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."[13]

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 $P = cV^2f$ 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."[14]

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 [9]. 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 [10] 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[7] 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.

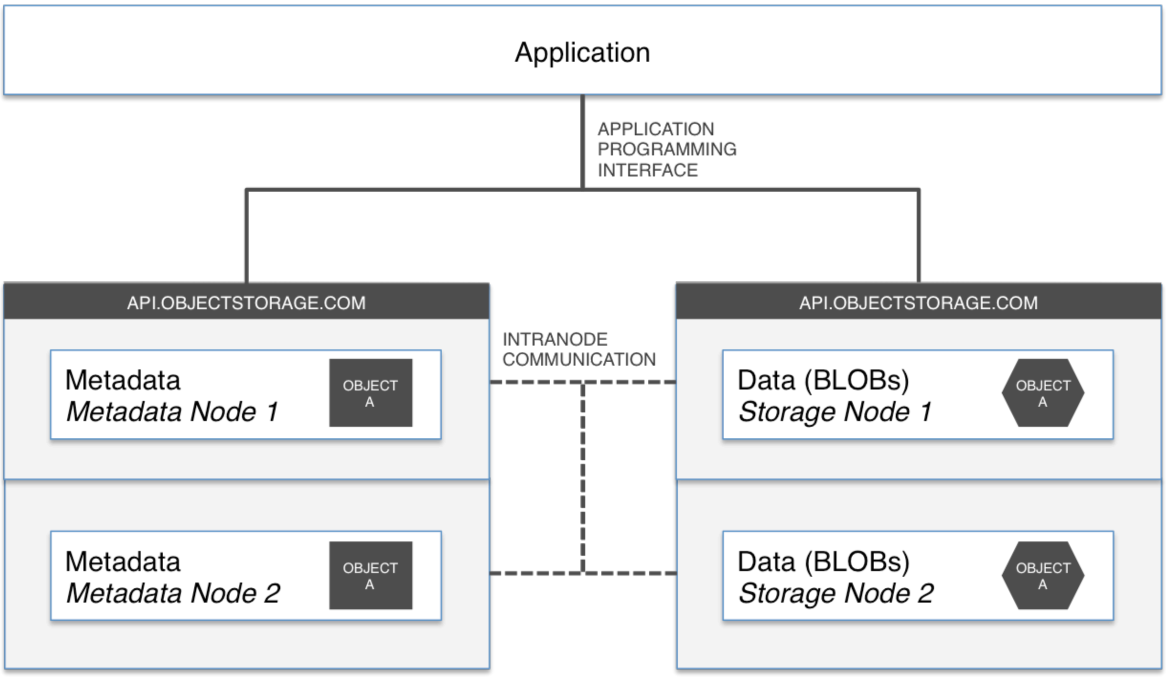

Figure 2.1: Storage Abstraction

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[18] 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

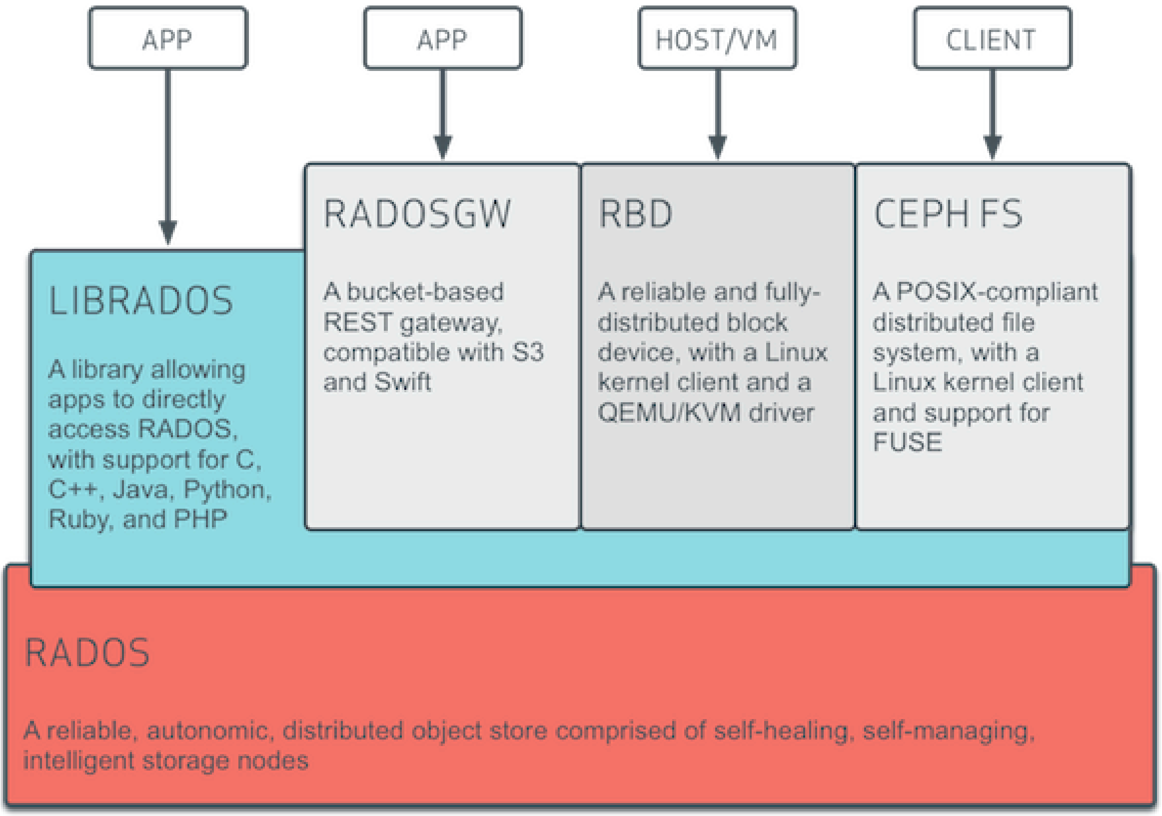

Figure 2.2: Ceph abstraction

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[11]. 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.

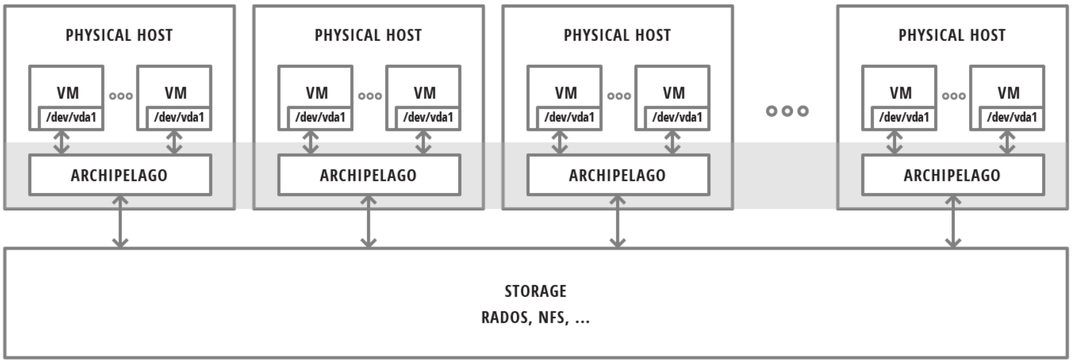

Figure 2.3: Archipelago Overview

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.

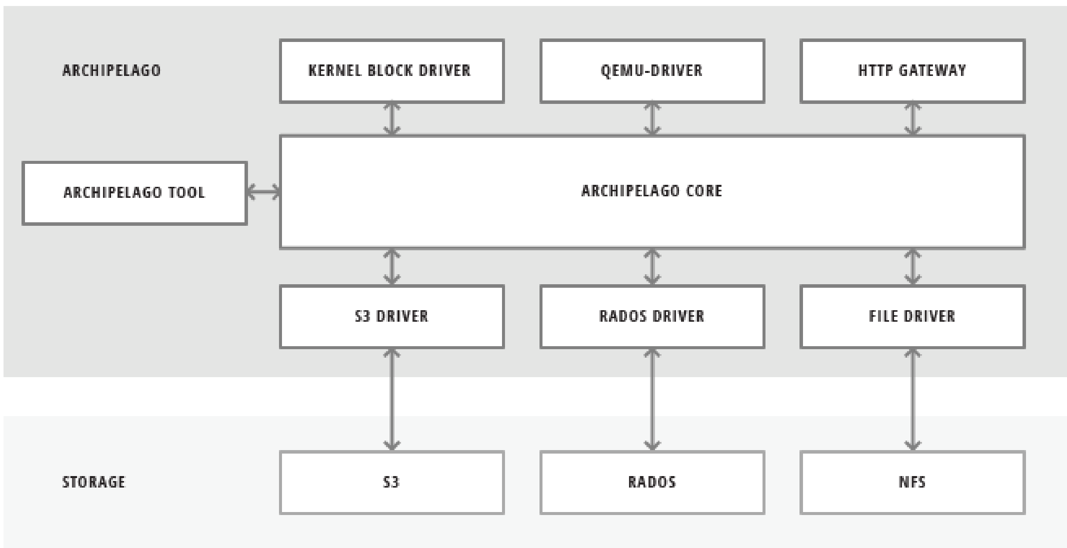

Figure 2.4: Archipelago APIs

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.

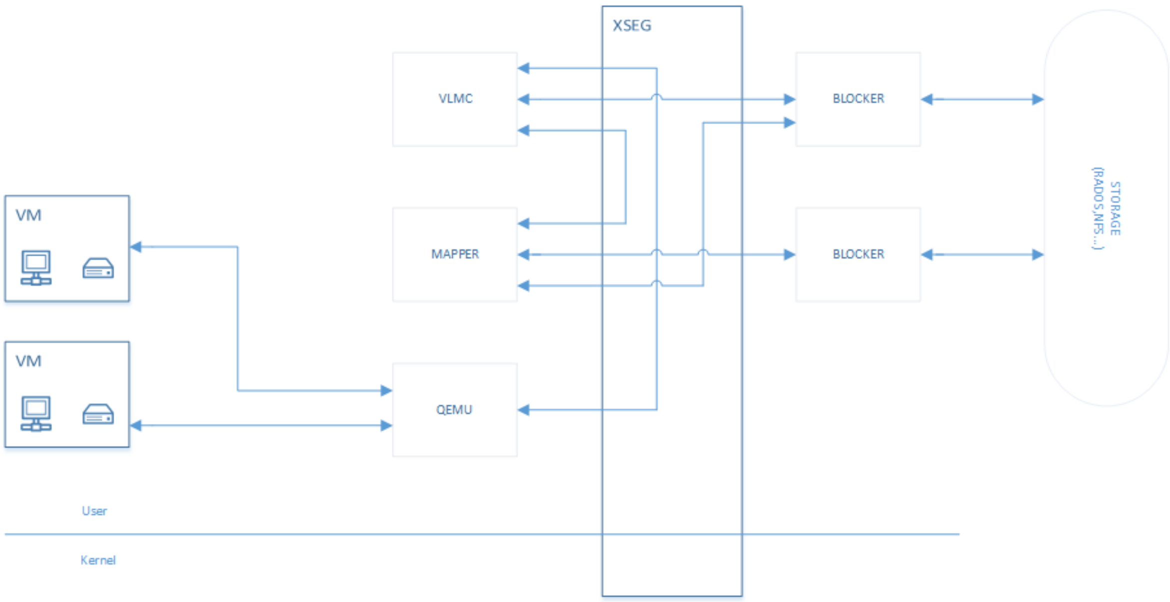

Figure 2.5: XSEG communication

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 [8] 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 [5] 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 [1] 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 [3]. 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 [4] 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 [15] 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[12] 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/

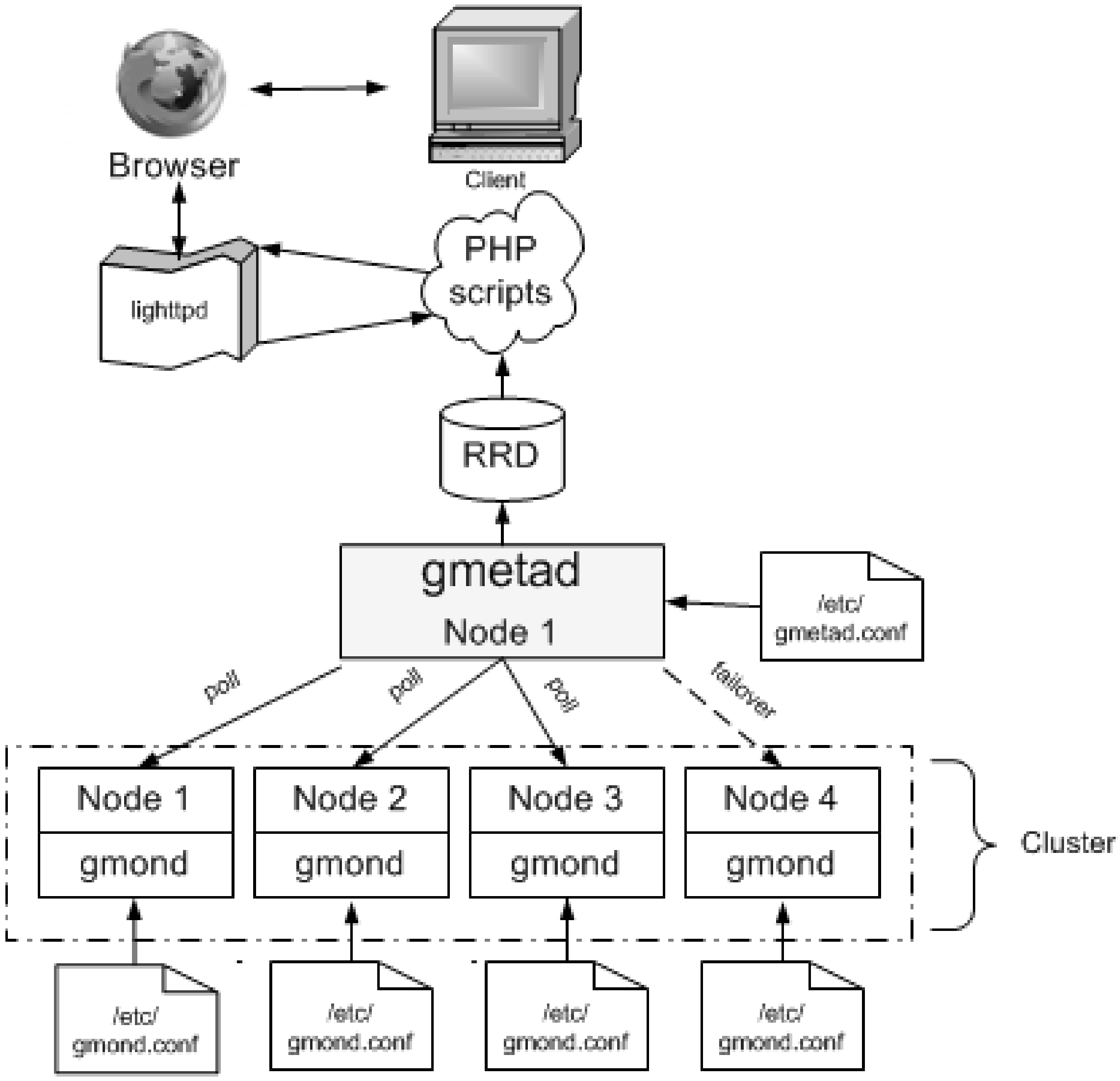

Figure 2.6: Ganglia architecture

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)

3.1 Overview

3.2 Buffering scheme

3.3 Kernelspace tracing

3.4 Userspace tracing

3.5 Common Trace Format (CTF)

3.6 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[17] these categories are:

black-box schemes [2, 16] 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 [3, 6] 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[17] 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*.

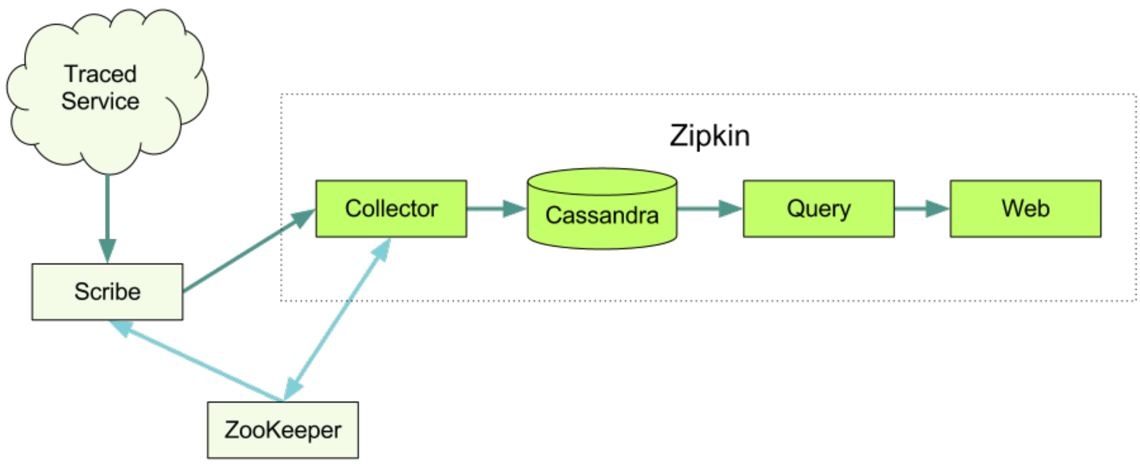

Figure 4.1: Zipkin Architecture

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

Bibliography

[1] 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.

[2] 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.

[3] 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.

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

[5] 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.

[6] 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 & implementation, NSDI'07, pages 20–20, Berkeley, CA, USA, 2007. USENIX Association.

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

[8] 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.

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

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

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

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

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.