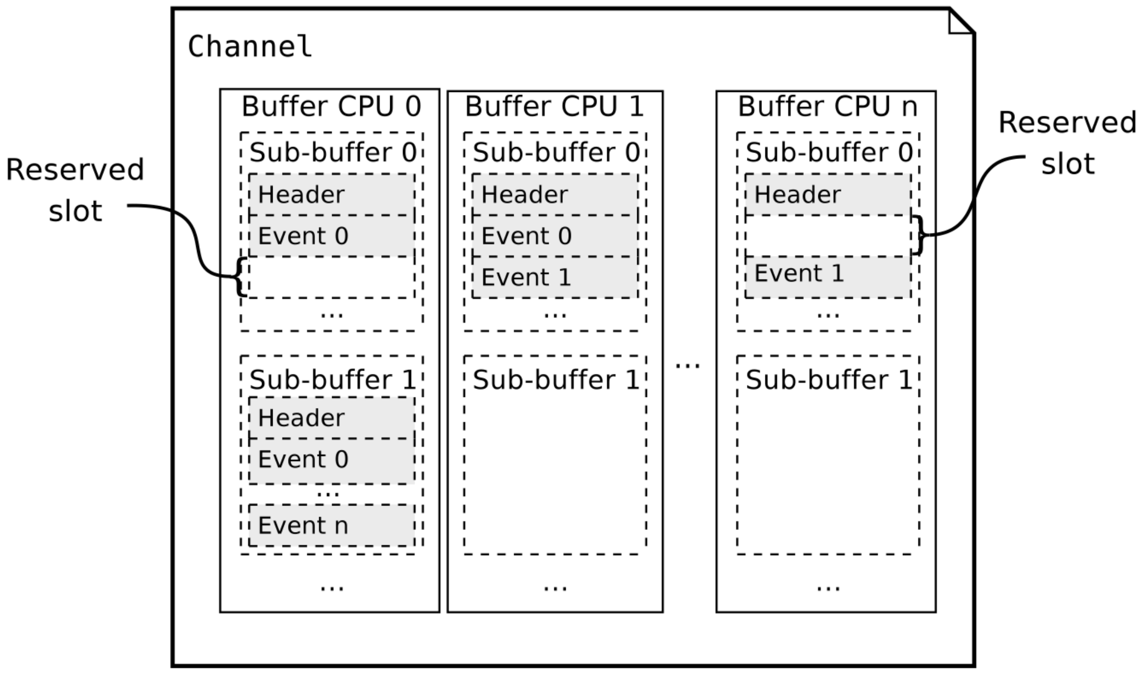


Figure 3.2: Channel layout

In this buffer, the header and event 0 elements have been complelety written to the buffer. The grey area represents slots for which associated commit count increment has been done. Committing a reserved slot makes it available for reading. On CPU n, a slot is reserved in sub-buffer 0 but is still uncommitted. It is however followed by a committed event. This is possible due to the non serial nature of event write and commit operations. This situation happens when execution is interrupted between space reservation and commit count update and another event must be written by the interrupt handler. Sub-buffer 1, belonging to CPU 0, shows a fully committed sub-buffer ready for reading.

Events written in a reserved slot are made of a header and a variable-sized payload. The header contains information containing the time stamp associated with the event and the event type (an integer identifier). The event type information allows parsing the payload and determining its size. The maximum slot size is bounded by the sub-buffer size. Both the number of the sub-buffers and their size can be configured by the `lttng` command line tool.

In order to synchronize the producer and consumer scheme, LTTng makes use of atomic operations. The two atomic instructions required are the `CAS` (Compare-And-Swap) and a simple atomic increment. Each per-CPU buffer has a control structure which contains the *write count*, the *read count*, and an array of *commit counts* and *commit seq counters*. The counters *commit count* keep track of the amount of data committed in a sub-buffer using a lightweight increment instruction. The *commit seq* counters are updated with a concurrency-aware synchronization primitive each time a sub-buffer is filled. The read count is updated using a standard SMP-aware `CAS` operation. This is required because the reader thread can read sub-buffers from buffers belonging to a remote CPU.

In the next two sections we will present how tracing is achieved in the different domains, kernel and user space.

3.3 Kernel-space tracing

In the previous section we described the buffering scheme used by LTTng. In this chapter we will analyze the kernel mechanism that enables LTTng to add a minimum overhead to the instrumented application during tracing or when the tracing is stopped.

In order to trace the Linux kernel with minimum overhead and without hurting the performance when the tracing is disabled, the equivalent mechanism should be provided by the kernel. The initial approach of *Kprobes*[24] Kprobes are a hardware breakpoint-based instrumentation approach. The Kprobe infrastructure dynamically replaces each kernel instruction to instrument with a breakpoint, which generates a trap each time the instruction is executed. A tracing probe can then be executed by the trap handler. However, due to the heavy performance impact of breakpoints, the inability to extract local variables anywhere in a function due to compiler optimizations, and the maintenance burden of keeping instrumentation separate from the kernel code, a more elaborate solution was needed.

This solution was given by Mathew Desnoyers the Linux Kernel Markers [9] and Tracepoints infrastructure. The markers and tracepoints allow us to declare instrumentation statically at the source-code level without affecting performance significantly and without adding the cost of a function call when instrumentation is disabled. A marker placed in code provides a hook to call a function (probe) that can be provided at runtime. A marker can be 'on' (a probe is connected to it) and the function is called so information is logged, or 'off' (no probe is at- tached). When a marker is 'off' it has no effect, except for adding a tiny time penalty (checking a condition for a branch). This instrumentation mechanism enables the instrumenta- tion of an application at the source-code level. Markers con- sists essentially of a C preprocessing macro which adds, in the instrumented function, a branch over a function call. By doing so, neither the stack setup nor the function call are ex- ecuted when the instrumentation is not enabled. At runtime, each marker can be individually enabled, which makes the branch execute both the stack setup and the function call

Having extremely low-overhead when instrumentation is dynamically disabled is crucially important to provide Linux distributions the incentive to ship instrumented programs to their customers. Markers and tracepoints consist in a branch skipping a stack setup and function call when instrumentation is dynamically disabled (dormant). These individual instrumentation sites can be enabled dynamically at runtime by dynamic code modification, and only add low overhead when tracing. The typical overhead of a dormant marker or tracepoint has been measured to be below 1 cycle [12] when cache-hot. Static declaration of tracepoints helps manage this instrumentation within the Linux kernel source-code. Given that the Linux kernel is a distributed collaborative project, enabling each kernel subsystem instrumentation to be maintained by separate maintainers helps distributing the burden of managing kernel-wide instrumentation.

However, statically declaring an instrumentation site for dynamic activation typically incurs a non-zero performance overhead due to the test and branch required to skip the instrumentation call. To overcome this limitation, Desnoyers created the concept of activating statically compiled code efficiently by dynamically modifying an immediate operand within an instruction, which is called Immediate Values [12]. This mechanism replaces the standard memory read, loading the condition variable, by a constant folded in the immediate value encoding of an instruction operand. This removes any data memory access to test for disabled instrumentation by keeping all the information encoded in the instruction stream. However, this involves dynamically modifying code safely against concurrent multiprocessor accesses. This requires either stopping all processors for the duration of the modification, or using a more complex, yet more lightweight, core synchronization mechanism. The choice made was the temporary breakpoint bypass [18].

In order to overcome a Kernel Markers' drawback, which was the limited type verification to scalar types due because its API is based on format string, *Tracepoints* [3] were created.

Two elements are required for tracepoints :

- A tracepoint definition, placed in a header file.
- The tracepoint statement, in C code.

In order to use tracepoints, you should include `linux/tracepoint.h`.

Define an event in `include/trace/events/subsys.h` as shown in Listing 3.1. You can use the Tracepoint within kernel code as shown in Listing 3.2.

As far as LTTng is concerned, the traced data is entirely controlled by the kernel. According to [11], the kernel exposes a transport pipeline (Ex: character device or anonymous file descriptor) and a user space daemon (session daemon) simply extracts data through this mechanism. This mechanism is based on `DebugFS` [4]

3.4 User-space tracing

User-space tracing needs a different approach from kernel-tracing. Approaches like SystemTap[5] or DTrace[6] based user-space tracing on breakpoints or system-calls whenever a tracing point is reached. However, this has a severe performance impact on the instrumented application and makes them inappropriate for system monitoring.

[3] https://www.kernel.org/doc/Documentation/trace/tracepoints.txt
[4] https://www.kernel.org/doc/Documentation/filesystems/debugfs.txt
[5] https://sourceware.org/systemtap/
[6] http://dtrace.org/blogs/

```c
#undef TRACE_SYSTEM
#define TRACE_SYSTEM subsys

#if !defined(_TRACE_SUBSYS_H) || defined(TRACE_HEADER_MULTI_READ)
#define _TRACE_SUBSYS_H

#include <linux/tracepoint.h>

DECLARE_TRACE(subsys_eventname,
        TP_PROTO(int firstarg, struct task_struct *p),
        TP_ARGS(firstarg, p));

#endif /* _TRACE_SUBSYS_H */

/* This part must be outside protection */
#include <trace/define_trace.h>
```

Listing 3.1: Kernel event definition

```c
#include <trace/events/subsys.h>

#define CREATE_TRACE_POINTS
DEFINE_TRACE(subsys_eventname);

void somefct(void)
{
    ...
        trace_subsys_eventname(arg, task);
    ...
}
```

Listing 3.2: Kernel Tracepoint activation

During BlkKin implementation, we tried to implement a custom tracing mechanism based on a memory-mapped ring buffer. However, this mechanism should handle with all the consumer-producer concurrency issues. Inspecting the LTTng user-space tracer, we found out that the aforementioned buffering scheme (3.2 is implemented for user-space tracing as well. This mechanism is not based on breakpoints or system-calls and does not affect the system's performance. So we decided to base out backend on LTTng ust-trace.

While the kernel tracer is the most complex entity in terms of code and algorithms, it is the simplest to handle. For the session daemon, this tracer is a single tracing source. However, tracing in user-space sets challenges concerning multiple users and concurrency. D. Goulet in his master thesis [17] created the `lttng-tools` project, which provides the needed unified user and kernel tracing. This project handles with all the issues concerning multiple concurrent tracing sources and the mechanism for their synchronization.

Since all these problems are handled by LTTng, in this section we will describe the mechanism behind a single tracing session.

As seen in Figure 3.3, each instrumented application creates a dedicated thread for tracing. This thread communicated with the sessiond over a UNIX-domain socket. The creation of this dedicated thread is created when the instrumented application is launched. Its creation is coded within functions labeled with `__attribute__((constructor))`. The instrumented applications are dynamically linked with the ust libraries. So, when the object files are loaded, the specific code is executed and the threads are

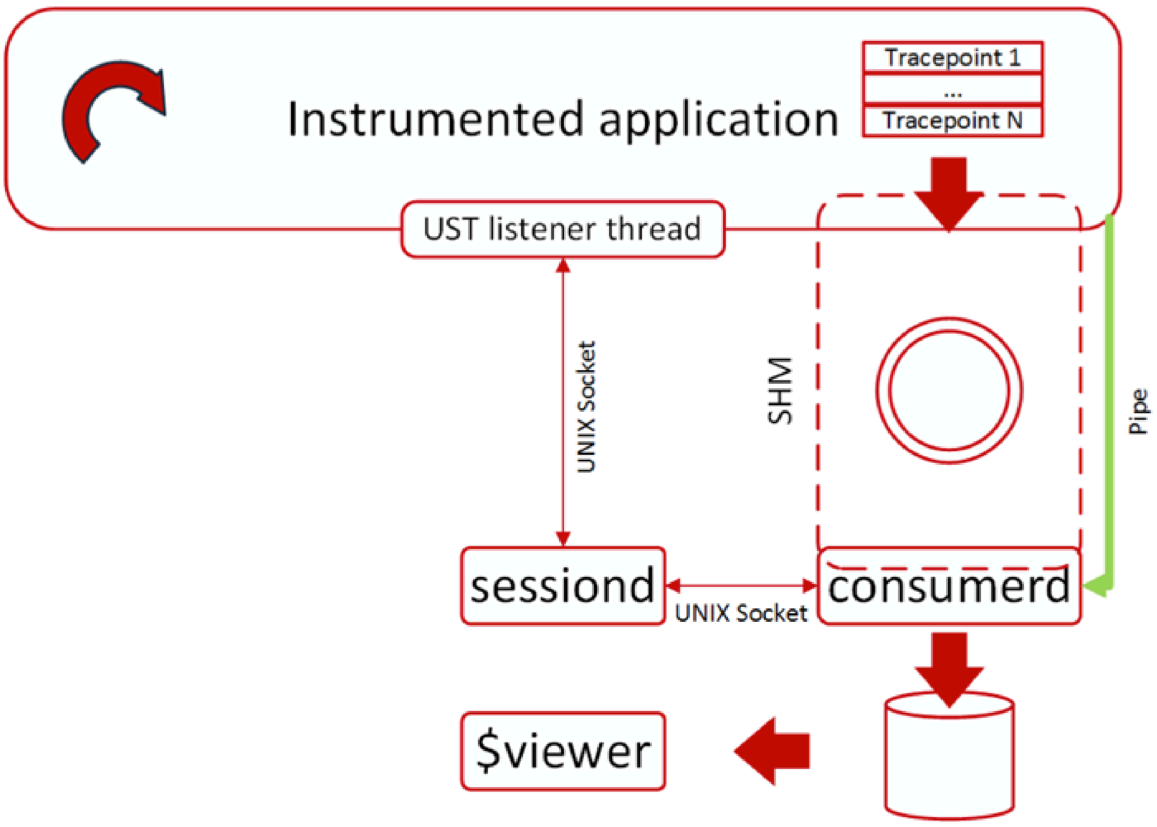


Figure 3.3: User-space tracer architecture

created. The session daemon communicates with the consumer over a UNIX-domain socket. Over this path pass all the control messages. For example, over these UNIX sockets pass the file descriptors of the shared memory segment, so that the consumer and the instrumented application refer to the same segment. The elaborate buffering scheme is deployed on a shared memory segment. The synchronization issues for the access to the segment are handled by the `liburcu`[7]. Whenever there are data available, the instrumented application notifies the consumer over a UNIX pipe. After that the consumer (which is different from the kernel consumer), writes the tracing data to a local folder. The tracing data will be available for viewing using viewers like Babeltrace[8] or LTTTV[9] only after the end of the session. This will be furthered discussed in Section 3.5.

The mechanism that supports the Tracepoints was ported in user-space as well, as mentioned in [6]. The user-space Tracepoints are defined in a header file. This file is compiled into an object file, which is finally linked along with the `liblttng-ust` with the instrumented application. So, the tracing threads will be created as mentioned and the tracepoint function calls will trace information only when tracing is enabled.

3.5 Common Trace Format (CTF)

LTTng makes use of *Common Trace Format* (CTF)[10] for the traces created. CTF is a trace format based on the requirements of the industry. It is the result of the collaboration between the Multicore

[7] https://lttng.org/urcu
[8] https://lttng.org/babeltrace
[9] https://lttng.org/lttv
[10] http://www.efficios.com/ctf

assosication[11] and the Linux community. This format was created to cover the tracing needs from versatile communities like the embedded, telecom, high-performance and kernel communities. It is a high-level model meant to be an industry-wide, common model, fulfilling the tracing requirements. It is meant to be application-, architecture-, and language-agnostic. One major element of CTF is the Trace Stream Description Language (TSDL) which flexibility enables description of various binary trace stream layouts. The CTF format is formally described in RFC.

According to this abstract model:

A *trace* is divided into multiple event streams. Each event stream contains a subset of the trace event types. The final output of the trace, after its generation and optional transport over the network, is expected to be either on permanent or temporary storage in a virtual file system. Because each event stream is appended to while a trace is being recorded, each is associated with a distinct set of files for output. Therefore, a stored trace can be represented as a directory containing zero, one or more files per stream.

An *event stream* can be divided into contiguous event packets of variable size. An event packet can contain a certain amount of padding at the end. The stream header is repeated at the beginning of each event packet

CTF offers a variety of *data types* for tracing, like integers, arrays or strings, which are defined in the RFC. These types allow inheritance so that other types can be derived.

So an event representation in CTF consists of

The overall structure of an event is:

Event Header
: (as specified by the stream meta-data). These information are the same for all streams in the trace. Example information: trace UUID

Stream Event Context
: (as specified by the stream meta-data) The stream context is applied to all events within the stream. Example information: pid, payload size

Event Context
: (as specified by the event meta-data) The event context contains information relative to the current event. Example information: missing fields

Event Payload
: (as specified by the event meta-data) An event payload contains fields specific to a given event type

Each trace is associated with some metadata. For example, the trace stream layout description is located in the trace meta-data or the fields belonging to an event type are described in the event-specific meta-data. The meta-data is itself located in a separate stream identified by its name: ‘metadata’.

The fact that the trace metadata are located in a different stream, prevents an LTTng ‘local’ trace from being read (reliably) without stopping the tracing session. LTTng offers no guarantee that the metadata on disk contains all the layout information needed to read any packet previously flushed to disk. For example, a new application, instrumented with previously unknown events, could be launched and fill a buffer with events. That buffer would then be flushed to disk. At that point, there would be no guarantee that the lttng-sessiond would have had the chance to flush the updated metadata to disk. Thus, reading that trace would fail.

[11] http://www.multicore-association.org/

So, in order to read an LTTng CTF-formatted event, LTTng created `relayd` and enabled live tracing, which is furthered analyzed in Section 3.6.

3.6 Live tracing

In version 2.4.0 LTTng integrated live tracing support. Instead of waiting the end of the session in order to read the traces, lttng-live enabled developers to read the traces live while they are being created using Babeltrace.

In order to live read trace data, traces have to be streamed, even if the tracer and the viewer operate on the same machine. Live tracing is achieved through the use of special daemon called `relayd`. So, when creating the tracing session, you can define whether you prefer live tracing and then you have to provide the IP address of the realyd which runs on the remote machine. When the session starts, relayd stores data to the remote machine, so that they will be saved after the end of the session. In order to view the traces, you have to use Babeltrace which connects to the relayd and prints text data to the stdout, when they arrive over the network. Again Babeltrace can run on different machine from the one being traced and the one where relayd runs.

LTTng relayd handles with the previous-mentioned metadata inconsistencies. Whenever new events appear, when a new instrumented application is launched for example, the relayd updates the metadata accordingly. As a result, the viewer (Babeltrace) receives from relayd a data packet with the actual tracing information and an index packet to properly locate the information. The updated metadata are also streamed to the client in a separate stream, as already mentioned. At any point, the live client must have all the metadata associated with the data packets it receives. The resulting interconnection is seen in Figure 3.4.

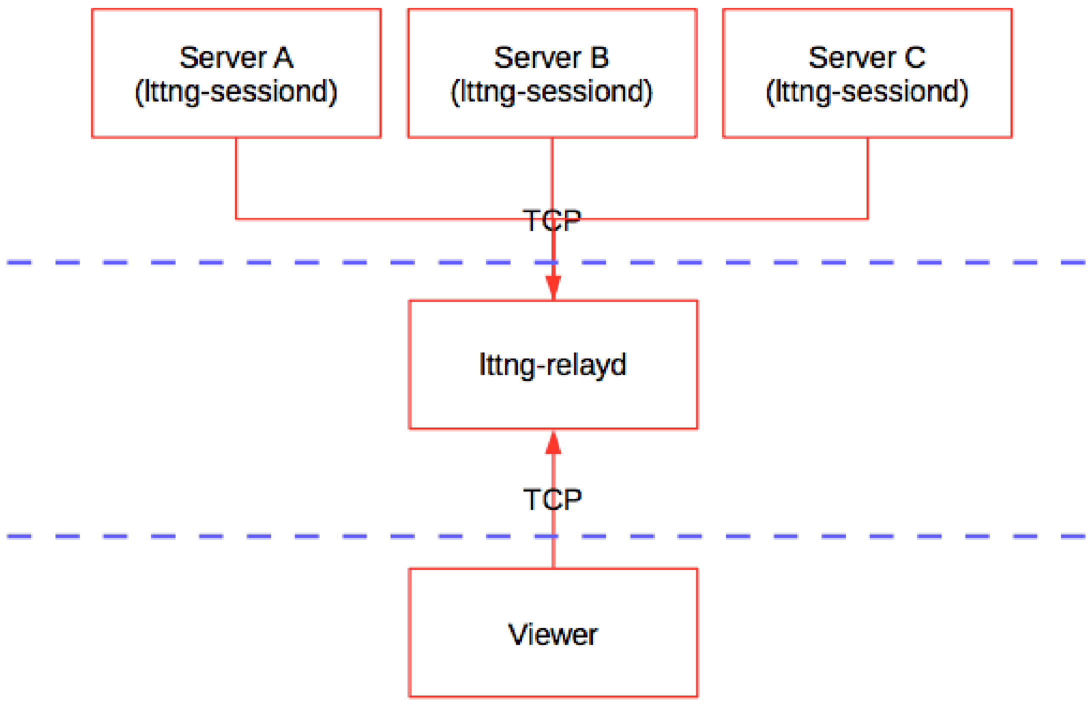


Figure 3.4: LTTng live tracing

Chapter 4

Tracing semantics

Apart from the mechanism that enable tracing, it is very important to choose what kind of information concerning the application is finally logged. A wise choice will ease the process of data correlation and assumption extraction. In this chapter, we cite the different schools behind tracing semantics, we analyze our tracing choice and finally we discuss how this choice is implemented.

4.1 Data correlation

Data resulting after tracing can be very bulky. Consequently, the process of extracting the information needed that triggered tracing is challenging. Data should be correlated and only the needed parts of the logs should be isolated and processed in order to extract meaningful information. This requires that tracing data are capable of being correlated. By data correlation we refer to data that refer to a specific subsystem or a specific request grouping.

Although there have been proposed many different tracing schemes, according to specific applications' needs, all these schemes can be summarized into two categories. According to Google's Dapper paper[29] these categories are:

black-box schemes [4, 28] assume there is no additional information other than the message record described above, and use statistical regression techniques to infer that association.

annotation-based schemes [5, 14] rely on applications or middleware to explicitly tag every record with a global identifier that links these message records back to the originating request.

While black-box schemes are more portable than annotation-based methods, they need more data in order to gain sufficient accuracy due to their reliance on statistical inference. The key disadvantage of annotation-based methods is, obviously, the need to manually instrument programs by adding instrumentation points in their source code.

As mentioned, BlkKin wants to achieve an end-to-end tracing so that latencies and faults between the different subsystem layers to become obvious. So, we decided to use an annotation-based tracing schema.

4.2 Dapper tracing concepts

Dapper[29] is a large scale distributed systems tracing infrastructure created by Google. It uses an annotation-based tracing scheme, which enables Google developers to monitor Google infrastructure only by instrumenting a small set of common libraries (RPC system, control flow). Although it is closed source, the tracing semantics used in Dapper are publicly available and have been used in BlkKin's development. Indeed, Google proposed a complete annotation-based scheme, which describes the following concepts 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 processing phase can be depicted as a different span. Each span should have a specific name and a distinct span id. It is important to note that each span can contain information from multiple hosts.

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

The previous concepts fit out demands for end-to-end tracing. So, we implemented them in a tracing library for C/C++ applications, which is described thoroughly in Chapter 1.1.

4.3 Zipkin: a Dapper open-source implementation

Dapper does not only describe the tracing semantics mentioned before, but is a full stack tracing infrastructure which includes subsystems to aggregate data per host, a central collector, a storage service and a user interface to query across the collected information. BlkKin, also has the same needs. So, to cover some of them, instead of rewriting the needed subsystems, we decided to use *Zipkin*.

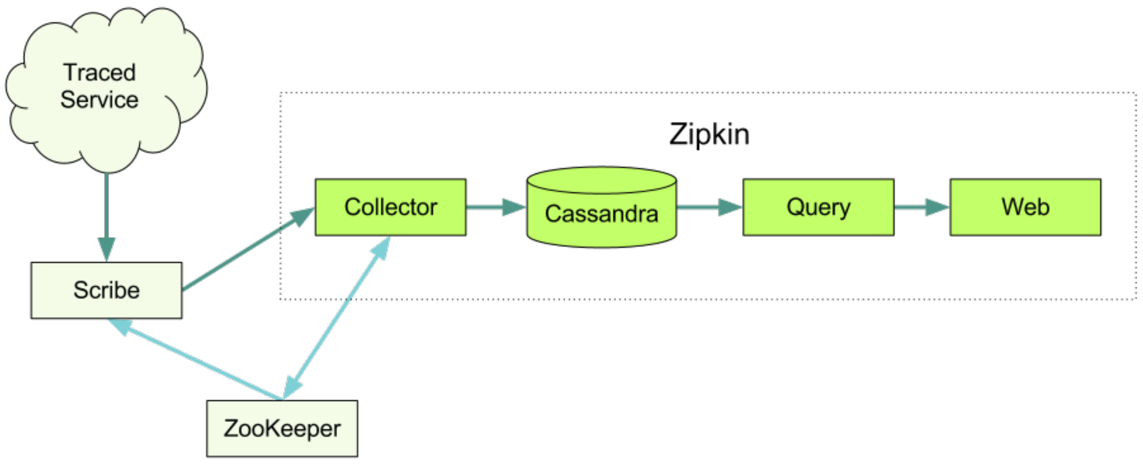


Figure 4.1: Zipkin Architecture

Zipkin[1] is an open-source implementation of the Dapper paper by Twitter. It is used to gather timing data for all the disparate services at Twitter. It manages both the collection and lookup of this data through a Collector and a Query service as well as the data presentation through a Web UI. Zipkin

[1] http://twitter.github.io/zipkin/

is written in Scala, while the UI is written in Ruby and Javascript using the D3.js[2] framework. So, Zipkin is a full-stack systems that encapsulates the Dapper tracing semantics out of the box. This is why we chose to use Zipkin.

Concerning transportation, Zipkin uses Scribe, which enables Zipkin to Scale. So, in order to feed Zipkin with data a Scribe client is needed. As metioned in Chapter 2.4 about Scribe, a category and a message is needed to log to Scribe. Zipkin messages are also Thrift encoded so that the collector and treat them and add them in the database. Zipkin thrift messages are encoded according to the following thrift file (Listing 4.1)

This thift file defines:

Endpoint is the location when an annotation took place. An endpoint is identified by its name, ip and port.

Annotation is the tracing information itself, exactly like the Dapper annotation

Span is also the Dapper span identified by its name, id, trace and parent ids and can contain multiple annotations.

Concerning the final data storage Zipkin provides various choices including SQL databases like SQLite, MySQL, and PostgreSQL as well as NoSQL databases like Cassandra and even Redis. However, like Twitter, we preferred to use Cassandra for our installation because of its performance.

So, the resulting Zipkin architecture can be seen in Figure 4.1

[2] http://d3js.org/

```
//************** Collection related structs **************

// these are the annotations we always expect to find in a span
const string CLIENT_SEND = "cs"
const string CLIENT_RECV = "cr"
const string SERVER_SEND = "ss"
const string SERVER_RECV = "sr"

// this represents a host and port in a network
struct Endpoint {
  1: i32 ipv4,
  2: i16 port                      // beware that this will give us negative ports. some conversion needed
  3: string service_name           // which service did this operation happen on?
}

// some event took place, either one by the framework or by the user
struct Annotation {
  1: i64 timestamp                 // microseconds from epoch
  2: string value                  // what happened at the timestamp?
  3: optional Endpoint host                 // host this happened on
}

enum AnnotationType { BOOL, BYTES, I16, I32, I64, DOUBLE, STRING }

struct BinaryAnnotation {
  1: string key,
  2: binary value,
  3: AnnotationType annotation_type,
  4: optional Endpoint host
}

struct Span {
  1: i64 trace_id                  // unique trace id, use for all spans in trace
  3: string name,                  // span name, rpc method for example
  4: i64 id,                       // unique span id, only used for this span
  5: optional i64 parent_id,                // parent span id
  6: list<Annotation> annotations, // list of all annotations/events that occured
  8: list<BinaryAnnotation> binary_annotations // any binary annotations
}
```

Listing 4.1: Zipkin message thrift definition file

Chapter 5

BlkKin Design

In the previous chapters we described the various challenges faced when dealing with distributed tracing and tools that can be used in order to achieve our tracing goals. In this chapter we cite the design of the tracing infrastructure called BlkKin. The name comes from the amalgamation of *Block storage* and *Zipkin*, which is one of the used building blocks and was described thoroughly in Section 4.3. BlkKin uses different open-source technologies and is designed to scale.

By building BlkKin we wanted to create a tracing infrastructure intended to cover the tracing needs created in software defined and distributed storage systems (but of course not restricted to them). After investigating the various needs that this kind of systems and the people developing and monitoring them have, we tried to summarize the points that are needed for our tracing infrastructure. We defined the following prerequisites that should be present in BlkKin:

low-overhead tracing
The traced system should be able to continue working in production scale serving real workloads in order to locate deficiencies and faults that are not obvious in debugging or tracing mode.

live-tracing BlkKin should be able to send traces at the time the are being produced so that the developer or the administrator can have an overview of the system at that specific time.

Dapper tracing semantics Tracing logic should be implemented in accordance with the concepts used by Dapper so that causal relationships and cross-layer architecture are depicted.

User interface BlkKin should provide various endpoints for the end user to collect and analyze data. One of those should be a graphical user interface that gives a graphical overview of the system's performance per specific layer.

In the following sections we will step by step examine BlkKin. Specifically, in Section 5.1 we will describe the rationale behind Blkin, while Section 5.2 goes through BlkKin's building blocks and Section 5.3 analyzises the BlkKin contributions. Finally, in Section 5.4 we illustrate the resulting BlkKin architecture and the flow of a traced request from the time created in the instrumented infrastructure until it ends up handled by BlkKin and stored.

5.1 Design rational

Since distributed systems follow a service-oriented architecture and consist of different software layers communicating with each other while running on different cluster nodes, we have to implement a tracing architecture that provides distributed tracing on each node, but collects all tracing data to a central repository. Thus, we can discover the relationships between the different layers. Otherwise, it would be impossible to find out what actions did a specific event on a specific node triggered throughout the cluster.

So, imitating the monitoring systems architecture as described in Section 2.4, BlkKin consists of the following parts

tracing agent
This agent runs on every cluster node. It is responsible for capturing traces both from user and kernel-space. Is supposed to add minimum overhead to the instrumented application.

central data collector The tracing agents from all the cluster nodes that we are interested about, should send the aggregated tracing information to a central collecting place so that they can be correlated. This system includes both the collecting part, a server that receives the data, and a storage system (database, file system), where information is finally kept.

User interface The aggregated information should be available through a Web user-interface that depicts the correlation between the systems' distinct software layers. Through that interface the user should be able to locate the information he wants and make basic queries. Also, this interface should be able to be used as a monitoring and alerting tool in case something misoperates.

5.2 BlkKin building blocks

After having identified our basic needs for BlkKin and respecting the proverb *'not to reinvent the wheel'*, we turned to the open-source community to find projects that met our requirements and we could combine to build BlkKin.

First we came across *Zipkin* (see Section 4.3 for more). Zipkin implements the Dapper semantics and provides a mechanism for data aggregation, data storage and a Web UI. So, we decided to employee Zipkin. Also, another crucial factor in favor of using Zipkin was the fact that it makes use of Scribe as a collector server. This is important because instead of storing tracing information in a database, we can store them in HDFS and run Map-Reduce jobs on them. An mentioned, tracing is notorious for the amount of data it produces. In order to manipulate that amount of data, Twitter engineers used distrubuted NoSQL databases and especially Cassandra. However, data from Zipkin that are stored in Cassandra follow a specific indexing pattern that is created in accordance with Zipkin's quering needs. This pattern makes hard (or even prevents) to run ad-hoc custom queries to extract any kind of correlation or information such as average values. Even Twitter uses Hadoop for these purposes. So we could use Zipkin for some visualizing purposes and HDFS for custom data manipulation using the same collecting mechanism and without the need to change data for the one or the other purpose. The same data stored in the Zipkin database and used to depict the causal relationships in the Web UI, can be stored in HDFS and investigated through Map-Reduce jobs.

Having covered the data collection and storage and the user-interface part, we had to create or use a system for the tracing agent. After some some custom endeavours to create such a system, we concluded that in order to be fast, this system has to use memory-mapped IO and specifically a ring buffer where a producer and a consumer exchange data. We created such a mechanism using a shared memory segment where the instrumented application wrote binary data and another daemon consumed them. However, we found out that to build such a mechanism, we had to solve multiple synchronization and concurrency problems and the situation would become more difficult with multi-threaded applications. Since BlkKin was designed to cope with production scale environments, we searched for a tested, open-source technology that covers that need. So, we decided to use LTTng. An mentioned in Section 3.4, LTTng enables us to trace both kernel and user-space applications with the same infrastructure. Since LTTng uses tracepoints inserted in the source code, by using it we can deploy any tracing logic we want. In our case, we had decided beforehand to use an annotation-based logic through the Dapper semantics, custom instrumentation was exactly what we needed. Also, LTTng has live tracing support, which could make use of. So, instantly another prerequisite was covered. Finally, like Syslog, LTTng implements different severity levels, namely a tracepoint can be considered emergency or waring for example. So, this enabled us to create different tracepoint with different severity levels and in case of live monitoring enable only the most basic ones, while in case of extensive tracing all tracepoints should be logged.

Finally, since we design BlkKin to be a distributed tracing infrastructure, we should take special care about the time skews among the cluster node clocks. When tracing a single node, a single clock is used. However, distributed applications create challenges concerning clock synchronizations. If we do not care about clock synchronization, there is serious possibility to find a request reply virtually taking place before the request itself, because there is time skew between the sever and the client. Older approaches ([2]) used post-processing techniques in order to adjust the time skews. According to these techniques, each cluster node collected tracing information based on its local clock, while a specific cluster node is considered anchor. During the tracing session, periodically each host sends an echo message to the anchor, and the anchor replies with the sender's timestamp and the anchor's current timestamp. The sending host receives the reply and records a time measurement consisting of three timestamps: send, remote (i.e., anchor), and reply. Relying on the fact that the network communication time is the same for sending and replying, we can compute the time skew. After the end of the tracing session these timeskews are interpolate to create the each host's time-skew line throughout the tracing session. Theses skews are applied to the logged timestamps before the events end up to the database. These approaches performed quite well, but their major disadvantages were the post-processing overhead to calculate the skew, which could be significant in case of long tracing sessions, and the fact that live-tracing was impossible due to this needed post processing.

However, when the previous methods were developed, NTPs performance was not acceptable for distributed tracing. According to [2] using NTP caused skews over 1000 μsec. In 2010 NTP version 4 was developed. According its RFC[1] NTPv4 NTPv4 includes fundamental improvements in the mitigation and discipline algorithms that extend the potential accuracy to the tens of microseconds with modern workstations and fast LANs. After experimenting with NTPv4 performance and accuracy, we decided that it was adequate for our tracing needs, namely there was no response happening before its request, only after a few hours of NTP syncing. One deployment that we also tried without remarkably better result, was to use a cluster node as NTP server, since we care only about relative timestamps and not absolute. Thus, exploiting the fast LAN we waited to have better synchronization results, but we concluded that even a global NTP server operated well.

5.3 BlkKin contribution

At that point, after having decided about the different building blocks that BlkKin would use, we had to find a way to make them communicate. In this section we will explain how we made the above systems communicating with each other in order to created a unified tracing infrastructure and what we needed to add in order to complete BlkKin as an end-to-end, unified tracing infrastructure. This contribution is returned back to the open-source community and the major part was created during my participation in the Google Summer of Code 2014 in the LTTng project, under the supervision of Jeremie Galarneau.

[1] http://tools.ietf.org/html/rfc5905

5.3.1 BlkKin library

Since we had decided on the tracing infrastructure, we had to find a way to trace low-overhead application in accordance with the Dapper semantics. Although Zipkin provides a variety of instrumentation libraries for languages such as Java, Scala or Python, there was no such library for C/C++. So we created a C/C++ library in accordance with the Dapper semantics which provides a useful API with all the functions that a programmer would need to instrument such an application. This API is further explained in 1.1. Although this library in backend-independent, which means that anybody could just keep the API and implement a custom tracing infrastructure, we implemented a backed based on LTTng. So, our library makes use of LTTng `tracepoint` in order to log the information we want. In addition, since we wanted to use BlkKin for monitoring purposes as well, BlkKin library should implement a kind of sampling. Otherwise, the amount of data created would be huge and the network traffic would be really high. Although a more elaborate mechanism for sampling could have been created, BlkKin currently implements only a rate sampling which means only 1/N root spans are actually initiated. Since we are talking about IO requests, the route of only one out of N requests will be actually traced.

5.3.2 Babeltrace plugins

As mentioned, Scribe uses Thrift as a communication protocol. So, we needed to implement a mechanism that would tranform LTTng data and send them to Scribe. LTTng data are encoded using the CTF format and Babeltrace is responsible for transforming these data to a human readable format. Consequently, we had to extend Babeltrace to communicate with Scribe. At first, we tried to implement this functionality within Babeltrace. So firstly we created a Scribe client written in C[2] which is used in Babeltrace. This version of Babeltrace[3] was abandoned because Babeltrace internal code architecture was hard to extend. Instead, we decided to use the Babeltrace Python bindings and deploy this functionality as a babeltrace plugin.

So, using these Python bindings and the `facebook-scribe` module from `pip` we created two different plugins. The first plugin in generic and send to Scribe any kind of LTTng data after transforming them in a *JSON format*. Using this plugin, we can avoid the tedious job of searching for information within log-files. Instead, we can send our data to Scribe and store them in HDFS. After that creating simple Map-Reduce jobs that read JSON encoded data can subtract any information we want. The motivation for this functionality came from the Facebook's equivalent tracing infrastructure called Scuba[4] [1].

Finally, LTTng live tracing supports only CTF to text transformation and the above plugins could not be used for live tracing. So we had to extend Babeltrace live support. Because at the spefic moment of BlkKin's development, Babeltrace underwent a generic code refactoring. After that refactoring the above plugins would work with live support as well. Consequently, we offered just an evanescent solution for LTTng live tracing for Zipkin data. According to this solution, LTTng is obliged to log only specific tracepoints. After that, Babeltrace sends these data to a Python consumer communicating over a named Pipe. Implementation details are further explained in Section 1.1.

5.3.3 BlkKin Monitoring UI

Although Zipkin's UI was adequate for investigating the correlations between the different storage layers, it didn't cover our needs as an alerting tool. So, we developed a simple Python-django[5] applica-

[2] https://github.com/marioskogias/scribe-c-client

[3] https://github.com/marioskogias/babeltrace/tree/scribe-client

[4] https://www.facebook.com/notes/facebook-engineering/under-the-hood-data-diving-with-scuba/10150599692628920

[5] https://www.djangoproject.com/

tion, what communicates with the Zipkin database. This application is responsible to gather particular metrics, mostly average values, such as the average network communication over the last 10 minutes, and present them accordingly. This application includes thresholds for these metrics. If the values are under the thresholds the UI shows them green, but when the metrics are over the threshold values, they are illustrated red. These threshold values are the result from elaborate data analysis over a wide series of data in the HDFS.

5.4 BlkKin architecture and data flow

In this section we present the overall BlkKin architecture and data flow. As we can see in Figure 5.1 we have an application written in C/C++ we are interested to trace or monitor. After we have identified the different application layers and decided how to implement the Dapper semantics for it, we instrument its source code using the BlkKik library. The host where the application runs, run LTTng daemons as well. So, whenever an instrumentation point is reached, a `tracepoint` logs the tracing information to LTTng. After that, depending on whether we are having a live tracing session or not, the CTF data will be send either to the `relayd` or to local storage. In case of live tracing, our version of Babeltrace will communicate with the `relayd`, get the CTF data, turn them into binary C-types and send them to the Python consumer over a named Pipe. Then this consumer will transform them into Scribe messages and send them to the Scribe server. In case of a non-live tracing session, the CTF data will end up to the local disk. After the end of the session, using our Babeltrace plugins, will transform them again into Scribe messages and send them to Scribe.

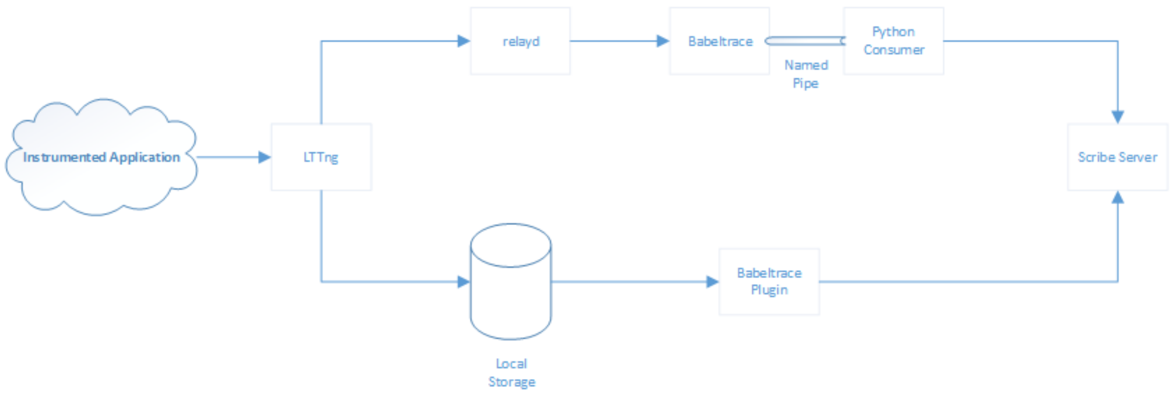


Figure 5.1: BlkKin Internal Communication

One important thing to mention is that, every communication arrow in Figure 5.1 is a TCP connection. This, in conjunction with the Scribe characteristics, gives us the ability for a variety of BlkKin deployments within the instrumented cluster. For example we can have multiple relayds and a single Scribe server. However, as it is going to be clarified in the Evaluation Section 1.1, the most appropriate and even suggested by Twitter deployment is the one illustrated in Figure 5.2. In this deployment, there is a whole BlkKin stack running on each cluster node, where the instrumented application runs. This includes a local Scribe daemon as well. This deployment enables us to take advantage of the Scribe batch messaging capability. All the CTF messages are send to relayd and Babeltrace over `localhost` which is faster and then end up to the local Scribe server. Scribe in optimized to batch messages or temporarily store them locally if the next Scribe server is unavailable. This lowers the network traffic and prevents us from data loss. Also, it enables us from changing the final data destination (Zipkin or HDFS) simply by changing the XML file from which each local Scribe daemon gets configured.

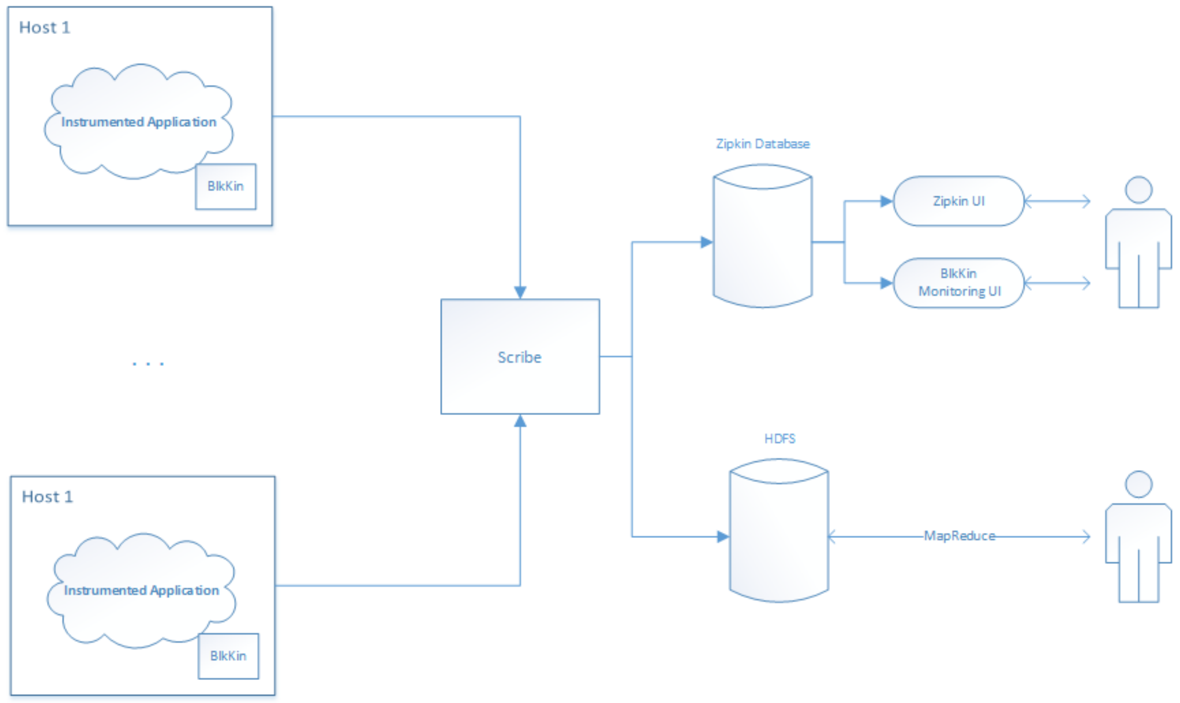


Figure 5.2: BlkKin Deployment

Chapter 6

BlkKin Implementation

In the previous chapter, we discussed how BlkKin was design to fulfil all the needed prerequisites. In this chapter, we will present how we implemented the BlkKin's interconnecting parts and the parts needed to use BlkKin and subtract useful information. There also included code snippets that clarify the implementation.

6.1 Instrumentation Library

As mentioned, we needed to implement a library in C/C++ that encapsulates the tracing semantics mentioned in Dapper. This API should give programmers the ability to perform any kind of tracing operation or correlation they want. Since Zipkin was designed for distributed Web services, the existing, equivalent Zipkin libraries for other languages, make use of HTTP headers in order to transport the correlating information. In fact, there are three distinct HTTP headers that travel along with the HTTP requests and used for tracing. These headers are `X-B3-TraceId`, `X-B3-SpanId` and `X-B3-ParentSpanId`. In our case, we have C/C++ applications communicating, so instead of these HTTP headers, we created the equivalent C-struct which includes the same information. This struct is the one in Listing 6.1.

```
/**
 * @struct blkin_trace_info
 * The information exchanged between different layers offering the needed
 * trace semantics
 * */
struct blkin_trace_info {
    int64_t trace_id;
    int64_t span_id;
    int64_t parent_span_id;
};
```

Listing 6.1: BlkKin basic struct

The above struct is exchanged between the different software layers and used for tracing their correlations.

According to Zipkin, in order to create a trace, you also need an `Endpoint` and a name for the trace. So, when an application receives or creates a new `struct blkin_trace_info`, it also creates a `blkin_trace` as seen in Listing 6.2 and then moves on to the other library operations.

After the above structs creation, the instrumented application could either create annotations (Listing 6.4), or create a *child* (Listing 6.3) of the current span. We have to mentioned that there is also a C++

```
/**
 * @struct blkin_endpoint
 * Information about an endpoint of our instrumented application where
 * annotations take place
 * */
struct blkin_endpoint {
    char *ip;
    int port;
    char *name;
};

/**
 * @struct blkin_trace
 * Struct used to define the context in which an annotation happens
 * */
struct blkin_trace {
    char *name;
    struct blkin_trace_info info;
    struct blkin_endpoint *trace_endpoint;
};
```

Listing 6.2: BlkKin trace struct

wrapper for these function calls, but is omitted since it shares exactly the same logic but implemented in an object-oriented way.

```
/**
* Initialize a new blkin_trace with the information given. The new trace
   will
* have no parent so the parent id will be zero.
*
* @param new_trace the blkin_trace to be initialized
* @param name the trace's name
* @param endpoint a pointer to a blkin_endpoint struct that contains info
    about
* where the specif trace takes place
*
* @returns 1 if success -1 if error
*/
int _blkin_init_new_trace(struct blkin_trace *new_trace, char *name,
        struct blkin_endpoint *endpoint);

/**
* Initialize a blkin_trace as a child of the given parent bkin_trace. The
    child
* trace will have the same trace_id, new span_id and parent_span_id its
* parent's span_id.
*
* @param child the blkin_trace to be initialized
* @param parent the parent blkin_trace
* @param child_name the blkin_trace name of the child
*
* @returns 1 if success -1 if error
*/
int _blkin_init_child(struct blkin_trace *child, struct blkin_trace *
   parent,
        struct blkin_endpoint *endpoint,
        char *child_name);

/**
* Initialize a blkin_trace struct and set the blkin_trace_info field to
   be
* child of the given blkin_trace_info. This means
* Same trace_id
* Different span_id
* Child's parent_span_id == parent's span_id
*
* @param child the new child blkin_trace_info
* @param info the parent's blkin_trace_info struct
* @param child_name the blkin_trace struct name field
*
* @returns 1 if success -1 if error
*/
int _blkin_init_child_info(struct blkin_trace *child,
        struct blkin_trace_info *info, struct blkin_endpoint *endpoint,
        char *child_name);
```

Listing 6.3: BlkKin child actions

```
/**
* @typedef blkin_annotation_type
* There are 2 kinds of annotation key-val and timestamp
*/
typedef enum {
    ANNOT_STRING = 0,
    ANNOT_TIMESTAMP
} blkin_annotation_type;

/**
* @struct blkin_annotation
* Struct carrying information about an annotation. This information can
   either
* be key-val or that a specific event happened
*/
struct blkin_annotation {
    blkin_annotation_type type;
    char *key;
    char *val;
    struct blkin_endpoint *annotation_endpoint;
};

/**
* Initialize a key-value blkin_annotation
*
* @param annotation the annotation to be initialized
* @param key the annotation's key
* @param val the annotation's value
* @param endpoint where did this annotation occured
*
* @returns 1 if success -1 if error
*/
int _blkin_init_string_annotation(struct blkin_annotation *annotation,
   char *key,
        char *val,
        struct blkin_endpoint * endpoint);

```

```
/**
* Initialize a timestamp blkin_annotation
*
* @param annotation the annotation to be initialized
* @param event the event happened to be annotated
* @param endpoint where did this annotation occured
*
* @returns 1 if success -1 if error
*/
int _blkin_init_timestamp_annotation(struct blkin_annotation *annot, char
    *event,
        struct blkin_endpoint * endpoint);

/**
* Log an annotation in terms of a specific trace
*
* @param trace the trace to which the annotation belongs
* @param annotation the annotation to be logged
*
* @returns 1 if success -1 if error
*/
int _blkin_record(struct blkin_trace *trace,
        struct blkin_annotation *annotation);
```

Listing 6.4: BlkKin annotations

All these mentioned structs include ids which are supposed to be unique not only per computer node, but per cluster as well, since we plan to implement distributed tracing. For our implementation, these ids are `uint_64` numbers that are randomly generated. In order to have a simple implementation, we use `rand()`. However, the normal procedure to feed it with the current timestamp failed for us, since we has multiple processes starting almost simultaneously on the same host and this resulted in all these services taking the same feed and producing the same ids. Instead, we feed `srand()` by reading from `\dev\urandom`.

After having defined the above API, we had to provide an implementation which is going to be based on LTTng. LTTng is activated only whenever the fucntion `record` is called. So, someone could keep the rest of the library and implement a custom tracing backend only by changing this function. Concerning the LTTng case, the `record()` function makes LTTng `tracepoint()` calls. These calls are predefined in a header file. This header file is used by LTTng to create the methods bodies and the object file which is finally linked to the final BlkKin object file, which in turn is linked with the instrumented application. In Listing 6.5 you can see how we defined the tracepoint for the key-value annotations. In this proof-of-concept version of BlkKin, all the tracepoints are considered WARNINGS, and the severity level configured is such, so that all of them are eventually logged. To avoid repetition, in the Listing only the key-value tracepoint is illustrated. The timestamp tracepoint is the same, but instead of key-value information, we have the event name.

LTTng assigns a timestamp to every event it records, so in case of the timestamp events we do not need to care about timing information by calling `gettimeofday()` for example. Instead, LTTng makes use of `CLOCK_MONOTONIC`, which is transformed to real timestamp during the reading process in Babeltrace. As part of the CTF metadata LTTng also sends the timestamp and the monotonic value from the time the session started, so that the real timestamp can be formed during reading.

As far as the sampling is concerned, in order not to trace all the requests, one has to export an environmental variable called `RATE`. This variable is an integer N which indicates that 1/N calls to `blkin_-init_new_trace` should actually create a new trace. This way we can regulate the amount of traces we produce.

```
TRACEPOINT_EVENT(
        zipkin,
        keyval,
        TP_ARGS(char *, trace_name, char *, service,
            int, port, char *, ip, long, trace,
            long, span, long, parent_span,
            char *, key, char *, val ),

        TP_FIELDS(
                /*
                 * Each span has a name mentioned on it in the UI
                 * This is the trace name
                 */
                ctf_string(trace_name, trace_name)
                /*
                 * Each trace takes place in a specific machine-endpoint
                 * This is identified by a name, a port number and an ip
                 */
                ctf_string(service_name, service)
                ctf_integer(int, port_no, port)
                ctf_string(ip, ip)
                /*
                 * According to the tracing semantics each trace should have
                 * a trace id, a span id and a parent span id
                 */
                ctf_integer(long, trace_id, trace)
                ctf_integer(long, span_id, span)
                ctf_integer(long, parent_span_id, parent_span)
                /*
                 * The following is the real annotated information
                 */
                ctf_string(key, key)
                ctf_string(val, val)
        )
)
TRACEPOINT_LOGLEVEL(
        zipkin,
        keyval,
        TRACE_WARNING)
```

Listing 6.5: BlkKin tracepoints

So, the BlkKin library provides a header file to be included in the source code and a dynamically linked object file to be linked with the application. This dll includes all the necessary LTTng functions as well. However, as we described in Section 3.4, normally the LTTng threads are created whenever the dll is loaded. This would cause problems for applications that fork(), because the child would not have its own LTTng threads to trace. So, instead, we used `dlopen` `dlsym` and manually load the BlkKin functions which in turn load the LTTng object file and create the needed threads whenever the function `blkin_init()` is called.

To sum up, we site an execution example in Listing 6.7 and its Makefile in Listing 6.6.

```
test: test.c $(DLIBFRONT).so
	gcc test.c -o test -g -I. -L. -lblkin-front

libblkin-front.so: blkin-front.o
	gcc -shared -g -o $@ $< -ldl

blkin-front.o: blkin-front.c
	gcc -I. -g -Wall -fpic -c -o $@ $<

libzipkin-c.so: zipkin_c.o tp.o
	gcc -shared -o $@ $^ $(LIBS)

zipkin_c.o: zipkin_c.c zipkin_c.h zipkin_trace.h
	gcc -I. -Wall -fpic -g -c -o $@ $<

tp.o: tp.c zipkin_trace.h
	gcc -I. -fpic -g -c -o $@ $<
```

Listing 6.6: BlkKin example Makefile

```c
#include <blkin-front.h>

int main() {
    r = blkin_init();
    if (r < 0) {
        fprintf(stderr, "Could␣not␣initialize␣blkin\n");
        exit(1);
    }

    /*initialize endpoint*/
    struct blkin_endpoint endp;
    blkin_init_endpoint(&endp, "10.0.0.1", 5000, "service␣a");

    /*Initialize trace*/
    struct blkin_trace trace;
    blkin_init_new_trace(&trace, "process␣a", &endp);

    /*Initialize annotation*/
    struct blkin_annotation ant;
    blkin_init_timestamp_annotation(&ant, "Test␣annotation", &endp);

    /*Log a timestamp event*/
    blkin_record(&trace, &ant);
}
```

Listing 6.7: BlkKin execution example

6.2 Babeltrace plugins implementation

In this section we will describe how we implemented the Babeltrace plugins that convert CTF data to Scribe messages and send them to the Scribe server. As mentioned, we implemented two different plugins one generic that sends JSON data to Scribe and one Zipkin-specific that creates Scribe messages that end up to Zipkin. Since Scribe messages are simple strings after all, both plugins share a common core that handles with the connections and message transportation. Each plugin implements a different message formation part which results to a string message to be sent to Scribe.

First we will explain how we extract the tracing information. Babeltrace exposes a Python library which is created using swig2.0[1]. According to this library, in order to read the CTF data you have to create a `Tracecollection` object and call the method `add_trace` on it passing the trace path. After that, a Python generator is created in `Tracecollection.events`. If we iterate over that generator, we can take the CTF event trace information formated as a Python object which has properties like the event name and the timestamp and another generator that returns the event information in the form of a tuple (item-name, item-value), namely the values passed to the `tracepoint` function call. After that we can manipulate the data the way we want.

The case of the JSON format is easy since after that we can easily create a Python dictionary and transform it into a JSON object which is ready to be forwarded to Scribe as it is already a string. On the other hand, in order to feed data to Zipkin the procedure is different. As mentioned in Section 4.3, Zipkin makes use of Thrift as well in order to create a binary representation of the events and the Thrift file used can be found in Listing 4.1. So, in order to create these messages, we used Thrift to create the Python code from the Zipkin Thrift file. Thrift created the Python classes needed. So, when obtained the tracing information with the method mentioned above, we passed them to the classes constructors and created the equivalent `Span`, `Annotation` and `Endpoint` objects. After that each span is encoded using the Thrift `TBinaryProtocol` and this value, in order to become a string, is base64 encoded. After this procedure, we have the final string to forward to Scribe.

After having formatted the messages, we have to send them to Scribe. To do that we used the Python module `facebook-scribe` from `pip`. Sending a message to Scribe is as simple as it seen in Listing 6.8.

```python
socket = TSocket.TSocket(host="HOST", port=9999)
transport = TTransport.TFramedTransport(socket)
protocol = TBinaryProtocol.TBinaryProtocol(trans=transport, strictRead=
    False, strictWrite=False)
client = scribe.Client(protocol)
transport.open()

category = 'zipkin'
message = 'hello␣world'

log_entry = scribe.LogEntry(category, message)
result = client.Log(messages=[log_entry])
if result == 0:
      print 'success'
```

Listing 6.8: Scribe messaging

In case we want to annotate on a subsystem that does not have distinct beginning and end, Zipkin provides some special annotations that are used to start and end a span. They are used only for instrumentation and visualization purposes. However, using these annotations annotations we can be sure that a span has ended and then forward a packed message to Scribe, namely a message that includes multiple annotations. This reduces the number of messages and consequently the network traffic and the server load. Depending on the instrumentation this may not be possible when having a span that collects annotations from multiple computer nodes in cases of distributed traces. So, the Zipkin plugin has the option to collect annotations, temporarily stored them in a dictionary using as a key the trace and span id pair and when a predefined annotation is asked to be logged then create a message including all the annotations for this specific span. In case this is not possible, the plugin just creates single-annotation messages and forwards them to Scribe.

[1] http://www.swig.org/

6.3 BlkKin Live tracing

Although, as already mentioned, the solution we gave for live tracing is only termporary since Babeltrace is under code refactoring at this specific moment and after this the above plugins will operate with the live tracing as well, for completion reasons, in this section we explain how we implemented live tracing for BlkKin.

Babeltrace is a CTF-to-text converter written in C. Instead of text we wanted to produce Scribe messages. So, at first we tried to implement this functionality within Babeltrace. For this reason we needed to implement a Scribe client written in C. Although we created a working prototype of Babeltrace sending messages to Scribe, this idea was abandoned after the urge of my GSoC mentor and started working with the plugins. Although we could redirect Babeltrace output to a process that would create the Scribe messages, we chose to avoid this CTF-to-text and text-to-Scribe conversion. Instead, we created a version of Babeltrace especially for BlkKin. This version instead of reading generic CTF-events is aware of the events' content as described in 6.5. Consequently, instead of iterating over the CTF trace for generic events, this version of Babeltrace extracts the included information and creates a C-struct with them as seen in Listing 6.9

```
struct zipkin_trace {
    char trace_name[20];
    char service_name[20];
    int port;
    char ip;
    long trace_id;
    long span_id;
    long parent_span_id;
    int kind; // 0 for timestamp 1 for key-val
    char key[20];
    char val[50];
    uint64_t timestamp;
};
```

Listing 6.9: Babeltrace live struct

After that this struct is forwarded to a Unix pipe. On the other side of the pipe there is a Python process. This process makes use of Python ctypes[2]. So whenever a struct is read, a Python object is created and from that point the Python consumer reuses the Zipkin Babeltrace plugin code to create the equivalent Span, Annotation and Endpoint objects and finally send the base64-encoded message to Scribe.

6.4 BlkKin Endpoints

6.4.1 BlkKin monitoring tool

As we mentioned in the design part, Zipkin covered a basic need for the user interface with its Web UI. However, this specific UI can be used mostly for trace visualization and not as a tool that can be used to detect abnormalities. Thus, we created a simple Python-django application. In this proof-of-concept implementation, this application communicates with the Zipkin databse which is a MySQL database. It queries the database for aggregate and average information that depict the cluster state over the last N minutes. After that either using a threshold for these values or judging on other criteria, it illustrates this information as green when everything is working properly, or red when there is a possible anomaly. There screenshots of this tool in the evaluation part (1.1).

[2] https://docs.python.org/2/library/ctypes.html

6.4.2 Hadoop

Scribe can be configured to store the data it receives in HDFS. So, we used Hadoop 0.21.0 to store tracing data from BlkKin for further analysis. The usecase scenario includes extensive tracing (not real time) without sampling and post-processing the tracing data in order to locate the thresholds and metrics used in the BlkKin monitoring tool. After that BlkKin can use sampling and send data to Zipkin.

In our described configuration, if we use the JSON Babeltrace plugin we end up having JSON data in the HDFS which can be manipulated easily with Map-Reduce jobs. However, if we use the Zipkin Babeltrace plugin, then in the HDFS we have multiple files containing base64 encoded strings. In order to extract the information wanted through Map-Reduce, we had to decode this information following the opposite direction. Again using Thrift and the Zipkin Thrift file we created the equivalent Java code which created the Annotation, Span and Endpoint classes in Java. Then after reading from HDFS, we base64 decoded the strings and then TBinaryProtocol decoded them. After that we could create the Span object. Since most of the data we wanted to extract where the duration between two specific annotations, we created a simple Java interface to help us. Whenever a Span with multiple annotations was read we had to define whether to keep the span or not and then to forward the annotation to the reducers or not. Then a key-value pair (id, timestamp) was emitted. The reducer's job was just to subtract the two different values belonging to the same id. This id is a string and its creation is also defined by the interface. The interface can be seen in Listing 6.10 and its up to the user how to implement it.

```java
public interface AnnotationChooser {
    /**
     * Function to decide or not to keep annotations from the specific
    span
     * @param s the specific span
     * @return true for keep false for the opposite
     */
    boolean shouldKeepSpan(Span s);
    /**
     * Function to decide whether to keep the specific annotation or not
     * @param a the annotation
     * @return true or false
     */
    boolean shouldKeepAnnotation(Annotation a);
    /**
     * This function returns the string value used as id for the emited
    timestamps
     * @param s The span
     * @param a The annotation
     * @return the string id
     */
    String getId(Span s, Annotation a);
}
```

Listing 6.10: Hadoop interface

Chapter 7

Evaluation

In this section we will describe our experience from using BlkKin in a real usecase senario. The instrumented infrastructure is described in Section 7.1. After that, in Section 7.3 we analyse performance metrics concerning the network and system overhead that justify our design and deployment choices. Finally, in Section 7.4 we explain how we used BlkKin to identify system faults which are virtually inserted by us, but reflect possible real failures or bottlenecks.

7.1 Instrumented infrastructure

As a use-case, we used BlkKin to instrument Archipelago and RADOS. These systems were examined in Sections 2.2 and 2.1.1 respectively. Archipelago is written is C and RADOS in C++. So we used the BlkKin library C++ wrapper for the RADOS instrumentation. Instead of using the `archip-bench` tool, which is part of Archipelago, to initiate IO requests, we instrumented the Qemu Archipelago driver. So, using Qemu we start a virtual machine which has an Archipelago drive and create different IO loads to this driver using `fio`[1]. Thus we can track the IO request from the time Qemu receives it until it is finally served by RADOS.

The Qemu Archipelago driver receives the IO requests from Qemu and creates XSEG requests for the VLMC. Qemu initiates the tracing information as well and Qemu spans are the root spans. After that, these tracing information is carried as part of the XSEG request. To do that, we needed to extend `libxseg`[2] and add the tracing information needed as shown in Listing 6.1 nested in the XSEG request. So, as far as Archipelago in concerned, the tracing information is transmitted as part of the XSEG request. Each Archipelago peer is considered a different service, with a different endpoint that creates a single span per IO request in the general case. So, in the Zipkin UI we expect to see each peer represented as a single bar, whose length indicates the time this peer needed to serve this specific request.

Unlike Archipelago where the instrumentation was obvious, instrumenting RADOS was more challenging. RADOS exposes a C-API (librados) which is used in the Archipelago rados-peer. So, the first thing we did was to instrument the read and write calls of this API. Then, we needed to extend the RADOS classes to transfer the tracing information. In a nutshell, after librados, Ceph protocol which is TCP-based transfers the IO request to the cluster. So, the tracing information is encoded as part of the `MOSDOp` Ceph object. Then the request after being decoded, enters a dispatch queue and waits to be served. Based on the objects affected, a different placement group handles it. After the dispatch queue, the request is handled by this pg's primary OSD and then based on the replication factor, equal number of replication operations are issued that follow the same route. Request handling includes journal access and filestore access. In an attempt not to expose much of the RADOS internals, so that

[1] http://linux.die.net/man/1/fio
[2] https://github.com/grnet/libxseg

the Zipkin UI would be self-explanatory even for someone that is not familiar with the RADOS code architecture, we tried to instrument the code so that we can extract information such as the time spent in the dispatch queue, the network communication time, or the journaling duration and at the same time we follow the causal relations used by Zipkin. For example, the IO handling by the primary OSD causes the replication operations. So the replication operations are children spans.

As far as the test-bed is concerned, we used two physical nodes LAN interconnected, and set up two OSDs on each node. On one of this nodes we installed Archipelago and Qemu. So, on the one node we had the running VM, Archipelago and 2 OSDs and on the other just two OSDs. Each node had a whole BlkKin stack running and a local Scribe server. Each Scribe server communicated with the central Zipkin collector or the Scribe server logging to the Hadoop cluster. For Zipkin we used a 4-core. 8-gb RAM virtual machine, while for the Hadoop cluster, as it was used only as a proof of concept, we used 2 2-core, 4-gb virtual machines.

You can find some specs regarding the hardware and software infrastructure in Tables 7.1 and 7.2.

Component	Description
CPU	2 x Intel(R) Xeon(R) CPU E5645 @ 2.40GHz [13] Each CPU has six cores with Hyper-Threading enabled, which equals to 24 threads.
RAM	2 banks x 6 DIMMs PC3-10600 Peak transfer rate: 10660 MB/s
Hard disks	12x 7.2k RPM 2TB SAS HDs, 12x 7.2k RPM 600GB SAS HDs, 6x 100GB SSD SATA HDs

Table 7.1: Test-bed hardware specs

The Ceph OSDs on the one node used two SSD disks in RAID 0, one each, and the other two on the other node two SAS disks in RAID 0, one each.

Software	Version
OS	Debian Wheezy
Linux kernel	3.2.0-4
lttng-tools	2.4

Table 7.2: Test-bed software specs

7.2 BlkKin tracing environment

In order to set up the tracing environment, apart from the central Scribe collector which could be either the Zipkin collector or a Scribe server connected to HDFS, on each cluster node you have to follow the next steps to live trace:

1. Start the local Scribe daemon
   The local Scribe daemon is configured to send the received messages to the next Scribe server. In case of connection loss or congestion, the data are stored locally and forward when the problem is solved.

2. Start LTTng live tracing
   Create a live tracing session and enable all the userspace events for it. Then start Babeltrace live and redirect its output to a named pipe.

3. Start the consumer
   Start the python consumer that reads from the named pipe and send data to the local Scribe server.

The overall procedure can be seen in Listing 7.1

```
#!/bin/bash

#Start the local Scribe server

src/scribd examples/example2client.conf

#Create the lttng session
lttng create --live 200000 -U net://localhost
#Enable all userspace events
lttng enable-evenut -u -a
#Start the session
lttng start

#Start babeltrace live
babeltrace -i lttng-live net://localhost/host/<hostname>/<session_name> >
    /var/run/blkin

#Start the python consumer to send data to localhost to the predefined
   Scribe port
./consumer.py localhost 1464 /var/run/blkin
```

Listing 7.1: Setting up the tracing environment

7.3 Evaluation metrics

In this section we analyze some metrics concerning the network and the system overhead that BlkKin poses to the system. These metrics led us to the previous deployment architecture.

7.3.1 LTTng vs Syslog

First of all, we should evaluate the use of LTTng versus another logging system that is based on system calls every time some information needs to be logged. SystemTap, DTrace or even syslog make a system call or a stop at a break point whenever they need to trace something. This is claimed to have high overhead for a system that need to run in full load. As mentioned the BlkKin library was backend independent, so we we changed the LTTng backend for a syslog backend and instrumented a simple client server application with the BlkKin library. Then we measured the time eacg backend took to finish tracing. The application was a simple two-process application, that communicated over a UNIX pipe. The server created a root span, annotated and then passed a message to the client including tracing information. The client created a child span annotated and answered back. A single iteration triggers four annotations and we repeated this message passing for 10000 times. The results can be seen in Table 7.3.

Backend	Time for 10000 iterations
LTTng	1.8 sec
Syslog	3.38 sec

Table 7.3: LTTng vs Syslog comparison

As it was expected the syslog backend was about 90% slower than LTTng even for this single example. So LTTng was the only choice possible that combined the low-overhead capability with the variety of tracing options such as user/kernel space tracing and performance counter access.

7.3.2 Thrift vs JSON format

As we mentioned we created two similar Babeltrace plugins, one sending JSON formatted messages and the other Thrift encoded messages to Scribe. It worths measuring the message size in these two cases because the smaller the message the lower the network overhead. In order to evaluate this parameter we created a simple BlkKin message and send it to a Scribe server both with the JSON and the Zipkin plugin. The message was as simple as seen in Listing 7.2

```
{
   "event":"start",
   "name":"process a",
   "timestamp":1412236861119767739,
   "service_name":"service a",
   "parent_span_id":0,
   "trace_id":8651812401464566866,
   "ip":"10.0.0.1",
   "port":5000,
   "span_id":7498859655107060208
}
```

Listing 7.2: A simple JSON formatted message

The packet size of the Zipkin-Thrift-encoded and the JSON encoded messages sent to Scribe can be seen in Table 7.4

Protocol	Packet size in bytes
Thrift	246
JSON	316

Table 7.4: Packet sizes per protocol used

As it was expected, the Thrift message is much smaller that the JSON one even in this case that the service and event names are small.

7.3.3 Scribe vs relayd

Another comparison we need to make to decide on the deployment architecture is the network overhead created by Scribe and relayd. For example, instead of running a local Scribe server, we could run a central realyd server per cluster and then send the data to the central Scribe server. Scribe, as mentioned offers buffering and batch messaging. Also, the LTTng consumerd will be faster when writing to localhost rather than to a remote server, thus reducing the possibility to lose tracing information. However, we have to figure out the amount of network traffic produced in the two cases. To evaluate this, we created 10 simple messages as the previous ones and sent them to a local relayd and then they were forward using Babeltrace live to the Scribe server. We measured the network traffic to localhost and to the Scribe server.

Our first notice when using `tcpdump` on localhost is that relayd polls consumerd on a specific interval to find out if there are any new data available. So, we wouldn't like to have our cluster being flooded by polling messages. Concerning the tracing data themselves, excluding polling, the sums of all the TCP packets' payloads sent for the 10 messages mentioned, can be found in Table 7.5 for each daemon.

Daemon	Data size in bytes
Scribe	1974
relayd	1079

Table 7.5: Data sent for 10 Scribe messages

However, even if CTF-format is more compact that Thrift, we prefer to avoid the polling messages in the cluster LAN and restrict them to localhost and to make use of the Scribe batch messaging capability in favor of the less payload size that CTF has to offer.

7.3.4 System overhead

In this subsection we evaluate the overhead that BlkKin poses to the instrumented application. To do so, we created two different but typical IO loads using fio. The first one was 2GB of random 4Kb writes and the second was 2GB of 64Kb sequential writes. These loads are diverse but really common and help up evaluate our systems performance under different working conditions. So, we ran fio in the virtual machine towards the Archipelago volume and measured the bandwidth the system had under different conditions. The scenarios we chose was without instrumentation, live tracing without sampling, live tracing with 1/500 sampling and tracing without live support. The results can be seen in Listings 7.1 and 7.2 and Tables 7.7 and 7.6 respectively.

Scenario	Bandwidth in Kbytes/sec
no tracing	16887
stopped tracing	16076
normal tracing	14882
live tracing	14941
live tracing sampling 500	15480

Table 7.6: Bandwidth for 64Kb sequential writes

Scenario	Bandwidth in Kbytes/sec
no tracing	1326.7
stopped tracing	1247.1
normal tracing	1100
live tracing	1107.8
live tracing sampling 500	11927

Table 7.7: Bandwidth for 4Kb random writes

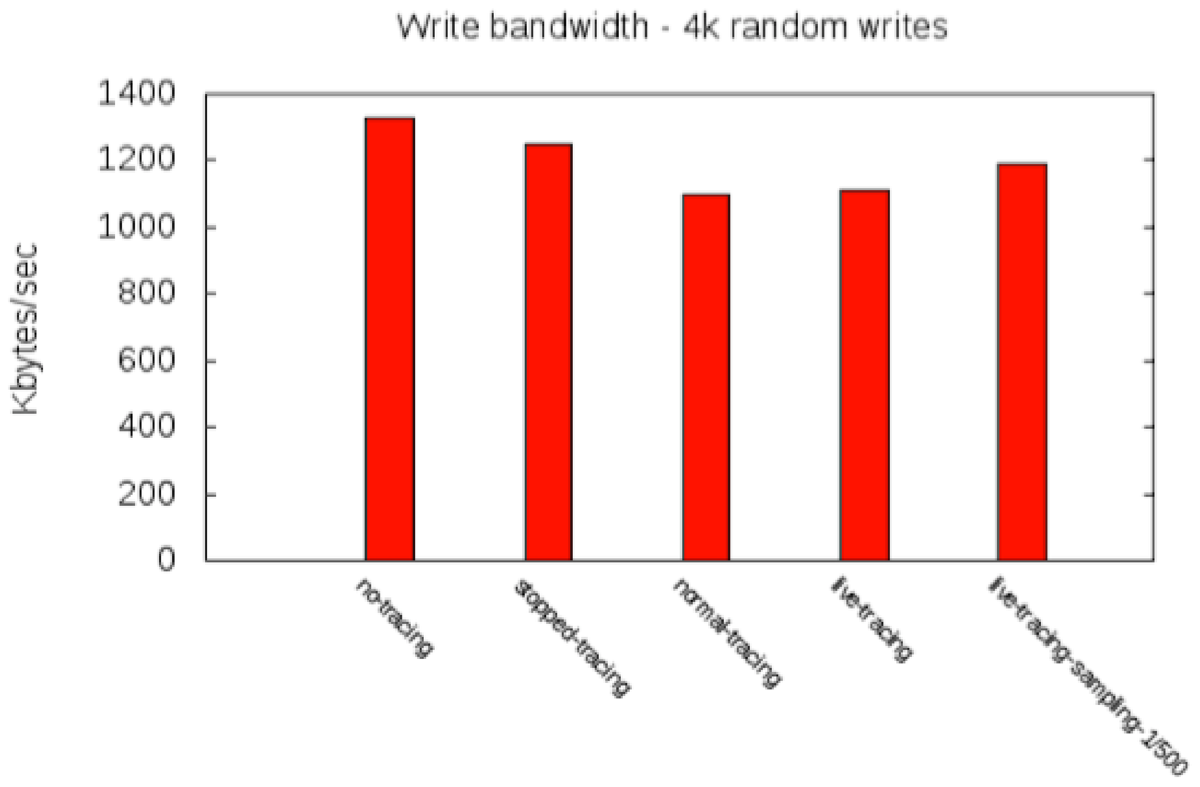


Figure 7.1: Performance overhead for 4k random writes

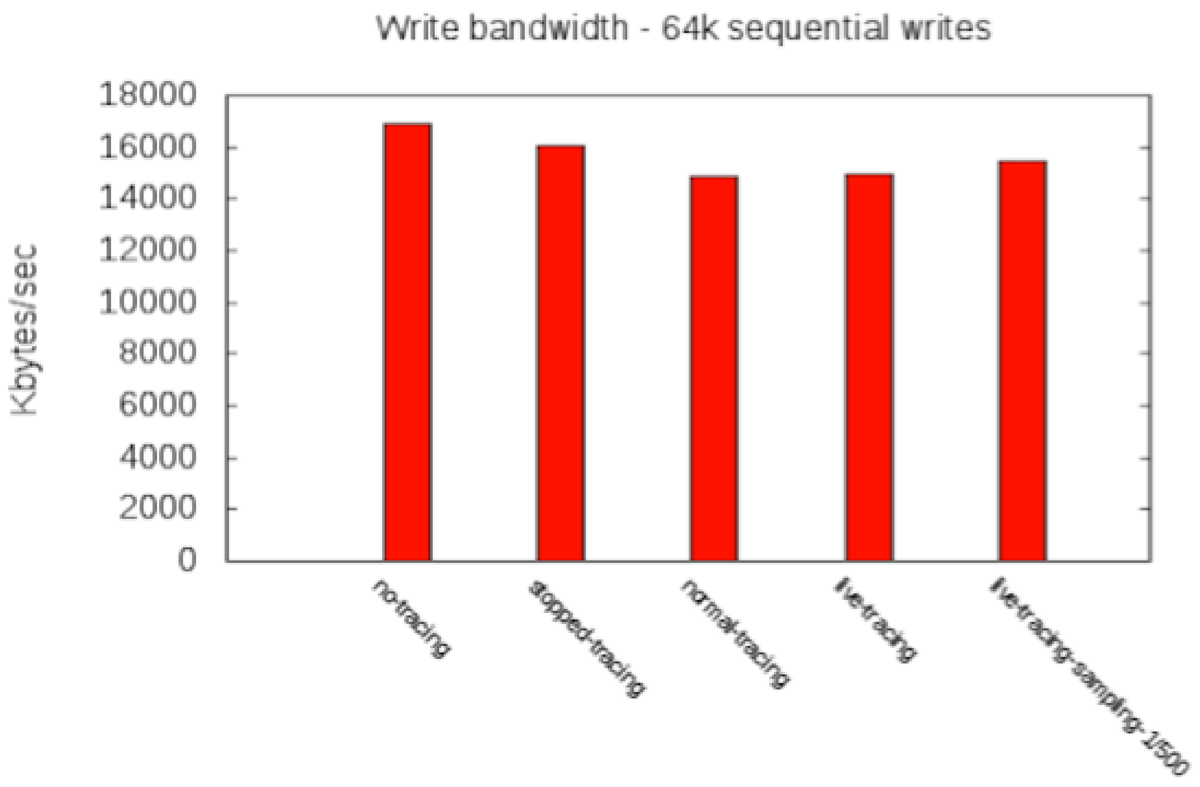


Figure 7.2: Performance overhead for 64k sequential writes

Although the above figures depict different loads and working conditions, BlkKin performs in a similar way. We can see that when tracing is disabled, we have about 5% bandwidth degradation. This degradation is caused by the check whether LTTng should trace or not. In case of full tracing the performance degradation increases at about 12% in case of sequential writes and at about 17% in case of small, random writes. This degradation is caused by the tracepoint function calls that actually log the information. The case of 4Kb writes is more affected because we have more IO request taking place fast one after the other. So the LTTng load is greater. Both live tracing without sampling and normal tracing affect the system in a similar way. The only change as far as LTTng is concerned is that in case if live tracing the consumerd writes the tracing information to a TCP socket instead of a local file descriptor. Finally, as we see in the scenario of sampled tracing, the degradation is such that we can afford tracing our system in production scale. In our scenario, we chose to sample 1/500 IO requests. Depending on the system load, this sampling rate can be either reduced or increased. It should be mentioned that in our instrumentation we used 113 annotations per single IO request in order to track its whole route. So, the case of so-sampling, live tracing produced a significant amount of network traffic and should be avoided not only because of this traffic, but also because of it, the tracing information take more time to reach to Zipkin. Consequently, we will need more time to detect a possible failure or problem.

7.4 Using BlkKin to detect bottlenecks and failures

As it has become obvious, BlkKin can be used for various reasons and trying to detect different problems either as part of the debugging process or as part of a fault detection mechanism. In this subjection we evaluate both of these uses.

7.4.1 Using Zipkin in debugging and system evaluation

The most simple use of BlkKin is to analyze the system's performance, measure the communication latencies, possible computation bottlenecks and generally to get a general overview of the request's evolution. For these reasons, the Zipkin UI is really usefull. This UI enables us to do simple queries and to understand the causal relations, to evaluate the time differences between the different software layers and to access the key-value annotations or even make queries based on them.

As seen in Figure 7.3 each span is represented as a separate bar whose length is commensurate to the duration of this specific processing phase. As far as time is concerned, each request is considered to start at time 0 and all timestamp annotations have timestamps relative to the first annotation of the trace. Also, if you click on a span that you are interested about, you can access this span's annotations as can be seen in Figure 7.4. In this screenshot we chose to investigate an 'OSD Handling op', namely the span that annotates the OSD's actions to serve and IO request. At the top we can see each timestamp annotation with its relative time, while at the bottom we can see the key-value annotations. For example, this operation handling refers to the specific RADOS object whose name can be seen as part of the binary annotation.

During this thesis evolution, Zipkin changed its UI. The old UI offered another visualizing capability that is planned to be added to the new UI soon. This capability was about endpoint dependencies. As it is seen in Figire 7.5, each service is depicted as a different circle whose radius is commensurate to the processing duration of that specific phase. This d3 visualization depicts exactly the information flow.

In case that these visual representations are not enough and we want to extract aggregate values, such as average values we have other choices. As mentioned Twitter suggest to deploy Zipkin with Cassandra. However, we deployed Zipkin using MySQL in order to have the ability of ad-hoc queries.

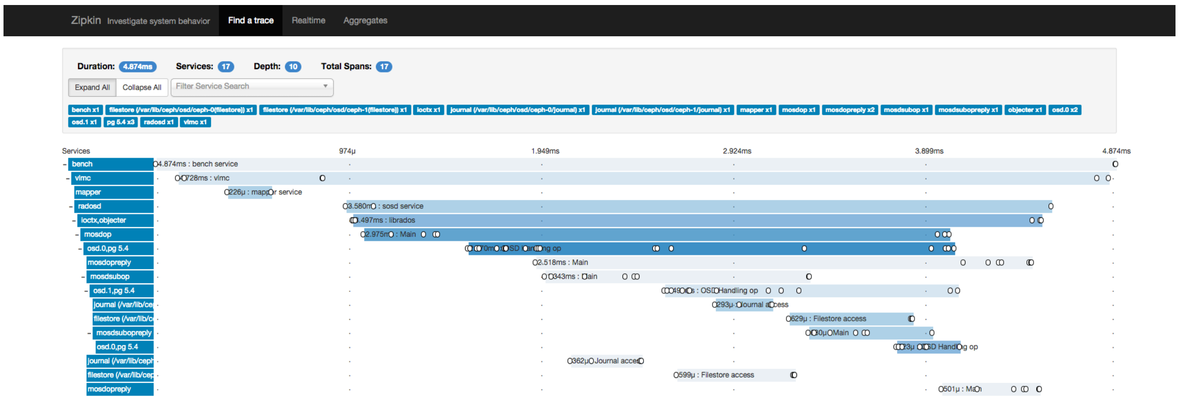
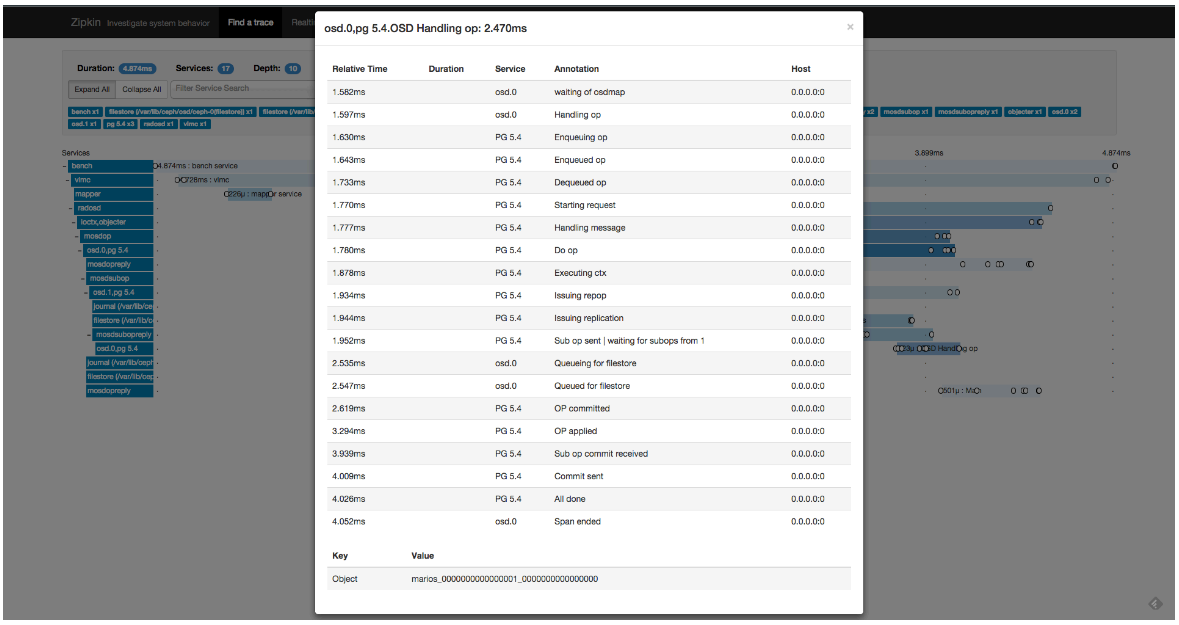


Figure 7.3: Zipkin UI overview


osd.0,pg 5.4.OSD Handling op: 2.470ms

Relative Time	Duration	Service	Annotation	Host
1.582ms		osd.0	waiting of osdmap	0.0.0.0:0
1.597ms		osd.0	Handling op	0.0.0.0:0
1.630ms		PG 5.4	Enqueuing op	0.0.0.0:0
1.643ms		PG 5.4	Enqueued op	0.0.0.0:0
1.733ms		PG 5.4	Dequeued op	0.0.0.0:0
1.770ms		PG 5.4	Starting request	0.0.0.0:0
1.777ms		PG 5.4	Handling message	0.0.0.0:0
1.780ms		PG 5.4	Do op	0.0.0.0:0
1.878ms		PG 5.4	Executing ctx	0.0.0.0:0
1.934ms		PG 5.4	Issuing repop	0.0.0.0:0
1.944ms		PG 5.4	Issuing replication	0.0.0.0:0
1.952ms		PG 5.4	Sub op sent | waiting for subops from 1	0.0.0.0:0
2.535ms		osd.0	Queueing for filestore	0.0.0.0:0
2.547ms		osd.0	Queued for filestore	0.0.0.0:0
2.619ms		PG 5.4	OP committed	0.0.0.0:0
3.294ms		PG 5.4	OP applied	0.0.0.0:0
3.939ms		PG 5.4	Sub op commit received	0.0.0.0:0
4.009ms		PG 5.4	Commit sent	0.0.0.0:0
4.026ms		PG 5.4	All done	0.0.0.0:0
4.052ms		osd.0	Span ended	0.0.0.0:0

Key	Value
Object	marios_0000000000000001_0000000000000000

Figure 7.4: Zipkin UI Annotations view

So, depending on the amount of tracing data we can extract the either from the database or by using Hadoop. In our case, we used Hadoop to calculate the average journaling time per OSD. So, simply by changing each local Scribe server's configuration file, we chose to send data to a Scribe server connected to HDFS and not Zipkin. Then, we created a Map-Reduce job to extract the information wanted as described in Section 6.4. The information gathered can be seen in Table 7.8.

OSD	Journaling time (μsec)
OSD1	401
OSD2	494
OSD3	475
OSD4	475

Table 7.8: Average journaling time for the 4 OSDs

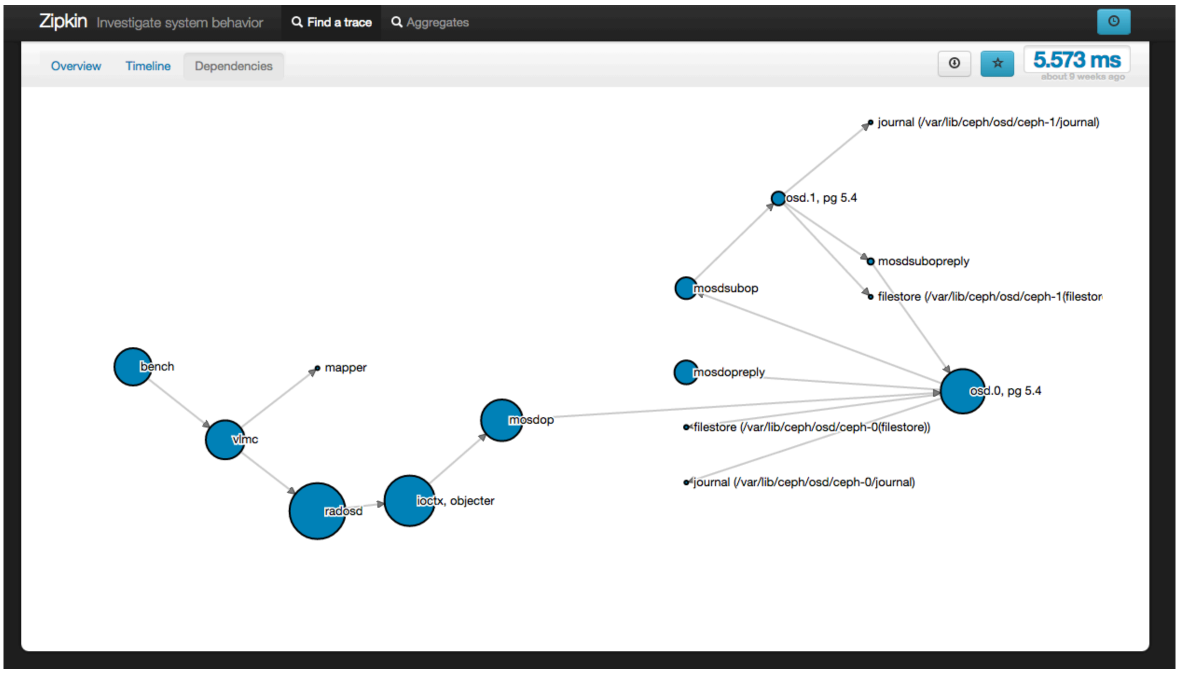

Zipkin Investigate system behavior
Find a trace
Aggregates
Overview
Timeline
Dependencies
5.573 ms
about 9 weeks ago
journal (/var/lib/ceph/osd/ceph-1/journal)
osd.1, pg 5.4
mosdsubopreply
mosdsubop
filestore (/var/lib/ceph/osd/ceph-1(filestor
bench
mapper
mosdopreply
osd.0, pg 5.4
vlmc
mosdop
filestore (/var/lib/ceph/osd/ceph-0(filestore))
journal (/var/lib/ceph/osd/ceph-0/journal)
radosd
ioctx, objecter

Figure 7.5: Zipkin UI Dependencies View

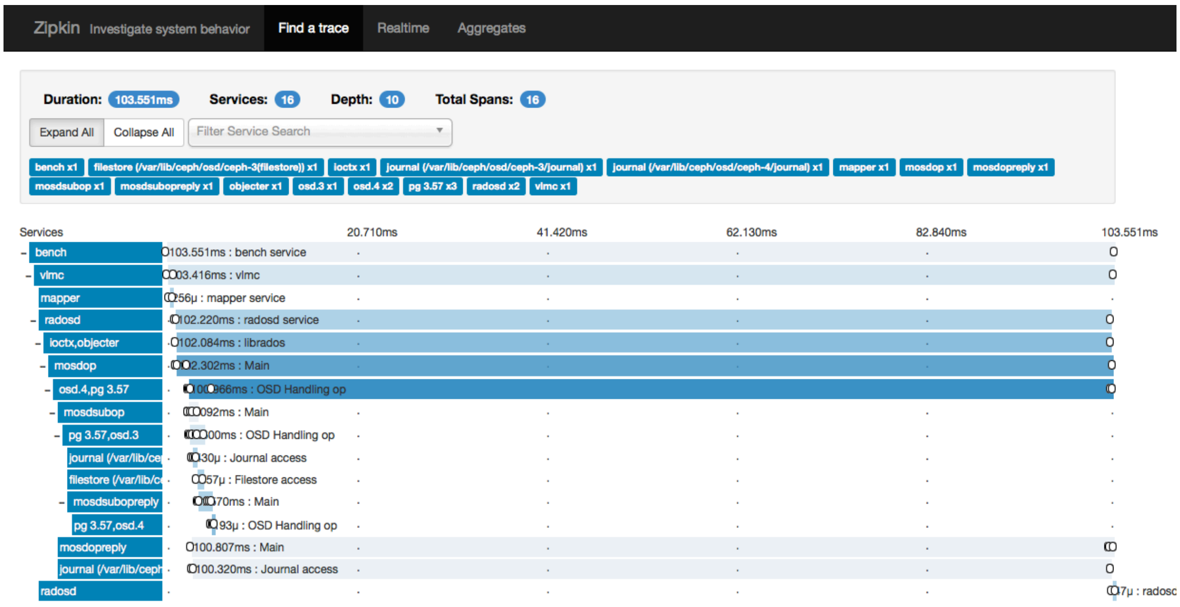
7.4.2 Using Zipkin to detect abnormal behaviors

The other BlkKin's use is as an online anomaly detection mechanism. According to [8], numerous techniques have been proposed for detecting system anomalies. Among them, the simplest ones are the threshold-based techniques which are a form of service level agreements (SLAs). They are very useful on the condition that their users clearly know the key metric to monitor and the best value of the thresholds in different scenarios. Unfortunately, it is very difficult, even for an expert, to correctly choose the necessary metrics to monitor and set the right values of the thresholds for different scenarios in the context of today's complex and dynamic computer systems. In addition, statistical learning or data mining techniques are widely employed to construct probability models for detecting various anomalies in large-scale systems based on some heuristics and assumptions, although these heuristics and assumptions may only hold in some particular systems or scenarios. Other methods ([15]), include artificial intelligence and neural networks, but they require large training datasets.

In this part of the evaluation, we present how we used Zipkin to detect common cases of anomaly that are possible to happen in a storage cluster. This abnormalities refer to either network or disk problem. We tried a threshold based alerting approach as a proof of concept, while the expressiveness and correlation capabilities of BlkKin allow further investigation for more correlative detection models. In our model, we conducted several tracing sessions for different IO loads and filled a Hadoop cluster with these data. Afterwards, through Map-Reduce jobs, we concluded on the thresholds we are going to use for the specific hardware and depending on the expected load. The thresholds included the communication times through the RADOS protocol and the times for journaling and filestore access. Then, we inserted these threshold values to the BlkKin monitoring tool.

Apart from BlkKin, we needed to find a way to simulate a faulty situation. For a network fault this was easy using the `tc` tool. This tool enables the user to insert common network faults, such as packet loss, latency or packet corruption. However, the case of disk faults is more complex. To simulate a faulty disk state there are multiple options. The most easy and the finally chosen is to add a significant IO load using fio with multiple threads making dummy read request to the specific disk. In our case of RADOS, since we constantly write 4Mb objects, this will increase the disk latency. The second choice is to use `cgroups` and the `blockio` controller. This choice enables us to throttle a disk bandwidth, by throttling the bandwidth of the processes writing to this disk, namely the OSD process. Another choice is to use the device mapper to create a faulty sector. The final choice is to use the Linux kernel fault injection capabilities[3]. The complexity of the latter choices and the kernel dependencies made us avoid them and just a dummy read IO load to the disk that we need to act as in case of a fault.

So, to simulate a disk fault we added a read IO load in the OSD4 journal partition. The results in Zipkin can be seen in Figure 7.6. As we can see, the Journaling span of the OSD4 takes significantly more time, when compared to a normal behaviour as seen in Figure 7.3 and as a result the whole request completion is affected since journaling is synchronous and encapsulated within the request process.



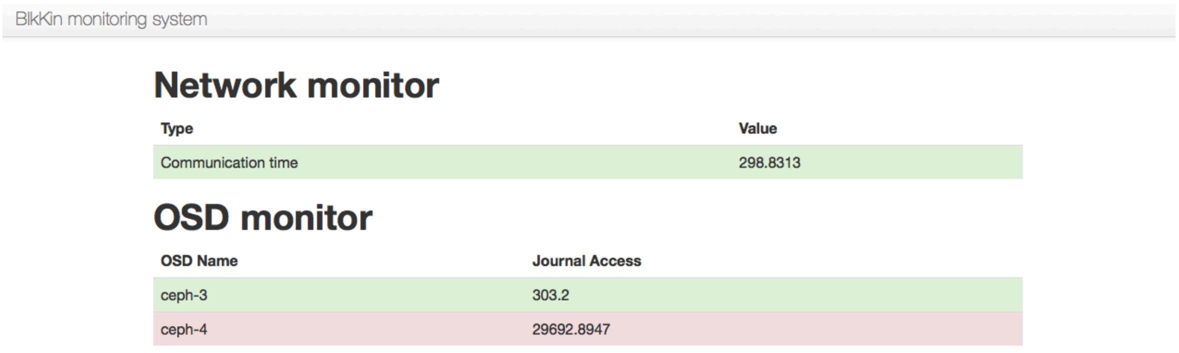
Duration: 103.551ms Services: 16 Depth: 10 Total Spans: 16

bench x1 | filestore (/var/lib/ceph/osd/ceph-3(filestore)) x1 | ioctx x1 | journal (/var/lib/ceph/osd/ceph-3/journal) x1 | journal (/var/lib/ceph/osd/ceph-4/journal) x1 | mapper x1 | mosdop x1 | mosdopreply x1 | mosdsubop x1 | mosdsubopreply x1 | objecter x1 | osd.3 x1 | osd.4 x2 | pg 3.57 x3 | radosd x2 | vlmc x1

```
Services                       20.710ms    41.420ms    62.130ms    82.840ms    103.551ms
- bench               103.551ms : bench service
 - vlmc               03.416ms : vlmc
   mapper             256μ : mapper service
  - radosd            102.220ms : radosd service
   - ioctx,objecter   102.084ms : librados
    - mosdop          2.302ms : Main
     - osd.4,pg 3.57  966ms : OSD Handling op
      - mosdsubop     092ms : Main
       - pg 3.57,osd.3   00ms : OSD Handling op
          journal (/var/lib/ce   30μ : Journal access
          filestore (/var/lib/c  57μ : Filestore access
        - mosdsubopreply         70ms : Main
            pg 3.57,osd.4        93μ : OSD Handling op
      mosdopreply     100.807ms : Main
      journal (/var/lib/ceph   100.320ms : Journal access
   radosd                                                                    7μ : radosc
```

Figure 7.6: Injected disk fault - Journaling Latency

Of course, this journaling duration is above the accepted threshold. So, the BlkKin monitoring UI would illustrate the problem as seen in Figure 7.7.


BlkKin monitoring system

Network monitor

Type	Value
Communication time	298.8313

OSD monitor

OSD Name	Journal Access
ceph-3	303.2
ceph-4	29692.8947

Figure 7.7: Journaling Latency - BlkKin monitoring UI

A similar situation was simulated for a network problem. Using `tc` we added 10 ms of latency to the NIC attached to the host where OSD1 and OSD3 run. The result are extended communication times that can be seen as part of the `Main` spans in Figure 7.8. Again this faulty situation is observed as seen in Figure 7.9 through the BlkKin monitoring tool.

[3] https://www.kernel.org/doc/Documentation/fault-injection/fault-injection.txt

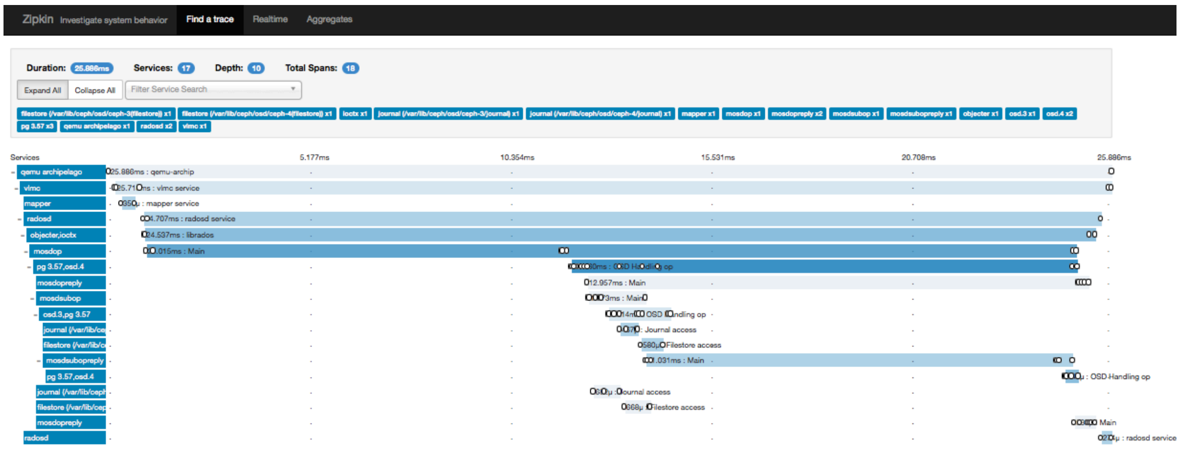
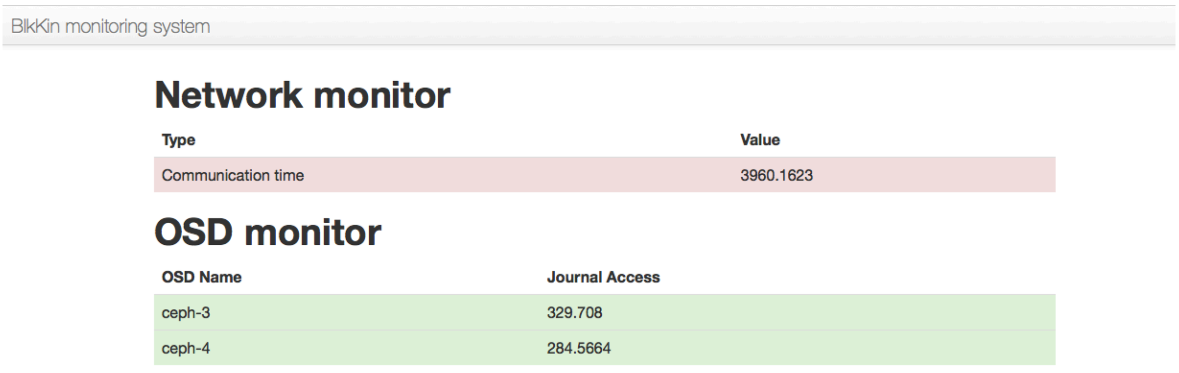


Figure 7.8: Network Latency - Zipkin UI



BlkKin monitoring system

Network monitor

Type	Value
Communication time	3960.1623

OSD monitor

OSD Name	Journal Access
ceph-3	329.708
ceph-4	284.5664

Figure 7.9: Network Latency - BlkKin monitoring UI

Chapter 8

Conclusion

8.1 Concluding Remarks

This thesis handles with the problem of low overhead distributed tracing aimed to analyze software defined storage systems. In this kind of multi-layered software architectures finding and locating bottlenecks and potential or even real faults is pretty challenging because of their complexity. After evaluating both various logging and tracing mechanism as well as different tracing schemas, we decided to implement our own tracing infrastructure called BlkKin. BlkKin is based on opensource technologies, namely LTTng and Zipkin, and implements the tracing semantics used by Google's Dapper tracing infrastructure. So BlkKin is a low-overhead tracing infrastructure that enables live tracing for applications written in C/C++ and provides two distinct user interfaces, so that the end user can take real time information about the system.

In order to accomplish this endeavour, we had to work with the LTTng community and extend their software so that it can communicate with Zipkin and create an instrumentation library for easy application instrumentation.

As a proof of concept for our system, we instrumented Archipelago and RADOS source code. Consequently, we were able to track an IO request's route from the time Qemu accepted it until it was finally served by RADOS and investigate any kind of latencies or bottlenecks each processing phase may have. Also, we simulated different faulty situations that a distributed storage system may face and investigated the use of BlkKin as an alerting mechanism for such kind of faults.

However, this work was just the beginning of cross-layered distributed tracing, since it provided the framework for further investigation. Our mechanism can be used in any kind of low-overhead application that needs a tracing and visualization infrastructure. So, far the Ceph community has showed interest for BlkKin and we are in close contact so that BlkKin can become the main tracing infrastructure for RADOS.

8.2 Future Work

BlkKin future plans/work include:

- Better live support, after the Babeltrace plugin system is released.
- Better sampling mechanism that takes into account the request's special characteristics, so that meaningful information is not lost because of sampling
- Offer Babeltrace plugins as part of the LTTng source tree in the form of a Python module availabe at `pip`.

- Use BlkKin in different kinds of distributed systems such as parallel applications, MPI for example.

As far as the RADOS instrumentation is concerned, future work includes:

- Better RADOS instrumentation not only for read and write requests. This instrumentation requires deep knowledge of the software's internals so that its bottlenecks are found.
- Implementation of a correlative alerting system that relates the replica operations with the cluster's health in order to avoid our threshold based alerting mechanism.
- Use of BlkKin tracing data from RADOS instrumentation to create an AI-based failure detector.

Bibliography

[1] Lior Abraham, John Allen, Oleksandr Barykin, Vinayak Borkar, Bhuwan Chopra, Ciprian Gerea, Daniel Merl, Josh Metzler, David Reiss, Subbu Subramanian, Janet L. Wiener, and Okay Zed. Scuba: Diving into data at facebook. Proc. VLDB Endow., 6(11):1057–1067, August 2013.

[2] Eric Anderson, Christopher Hoover, Xiaozhou Li, and Joseph Tucek. Efficient tracing and performance analysis for large distributed systems. In MASCOTS, pages 1–10. IEEE, 2009.

[3] Jonathan Appavoo, Marc Auslander, Dilma DaSilva, David Edelsohn, Orran Krieger, Michal Ostrowski, Bryan Rosenburg, Robert W. Wisniewski, and Jimi Xenidis. K42's Performance Monitoring and Tracing. IBM Research.

[4] Paramvir Bahl, Ranveer Chandra, Albert Greenberg, Srikanth Kandula, David Maltz, and Ming Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In SIGCOMM. Association for Computing Machinery, Inc., August 2007.

[5] Paul Barham, Rebecca Isaacs, Richardand Mortier, and Dushyanth Narayanan. Magpie: online modelling and performance-aware systems. In Proceedings of the 9th conference on Hot Topics in Operating Systems, 2003.

[6] Jan Blunck, Mathieu Desnoyers, and Pierre-Marc Fournier. Userspace application tracing with markers and tracepoints. In Proceedings of the 2009 Linux Kongress, October 2009.

[7] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instru- mentation of production systems. USENIX.

[8] Xin Chen, Xubin He, He Guo, and Yuxin Wang. Design and evaluation of an online anomaly detector for distributed storage systems. JSW, 6(12):2379–2390, 2011.

[9] Jonathan Corbet. Kernel Markers. In Linux Weekly News. August 2007.

[10] Luize Derose, Jefrey K. Hollingsworth, and Ted Hoover Jr. The Dynamic Probe Class Library - An Infrastructure for Developing Instrumentation for Performance Tools. In Parallel and Distributed Processing Symposium., Proceedings 15th International., 2001.

[11] M. Desnoyers. Low-impact Operating System Tracing. PhD thesis, Ecole Polytechnique de Montreal, 2009.

[12] Mathieu Desnoyers and Michel R. Dagenais. LTTng: Tracing across execution layers, from the hypervisor to user-space. In Linux Symposium 2008, page 101, July 2008.

[13] Intel(r) xeon(r) cpu e5645 specifications. http://ark.intel.com/products/48768/Intel-Xeon-Processor-E5645-12M-Cache-2_40-GHz-5_86-GTs-Intel-QPI.

[14] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-trace: a pervasive network tracing framework. In Proceedings of the 4th USENIX conference on Networked systems design &#38; implementation, NSDI'07, pages 20–20, Berkeley, CA, USA, 2007. USENIX Association.

[15] Errin W. Fulp, Glenn A. Fink, and Jereme N. Haack. Predicting computer system failures using support vector machines. In Proceedings of the First USENIX Conference on Analysis of System Logs, WASL'08, pages 5–5, Berkeley, CA, USA, 2008. USENIX Association.

[16] Filippos Giannakos, Vangelis Koukis, Constantinos Venetsanopoulos, and Nectarios Koziris. okeanos large-scale cloud service using cepg. ;login, 39(3):6–9, June 2014.

[17] D. Goulet. Unified Kernel/User-Space Efficient Linux Tracing Architecture. PhD thesis, Ecole Polytechnique de Montreal, 2012.

[18] M. Hiramatsu and S. Oshima. Djprobe—Kernel probing with the smallest overhead. In Proceedings of the 2006 Linux Symposium, Ottawa, Canada, July 2006.

[19] Jefrey K. Hollingsworth, Barton P. MILLER, and Jon Cargille. Dynamic program instrumentation for scalable performance tools. In Proceeding of the Scalable High Performance Computing Conference, 1994.

[20] Vangelis Koukis, Constantinos Venetsanopoulos, and Nectarios Koziris. okeanos: Building a cloud, cluster by cluster. IEEE Internet Computing, 17(3):67–71, May 2013.

[21] Vangelis Koukis, Constantinos Venetsanopoulos, and Nectarios Koziris. Synnefo: A complete cloud stack over ganeti. ;login, 38(5):6–10, October 2013.

[22] Leslie Lamport. The part-time parliament. ACM Trans. Comput. Syst., 16(2):133–169, May 1998.

[23] Matthew L. Massie, Brent N. Chun, and David E. Culler. The ganglia distributed monitoring system: Design, implementation and experience. Parallel Computing, 30:2004, 2003.

[24] Ananth Mavinakayanahalli, Prasanna Panchamukhi, Jim Keniston, Anil Keshavamurthy, and Masami Hirama. Probing the guts of kprobes. In Proceedings of the 2006 Linux Symposium, Ottawa, Canada, July 2006.

[25] Gordon E. Moore. Cramming more components onto integrated circuits. In Mark D. Hill, Norman P. Jouppi, and Gurindar S. Sohi, editors, Readings in computer architecture, pages 56–59. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[26] National Institute of Standards and Technology. The NIST definition of cloud computing. Special Publication 800-145, September 2011.

[27] Vara Prasad, William Cohen, Frank Ch. Eigler, Martin Hunt, Jim Keniston, and Brand Chen. Locating System Problems Using Dynamic Instrumentation. In Proceedings of the Ottawa Linux Symposium, 2005.

[28] Patrick Reynolds, Janet L. Wiener, Jeffrey C. Mogul, Marcos Kawazoe Aguilera, and Amin Vahdat. Wap5: black-box performance debugging for wide-area systems. In Les Carr, David De Roure, Arun Iyengar, Carole A. Goble, and Michael Dahlin, editors, WWW, pages 347–356. ACM, 2006.

[29] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[30] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: a scalable, high-performance distributed file system. In Proceedings of the 7th symposium on Operating systems design and implementation, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.