KTH Information and
Communication Technology

Towards Elastic High Performance Distributed Storage Systems in the Cloud

YING LIU

Licentiate Thesis Proposal
School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden

TRITA xxx yyyy-nn

Abstract

With the growing popularity of Internet-based services and the trend of hosting them in the Cloud, more powerful back-end storage systems are needed to support these services. On one hand, the storage system itself should be capable of handling workloads with higher concurrency and intensity. Distributed storage solutions are proposed. They are expected to achieve some of the following properties: scalability, availability, durability, consistency, and partition tolerance. It is difficult or even impossible to have a distributed storage solution that fully satisfies all of these properties. Thus, there are researches that focus on pushing the limit on achieving those properties in the designs of distributed storage solutions satisfying different usage scenarios. On the other hand, the increasing trend of Cloud computing brings new challenges to the design of distributed storage systems. In order to better enjoy the benefits from the pay-as-you-go pricing model in the Cloud, a dynamic resource provisioning middleware, i.e., an elasticity controller, is needed. An elasticity controller is able to help saving the provisioning cost of an application without compromising its performance. The design of an elasticity controller to auto-scale a distributed storage system is not trivial. The challenging issue is the data migration overhead (data consume time and resources to be transferred to added instances or transferred out from removed instances) when scaling the system.

The main goal of this thesis is to improve access latency in distributed storage services hosted in a Cloud environment. The goal is approached from two aspects: First, the dynamicity, in terms of intensity, concurrency and locality, in workloads introduces challenges to have a low latency storage solution. We propose smart middlewares, such as elasticity controllers, to effectively and efficiently achieve low latency with minimum resource provisioning under dynamic workloads. Another aspect that we consider is the efficiency of a storage solution itself. We improve access latency of a storage solution under the requirements of scalability, consistency, availability. We start by summarizing the current state of the research field by surveying representative design solutions and identify trends and open issues.

We proceed by presenting a performance evaluation of a distributed storage system (Swift) in a simulated Cloud platform. The project identifies the feasibility of deploying a distributed storage system in a community Cloud environment, which is different from a traditional data center environment. We start by characterizing environment parameters that influence the performance of Swift. We quantify the performance influences by varying the environment parameters including virtual machines (VMs) capability, network connectivity, and potential VM failures. We conclude that Swift is sensitive to network connectivity among storage instances and using a simple self-healing solution, it is able to survive with VM failures to some extent.

We continue by introducing GlobLease, a distributed storage system that uses leases to achieve consistent and low latency read/write accesses in a global scale. Leases guarantee that a data object is updated in a period of time. Thus, the data with valid lease can be read from any replica consistently. Leases are used to improve the access latency of read/write requests by reducing the communication cost among replicas.

Next, we describe BwMan, a bandwidth manager that arbitrates bandwidth allocation among multiple services sharing the same network infrastructure. BwMan guarantees network bandwidth allocation to specific service based on predefined rules or machine learning models. In the scenario of guarantee latency SLAs of a distributed storage system under dynamic workload, BwMan is able to significantly reduce SLA violations when there are other services sharing the same network.

Finally, we describe ProRenaTa, an elasticity controller for distributed storage systems that combines proactive and reactive control techniques to achieve better SLA commitments and resource utilization. ProRenaTa explicitly considers and models the data migration overhead when scaling a distributed storage system. Furthermore, ProRenaTa takes advantages of the complimentary nature of the proactive and reactive scaling approach and makes them operate coherently. As a result, ProRenaTa yields a better performance modeling and prediction of the system and thus achieving higher resource utilization and less SLA violation.

Contents

List of Figures

Part I

Thesis Overview

Chapter 1

Introduction

With the growing popularity of Internet-based services, more powerful back-end storage systems are needed to match the increasing workloads in terms of concurrency, intensity, and locality. When designing a high performance storage system, a number of major properties, including scalability, elasticity, availability, consistency guaranties and partition tolerance, need to be satisfied.

Scalability is one of the core aspects in the design of high performance storage solutions. Centralized storage solutions are no longer able to support large scale web applications because of the high access concurrency and intensity. Under this scenario, distributed storage solutions, which are designed with greater scalability, are proposed. A distributed storage solution provides an unified storage service by aggregating and managing a large number of storage instances. A scalable distributed storage system can, in theory, aggregate unlimited number of storage instances, therefore providing unlimited storage capacity if there are no bottlenecks in the system.

Availability is another desired property for a storage system. Availability means that data stored in the system are safe and always or most of the time available to its clients. Replications are usually implemented in a distributed storage system in order to guarantee data availability in the presence of server or network failures. Specifically, several copies of the same data are preserved in the system at different servers, racks, or data centers. Thus, in the case of server or network failures, data can still be served to clients from functional and accessible servers that having copies of the data. Maintaining multiple copies of the same data brings the challenge of **consistency** issues. Based on application requirements and usage scenarios, a storage solution is expected to provide some level of consistency guarantee. For example, strong consistency provides that all the data copies act synchronously like one single copy and it is desired because of its predictability. Other consistency models, for example, eventual consistency allows data copies to diverge within a short period of time. In the general case, stricter consistency involves more overhead for a system to achieve and maintain. Hosting multiple data replicas in multiple servers also needs to survive with **network partitions**. Partition of networks blocks the communications among data copies. As a result, either inconsistent result or no result can be returned to clients in this scenario.

Despite researching on the above properties to design high performance storage solutions, we narrow down the design scope by targeting storage systems that are hosted in public/private Cloud platforms. Cloud computing not only shifts the paradigm that companies used to host their Internet-based businesses, but also provides end users a brand new way of accessing services and data. A Cloud is the integration of data center hardware and software that provides "X as a service (XaaS)"; value of X can be infrastructure, hardware, platform, and software. The scope of Cloud computing focused in this thesis is infrastructure as a service (IaaS), where Cloud resources are provided to consumers in the form of physical or more often virtual machines (VMs). Cloud computing provides the choice for companies to host their services without provisioning a data center. Moreover, the pay-as-you-go business model allows companies to use Cloud resources on a short-term basis as needed. On one hand, companies benefit from letting resources go when they are no longer needed. On the other hand, companies are able to request more resources anytime from the Cloud platform when their businesses grow without planning ahead for provisioning. The autonomic tuning of resources according to the needs of applications hosted in the Cloud is often called dynamic resource provisioning, auto-scaling or **elasticity**.

1.1 Research Objectives

The main goal of this work is to optimize service latency of distributed storage systems that are hosted in a Cloud environment. From a high-level view, there are two main factors that significantly impact the service latency of a distributed storage system, assuming a static execution environment and available resources: (a) the efficiency of the storage solution itself and (b) the intensity of the workload that need to be handled by the system. Naturally, less efficient storage solution slows down the request processing and also, handling more intensive workload saturates the system. Thus, in order to provide low service latency guarantee, we define two main corresponding goals:

1. to improve the efficiency of storage solutions, and

2. to make storage systems adapt to changing workload with low latency guarantees

Our vision towards the first goal is make storage systems to be distributed in a larger scale, so that more requests can be served by servers that are close to them, which significantly reduce the portion of high latency requests. We specify optimizations on data consistency algorithm in this usage case with the objective of reducing the replica communication overhead under the requirements of scalability, consistency, availability. The core challenge to achieve the second goal is introduced by the complexity of workload patterns, which can be highly dynamic in intensity, concurrency, and locality. We propose smart middlewares, i.e., elasticity controllers, to effectively and efficiently achieve low latency with dynamic resource provisioning under dynamic workloads.

With respect to these goals, [1] is a study on state of the art optimizations on designing efficient storage solutions, [2] is a work done with respect to improving the efficiency of a storage solution, [3] presents a bandwidth manager for storage solutions under dynamic workload patterns, 10 contains work towards improving resource utilization and guaranteeing service latency under dynamic workload.

1.2 Research Methodology

In this section, we describe the methods that we used in this research work. We provide descriptions of the process that we followed and design decisions that we made in order to achieve our goals. We also discuss challenges that we faced during the process and how we overcame them.

1.2.1 Design of Efficient Storage Solutions

The work on this matter does not follow analytical, mathematical optimization methods, but is rather based on an empirical approach. We approach the problem by first studying techniques/algorithms used in the design of distributed storage solutions. This process provides us the knowledge base for inventing new algorithms to improve the efficiency of storage solutions. After understanding the state of the art solutions, we then start investigating the usage scenarios of distributed storage solutions. The efficiency of a storage solution varies on different usage scenarios. We focus on analyzing solutions' efficiency with respect to the operating overhead when deployed in different usage scenarios. This overhead usually differs because of different system architectures, implementations and algorithms. We place our innovation when there is no efficient storage solution for a needing usage scenario. Once we have confirmed our innovation direction, we investigate the causes of inefficiency of the current solutions and avoid them in our design. After examining sufficient number of leading storage solutions, we choose the most suitable system architecture that can be applied in this scenario. We tailor algorithms for this scenario by avoiding the known performance bottlenecks. Finally, we evaluate our design and implementation by comparing it with the several leading storage solutions. We use request latency as the performance measure and also present algorithm overhead, when applicable.

1.2.2 Design of Elasticity Controller

The work on this matter also follows an empirical approach. Our approach is based on first understanding the environmental and system elements/parameters that are influential to the effectiveness and accuracy of an elasticity controller, which directly affects service latency of the controlled systems. Then, we study the technologies that are used in building performance models and the frameworks that are applied in implementing the controller. The result of the studies allows us to discover the unconsidered elements/parameters that influences the effectiveness and accuracy of an elasticity controller. We verify our assumptions on the performance degradation of an elasticity controller if not including the elements/parameters by experiments. Once we have confirmed the space for improving the effectiveness and accuracy of an elasticity controller, we innovate on designing new performance models that consider those environmental and system elements/parameters. After implementing our controllers using the novel performance model, we evaluate it by comparing it to the original implementation. For the evaluation, we deploy our systems in real platforms and test them with real-world workload, where possible. We use the latency SLA commitment and the resource utilization as the performance measure.

1.2.3 Challenges

We have faces several challenges during this work. First, most of the systems presented in literature are not publicly available to download and experiment with. Thus, we have to implement our own versions of the algorithms/architectures following the description in the literatures for comparisons in most of the cases. Another significant obstacle that we face when doing the experiments are the lack of available real-world workload or data set. For example, in the work of ProRenaTa, we have to create our own synthetic workload by downloading access log traces of Wikipedia from Amazon public dataset. Finally, we have experienced performance interference of virutal machines (VMs) on Cloud platforms since we are often not able to control the allocation of VMs on physical machines. Worse, service latency is a sensitive measure to VM performance interference. We have to deploy and evaluate our systems multiple times to filter out potential performance interference introduced by the activities of other VMs in order to assure the credibility of our results.

1.2.4 Experimental Evaluation

We conduct all of our experiments using virtual machines in a Cloud environment. By using VMs, we are able to use a clean and relatively isolated environment for experiments. However, we have experience performance interference from other VMs on the same platform as explained in the previous section. For the systems that are used for comparisons, since most of them are not open source, we have to implement prototypes by following the algorithms and methodologies described in the literature. We choose to use real-world workloads to increase the credibility of our evaluations. We construct the workload from public data sets in order to facilitate reproducibility.

1.3 Thesis Contributions

The contributions of this thesis are as follows.

1. A performance evaluation of a distributed storage system in a simulated Cloud platform. It identifies the requirements of deploying distributed storage systems in a community Cloud environment.

2. A survey of replication techniques for distributed storage systems, that summarizes the state-of-the-art data replication methods and discusses the consistency issues come with data replication. The survey concludes with open issues, topics for further research.

3. GlobLease, a distributed storage system that uses leases to achieve consistent and low latency read/write accesses in a global scale This work proves that using leases is able

to improve the access latency of read/write requests by reducing the communication cost among replicas distributed in a massive scale.

4. BwMan, a bandwidth manager that arbitrates bandwidth allocation among multiple services sharing the same network infrastructure. BwMan can be used to guarantee latency SLAs for distributed storage systems.
5. ProRenaTa, an elasticity controller for distributed storage systems that combines proactive and reactive control techniques to achieve better SLA commitments and resource utilization. It is also innovative in the modeling of data migration overhead when scaling a distributed storage system.

1.4 Thesis Organization

The rest of this thesis organized as follows. Chapter 2 gives the necessary background and describes the systems used in this research work. Thesis contributions are discussed in Chapter 4. Chapter 5 contains conlusions and future work. Part II contains the research papers produced during this work.

Chapter 2

Background

Hosting services in the Cloud are becoming more and more popular because of a set of desired properties provided by the platform, which include low setup cost, unlimited capacity, professional maintenance and elastic provisioning. Services that are elastically provisioned in the Cloud are able to use platform resources on demand, thus saving hosting costs by appropriate provisioning. Specifically, instances are spawned when they are needed for handling an increasing workload and removed when the workload drops. Enabling elastic provisioning saves the cost of hosting services in the Cloud, since users only pay for the resources that are used to serve their workload.

In general, Cloud services can be coarsely characterized in two categories: state-based and stateless. Examples of stateless services include front-end proxies, static web servers, etc. Distributed storage service is a state-based service, where state/data needs to be properly maintained to provide a correct service. Thus, managing a distributed storage system in the Cloud exposes many challenging issues. In this thesis, we focus on the self-management and performance aspect of a distributed storage system deployed in the Cloud. Specifically, we examine techniques in order to design a distributed storage system that can operate efficiently in a Cloud environment [1, 2]. Also, we investigate approaches that support a distributed storage system to perform well in a Cloud environment by achieving a set of desired properties including elasticity, availability, and performance guarantees (Service Level Agreements).

In the rest of this chapter, we present the concepts and techniques used in the papers included in this thesis; Cloud environment, a visualized environment to effectively and economically host services; distributed storage systems, storage systems that are organized in a decentralized fashion; auto-scaling techniques, methods that are used to achieve elasticity in a Cloud environment; OpenStack Swift, a distributed storage system that is used as a study case.

2.1 Cloud Computing

"Cloud Computing refers to both the applications delivered as services over the Internet and the hardware and systems software in the data centers that provide those services [4]." A

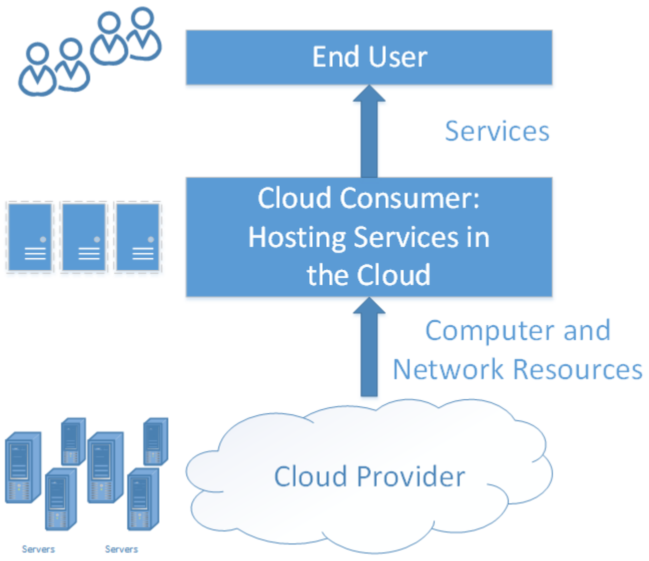
End User
Services
Cloud Consumer: Hosting Services in the Cloud
Computer and Network Resources
Cloud Provider
Servers
Servers

Figure 2.1. The Roles of Cloud Provider, Cloud Consumer, and End User in Cloud Computing

Cloud is the integration of data center hardware and software that provides "X as a service (XaaS)" to clients; value of X can be infrastructure, hardware, platform, and software. Those services in the Cloud are made available in pay-as-you-go manner to public. The advantages of Cloud computing to Cloud providers, consumers, and end users are well understood. Cloud providers make profits in renting the resources, providing services based on their infrastructures to Cloud consumers. Cloud consumers, on the other hand, greatly enjoy the simplified software and hardware maintenance and pay-as-you-go pricing model to start their business. Also, Cloud computing makes a illusion to Cloud consumers that the resources in the Cloud are unlimited and is available whenever requested without building or provisioning their own data centers. End users is able to access the services provided in the Cloud anytime and anywhere with great convenience. Figure 2.1 demonstrates the roles of Cloud provider, Cloud consumer, and end user in Cloud computing.

Based on the insights in [4], there are three innovations in Cloud computing:

1. The illusion of infinite computing resources available on demand, thereby eliminating the need for Cloud consumers to plan far ahead for provisioning;

2. The elimination of an up-front commitment by Cloud consumers, thereby allowing companies to start small and increase hardware resources only when there is an increase in their needs;

3. The ability to pay for use of computing resources on a short-term basis as needed (e.g., processors by the hour and storage by the day) and release them when they are no longer needed.

Elasticity

Elasticity is a property of a system, which allows the system to adjust itself in order to offer satisfactory service with minimum resources (reduced cost) in the presence of changing workloads. Typically, an elastic system is able to add or remove service instances (with proper configurations) according to increasing or decreasing workloads with respect to meet the Service Level Agreements, if any. To support elasticity, a system needs to be scalable, which means that its capability to serve workload is proportional to the number of service instances deployed in the system. Then, the hosting platform needs to be scalable, i.e., having enough resources to allocate whenever requested. Cloud computing is a perfect suit in this aspect. Another requirement to achieve elasticity is to have a controller that can automatically scale the system based on the intensity of the incoming workload. A simple example is the elastic scaling service provided by Amazon using Amazon Cloud Watch [5]. Background of auto-scaling techniques are introduced in Section 2.3.

Service Level Agreement

Service Level Agreements (SLA) define the quality of service that are expected from the service provider. SLAs are usually negotiated and agreed between Cloud service providers and Cloud service consumers. A SLA can define the availability aspect and/or performance aspect of a service, such as service up-time, service percentile latency, etc. A violation of SLA affects both the service provider and consumer. When a service provider is unable to uphold the agreed level of service, they typically pay penalties to the consumers. From the consumers perspective, a SLA violation can result in degraded service to their clients and consequently lead to loss in profits. Hence, SLA commitment is essential to the profit of both Cloud service providers and consumers.

2.2 Distributed Storage System

A distributed storage system provides an unified storage service by aggregating and managing a large number of storage nodes. A scalable distributed storage system can, in theory, aggregate unlimited number of storage nodes, therefore providing unlimited storage capacity. Distributed storage solutions include relational databases, NoSQL databases, distributed file systems, array storages, and key-value stores. The rest of this section provide backgrounds on the three main aspects of a distributed storage system, i.e., organizing structure, data replication and data consistency.

2.2.1 Structures of Distributed Storage Systems

Usually, a distributed storage system is organized either using a hierarchical or symmetric structure.

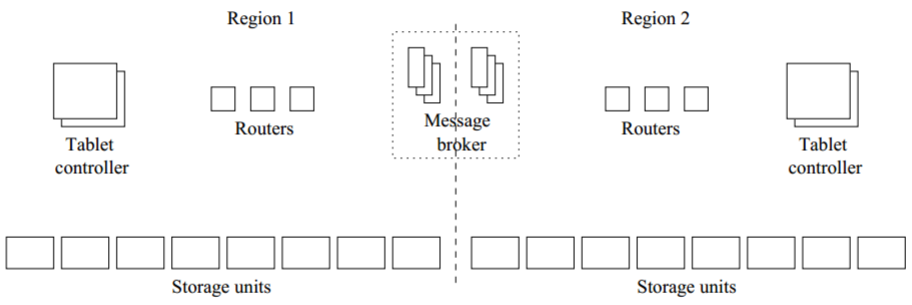

Figure 2.2. Storage Structure of Yahoo! PNUTS

Hierarchical Structure

A hierarchical distributed storage system is constructed with multiple components responsible for different functionalities. For example, separate components can be designed to maintain storage namespace, request routing or the actual storage. Recent representative systems organized in hierarchical structures are Google File System [6], Hadoop File System [7], Google Spanner [8] and Yahoo! PNUTS [9]. An example is given in Figure 2.2, which shows the storage structure of Yahoo! PNUTS. It uses tablet controller to maintain storage namespace, separate router components to route requests to responsible tablet controllers, message brokers to asynchronously deliver message among different storage regions and storage units to store data.

Symmetric Structure

A symmetrically structured distributed storage system can also be understood as a peer to peer (P2P) storage system. It is a storage system that does not need centralized control and the algorithm running at each node is equivalent in functionality. A distributed storage system organized in this way has robust self-organizing capability since the P2P topology often changes as nodes join or leave the system. It is also scalable since all nodes function the same and organized in a decentralized fashion, i.e., there is no centralized bottleneck. Availability is achieved by having data redundancies in multiple peers in the system.

An efficient resource location algorithm in the P2P overlay is essential to the performance of a distributed storage system built with P2P structure. One core requirement of such algorithm is the capability to adapt to frequent topology changes. Some systems use a centralized namespace service for searching resources, which is proved to be a bottleneck REF. An elegant solution to this issue is distributed hash table (DHT). It uses the hash of resource names to locate the object. Different routing strategies and heuristics are proposed to improve the routing efficiency.

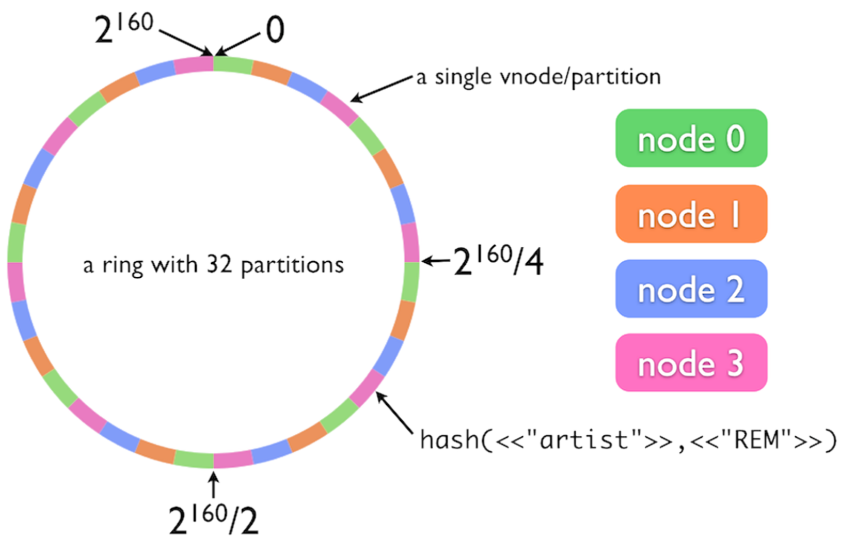

Figure 2.3. Distributed Hash Table with Virtual Nodes

Distributed Hash Table

Distributed Hash Table (DHT) is widely used in the design of distributed storage systems [10, 11, 2]. DHT is a structured peer to peer overlay that can be used for namespace partitioning and request routing. DHT partitions the namespace by assigning each node participating in the system a unique ID. According to the assigned ID, a node is able to find its predecessor (first ID before it) or successor (first ID after it) in the DHT space. Each node maintains the data that fall into the range between its ID and its predecessor's ID. As an improvements, nodes are allowed to hold multiple IDs, i.e., maintaining data in multiple hashed namespace ranges. These virtual ranges are also called virtual nodes in literature. Applying virtual nodes in a distributed hash table brings a set of advantages including distributing data transfer load evenly among other nodes when a node joins/leaves the overlay and allowing heterogeneous nodes to host different number of virtual nodes, i.e., handling different loads, according to their capacities. Figure 2.3 presents a DHT namespace distributed among four nodes with virtual nodes enabled.

Request routing in a DHT is handled by forwarding request through predecessor links, successor links or finger links. Finger links are established among nodes based on some criteria/heuristics for efficient routing [12, 13]. Algorithms are designed to update those links and stabilize the overlay when nodes join and leave. Load balancing among nodes is also possible by applying techniques, such as virtual nodes, in a DHT.

2.2.2 Data Replication

Data replication is usually employed in a distributed storage system to provide higher data availability and system scalability. In general approaches, data are replicated in different disks, physical servers, racks, or even data centers. In the presence of data loss or corruption caused by server failures, network failures, or even power outage of the whole data center, the data can be recovered from other correct replicas and continuously serving to the clients. The system scalability is also improved by using the replication techniques. Concurrent clients is able to access the same data at the same time without bottlenecks by having multiple replicas of the data properly managed and distributed. However, data consistency needs to be properly handled as a side effect of data replication and will be briefly introduced in Section 2.2.3.

Replication for Availability

A replicated system is designed to provide services with high availability. Multiple copies of the same data are maintained in the system in order to survive server failures. Through well-designed replication protocol, data loss can be recovered through redundant copies.

Replication for Scalability

Replication is not only used to achieve high availability, but also to make a system more scalable, i.e., to improve ability of the system to meet increasing performance demands in order to provide acceptable level of response time. Imagine a situation, when a system operates under so high workload that goes beyond the system's capability to handle it. In such situation, either system performance degrades significantly or the system becomes unavailable. There are two general solutions for such scenario: *vertical scaling*, i.e., scaling up, and *horizontal scaling*, i.e., scaling out. For vertical scaling, data served on a single server are partitioned and distributed among multiple servers, each responsible for a part of data. In this way, the system is capable to handle larger workloads. However, this solution requires much knowledge on service logic, based on which, data partitioning and distribution need to be performed in order to achieve scalability. Consequently, when scaled up, the system might become more complex to manage. Nevertheless, since only one copy of data is scattered among servers, data availability and robustness are not guaranteed. One the other hand, horizontal scaling replicates data from one server on multiple servers. For simplicity without losing generality, assume all replication servers are identical, and client requests are distributed evenly among these servers. By adding servers and replicating data, system is capable to scale horizontally and handle more requests.

Geo-replication

In general, accessing the data in close proximity means less latencies. This motivates many companies or institutes to have their data/service globally replicated and distributed by using globally distributed storage systems, for example [8, 2]. New challenges appear when designing and operating a globally distributed storage system. One of the most essential issues is the communication overhead among the servers located in different data centers. In this case, the communication latency is usually higher and the link capacity is usually lower.

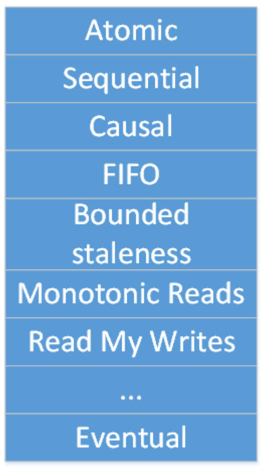

Figure 2.4. Different Data Consistency Levels

2.2.3 Data Consistency

Data replication also brings new challenges for system designers including the challenge of data consistency that requires the system to tackle with the possible divergence of replicated data. Various consistency levels, as shown in Figure 2.4, are proposed based on different usage scenarios and application requirements. There are two general approaches to maintain data consistency among replicas: master-based and quorum-based.

Master-based consistency

A master based consistency protocol defines that, within a replication group, there is a master replica of the object and the other replicas are slaves. Usually, it is designed that the master replica is always uptodate while the slave replicas can be a bit outdated. The common approach is that the master replica serialize all write operations while the slave replicas are capable of serving parallel read operations.

Quorum-based consistency

A replication group/quorum involves all the nodes that maintain replicas of the same data object. The number of replicas of a data object is the replication degree and the quorum size (N). Assume that read and write operations of a data object are propagated to all the replicas in the quorum. Let us define R and W responses are needed from all the replicas to complete a read or write operation. Various consistency levels can be achieved by configuring the value or R and W. For example, in Cassandra [11], there are multiple consistency choices telling the system how to handle a read or write operation. Specifically, what is the minimum number of R and W to complete a request. Different combinations of R and W values result in different consistency level. For example, sequential consistency can be achieved by having R+W>N.

2.2.4 OpenStack Swift

OpenStack Swift is a distributed object storage system, which is part of OpenStack Cloud Software [14]. It consists of several different components, providing functionalities such as highly available and scalable storage, lookup service, and failure recovery. Specifically, the highly available storage service is achieved by data replication in multiple *storage servers*. Its scalability is provided with the aggregated storage from multiple storage servers. The lookup service is performed through a Swift component called *proxy server*. Proxy servers are the only access entries for the storage service. The main responsibility of a proxy server is to process the mapping of the names of the requested files to their locations in the storage servers, similar to the functionality of NameNodes in GFS [6] and HDFS [7]. The namespace mapping is provided in a static file called *the Ring file*. Thus, the proxy server itself is stateless, which ensures the scalability of the entry points in Swift. The Ring file is distributed and stored on all storage and proxy servers. When a client accesses a Swift cluster, the proxy server checks the Ring file, loaded in its memory, and forwards client requests to the responsible storage servers. The high availability and failure recovery are achieved by processes called *the replicators*, which run on each storage server. Replicators use the Linux rsync utility to push data from a local storage server to other storage servers, which should maintain the same replicated data based on the mapping information provided in the Ring file. By doing so, the under-replicated data are recovered.

2.3 Auto-scaling Techniques

Typical methods used for auto-scaling are threshold-based rules, reinforcement learning or Q-learning (RL), queuing theory, control theory and time series analysis. We have used techniques of reinforcement learning, control theory and time series analysis in the papers presented in the second section of the thesis.

2.3.1 Threshold-based Rules

The representative systems that use threshold-based rules to scale a service are Amazon Cloud Watch [5] and RightScale [15]. Simply speaking, this approach defines a set of thresholds or rules in advance. Violating the thresholds or rules to some extent will trigger the action of scaling. Threshold-based rule is a typical implementation of reactive scaling.

2.3.2 Reinforcement Learning or Q-learning (RL)

Reinforcement learning are usually used to understand the application behaviors by building empirical models. The empirical models are built by learning through direct interaction between monitored metrics and control metrics. After sufficient training, the empirical models are able to be consulted and referred to when making system scaling decisions. The accuracy of the scaling decisions largely depend on the consulted value from the model. And the accuracy of the model depends on the metrics and model selected, as well as the amount of data trained to the model. For example, [16] presents an elasticity controller that

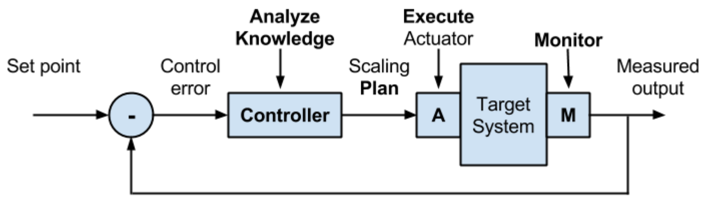

Figure 2.5. Block Diagram of a Feedback Control System

integrates several empirical models and switches among them to obtain better performance predictions. The elasticity controller built in [17] uses analytical modeling and machine-learning. They argued that by combining both approaches, it results in better controller accuracy.

2.3.3 Queuing Theory

Queuing theory can be also applied to the design of an elasticity controller. It makes reference to the mathematical study of waiting lines, or queues. For example, [18] uses the queueing theory to model a Cloud service and estimates the incoming load. It builds proactive controllers based on the assumption of a queueing model with metrics including the arrival rate, the inter-arrival time, the average number of requests in the queue. It presents an elasticity controller that incorporates a reactive controller for scale up and proactive controllers for scale down.

2.3.4 Control Theory

Elasticity controllers using control theory to scale systems are mainly reactive, but there are also some proactive approximations such as Model Predictive Control (MPC), or even combining a control system with a predictive model. Control systems are be broadly categorized into three types: open-loop, feedback and feed-forward. Open-loop controllers use the current state of the system and its model to estimate the coming input. They do not monitor (use feedback signals) to determine whether the system output has met the desired goal. In contrast, feedback controllers use the output of the system as a signal to correct any errors from the desired value. Feed-forward controllers anticipate errors that might occur in the output before they actually happen. The anticipated behavior of the system is estimated based on a model. Since, there might exist deviations in the anticipated system behavior and the reality, feedback controllers are usually combined to correct prediction errors.

Figure 2.5 illustrates the basic structure of a feedback controller. It usually operates in a MAPE-K (Monitor, Analysis, Plan, Execute, Knowledge) fashion. Briefly, the system monitors the feedback signal of a selected metric as the input. It analyzes the input signal using the method implemented in the controller. The methods can be broadly placed into four categories: fixed gain control, adaptive control, reconfiguring control and model predictive control. After the controller has analyzed the input (feedback) signal, it plans the scaling actions and sends them to actuators. The actuators are the methods/APIs to resize the target system. After resizing, another round of feedback signal is input to the controller.

2.3.5 Time Series Analysis

A time-series is a sequence of data points, measured typically at successive time instants spaced at uniform time intervals. The purpose of applying time series analysis in auto-scaling problem is to provide predicted value of an interested input metric (CPU load or input workload) to facilitate the decision making of an elasticity controller. Techniques used for this purpose in the literature are Moving Average, Auto-regression, ARMA, exponential smoothing and machine learning based approaches.

Chapter 3

Performance Optimization for Distributed Storage Systems Hosted in the Cloud

We optimize the performance of a distributed storage system deployed in the Cloud from two different angels. One optimization is conducted on the design of distributed storage systems. Another approach is to design smart agents/middlewares that helps a distributed storage system to operate efficiently in a Cloud environment.

3.1 Optimizations on the Design of Distributed Storage Systems

We optimize the performance of distributed storage systems with respect to service latency. Service latency of a distributed storage systems is determined by the load of the system and the algorithm designed to process the request. Discussions on alleviating load to improve the performance of a distributed storage system are presented in the next section. In this section, we investigate from the design of storage systems. Data are usually stored in a replicated fashion in a distributed storage system. Data replication guarantees the high availability of the data and increase service concurrency if there is no data consistency constraints. However, some applications may expect the underlying storage system to return strongly consistent data. The overhead to synchronize the replicated data can significantly increase service latency. This effect gets more obvious when the communication cost among replicas increase. Geo-replication is such an example. Our focus is to design and implement data consistency algorithms that have low communication overhead with respect to specific workload patterns and system deployment. The reduced data synchronization overhead in a replicated storage system can largely improve its service latency.

3.1.1 Summary of Proposed Optimizations

In Chapter 7, we present a survey on the state of the art replication techniques used in the design of distributed storage systems. Data replication provides the system with a set of desired properties include high performance, tolerant to failures, and elasticity. We have compared and categorized replication techniques used in 4 state of the art systems. We have defined a design space for replication techniques, identified current limitations, challenges. Then, in Chapter 8, we have designed our own replication strategy that has low data synchronization overhead in order to achieve strong consistency.

Then, we present GlobLease, an elastic, globally distributed and consistent key-value store. It is organised as multiple distributed hash tables (DHTs) storing replicated data and namespace. Across DHTs, data lookups and accesses are processed with respect to the locality of DHT deployments. We explore the use of leases in GlobLease to maintain data consistency across DHTs. The leases enable GlobLease to provide fast and consistent read access in a global scale with reduced global communications. The write accesses are optimized by migrating the master copy to the locations, where most of the writes take place. The elasticity of GlobLease is provided in a fine-grained manner in order to precisely and efficiently handle spiky and skewed read workloads. In our evaluation, GlobLease has demonstrated its optimized global performance, in comparison with Cassandra, with read and write latency less than 10 ms in most of the cases.

3.2 Optimizations on the Load of Distributed Storage Systems

In order to benefit from the pay-as-you-go pricing model in the Cloud, a dynamic resource provisioning middleware (an elasticity controller) is needed. A core requirement of such system is that it should be able to help saving the provisioning cost of an application without sacrificing its performance. Specifically, the system needs to provision the minimum resources as needed by the application to achieve a desired quality of service, which is usually negotiated between Cloud provider and Cloud consumer and defined as the Service Level Agreement (SLA) . In the view of Cloud providers, the dynamic resource provisioning system needs to keep the performance promise in the SLA under any circumstances, e.g. dynamic incoming workloads to the application, otherwise a penalty needs to be paid. In the view of Cloud consumer, a violation in the SLA usually causes bad service experiences to their end users and, as a result, influences profit. Thus, it is essential to have a well designed elasticity controller for both Cloud provider and Cloud consumer that provides the following properties:

1. Accurate resource allocation that satisfy both constraints: minimize provisioning cost and SLA violations.

2. Swift adaptation to workload changes without causing resource oscillation.

3. Efficient use of resources under SLA constraints during scaling. Specifically, when scaling up, it is preferable to add instances at the very last possible moment. In

contrast, during scaling down, it is better to remove instances as soon as they are not needed anymore. The timings are challenging to control.

The services hosted in the Cloud can be coarsely categorized into two categories: stateless and stateful. Dynamic provisioning of stateless services is relatively easy since less/no overhead is needed to prepare a Cloud VM before it can serve workloads, i.e., adding or removing Cloud VMs affects the performance of the service immediately. On the other hand, scaling a stateful service requires states to be properly transferred/configured from/to VMs. Specifically, when scaling up a stateful system (adding VMs), a VM is not able to function until proper states are transferred to it. When scaling down a stateful system (removing VMs), a VM cannot be safely removed from the system until its states are arranged to be handled/preserved by other VMs. Furthermore, this scaling overhead creates additional workload for the other VMs in the system and can result in the degradation of system performance if not well handled. Thus, it is challenging to scale a stateful system and adds an additional requirement for the design of an elasticity controller;

4. Be aware of scaling overheads, including the consumption of system resources and time, and prevent them from causing SLA violations.

3.2.1 Summary of Proposed Optimizations

In Chapter 9, we present a bandwidth manager (BwMan) for distributed storage systems. BwMan predicts and performs the bandwidth allocation and tradeoffs between multiple service activities in a distributed storage system. The performed bandwidth allocation is able to guarantee the performance of the storage system with respect to meeting Service Level Agreements (SLAs). To make management decisions, BwMan uses statistical machine learning (SML) to build predictive models. This allows BwMan to arbitrate and allocate bandwidth dynamically among different activities to satisfy different SLAs.

Then, in Chapter 10, we present ProRenaTa, an elasticity controller for distributed storage systems. The advantage of ProRenaTa is that it guarantees the specified latency SLAs of the underlying storage system with the provisioning of a minimum number of VM instances. To achieve this, ProRenaTa needs to know an accurate prediction of the incoming workload, which is designed and tested with real-world traces in the chapter. Then, we have built and trained performance models to map the workload intensity and the desired number of VMs. The performance model is used by a proactive controller to prepare VMs in advance and a reactive controller to correct possible prediction errors, which is mapped to the number of VMs. In order to further guarantee the performance of the system and saving the provisioning cost, a data migration model is designed and presented. By using it, ProRenaTa scheduler is able to schedule data migration to/from added/removed VMs without hurting the performance of the system. Furtermore, it also facilitates ProRenaTa to add/remove VMs at the best possible time. With all this, ProRenaTa outperforms the state of the art approaches in guaranteeing a high level of SLA commitments while improving the overall resource utilization.

Chapter 4

Thesis Contribution

In this chapter, we describe the thesis contributions. First, we list the publications produced during this work. Then, we provide details on the contributions of each publication separately and highlight the individual contributions of the thesis author.

4.1 List of Publications

1. Y. Liu, V. Vlassov, L. Navarro, *Towards a Community Cloud Storage*, 28th International Conference on Advanced Information Networking and Applications (AINA), 2014

2. Y. Liu, V. Vlassov, *Replication in Distributed Storage Systems: State of the Art, Possible Directions, and Open Issues*, 5th IEEE International Conference on Cyber-enabled distributed computing and knowledge discovery (CyberC), 2013

3. Y. Liu, X. Li, V. Vlassov, *GlobLease: A Globally Consistent and Elastic Storage System using Leases*, 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS), 2014

4. Y. Liu, V. Xhagjika, V. Vlassov, A. Al-Shishtawy, *BwMan: Bandwidth Manager for Elastic Services in the Cloud*, 12th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA), 2014

5. Y. Liu, N. Rameshan, E. Monte, V. Vlassov and L. Navarro, *ProRenaTa: Proactive and Reactive Tuning to Scale a Distributed Storage System*, Accepted for publication in 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2015

4.2 Performance Evaluation of a Distributed Storage System

The evaluation of OpenStack Swift regarding its feasibility to be deployed in a community Cloud environment is published as a research paper [19] and appears as Chapter 6 in this thesis.

Paper Contribution

This paper contributes in two aspects. The first contribution is the identification of the characteristics in a community Cloud environment regarding deploying a distributed storage service. The second contribution is the performance evaluation of Swift, a distributed object storage system, running under different platform factors including hardware, network, and failures. Our results facilitates the understanding of community Cloud characteristics and proves the feasibility of deploying Swift in a community Cloud environment. A community Cloud is established by connecting heterogeneous computing and networking resources in a wide geographic area REF. We have identified three main differences between a community Cloud and data center Cloud:

1. The types of computing resources: powerful dedicated servers vs. heterogeneous computing components;

2. The network features: exclusive high speed networks vs. shared ISP broadband and wireless connections;

3. The maintenance: regular vs. self-organized.

Based on the identification above and real studies on community Clouds REF, we simulate a community Cloud environment in the aspects of hardware resource, network connectivity, and failure rates.

Our evaluation results have established the relationship between the performance of a Swift cluster and the major environment factors in a community Cloud, including limited hardware resources and unstable network features. Furthermore, in order to tackle with relatively frequent server failures in a community Cloud, a self-healing control system is implemented for Swift and proved to be useful. We conclude that it is possible to deploy Swift in a community Cloud environment if those environment factors are in the range of our simulations, which is our future work.

Thesis Author Contribution

The author was the main contributor in this work, including writing most of the article. He came up with the methodology to evaluate the performance aspect of Swift in a simulated community Cloud environment. He also finished all the implementations and evaluations on the platform and presents the results in the paper.

4.3 State-of-the-Art Survey on Distributed Storage Systems

Our study on the replication techniques that are used to design distributed storage systems is published as a research paper [1] and appears as Chapter 7 in this thesis.

Paper Contribution

In this paper, we investigate the following three most essential issues related to data replication, to be considered when developing a replicated distributed storage service: replication for availability, replication for performance, and replica consistency.

To examine these issues, we first present the general techniques that are used to achieve replication in a distributed storage system. Then, we present different levels of data consistency that is usually achieved in production systems according to different usage scenarios and application requirements. Finally, we consider, as case studies, replication and consistency in four production distributed storage systems, namely: Dynamo of Amazon [10], Cassandra of Facebook [11], Spanner of Google [8], and Yahoo! PNUTS [9]. In our view, these four systems and others not considered here (e.g., Voldemort, Swift, and Windows Azure Storage), define state of the art in the field of large-scale distributed storage systems, and furthermore, each of the systems has its own unique properties that distinguish it from the other systems. Our discussion focuses on the replication rationales, usage scenario, and their corresponding consistency models. To be concise, we have investigated flexibility of replica placement and migration mechanisms, management of replication degree, and tradeoff of consistency guarantees for performance in the studied systems. At the end of the paper, we identify open issues and suggest future research directions regarding replication techniques for designing distributed storage systems.

Thesis Author Contribution

The author was the main contributor in this work, including writing most of the article. The survey on the related work and the characterization was studied by the thesis author. He is also the main contributor in summarizing the techniques used for data replication in distributed storage systems. The author is also responsible for discussing the limitations and proposing future research directions based on surveying a large number of research articles.

4.4 A Manageable Distributed Storage System

In [2], we propose a distributed storage system, GlobLease, which is designed for low latency access in a global scale with exposure of rich configuring APIs. The paper appears as Chapter 8 in this thesis.

Paper Contribution

In this work, we present GlobLease, an elastic, globally-distributed and consistent key-value store. GlobLease differs from the state of the art systems in three ways. First, it is organised as multiple distributed hash tables (DHTs) storing replicated data and namespace, which allows more flexible management when different DHTs are placed in different locations. Specifically, across DHTs, data lookups and accesses are processed with respect to the locality of DHT deployments. Second, GlobLease uses leases to maintain data consistency among replicas. Using leases allows GlobLease to provide fast and consistent read access in a global scale with reduced global communications. Write accesses are also optimized by migrating the master copy of data to the locations, where most of the writes take place. Third, GlobLease is high adaptable to different workload patterns. Specifically, fine-grained elasticity is achieved in GlobLease using key-based multi-level lease management, which allows GlobLease to precisely and efficiently handle spiky and skewed read workloads.

GlobLease is not only a storage system that achieves optimized global performance, but also a system that can be tuned during runtime. GlobLease exposes many performance tuning APIs, such as, API to adjust a key's replication degree, API to migrate replicas and API to add lightweight affiliated nodes for handling transient spiky or skewed workloads.

Thesis Author Contribution

The author is a major contributor in this work, including writing all of the paper. He played a major role in designing and implementing the storage system that uses leases. He led the implementation and actively refining the design during the implementation. The author is also responsible in designing the experiments and conducting them in Amazon EC2.

4.5 Arbitrating Resources among Different Workloads in a Distributed Storage System

Our work on bandwidth arbitration among different workloads in a distributed storage system, is published as a conference paper [3]. The complete paper appears as Chapter 9 of this thesis.

Paper Contribution

In this work, we first identify that there are mainly two types of workloads in a distributed storage system: user-centric workload and system-centric workload. User-centric workload is the load that are created by client requests. Workloads that are associated with system maintenance including load rebalancing, data migration, failure recovery, and dynamic reconfiguration are system-centric workload. We demonstrate that without explicitly managing the resources among different workloads leads to unstable performance of the system and results in SLA violations.

From our experimental observations, in a distributed storage system, both user-centric and system-centric workloads are network bandwidth intensive. Then, we propose BwMan, a network bandwidth manager for distributed storage systems. BwMan arbitrates the bandwidth allocation among individual services and different service activities sharing the same infrastructure. Each service can have a demanded and dedicated amount of bandwidth allocation without interfering among each other. Dynamic and dedicated bandwidth allocation to services supports their elasticity properties with reduced resource consumption and better performance guarantees. In the perspective of user-centric workload, from our evaluation, we show that more than half of the SLA violations is prevented by using BwMan for an elastic distributed storage. In the perspective of system-centric workload, we are able to have a better estimation on the finish time of, for example, failure recovery jobs and data replication jobs. Furthermore, since BwMan controls bandwidth in port granularity, it can be easily extended to adapt to other usage scenarios where network bandwidth is a sharing resource and creates potential bottlenecks.

Thesis Author Contribution

The author is a major contributor in this work, including writing most of the paper. The machine learning techniques used to model the bandwidth arbitration among multiple services is his individual work. The author also motivates the bandwidth management idea by finding concrete usage cases, i.e., SLA commitment while there are data corruptions in a distributed storage system. Also, he is the main contributor in implementing the bandwidth manager. In addition, he proposes and conducts the evaluation plans to validate BwMan.

4.6 Auto-scaling of a Distributed Storage System

Our work on a elasticity controller for a distributed storage system has been accepted to be published as a conference paper and appears as Chapter 10 in this thesis.

Paper Contribution

We provide a design of an elasticity controller for a distributed storage system that combines both proactive and reactive auto-tuning. First, we demonstrate that there are limitations while relying solely on proactive or reactive tuning to auto-scale a distributed storage system. Specifically, reactive controller can scale the system with a good accuracy since scaling is based on observed workload characteristics. However, a major disadvantage of this approach is that the system reacts to workload changes only after it is observed. As a result, SLA violations are observed in the initial phase of scaling because of data/state migration in order to add/remove instances in a distributed storage system and causes a period of disrupted service. Proactive controller, on the other hand, is able to prepare the instances in advance and avoid any disruption in the service. However, the accuracy of workload prediction largely depends on application-specific access patterns. Worse, in some cases workload patterns are not even predictable. Thus, proper methods need to be designed and applied to deal with the workload prediction inaccuracies, which directly influences the accuracy of scaling that in turn impacts SLA guarantees and the provisioning costs.

Then, we identify that, in essence, proactive and reactive approach complement each other. Proactive approach provides an estimation of future workloads giving a controller enough time to prepare and react to the changes but having the problem of prediction inaccuracy. Reactive approach brings an accurate reaction based on current state of the system but without leaving enough time for the controller to execute scaling decisions. So, we design ProRenaTa, which is an elasticity controller to scale a distributed storage system combining both proactive and reactive approaches.

Lastly, we build a data migration model to quantify the overhead to finish a scale plan in a distributed system, which is not explicitly considered or model in the state of the art works. The data migration model is able to guarantee less SLA violations while scaling the system. Because, by consulting the data migration model, ProRenaTa scheduler is able to smartly arbitrating the resources that are allocated to system resizing under the constraint of guaranteeing the SLA.

In sum, ProRenaTa improves the classic prediction based scaling approach by taking into account the scaling overhead, i.e., data/state migration. Moreover, the reactive controller helps ProRenaTa to achieve better scaling accuracy, i.e., better resource utilization and less SLA violations, without causing interference with the scaling activities scheduled by the proactive controller. Our results indicate that ProRenaTa outperforms the state of the art approaches by guaranteeing a high level of SLA commitments while also improving the overall resource utilization.

Thesis Author Contribution

The thesis author was a major contributor in this work, including writing most of the article. He is the main contributor in coming up with the idea and formalizing it. He motivates and approaches the problem in an unique angel. The author also finishes most part of the system implementation, including metric monitoring, proactive controller, reactive controller, ProRenaTa scheduler, and actuators to add and remove storage instances in GlobLease. He is also the main contributor in the evaluation design and implementation. Finally, the author also developed the data migration overhead model and validated its benefits in preventing SLA violations and boosting resource utilization.

Chapter 5

Conclusions and Future Work

In this thesis, we work towards building high performance elastic distributed storage solutions. We approach the problem by investigating the efficiency of the storage solutions themselves and smart agents to guarantee their performance under dynamic workload and environment. We give an introduction to distributed storage systems and we provide specific background knowledge on the systems and techniques we have applied in our research. Then, we summarize our contributions by presenting five research papers: a hands-on evaluation of a distributed storage system on a distributed platform, a survey on replication techniques used in distributed storage systems, a geo-distributed storage system that provides low latency services, a bandwidth manager to guarantee bandwidth allocation to elastic storage services, a elasticity controller that achieves better scaling accuracy and lower provision cost.

Providing low latency storage solutions has been a very active research area with a plethora of open issues and challenges to be addressed. Challenges for this matter include: the emerging of novel system usage scenarios, for example, global distribution, the uncertainty and dynamicity of incoming system workload, the performance uncertainty introduced by the virtualized deployment platform. Our work thrive to provide low latency storage solutions through investigating the above angles.

5.1 Summary of Results

The flexibility of Cloud computing allows elastic services to adapt to changes in workload patterns in order to achieve desired Service Level Agreements (SLAs) at a reduced cost. Typically, the service adapts to changes in workload by adding or removing service instances (VMs), which for stateful services will require moving data among instances. The SLAs of a distributed Cloud-based service are sensitive to the available network bandwidth, which is usually shared by multiple activities in a single service without being explicitly allocated and managed as a resource. With our work of BwMan, bandwidth is allocated smartly among multiple service activities with respect to meet desired SLAs. Experiments show that our work of BwMan is able to reduce SLA violations by a factor of two or more.

More and more IT companies are expanding their businesses and services to a global scale, serving users in several countries. Globally distributed storage systems are needed to reduce data access latency for clients all over the world. Unlike storage solutions that are deployed in one-site, global deployments of storage systems introduce new challenges. One essential difference is the communication overhead among data replicas has increased dramatically. In the scenario, where applications require strongly consistent data from the underlying storage systems, this difference will increase service latency dramatically. With our work of GlobLease, we tailored and invented novel data consistency algorithms to reduce the synchronization overhead among replicas. As a result, Globlease is able to reduce around 50% of high latency requests while guaranteeing the same data consistency level.

Provisioning stateful services in the Cloud that guarantees high quality of service with reduced hosting cost is challenging to achieve. There are two typical auto-scaling approaches: predictive and reactive. A prediction based controller leaves the system enough time to react to workload changes while a feedback based controller scales the system with better accuracy. Using either approach alone is not able to achieve satisfactory scaling results in this scenario. Limitations are shown when using a proactive or reactive approach in isolation to scale a stateful system. To overcome the limitations, we have implemented an elasticity controller, ProRenaTa, which combines both reactive and proactive approaches to leverage on their respective advantages. Furthermore, a data migration model is designed to explicitly handle the scaling overhead. As a result, we show that the combination of reactive and proactive approaches outperforms the state of the art approaches in guaranteeing a higher level of SLA commitments while improving the overall resource utilization.

5.2 Future Work

The research work described here have the opportunities to be improved in many ways. For the work to design low latency storage solutions in a global scale, we are particularly interested in data consistency algorithms that are able to provide the same consistency level while requiring less replica synchronization. We are investigating using metadata and novel message propagation patterns to reduce replica communication overhead.

Despite the contention of bandwidth, we would like to consider more factors that might influence the performance of a distributed storage system deployed in a Cloud environment. Particularly, we are heading towards analyzing the performance interference to the quality of a storage service. Performance interference happens very often when services are hosted in a Cloud environment, where there is usually no guarantee of VM collocations.

Furthermore, we are interested in improving the elasticity controller ProRenaTa. We are willing to consider more complicated workload scenarios to go beyond the assumption of uniformly distributed workload. Also, we would like to explore possibilities to reduce state migration overhead while scaling the system according to intensity of workload.

Bibliography

[1] Ying Liu and V. Vlassov. Replication in distributed storage systems: State of the art, possible directions, and open issues. In *Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), 2013 International Conference on*, pages 225–232, Oct 2013.

[2] Ying Liu, Xiaxi Li, and Vladimir Vlassov. Globlease: A globally consistent and elastic storage system using leases. `http://dx.doi.org/10.13140/2.1.2183.7763`. (To appear in) Proceedings of the 2014 IEEE International Conference on Parallel and Distributed Systems (ICPADS '14).

[3] Ying Liu, V. Xhagjika, V. Vlassov, and A. Al Shishtawy. Bwman: Bandwidth manager for elastic services in the cloud. In *Parallel and Distributed Processing with Applications (ISPA), 2014 IEEE International Symposium on*, pages 217–224, Aug 2014.

[4] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[5] Amazon cloudwatch. `http://aws.amazon.com/cloudwatch/`.

[6] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[7] K. Shvachko, Hairong Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10, May 2010.

[8] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google&rsquo;s globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, August 2013.

[9] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

[10] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[11] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[12] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM.

[13] Gurmeet Singh Manku, Mayank Bawa, Prabhakar Raghavan, et al. Symphony: Distributed hashing in a small world. In *USENIX Symposium on Internet Technologies and Systems*, page 10, 2003.

[14] Openstack cloud software. `http://www.openstack.org/`.

[15] Right scale. `http://www.rightscale.com/`.

[16] Simon J. Malkowski, Markus Hedwig, Jack Li, Calton Pu, and Dirk Neumann. Automated control for elastic n-tier workloads based on empirical modeling. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, pages 131–140, New York, NY, USA, 2011. ACM.

[17] Diego Didona, Paolo Romano, Sebastiano Peluso, and Francesco Quaglia. Transactional auto scaler: Elastic scaling of in-memory transactional data grids. In *Proceedings of the 9th International Conference on Autonomic Computing*, ICAC '12, pages 125–134, New York, NY, USA, 2012. ACM.

[18] A. Ali-Eldin, J. Tordsson, and E. Elmroth. An adaptive hybrid elasticity controller for cloud infrastructures. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pages 204–212, April 2012.

[19] Ying Liu, V. Vlassov, and L. Navarro. Towards a community cloud storage. In *Advanced Information Networking and Applications (AINA), 2014 IEEE 28th International Conference on*, pages 837–844, May 2014.

Part II

Research Papers

Chapter 6

Towards a Community Cloud Storage

Ying Liu, Vladimir Vlassov and Leandro Navarro

In *28th International Conference on Advanced Information Networking and Applications* (AINA), 2014

Abstract

Community Clouds, usually built upon community networks, operate in a more disperse environment compared to a data center Cloud, with lower capacity and less reliable servers separated by a more heterogeneous and less predictable network interconnection. These differences raise challenges when deploying Cloud applications in a community Cloud.

OpenStack Swift is an open source distributed storage system, which provides stand alone highly available and scalable storage from OpenStack Cloud computing components. Swift is initially designed as a backend storage system operating in a data center Cloud environment.

In this work, we illustrate the performance and sensitivity of OpenStack Swift in a typical community Cloud setup. The evaluation of Swift is conducted in a simulated environment, using the most essential environment parameters that distinguish a community Cloud environment from a data center Cloud environment.

6.1 Introduction

Many of Web2.0 applications, such as Wikis and social networks as well as Cloud-based applications may benefit from scalable and highly available storage systems. There are many distributed storage systems emerged to meet this goal, such as Google Spanner [1], Amazon Simple Storage Service [2], Facebook Cassandra [3],Yahoo! PNUTS [4], and OpenStack Swift [5]. However, all of these state-of-the-art systems are designed to operate in a data center environment, where under a normal workload, there are no hardware bottlenecks in the server capabilities as well as the network. We investigate the possibility to adapt these storage systems in an open area network environment. A realistic use case is the community Cloud [6, 7].

We consider a community Cloud built upon a community network. Specifically, community networks are large-scale, self-organized, distributed and decentralized systems constructed with a large number of heterogeneous nodes, links, content and services. Some resources, including nodes, links, content, and services, participating in a community network are dynamic and diverse as they are built in a decentralized manner, mixing wireless and wired links with diverse routing schemes with a diverse range of services and applications [8]. Examples of such community networks are Guifi.net in Spain [9] and AWMN in Greece [10]. A Cloud infrastructure in a community network can be used primarily to isolate and provide services, such as a storage service.

In order to provide Infrastructure as a Service Cloud in a community network and enable Cloud-based services and applications, a proper backend storage system is needed for various purposes, such as maintaining user information, storing Virtual Machine (VM) images as well as handling intermediate experiment results possibly processed with big data platforms [11, 12]. The first approach towards such a storage system is to evaluate the existing open source storage systems that may match our scenario. One of the widely-used open source Cloud software is OpenStack, which includes the storage system called Swift [5]. Studies have been conducted regarding the performance of Swift in a general Cloud environment, such as in the Amazon EC2, and in some private clouds [13, 14]. In this paper, we have provided the methodology and conducted a thorough evaluation of OpenStack Swift using a simulated environment regarding its properties to operate in a community Cloud environment. The evaluating results may also be extended to be used in multi-site Clouds and multi-Cloud federations with undedicated networks using best effort Internet, which has no guarantees in network latency and bandwidth.

Since Swift is originally designed to operate in a data center, we start our investigation by the identification of the differences between a data center environment and a community Cloud environment. In general, a data center is constructed with a large number of powerful dedicated servers connected with the exclusive, high bandwidth, and low latency network switches with regular maintenance. In contrast, a community Cloud upon a community network is established by connecting heterogeneous computing and networking resources in a wide geographic area. The three main differences between these two environments are:

- The types of computing resources: powerful dedicated servers vs. heterogeneous computing components;
- The network features: exclusive high speed networks vs. shared ISP broadband and wireless connections;
- The maintenance: regular vs. self-organized.

Based on the identification above, we have conducted the evaluations on a Swift cluster in the later sections.

The contributions of this paper are:

- We demonstrate a methodology to evaluate the performance of a distributed storage system in a simulated community Cloud environment;
- We quantify Swift performance under different hardware specifications and network features;
- We design and evaluate a self-healing algorithm on a Swift cluster to handle server failures;
- We investigate the feasibility to apply OpenStack Swift in a community Cloud environment.

The rest of this paper is organized as follows. Section 6.2 provides the background on the community Cloud and OpenStack Swift. Section 6.3 presents the experimental environment that we used for the Swift evaluation. Section 6.4 describes our evaluation plan. Section 6.5 and Section 6.6 present results of the evaluation of the Swift performance under different hardware and network configurations. Section 6.7 presents an implementation of a self-healing mechanism for Swift. We conclude in Section 6.8.

6.2 Background

6.2.1 Community Network and Community Cloud

Community networking, also known as bottom-up networking, is an emerging model for the Future Internet across Europe and beyond, where communities of citizens can build, operate and own open IP-based networks, a key infrastructure for individual and collective digital participation [15, 16]. Many services, including Internet access, cable TV, radio and telephone, can be provided upon the infrastructure of the community networks. An example is given in [17]. Furthermore, community networks can also be employed and extended to provide Cloud-based services, namely the *community Cloud*, to the community, researchers, and participating SMEs. Community Cloud provides alternative choices for its participants in the community to manipulate their own Cloud computing environment without worrying about the restrictions from a public Cloud provider.

However, building a community Cloud is very challenging. One of the major challenges is to achieve the self-management, high performance, and low cost services based on the overlay of the community networks, which is usually organized as the aggregation of a large number of widespread low-cost unreliable networking, storage and home computing resources. In this work, we provide an evaluation of an existing open source software, OpenStack Swift, in a simulated community Cloud environment.

6.2.2 OpenStack Swift

OpenStack Swift is one of the storage services of the OpenStack Cloud platform [18]. It consists of several different components, providing functionalities such as highly available and scalable storage, lookup service, and failure recovery. Specifically, the highly available storage service is achieved by data replication in multiple *storage servers*. Its scalability is provided with the aggregated storage from many storage servers.

The lookup service is performed through a Swift component called *the proxy server*. The proxy servers are the only access entries for the storage service. The main responsibility of a proxy server is to process the mapping of the names of the requested files to their locations in the storage servers. This namespace mapping is provided in a static file called *the Ring file*. Thus, the proxy server itself is stateless, which ensures the scalability of the entry points in Swift. The Ring file is distributed and stored on all the storage and proxy servers. When a client accesses a Swift cluster, the proxy checks the Ring file, loaded in its memory, and forwards client requests to the responsible storage servers.

The high availability and failure recovery are achieved by processes called *the replicators*, which run on every storage server. Each replicator uses the Linux rsync utility to push the data from the local storage server to the other storage servers, which should maintain the same replicated data, based on the information provided in the Ring file. By doing so, the under-replicated data are recovered.

6.2.3 Related Work

One of the leading companies for cloud backups called Zmanda has measured the performance of Swift on Amazon EC2 [13]. They have evaluated Swift performance with different proxy server and storage server capabilities by employing different flavors of Amazon instances. The result provides premium deployment strategy to satisfy the throughput requirements at a possible low cost. However, the evaluations only focus on the hardware setup of a Swift cluster. In addition, our work provides the evaluation of Swift's performance under different network environments, which is an important aspect in a community Cloud.

FutureGrid, the high performance experimental testbed for computer scientists, has evaluated several existing distributed storage systems towards a potential usage as VM repositories [19]. The evaluations are conducted by the comparison of MongoDB [20], Cumulus [21], and OpenStack Swift. They have concluded that all these three systems are not perfect to be used as an image repository in the community Cloud. However, Swift and Cumulus have the potential to be improved to match the requirements. In comparison, instead of comparing the performance of several open source Cloud storage systems, we analyze Swift performance under identified metrics described in Section 6.4.

Another scientific group has investigated the performance of Swift for CERN-specific data analysis [14]. The study investigated the performance of Swift with the normal use case of the ROOT software [22]. However, their work mainly focuses on the evaluation of Swift under the CERN-specific data usage pattern. Our work evaluates Swift performance under a general usage pattern but with different resource constrains, namely, server capacities and network features.

6.3 Experiment Environment

6.3.1 The Platform

In order to simulate a community Cloud environment, we start by constructing a private Cloud and tune some of its parameters according to the aspects that we want to evaluate. These parameters are explained at the beginning of every evaluation section. For the private Cloud environment, we have configured the OpenStack Compute (Nova) [23] and the Dashboard (Horizon) on a large number of interconnected high-end server computers. On top of the OpenStack Cloud platform, we have configured a Swift cluster with different VM flavors for our experiments. Specifically, the following four different VM flavors, which decide the capabilities of the servers in terms of CPU and memory, are constructed.

- XLarge Instance: 8 Virtual Core, 16384 MB Memory;
- Large Instance: 4 Virtual Core, 8192 MB Memory;
- Medium Instance: 2 Virtual Core, 4096 MB Memory;
- Small Instance: 1 Virtual Core, 2048 MB Memory.

Each virtual core is 2.8 GHz. The different capabilities of VMs are used in the experiments described in Section 6.5, which identify the bottom-line hardware requirements for Swift to operate according to a specific level of performance. These VMs are connected with 1Gbps sub-networks. Hard drives with 5400 rpm are mounted for the storage servers. In our view, this setup represents a reasonable environment in a community Cloud infrastructure.

6.3.2 The Swift Setup

Our Swift cluster is deployed with a ratio of 1 proxy server to 8 storage servers. According to OpenStack Swift Documentation [24], this proxy and storage server ratio achieves efficient usage of the proxy and storage CPU and memory under the speed bounds of the network and disk. Under the assumption of uniform workload, the storage servers are equally loaded. This implies that the Swift cluster can scale linearly by adding more proxy servers and storage servers following the composition ratio of 1 to 8. Due to the linear scalability, our experiments are conducted with 1 proxy server and 8 storage servers.

6.3.3 The Workload

We have modified the Yahoo! Cloud Service Benchmark [25] (YCSB) to generate the workloads for a Swift cluster. Our modification allows YCSB to support read, write, and delete operations to a Swift cluster with best effort or a steady workload according to a requested throughput. If the requested throughput is so high that requests cannot be admitted by the system, then the requests are queued for later execution, thus achieving the average target system throughput in the long run. Furthermore, YCSB is given 16 concurrent client threads and generates uniformly random read and write operations to the Swift cluster.

The Swift cluster is populated using randomly generated files with predefined sizes. Our experiment parameters are chosen based on parameters of one of the largest production Swift clusters configured by Wikipedia [26] to store images, texts, and links. The object size is 100KB as a generalization of the Wikipedia scenario.

6.3.4 The Network Instrumentation

We apply "tc tools" by NetEm [27] to simulate different network scenarios, and to be able to manage the network latency and bandwidth limitations for every thread or service.

6.4 Evaluation Plan

Swift is designed to achieve linear scalability. However, potential performance bounds can be introduced by the lack of computing resources of the proxy servers, disk read/write speed of the storage servers, and the network features of the Swift sub-networks, which connect the proxy servers and the storage servers. By understanding these potential performance bounds of Swift, as well as the major differences between a community Cloud environment and a data center Cloud environment identified in Section 6.1, we have set up the following three sets of experiments, which in our view, cover the major concerns to achieve a community storage Cloud using Swift.

Hardware requirement by Swift proxy servers

In this experiment, we focus on the identification of the minimum CPU and memory requirement of a Swift proxy server to avoid bottlenecks and achieve efficient resource usage, given a network and disk setup described in Section 6.3.1.

Swift Performance under different networks

In this set of experiments, we correlate Swift performance with different network features given that the proxy servers and the storage servers are not the bottleneck. The evaluations quantify the influences introduced by network latencies and insufficient bandwidth to the performance of a Swift cluster. The results can help a system administrator to predict and guarantee the quality of service of a Swift cluster under concrete network setups.

Swift's Self-healing property

According to our experience in a community Cloud, the servers are more prone to fail comparing to the servers in a data center. Thus, the applications deployed in a community Cloud are expected to have self-* properties to survive with server failures. In this experiment, we have examined the failure recovery mechanism in Swift and extended it with a self-healing control system. In our view, this self-healing mechanism is essential when Swift is deployed in a community Cloud.

6.5 Server Hardware Evaluation

It is intuitively clear and stated in Swift documentation [24] that the proxy servers are compute intensive and the storage servers are disk intensive. The following experiments are focused on the minimum hardware requirements of the proxy servers. The storage servers are configured with our Small VM flavors listed in Section 6.3.1, which are ensured not to be the bottleneck. The proxy servers are deployed with different flavors of VM instances.

Figure 6.1 and Figure 6.2 illustrate the 99th percentile read and write latencies (y-axis) under a specified steady workload (x-axis) generated from YCSB with the deployment of the proxy servers using different VM flavors. The figure is plotted using the results from five repeated runs in each setup.

Obviously, there are no significant performance differences when using Medium, Large, or even Xlarge VM flavors as Swift proxy servers, for both reads and writes. In contrast, when running the proxy server on a Small VM instance, the latencies of the write requests increase sharply after 30 operations per second (abbreviate as op/s). This phenomenon is also observed in the read experiments after the throughput of 120 op/s.

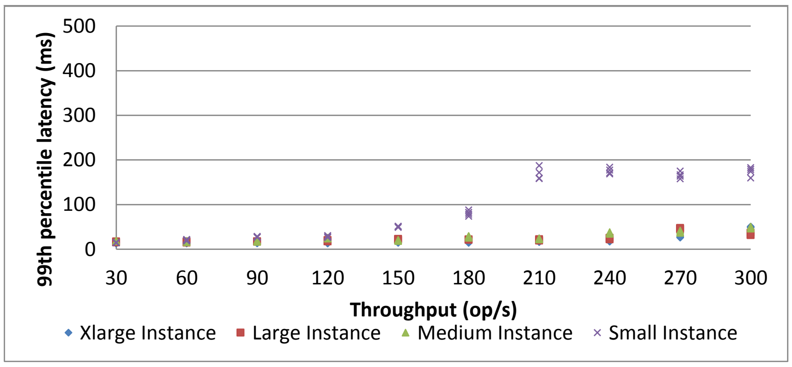

Figure 6.1. Read latency of Swift using proxies with different VM flavors

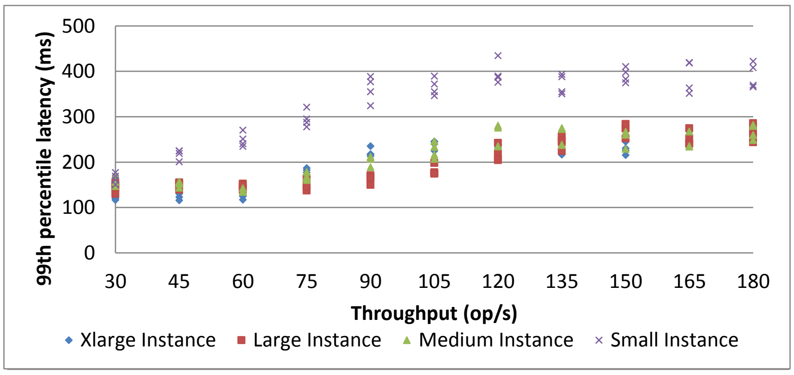

Figure 6.2. Write latency of Swift using proxies with different VM flavors

Discussion

In order to achieve efficient resource usage as well as acceptable performance without potential bottlenecks in Swift, our experiment demonstrates the correlation of the performance of a Swift cluster with the capabilities of the proxy servers with read or write workloads. In particular, the Small instance proxy servers, in our experiment, experience severe performance degradation after some throughput threshold values shown in Figure 6.1 and Figure 6.2 for read and write requests. We observe that the CPU of the Small instance proxy servers saturates after the threshold value, which results in the increase of request latency. The balance of request latency (system throughput) and the saturation of the CPU in the small instance proxy servers is reached at a later steady state shown in both Figure 6.1 and Figure 6.2.

It is important to know both the threshold value and the steady state value for every different deployment of the Swift proxy servers. This experiment demonstrates the methodology for identifying the potential bottlenecks in the capabilities of the proxy servers in a specific Swift setup. The threshold value defines the state where the latency of service starts to be affected significantly if the workload keeps increasing. The steady state value can be used to calculate the maximum reachable throughput of the system, given the number of concurrent clients. In between the threshold value and the steady state, a linear correlation of the request latency and the system workload with different coefficients for reads and writes is observed.

6.6 Network Evaluation

In this section, we focus on the evaluation of a Swift cluster under different network features. Specifically, we examine the effect of the network latency and the bandwidth limit to Swift's performance. We employ a Swift setup with the Medium proxy servers and the Small storage servers, which is demonstrated to have no bottleneck in our operational region in the previous section.

As introduced in Section 6.3.4, we use NetEm "tc tools" to simulate different network features in a Swift cluster. We have run two sets of experiments to evaluate the Swift under two important network factors, namely the network latency and the available bandwidth. In each set of the experiments, our results provide the intuitive understanding of the performance impact of these network factors on Swift. Furthermore, these performance impacts are compared with the results from the experiments conducted separately on storage servers and proxy servers. The storage server and proxy server comparison provides more fine-grained guidelines for the possible deployment of a Swift cluster in a community Cloud to achieve efficient network resource usage and desired performance.

6.6.1 Swift Experiment with Network Latencies

In this section, we present the evaluation results of Swift's performance under specific network latencies. The network latencies are introduced on the network interfaces of the proxy servers and the storage servers separately. In particular, for the read experiments, we have introduced the latencies on the outgoing links while, for the write experiments, latencies are introduced on the incoming links. Latencies are introduced in an uniform random fashion in a short window with the average values from 10 ms to 400 ms. After 400 ms latency, the Swift cluster might become unavailable because of request timeouts.

Figure 6.3 and Figure 6.4 demonstrate the influence of network latencies to the read and write performance of a Swift cluster. In both figures, the x-axis presents the average latencies introduced to either the proxy servers or the storage servers network interfaces. The y-axis shows the corresponding performance, quantified as system throughput in op/s. Experiments on different latency configurations last for 10 minutes. Data are collected every 10 seconds from the system performance feedback in YCSB. The plot shows the mean (the bar) and standard deviation (the error line) of the results from each latency configuration.

It is obvious that Figure 6.3 and Figure 6.4 share similar patterns, which indicates that the network latencies on either the storage servers or the proxy servers bound the read and write performance of the cluster. Furthermore, it is shown in both figures that the network latencies on the proxy servers result in further performance degradation. The throughput of Swift becomes more stable (shown as the standard deviation), although decreasing, with the increasing of network latencies. The decreasing of throughput causes less network congestions in the system and results in more stable performance.

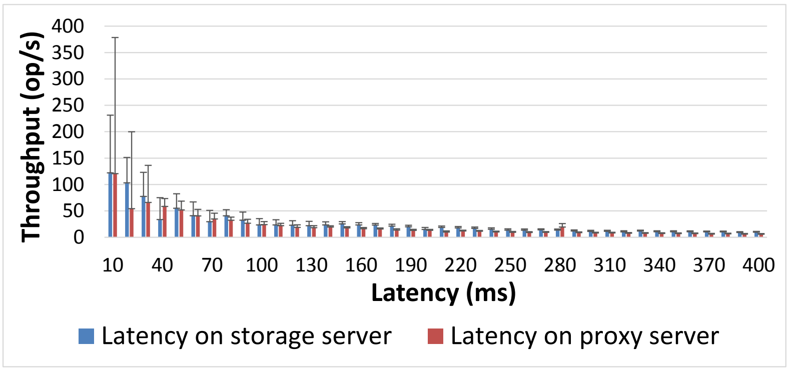

Figure 6.3. Read Performance of Swift under Network Latencies

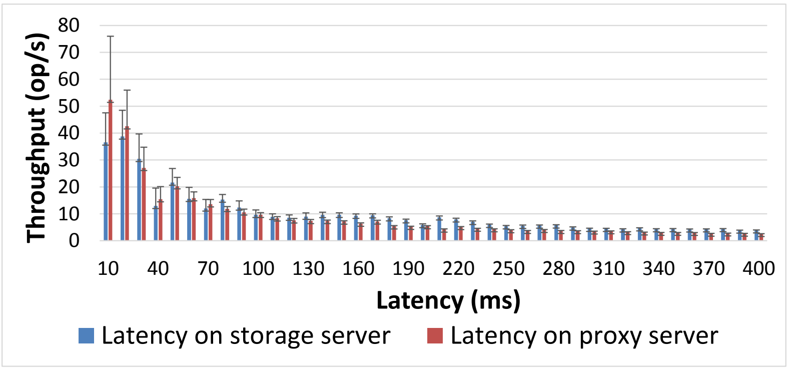

Figure 6.4. Write Performance of Swift under Network Latencies

Discussion

From the experiment data, we could estimate the bounded performance of a Swift cluster by deriving a mathematical correlation model of the system throughput and the network latencies. For example, a logarithmic approximation may best fit the performance boundaries shown in both figures. The correlation model can be used to predict system throughput under a given network latency and further be used to make guarantees for the quality of service. In particular, by knowing the network latencies on every incoming and outgoing links of a server in a community Cloud, we could estimate the bounded performance (throughput) for a particular requested data unit.

In order to estimate the bounded performance for a requested data unit, we need to understand the consistency mechanism implemented in Swift. Every read and write request is sent to several storage servers, depending on the configured replication degree, from the proxy servers. Not all the responses from the storage servers are needed to successfully complete a client request. Specifically, Swift is an eventual consistent system that, for a read request, only one correct response from the storage servers is needed. In contrast, the write operations require stricter scenario where the majority of the storage servers, which store the same replicated data, are needed. By understanding this consistency mechanism

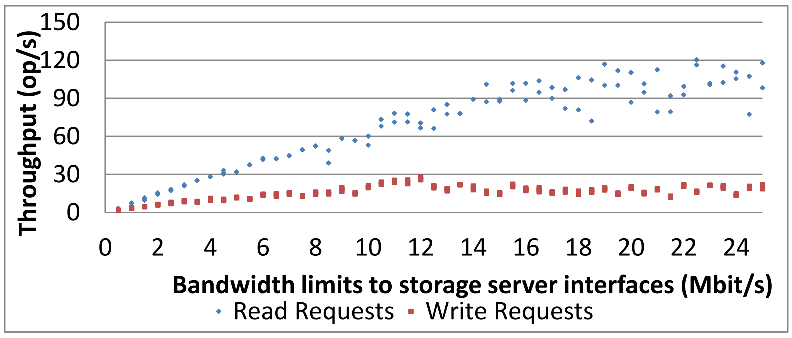

Figure 6.5. Performance of Swift with Bandwidth Limits on Storage Servers

in Swift, the read performance is bounded by the fastest (lowest latency) outgoing links of the storage servers that store the requested data. In comparison, the write performance is bounded by slowest incoming links of the storage servers in the majority quorum, which is formed by the faster majority from all the storage servers that should store the target data. By knowing the network latencies of all the links in a community Cloud and the mappings of the namespace, we could use the results in Figure 6.3 and Figure 6.4 to estimate a bounded performance for a read or write request on a single data unit.

6.6.2 Swift Experiment with Network Bandwidth

In this section, we present the evaluation results regarding the influence of the available network bandwidth to the performance of a Swift cluster. First, we illustrate the read and write performance bounds that correspond to the available bandwidth in the storage servers. Then, we present the monitored results of the network bandwidth consumption in the proxy server and come up with deployment suggestions.

Storage Server Bandwidth Consumption

In this experiment, the available network bandwidth allocated to each storage server is shown in the x-axis in Figure 6.5. The y-axis represents the system throughput that corresponds to the limited network bandwidth allocated to the storage servers. The plot shows system throughput under different bandwidth allocations averaged in ten minutes. Data are collected through three individual repeated experiments.

Discussion

As shown in Figure 6.5, there is a clear linear correlation of the bandwidth consumption and the system throughput for both read and write. Different gains can be calculated for the read and write workloads before reaching the saturation point and entering the plateau. Specifically, with the gradual increase of the available bandwidth allocated to the read and write workloads, the write workloads reach the saturation point earlier than the read workloads. There are mainly two reasons for this difference. One reason is that usually the disk write speed is slower than the read speed. Another reason is that a write request in a

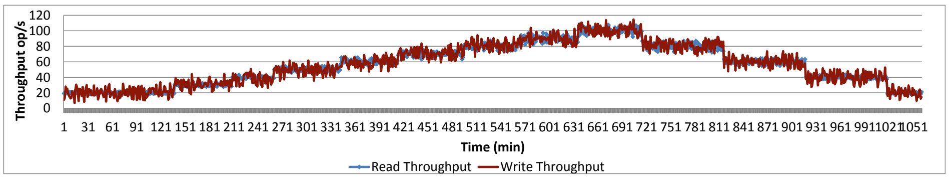

Figure 6.6. Workload Patterns

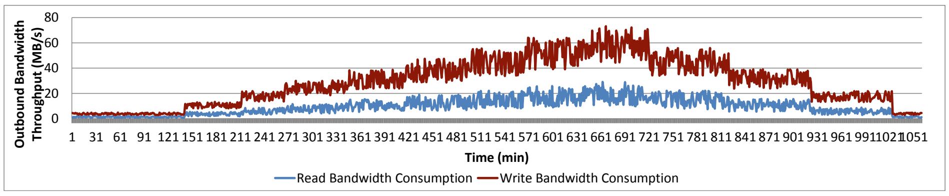

Figure 6.7. Proxy Server Outbound Bandwidth Usage for Reads and Writes

Swift cluster need the majority of the responsible storage servers to perform while a read request only require the fastest respond from one of the responsible storage servers.

In order to achieve the efficient usage of the network bandwidth in community Cloud platform, a system maintainer needs to find out the read and write saturation balance among the available bandwidth and the above mentioned two potential bottlenecks. For example, using our hard disk and the Swift setup described in Section 6.3.2, the saturation point for the write requests corresponds to 11 Mbit/s bandwidth allocation shown in the x-axis in Figure 6.5. Understanding the saturation balance among multiple resources, a system maintainer can deploy the storage nodes of a Swift cluster based on the available outbound and inbound bandwidth that fit the request patterns (read-intensive or write-intensive).

Proxy Server Bandwidth Consumption

When evaluating the correlations between the available bandwidth and the Swift performance, we have monitored the bandwidth usage by all the storage and proxy servers. The bandwidth consumption patterns in the proxy servers are of interest.

Figure 6.6 presents the read and write workloads generated from our customized YCSB. Data are collected every 10 seconds. Since the proxy servers are the only entry points for a Swift cluster, it is intuitively clear that the workload in Figure 6.6 is equal to the inbound network traffic imposed on the proxy servers when using the setup of only one workload generator and one proxy server. Figure 6.7 shows the outbound bandwidth consumption on the proxy server for the read and the write requests. The bandwidth consumed by the reads and the writes shows correlations. Specifically, the outbound bandwidth consumed by write requests is X times more than the bandwidth consumed by the read requests, where X corresponds to the replication degree. In our setup, the replication degree is configured to three, thus writes consume three times more outbound bandwidth on the proxy server than the reads.

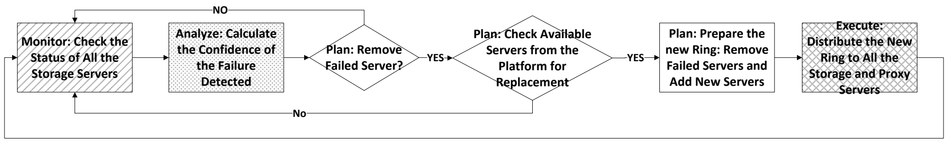
NO
Monitor: Check the Status of All the Storage Servers
Analyze: Calculate the Confidence of the Failure Detected
Plan: Remove Failed Server?
YES
Plan: Check Available Servers from the Platform for Replacement
YES
Plan: Prepare the new Ring: Remove Failed Servers and Add New Servers
Execute: Distribute the New Ring to All the Storage and Proxy Servers
No

Figure 6.8. Control Flow of the Self-healing Algorithm

Discussion

The extra bandwidth consumed by write requests shown in Figure 6.7, is because of the replication technique employed by Swift. Specifically, the replicated data are propagated from the proxy server in a flat fashion. The proxy servers send the X times replicated data to the X responsible storage servers directly, when the data is written to the cluster. Based on this knowledge, a system administrator can make decisions on the placement of proxy servers and differentiate the read and write workloads considering network bandwidth consumption. Based on the above analysis, it is desired to allocate X times (the replication degree) more outbound network bandwidth to the proxy servers under write-intensive workload because of the data propagation schema employed by Swift.

6.7 The Self-healing of Swift

Since a community Cloud is built upon a community network, which is less stable than a data center environment, systems operating in a community Cloud environment have to tackle with more frequent node leaves and network failures. Thus, we have developed a self-healing mechanism in Swift to operate in a community Cloud environment.

Originally, the failure recovery in a Swift cluster is handled by a set of replicator processes introduced in Section 6.2.2. Swift itself cannot automatically recover from server failures because of the Ring file, which records the namespace mapping. Specifically, the replicator processes on all the storage servers periodically check and push their local data to the locations recorded in the Ring files, where the same replicated data should reside. Thus, when the storage servers fail, the Ring files should respond to such changes swiftly in order to facilitate the replicators by providing the location of the substitute replication servers, and thus guarantee the availability of the data. However, this healing process is expected to be monitored and handled manually by a system maintainer in the Swift design. In the following paragraphs, we describe an algorithm to automate the healing process of Swift. The self-healing algorithm is validated under different failure rates introduced by our failure simulator.

6.7.1 The Self-healing Algorithm

In this section, we describe a MAPE (Monitor, Analysis, Plan, and Execute) control loop for the self-healing algorithm in Swift for the Ring files shown in Figure 6.8.

The control cycle illustrated in Figure 6.8 is implemented on a control server, which can access the local network of the Swift cluster. In the monitor phase, it periodically sends

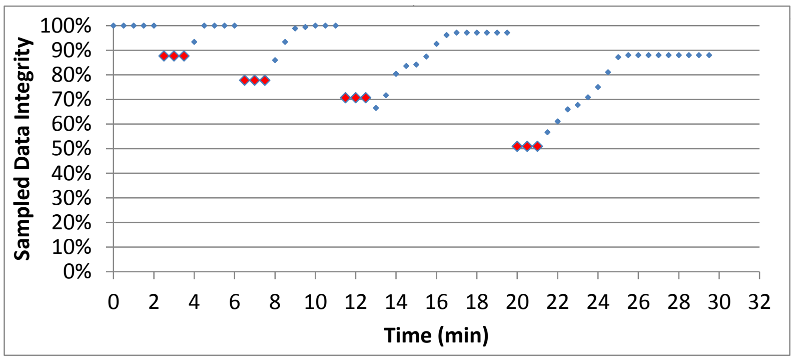

Figure 6.9. Self-healing Validation under the Control System

heartbeat messages to all the storage servers registered in the Swift cluster. We assume that the storage servers follow the fail-stop model. If there are no replies from some storage servers, these servers are added to the Failure Suspicious List (FSL) maintained in the control server. The FSL records the number of rounds that a server is suspected failed. Then, in the analysis phase, the control server compares the current round FSL with the previous FSL. Servers in the previous FSL are removed if they do not exist in the current FSL. Servers in both previous FSL and current FSL will be merged by summing up the number of suspicious rounds in previous FSL and current FSL. A configurable threshold value is introduced to manipulate the confidence level of the failure detect mechanism. Next, in the plan phase, if the number of the suspicious rounds associated with the storage servers in the FSL exceeds the threshold value, the control system marks these servers as failed and plans to remove them from the Ring files. If server removals are needed, the control system checks the available servers from the platform and plans to join them as the substitute servers for the failed ones. If there are available servers in the platform, new Ring files are prepared by removing the failed servers and adding the new ones. Finally in the execution phase, the new Ring files, which record the updated namespace and server participation information, are distributed to all the current storage servers and the proxy servers.

After the self-healing cycle completes, the data in the failed servers will be recovered from the other replicas to the newly added servers by the Swift replicator daemons running in the background. It is intuitively clear that the frequency of server failures, which cause the data loss, should not exceed the recovery speed by the replicators otherwise some data may be lost permanently.

In Figure 6.9, we illustrate the self-healing process with a Swift setup presented in Section 6.3.2. The horizontal axis shows the experiment timeline. The blue points along with the vertical axis present the cluster's health by showing the data integrity information obtained by a random sampling process, where the sample size is 1% of the whole namespace. The total number of files stored in our Swift cluster is around 5000. In order to simulate storage server failures, we randomly shut down a number of the storage servers. The decrease of the data integrity observed in Figure 6.9 is caused by shutting down 1 to 4 storage servers. The red points show the control latency introduced by the threshold value, which is two in our case, set for the failure detector of the control system to confirm a server failure. When we fail more than 3 (replication degree) servers, which may contain all the replicas of some data, the final state of the self-healing cannot reach 100% data integrity because of data loss.

The choice of the threshold value for our failure detector provides the flexibility to make trade-offs. Specifically, a larger threshold value may delay the detection of server failures but with higher confidence. On the other hand, a smaller threshold value makes the control system react to server failures faster. However, the commitments of the server failures come with a potential cost. This cost is the rebalance of data on the substitute servers. Thus, with higher failure detection confidence, the data rebalance cost is minimized, but it may slow down the system reaction time. Furthermore, the cost of data rebalance is proportional to the data stored in the failed server. Thus, it is a good strategy to set a larger threshold value when there is a large amount of data stored in the storage servers.

6.8 Conclusions

This paper presents the first evaluation results in a simulated environment on the feasibility of applying OpenStack Swift in a community Cloud environment. It presents a detailed technical evaluation of Swift, regarding its bottom-line hardware requirements, its resistance to network latencies and insufficient bandwidth. Furthermore, in order to tackle with frequent server failures in the community Cloud, a self-healing control system is implemented and validated. Our evaluation results have established the relationship between the performance of a Swift cluster and the major environment factors in a community Cloud, including the proxy hardware and the network features among the servers. Whether it is feasible to deploy a Swift cluster in the community Cloud environment depends on the further research on these environment factor statistics. There also exist many unknown community Cloud environment factors to be discovered in the real deployments and evaluations. Thus, the evaluation of OpenStack Swift as a real community Cloud deployment is our future work.

Acknowledgment

This work was supported in part by the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC) funded by the Education, Audiovisual and Culture Executive Agency (EACEA) of the European Commission under the FPA 2012-0030, and in part by the FP7 project CLOMMUNITY funded by the European Commission under EU FP7 Grant Agreement number 317879. The authors gratefully acknowledge Vamis Xhagjika for his collaboration in development and configuration of the experiment testbed.

Bibliography

[1] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[2] Amazon simple storage servie. `http://aws.amazon.com/s3/`.

[3] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[4] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

[5] Ken Pepple. *Deploying OpenStack*. O'Reilly Media, 2011.

[6] Gerard Briscoe and Alexandros Marinos. Digital ecosystems in the clouds: Towards community cloud computing. *CoRR*, abs/0903.0694, 2009.

[7] Alexandros Marinos and Gerard Briscoe. Community cloud computing. In *Proceedings of the 1st International Conference on Cloud Computing*, CloudCom '09, pages 472–484, Berlin, Heidelberg, 2009. Springer-Verlag.

[8] Bart Braem, Chris Blondia, Christoph Barz, Henning Rogge, Felix Freitag, Leandro Navarro, Joseph Bonicioli, Stavros Papathanasiou, Pau Escrich, Roger Baig Viñas, Aaron L. Kaplan, Axel Neumann, Ivan Vilata i Balaguer, Blaine Tatum, and Malcolm Matson. A case for research with and on community networks. *SIGCOMM Comput. Commun. Rev.*, 43(3):68–73, July 2013.

[9] Guifi.net. `http://guifi.net/`.

[10] Awmn the athens wireless metro-politan network. `http://www.awmn.net/`.

[11] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[12] Yi Wang, Wei Jiang, and Gagan Agrawal. Scimate: A novel mapreduce-like framework for multiple scientific data formats. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, CCGRID '12, pages 443–450, Washington, DC, USA, 2012. IEEE Computer Society.

[13] Zmanda: The leader in cloud backup and open source backup. `http://www.zmanda.com/blogs/?cat=22`.

[14] Salman Toor, Rainer Toebbicke, Maitane Zotes Resines, and Sverker Holmgren. Investigating an open source cloud storage infrastructure for cern-specific data analysis. In *Proceedings of the 2012 IEEE Seventh International Conference on Networking, Architecture, and Storage*, NAS '12, pages 84–88, Washington, DC, USA, 2012. IEEE Computer Society.

[15] Community networks testbed for the future internet. `http://confine-project.eu/`.

[16] A community networking cloud in a box. `http://clommunity-project.eu/`.

[17] Bryggenet community network. `http://bryggenet.dk/`.

[18] Openstack cloud software. `http://www.openstack.org/`.

[19] Javier Diaz, Gregor von Laszewski, Fugang Wang, Andrew J. Younge, and Geoffrey Fox. Futuregrid image repository: A generic catalog and storage system for heterogeneous virtual machine images. In *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*, CLOUDCOM '11, pages 560–564, Washington, DC, USA, 2011. IEEE Computer Society.

[20] C Chodorow. Introduction to mongodb. In *Free and Open Source Software Developers' European Meeting*, 2010.

[21] Nimbus project. `http://www.nimbusproject.org/`.

[22] Physics data analysis software. `http://root.cern.ch/drupal/`.

[23] K Pepple. Openstack nova architecture. *Viitattu*, 2011.

[24] Openstack swift's documentation. `http://docs.openstack.org/developer/swift/`.

[25] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[26] Scaling media storage at wikimedia with swift. `http://blog.wikimedia.org/2012/02/09/scaling-media-storage-at-wikimedia-with-swift/`.

[27] Stephen Hemminger et al. Network emulation with netem. In *Linux Conf Au*, pages 18–23. Citeseer, 2005.

Chapter 7

Replication in Distributed Storage Systems: State of the Art, Possible Directions, and Open Issues

Ying Liu and Vladimir Vlassov

In *5th IEEE International Conference on Cyber-enabled distributed computing and knowledge discovery* (CyberC), 2013

Abstract

Large-scale distributed storage systems have gained increasing popularity for providing highly available and scalable services. Most of these systems have the advantages of high performance, tolerant to failures, and elasticity. These desired properties are achieved mainly by means of the proper adaptation of replication techniques. We discuss the state-of-art in replication techniques for distributed storage systems. We present and compare four representative systems in this realm. We define a design space for replication techniques, identify current limitations, challenges and open future trends.

7.1 Introduction

Web applications and services, including Cloud-based ones, use storage systems that can scale horizontally and provide different levels of data consistency and availability guarantees. Amazon Dynamo [1], Cassandra [2], Voldemort [3], Google Spanner [4], Yahoo! PNUTS [5], OpenStack Swift [6], Windows Azure Storage [7] are few examples of storage systems. Usually, the design and implementation choices of system developers are driven by requirements of applications and application domains, in which a storage system is used or to be used.

A replication technique is usually designed and implemented in distributed storage systems in order to guarantee data availability in the presence of server failures. Specifically, it replicates several copies of the same data in different servers, racks, or data centers. These data copies are called replicas. Thus, in the case of server failures, data can be recovered from other servers having replicas of the data.

Furthermore, availability is not the only concern of using replication. Since there are multiple copies of the same data in the system, system designers observe the potential usage of replication to improve system performance and scalability. Specifically, multiple replica servers are expected to serve clients concurrently instead of just being the data backup. However, when further investigate this approach, new challenge appears: replica consistency.

In general, replica consistency requires that all replicas of the same data should perform synchronously like one single copy. Situation gets complicated when system is distributed and client workload is simultaneous, which means different clients can query different replicas of the same data at the same time. There already exist many consistency models from the previous experience of building distributed storage systems [8, 9].

In this paper, we discuss the following three most essential, in our view, issues related to replication, to be considered when developing a replicated distributed storage service: replication for availability, replication for performance, and replica consistency.

To examine these issues, we consider, as case studies, replication and consistency in four production distributed storage systems, namely: Dynamo of Amazon [1], Cassandra of Facebook [2], Spanner of Google [4], and Yahoo! PNUTS [5]. In our view, these four systems and others not considered here (e.g., Voldemort, Swift, and Windows Azure Storage), define state of the art in the field of large-scale distributed storage systems, and furthermore, each of the systems has its own unique properties that distinguish it from the other systems. Our discussion focuses on the replication rationales, usage scenario, and their corresponding consistency models. To be concise, we have investigated flexibility of replica placement and migration mechanisms, management of replication degree, and tradeoff of consistency guarantees for performance. Finally, we identify open issues and suggest future research directions regarding replication techniques for large scale distributed storage systems.

The contributions of the paper are:

- An overview of the basics of replication in large scale distributed storage systems;
- A comparison, analysis and classification of replication techniques in four production state-of-the-art storage systems considered as case studies;
- A synopsis of the current state of the research area, identifying trends and open issues;
- A vision on possible future directions.

The rest of the paper is structured as follows. Section 7.2 outlines motivations and basic approaches to data replication. Section 7.3 summarizes the main-trend consistency models. Section 7.4 presents a survey of four existing production systems. In Section 7.5, we analyze replication approaches and discuss the decision rationales of the surveyed systems. Section 7.5 also presents some possible future directions and open issues. We conclude in Section 7.6.

7.2 Replication in Distributed Storage

In distributed storage, replication techniques are used for two main purposes: first, to guarantee high availability of data, and second, to improve system scalability.

7.2.1 Replication for Availability

A replicated system is designed to provide services with high availability. Multiple copies of the same data are maintained in the system in order to survive server failures. Through well-designed replication protocol, data lost because of server failures can be recovered through redundant copies. However, high availability does not come for free. An essential side-effect of adapting replication techniques is the challenge of maintaining consistency among replicas. In general, a well-designed replication mechanism is able to provide the illusion of accessing a single copy of data to clients even though there are multiple replicas in the system. This property is called linearizability or strong consistency. It is usually managed under the concept of replication group.

A *replication group* is a group of servers that store copies of the same data. The responsibility of a replication group is to hide the replication degree and the replica consistency from clients. A replication group uses a distributed replication protocol to maintain the

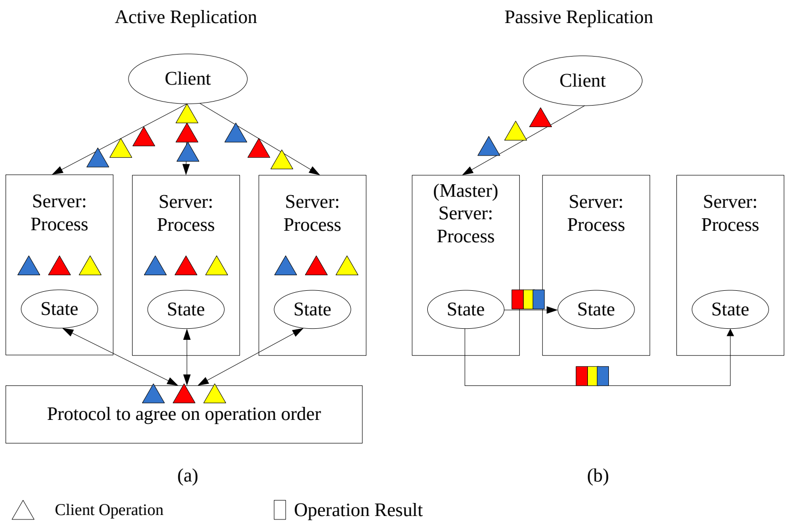

Figure 7.1. Active and Passive Replication

consistent view of memory to clients. One main task of the protocol is to keep replicas as synchronized as possible. This is because the consistent view of memory is easier to provide when all replicas are synchronized to have the same content. On the other hand, when replicas diverge, the maintenance of the consistent view requires more efforts. In this case, the consistency level provided by the system usually depends on the consistency requirements defined by clients. In some applications, some memory inconsistency can be tolerated. We discuss consistency challenges in Section 7.3.

Keeping replicas synchronized becomes a primary challenge. In general, either synchronized or diverged situation is fundamentally caused by the orders of messages received by different replicas [10]. The ordering of messages usually depends on the group communication protocol. An example of the mostly used group communication protocol is Atomic Broadcast [11]. It ensures all the replicas reliably receive broadcast messages in the same order. In this way, the views and behaviors of the replicas are synchronized.

7.2.2 Active and Passive Replication Techniques

Generally, there are two replication techniques: *active replication* and *passive replication*. In both techniques, the concept of replicated state machine [12] is introduced for the management of replica consistency. Essentially, it represents either a synchronized or diverged state of each replica.

In active replication (Fig. 7.1a), client operation requests are forwarded to all replicas in a coordinated order. In Fig. 7.1a, the order of operations is shown as the order of triangles of different colors representing operation requests. With an ordering protocol, such as the Atomic Broadcast, all replicas agree on the same order of execution of operations, and then, each replica executes the operations individually and finally reaches a result state. To guarantee that the result states of replicas are identical, client operations are required to be deterministic in order to maintain the following property: with the same replica initial states and the same sequence of operations, the output states of each replica are identical.

In passive replication (Fig. 7.1b), one of the replicas in each replication group is selected to be a master replica. Clients send operations to the master replica, which executes the operations, and sends output states to all other replicas in the same order as the order of operations executed by master, as illustrated with colored rectangles in Fig. 7.1b. This guarantees that the states of all replicas are identical.

Each of these two techniques, active replication and passive replication, retains its own advantages and drawbacks. In particular, passive replication is based on sending output states from the master to other replicas; whereas active replication is based on sending operations to all replicas. Thus, bandwidth consumption is expected to be higher in passive replication when the size of the output state is large. Furthermore, since roles of replicas in passive replication are not identical, some overhead is required to perform master replica's recovery and election.

However, passive replication outperforms active replication by saving computing resources since output state can be calculated only once for a number of concurrent or consecutive operations on the master replica. One possible optimization to save bandwidth is to transfer only an update difference rather than the entire state. Moreover, passive replication allows non-deterministic operations.

Practical solutions are hybrid based on combination of passive and active replication. Section 7.4 gives examples.

7.2.3 Replication for Scalability

Replication is not only used to achieve high availability, but also to make a system more scalable, i.e., to improve ability of the system to meet increasing performance demands in order to provide acceptable level of response time.

Imagine a situation, when a system operates under so high workload that goes beyond the system's capability to handle it. In such situation, either system performance degrades significantly or the system becomes unavailable. There are two general solutions for such scenario: *vertical scaling*, i.e., scaling up, and *horizontal scaling*, i.e., scaling out.

For vertical scaling, data served on a single server are partitioned and distributed among multiple servers, each responsible for a part of data. In this way, the system is capable to handle larger workloads. However, this solution requires much knowledge on service logic, based on which, data partitioning and distribution need to be performed in order to achieve scalability. Consequently, when scaled up, the system might become more complex to manage. Nevertheless, since only one copy of data is scattered among servers, data availability and robustness are not guaranteed.

One the other hand, horizontal scaling replicates data from one server on multiple servers. For simplicity without losing generality, assume all replication servers are identical, and client requests are distributed evenly among these servers. By adding servers and replicating data, system is capable to scale horizontally and handle more requests. Replication also makes system more robust and improves its availability. However, horizontal scaling has the same issue as replication for availability: both require maintaining replica consistency. In general, maintaining strong consistency is achievable but expensive. Thus, many hybrid approaches make acceptable tradeoff between data consistency and system performance. The consistency issue is further discussed in Section 7.3.

Even though scalability or elasticity is an attractive property that allows system to grow and shrink to handle dynamic workloads, it may lead to over-provisioning of resources. The difficulties exist in the proper identification of system workloads, efficient resource allocation and swift replica or data rebalance. Thus, well-managed system elasticity is desired which allows to improve resource utilization and achieve required (acceptable) quality of service at a minimal cost (amount of resources used). Elasticity can be managed manually by a human system administrator or automatically by an elasticity controller [13, 14, 15, 16].

7.2.4 Replica Placement

One of the issues to be considered when developing an efficient and effective replication for both scalability and availability is replica placement.

It is intuitively clear that benefits of replication for availability are sensitive to replica placement. The general solution for availability is replication, but when considering replica placement, the developer should try to avoid introducing single points of failures by placing replicas on the same hardware even if it is reliable.

As mentioned above, replication allows making the service horizontally scalable, i.e., improving service throughput. Apart from this, in some cases, replication allows also to improve access latency and hence the service quality. When concerning about service latency, it is always desirable to place service replicas in close proximity to its clients. This allows reducing the request round-trip time and hence the latency experienced by clients. In general, replica placement depends on two decisions: geographical placement of replication servers and distribution of replicas.

Ideally, a replicated system should place its replication servers geographically close to most of its clients. Server placement is affected by a number of factors, such as geographical distribution of (potential) clients, electricity cost, and network connectivity. Furthermore, most of these factors are dynamic. This makes the decision of optimal replica placement challenging. The discussion of this decision making is out of the scope of this paper.

Despite replication server placement, algorithms to distribute replicas on replication servers also greatly affect service quality. One can distinguish two kinds of data distribution techniques: *server-initiated* and *client-initiated*.

With server-initiated replication, the system takes the responsibility to decide whether to migrate or replicate data to particular servers. Replica distribution or placement can be made automatically with the help of machine-learning algorithms. Machine learning allows system to dynamically migrate or replicate frequently accessed data according to learned or predicted client access patterns. One typical example is content delivery networks.

Client-initiated replication is usually implemented as client-side caching. In contrast to server-initiated replication, client-side replication is not constrained by system replication rules. A client application can cache data wherever it benefits from caching. Furthermore, client-side caching can be implemented rather easily and it is proved to be relatively efficient for frequently read data. For example, this technique is applied maturely in most of current Internet browsers.

7.3 Consistency Challenges

Previous sections have shown that data replication does provide many advantages for distributed systems, such as high availability, horizontal scalability, and geo-optimized performance. However, data replication also brings new challenges for system designers including the challenge of data consistency that requires the system to tackle with the possible divergence of replicated data. Developers are heading two different directions to address this challenge.

One approach is to prevent replica divergence by implementing strict consistency models, such as atomic consistency [17], and sequential consistency [18]. However, the strict consistency model is proved to be very expensive and hard to implement in distributed systems. It is well known that when dealing with the possibility of network partitioning, strong consistency and high data availability cannot be achieved simultaneously [19]. Achieving high availability and strong memory consistency becomes more challenging when the system size grows in terms of the number of replicas and the coverage of geographical locations. Thus, many existing production systems turn to the other consistency models weaker than the strong consistency model in terms of consistency guarantees.

When maintaining strong consistency becomes impractical, developers try to hide data inconsistency in their systems and applications or to support multiple consistency levels to tradeoff consistency for performance. Depending on usage of storage services, many applications can tolerate obsolete data to some extent. For example, in the context of a social network, most of users might not mind reading slightly older posts within some time bounds. This scenario leads us to think about eventual consistency [20]. For example, this model is provided and successfully adapted in Cassandra [2] by Facebook. However, there are also some scenarios that require stronger consistency guarantees. Continuing with the example of a social network, assume the user wants to set limited access rights to a newly created album and then upload some photos. These two operations, set access rights and upload photos cannot tolerate bare eventual consistency that does not guaranty that the execution order of these two operations is preserved. Under the eventual consistency model, an unauthorized user might be able to access uploaded photos before access rights got consistent in all replicas. Thus, stricter consistency models are desired for this scenario. For example, causal consistency [18] model can be applied in this case, since it guarantees the order of operations with causal relations. Under the causal consistency model, the operation of setting access rights is guaranteed to be executed prior the upload operation takes place. Understanding consistency uncertainties, many applications are designed to tolerate inconsistency to some extent without sacrificing the user experience [20, 21].

Table 7.1 presents some of consistency models that, in our view, define state-of-the-art in memory consistency.

Table 7.1. Consistency Models

Consistency Models	Consistency Guarantees
Atomic Consistency	Atomic consistency is the strictest consistency model. It requires operations to be executed in the same order by all replicas. This order respects the real time that operations are issued. As a result, the execution of a series of commands in a replicated system with atomic consistency should be equivalent to a sequential execution scenario.
Sequential Consistency	Sequential consistency is weaker than atomic consistency. It does not demand that all operations are ordered respect to the actual issuing time. However, the orders of operations are strictly synchronized among replicas.
Causal Consistency	Comparing to sequential consistency, not all operations are strictly synchronized in the causal consistency model. It only requires that operations with a potential causal relation should be ordered to all replicas.
PRAM consistency (FIFO Consistency)	PRAM consistency [22] is short for Pipelined Random Access Memory consistency. It has fewer requirements than causal consistency in a sense that only write operations on the same replica are ordered in the view of other replicas, i.e., replicas can disagree on the order of write operations issued by different replicas.
Eventual Consistency	Eventual consistency is a general concept, which states that after a sufficient long time, updates will eventually propagate to the whole system and then replicas will be synchronized. A system with eventual consistency may have inconsistent replica states. Thus, client requests are not guaranteed to retrieve the most updated result from the system. Furthermore, there is no guarantee on the time when replica states will be consistent.
Delta Consistency	Delta consistency is stronger than eventual consistency. The delta value specifies the maximum period that replicas will finally converge to a consistent state.

7.4 Existing Systems

In this section, we give a survey of replication techniques implemented in four of the most popular distributed storage systems, namely Amazon's Dynamo, Facebook's Cassandra, Google's Spanner, and Yahoo! PNUTS. In our view, these four systems define the state of the art in the field of scalable distributed storage systems specially optimized for storing large amount of structured and unstructured data with associated metadata, used in Web-2 services and applications. They are characterized in three groups: key-value storage, column-based storage, and geographical database.

7.4.1 Key-value Store: Dynamo

Dynamo is one of the most influential highly available distributed key-value stores designed by Amazon. There are many open-source implementations inspired by Dynamo, such as Voldemort [3] used by LinkedIn, Dynomite and KAI.

Dynamo applies consistent hashing [23] to partition and replicate data. Servers are arranged in a ring like structure where each server is responsible for a specific hashed key range. Servers are called primary servers for their corresponding ranges. High availability is achieved by replicating data successively on several servers clockwise next to the primary server. This list of servers, who store the same particular key ranges, are called preference list. The size of preference list is configurable for different instances. Servers in the list are coordinated to achieve certain level of consistency and failure recovery.

The primary goal of Dynamo is to provide customers of 'always on' experience. Thus, the design decision taken by Dynamo is to compromise consistency while leverage availability. Specifically, there is no total order of operations to all the replicas. Dynamo allows replicas to diverge under certain coordination. This coordination is version technique in Dynamo, where all operations are marked with logical vector clock [10, 24, 25] as time stamps. The system allows taking write requests simultaneously on different replication servers to achieve high performance and availability. The conciliations of version divergences are triggered by read requests. Compatible version divergences are conciliated by system automatically. However, application-driven version conciliation is expected to tackle complex version conflicts.

To sum up, Dynamo trades off consistency for performance and availability by accepting operations on any replica servers. Since Dynamo is designed to be deployed as a background high availability storage service within Amazon, the conciliation of complex data version conflicts are left for the upper application layers.

7.4.2 Column-based Storage: Cassandra

Cassandra is a distributed storage system developed by Facebook, whose structure is similar to Dynamo. In addition, it provides BigTable-like [26] query interface. Specifically, Cassandra also adapts consistent hashing to partition data. Servers are primary responsible for one key range and replicate for several other ranges. Like in Dynamo, the replicated ranges are consecutive spread on the ring. In addition, Cassandra allows client to specify some policies for their replica placement to keep data more robust, such as "Rack Aware" (within a datacenter) or "Datacenter Aware" (in different datacenters). Similar to Dynamo, the concept of preference list is also implemented in Cassandra. The degree of replication is also configurable for different instances.

Cassandra provides client APIs to satisfy different consistency requirements. Write is successful when it is acknowledged by the majority of servers in the preference list. There are two consistency scenarios for reads. First, the client can achieve consistent read, which is usually expensive (slow), by reading from the majority of replicas in the preference list. Second, the client can perform read from the nearest replica, that is inexpensive (fast) but might be inconsistent.

In sum, Cassandra's replication technique is quite similar to Dynamo. Instead of leaving consistency concerns to application level conciliation, Cassandra implements quorum based majority reads and writes interfaces to achieve high consistency. In the case of consistent read, both systems need to contact all the replica servers at similarly high cost. Furthermore, because of quorum writes, theoretically Cassandra's write latency will be worse than Dynamo.

7.4.3 Geographical Database: Spanner

Spanner is the latest Google's globally-distributed database. It is aimed at a database system that is able to operate across different data centers at a global scale. The concept of zones is introduced to isolate the data in Spanner. Specifically, one datacenter is able to hold one or several zones while replicas are placed across different zones. Data is partitioned and grouped by prefix, which also decides data locality. Replication policy is configured in individual groups, which includes replication type, degree, and geographic placement.

Regarding consistency, the implementation of the 'true time' API allows Spanner to achieve atomic consistent writes in a global scale while exposing non-blocking consistent reads. This 'true time' API is implemented through GPS and atomic clock. It is able to bound the real time uncertainties among all the replication servers. Specifically, every operation in Spanner is marked with a timestamp provided by this API. Write operations are coordinated through Paxos [27] groups and commit wait after the pass of time uncertainty indicated by the 'true time' API. Thus, consistent read operations can be performed without any locks or quorums efficiently with the help of a current timestamp, which guarantees that stale data is not provided.

To conclude, Spanner achieves atomic consistent read and write operations on a global scale with an acceptable expense. Spanner will not outperform other similar systems with the implementation of writes using Paxos group. However, consistent read operations are expected to be more efficient, which are able to be executed on the nearest replica with atomic consistency guarantee by the global timestamp.

7.4.4 Geographical Database: Yahoo! PNUTS

PNUTS is Yahoo!'s globally distributed hosting service with high availability and performance guarantees. In this section, we focus on the replication placement, replication algorithm, and consistency concerns of PNUTS.

PNUTS replicates data in different regions, which may reside in the same or different data centers. PNUTS organizes data in tables. Each row in the table is called a record. Each record is associated with a hashed key, and grouped in tablets, which stored in the actual servers. Data partition is managed by a utility called router, which keeps all the mapping of key ranges to physical servers in its memory. Movement of data is managed by tablet controller.

PNUTS implements a novel per-record timeline consistency model. This consistency model guarantees that all the replicas of the same record apply updates in the same order. It is a little weaker than linearizability, which requires the replicas of all the records in the same transaction to be updated in the same order. The per-record consistency is implemented by assigning one master per-record, which coordinated the updates. According to PNUTS, the mastership of one record will be adjusted to the nearest replica which observes highest workloads. All the consistent read and write operations are serialized to the master replica of the target record. The master replica applies the updates and publishes it into Yahoo! Message Broker before commits to client. Yahoo! Message Broker is a Publish/-Subscribe system, which guarantees the updates accepted from the same source (master replica) will be delivered in the same order to all other replicas that subscribed (responsible) for the same record. The advantage of using a Pub/Sub system is to hide the latency of global communication with replicas.

Despite per-record consistent read and write performed by master replica, PNUTS also provides the interface to read a little stale record to leverage the performance in some cases. The fast read is operated by reading any replica that is close to the client. Another format of read API is implemented to require a record that is newer than the requested version.

In sum, PNUTS applies master-based active replication. Yahoo! Message Broker is a novel approach that can be regarded as an asynchronous broadcast protocol, which hides the master-slave communication latency from clients.

7.5 Discussion

In this section, we summarize the main replication features of the four systems surveyed above. Table 7.2 presents two main features: replication and consistency. We lead our discussion further by focusing on system-wise decision rationales and possible future research directions.

In Table 7.2, columns 2 and 3 show placement of replicas and dynamicity of replica migration. Dynamo and Cassandra place replicas on successors of consistent hashing overlay and this placement is static to some extent. Even though there are some rules, such as different physical servers or zones, that can be specified to omit some successive server placement, replica placement decision is still not fully flexible. In contrast, Spanner and PNUTS have not adopted consistent hashing. They adapt an additional controller component, called zone master in Spanner and router in PNUTS, to manage namespace and routing. Thus, in Spanner and PNUTS, placement of replicas is flexible by changing the mappings in controller components. The advantage of replica placement flexibility is that a

Table 7.2. Comparative Table of Replication Features in the Surveyed Systems

Systems	Replication Server Placement	Replica Migration	Replication Degree	Consistency Level	Consistency Mechanism
Dynamo	Successor Replication	Static	Configurable per instance	Version conciliation consistency	Vector clock versioning with version conciliation mechanism
Cassandra	Successor Replication	Static	Configurable per instance	Sequential and Eventual Consistency	Majority based quorum group
Spanner	Replicated in different zones	Configurable by choosing prefix	Configurable per directory	Atomic Consistency	Paxos group and global timestamp
PNUTS	Replicated in different regions	Configurable by tablet controller	Configurable per record	Per-record consistency and weak consistency	Master based replicated state machine

system can be dynamically (re)configured to satisfy different request patterns. Specifically, the system can dynamically place replicas close to a group of clients, thus achieving better performance. We observe great opportunity to use this dynamic configuration in a runtime self-configuration algorithm, which will automate and facilitate the system management.

Obviously, column 4 in Table 7.2 leads us to notice that each of the four systems allows changing the replication degree. There are basically two meanings of this flexibility. One rationale is to achieve high availability of critical (popular) items by replicating them more times. Another reason is to make system services elastic. Specifically, larger replication degree can be assigned to popular items to spread workloads that do not need a quorum to perform (in order to maintain consistency), such as inconsistent read operation. This increase of the replication degree will directly improve system throughput in many cases. However, one may notice that the increase of replication degree will also cause more overhead to quorum-based operations, because there will be more replicas in the quorum. This is an already mentioned tradeoff of performance for consistency.

Columns 5 and 6 in Table 7.2 outline consistency models in the four studied systems. Notice that most of these consistency models, except the one supported in Spanner, implement both strong and weak consistency. Strong consistency is usually expensive (high latency and overhead) to maintain: version conciliation in Dynamo, replica quorum operations in Cassandra, cross continent Paxos vote of all replicas in Spanner, and global master-slave communication in PNUTS. However, weak consistency is usually cheaper to maintain. It results in higher throughput of the same system as well as shorter client latency. Specifically, weak consistency operations are performed only on one of the closest and fastest replicas of the replication group.

7.5.1 Some Possible Future Directions and Open Issues

Based on the above survey of studied systems, we propose our vision on some possible future directions. One of the promising directions, in our, view, is support for efficient *fine-grained (individual) replica configuration management*, where each replica has its own configurations regarding placement, migration, leadership, and replication degree. We believe that the diverse fine-grained replica configuration management will enable storage systems to support more complex usage scenarios. This property is partially achieved by Spanner and PNUTS with specific middleware entity (zone master for Spanner and router for PNUTS) to manage each replica. However, Dynamo and Cassandra are bounded by structured overlay of DHT, which makes replica migration difficult. Furthermore, flexible data consistency management is also necessary in order to provide the ability to change the consistency level at runtime. This requires fine-grained control within a replication group, which can be achieved by providing a component similar to the replication master in PNUTS or the Paxos leader in Spanner.

In our view, *automatic switching of consistency guarantees (consistency levels)* for performance purposes is an interesting and promising research direction. For example, when a system is saturated under read dominant workload, the system can increase the replication degree of popular items and relax the consistency level in order to handle high workload. Some applications, such as social network, online shopping, and news hosting sites, can tolerate data inconsistency and, thus, tradeoff consistency level for performance under high workload. In some cases, consistency is over-provisioned, effective ways to dynamically tradeoff consistency level for performance in different scenarios deserve further investigations.

Even though distributed storage systems nowadays are able to provide services with relatively high availability (low latency), scalability and fault-tolerant, there still remain some open issues in replication techniques.

In our view, one of the open issues is to *efficiently handle skewed workload*. We believe that today's replication techniques enable storage systems to function well under relatively uniform workloads. Skewed workloads may cause poor system performance. For example, according to major media (CNN), Twitter and Wikipedia have experienced significant downtime because of highly skewed workloads [28].

In our view, *data consistency algorithms with low communication complexity* are one of the new trends in this realm. Most of the existing strong consistency algorithms, such as two-phase commit [29], anti-entropy protocols [30], and Paxos [27], are based on extensive network communication among replicas to reach consensus. These algorithms work well when system deployment is relatively small geographical scale. When considering geographic storage systems, which is a trend for large-scale distributed systems [4, 5], most of the existing consistency algorithms have relatively high cost to operate. Thus, novel consistency algorithms that can satisfy specific usage scenarios at low costs are highly desired. For example, a consistency protocol that optimizes cross-datacenter communication cost is presented in [31].

Additionally, since storage systems are spanning across broader geographical areas, *automated runtime migration of replicated data close to requests* for better user experience (low latency) might be also identified as a future direction. For example, PNUTS supports a master election algorithm for the replication group to achieve better overall performance based on the analysis of request frequencies in different geographical areas. Another related direction from the area of big-data analytics, that have attracted the high interest, is efficient migration of computations close to data.

We believe that machine learning techniques (which can be used to learn and predict workload patterns and user behaviors) combined with elements of control theory (which can be used to build autonomic managers for different management objectives in order to automate runtime reconfiguration and adaptation to changes in workload and execution environments) should be more intensively employed in storage systems and used by storage system developers. We believe that the combination of machine learning with automated controllers (autonomic managers) will make future large scale distributed storage system self-managing, truly elastic and robust.

Finally, we believe that the applications drive design choices of their developers, therefore, the possible future research directions outlined above should find their ways to and should be driven by real existing and novel applications.

7.6 Conclusion

Replication in large-scale distributed storage systems is a promising and essential research area. Systems that are capable of maintaining data with high availability and scalability provide solid grounds for the success of both research and business. In this paper we have discussed the state of the art in replication techniques in large-scale distributed storage systems. We have presented and compared four representative storage systems. We have identified some of the current open issues, possible future trends and directions, including support for fine-grained replica management, automated tradeoff of data consistency for performance under high workloads, automation of elasticity, consistency algorithms with low communication overhead, geographical replica placement, the use of control theory and machine learning techniques for self-management of storage systems, in particular, automated replication management.

Acknowledgement

This work was supported by the Erasmus Mundus Joint Doctorate in Distributed Computing funded by the EACEA of the European Commission under the FPA 2012-0030, by the End-to-End Clouds project funded by the Swedish Foundation for Strategic Research under the contract RIT10-0043, and by the Complex System Engineering project in the ICT-TNG Strategic Research Areas initiative at KTH. The authors would like to thank the anonymous reviewers for their valuable comments.

Bibliography

[1] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, volume 7, pages 205–220, 2007.

[2] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[3] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving large-scale batch computed data with project voldemort. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, pages 18–18. USENIX Association, 2012.

[4] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally-distributed database. In *Proceedings of OSDI*, volume 1, 2012.

[5] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008.

[6] Openstack cloud software, June 2013.

[7] Windows azure: Microsoft's cloud platform, June 2013.

[8] Fred B Schneider. Replication management using the state-machine approach, distributed systems. 1993.

[9] Bernadette Charron-Bost, Fernando Pedone, and Andre Schiper. *Replication: theory and Practice*, volume 5959. springer, 2010.

[10] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[11] Flaviu Cristian, Houtan Aghili, Raymond Strong, and Danny Dolev. *Atomic broadcast: From simple message diffusion to Byzantine agreement*. Citeseer, 1986.

[12] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[13] Bruno Abrahao, Virgilio Almeida, Jussara Almeida, Alex Zhang, Dirk Beyer, and Fereydoon Safai. Self-adaptive sla-driven capacity management for internet services. In *Network Operations and Management Symposium, 2006. NOMS 2006. 10th IEEE/IFIP*, pages 557–568. IEEE, 2006.

[14] Beth Trushkowsky, Peter Bodík, Armando Fox, Michael J Franklin, Michael I Jordan, and David A Patterson. The scads director: Scaling a distributed storage system under stringent performance requirements. In *FAST*, pages 163–176, 2011.

[15] M Amir Moulavi, Ahmad Al-Shishtawy, and Vladimir Vlassov. State-space feedback control for elastic distributed storage in a cloud environment. In *ICAS 2012, The Eighth International Conference on Autonomic and Autonomous Systems*, pages 18–27, 2012.

[16] Ahmad Al-Shishtawy and Vladimir Vlassov. Elastman: autonomic elasticity manager for cloud-based key-value stores. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing (HPDC '13)*, pages 115–116, New York, NY, USA, 2013. ACM.

[17] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[18] Michel Raynal and André Schiper. From causal consistency to sequential consistency in shared memory systems. In *Foundations of Software Technology and Theoretical Computer Science*, pages 180–194. Springer, 1995.

[19] International Business Machines Corporation. Research Division, BG Lindsay, PG Selinger, C Galtieri, JN Gray, RA Lorie, TG Price, F Putzolu, and BW Wade. *Notes on distributed databases*. 1979.

[20] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[21] Justin Mazzola Paluska, David Saff, Tom Yeh, and Kathryn Chen. Footloose: A case for physical eventual consistency and selective conflict resolution. In *Mobile Computing Systems and Applications, 2003. Proceedings. Fifth IEEE Workshop on*, pages 170–179. IEEE, 2003.

[22] Jerzy Brzezinski, Cezary Sobaniec, and Dariusz Wawrzyniak. Session guarantees to achieve pram consistency of replicated shared objects. In *Parallel Processing and Applied Mathematics*, pages 1–8. Springer, 2004.

[23] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM Computer Communication Review*, volume 31, pages 149–160. ACM, 2001.

[24] Colin Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, 1991.

[25] Michel Raynal and Mukesh Singhal. Logical time: Capturing causality in distributed systems. *Computer*, 29(2):49–56, 1996.

[26] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[27] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[28] Jackson dies, almost takes internet with him, June 2013.

[29] Butler Lampson and David B Lomet. A new presumed commit optimization for two phase commit. In *VLDB*, pages 630–640, 1993.

[30] Robbert Van Renesse, Dan Dumitriu, Valient Gough, and Chris Thomas. Efficient reconciliation and flow control for anti-entropy protocols. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*. ACM, 2008.

[31] Tim Kraska, Gene Pang, Michael J Franklin, Samuel Madden, and Alan Fekete. Mdcc: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 113–126. ACM, 2013.

Chapter 8

GlobLease: A Globally Consistent and Elastic Storage System using Leases

Ying Liu, Xiaxi Li and Vladimir Vlassov

In *20th IEEE International Conference on Parallel and Distributed Systems* (ICPADS), 2014

Abstract

Nowadays, more and more IT companies are expanding their businesses and services to a global scale, serving users in several countries. Globally distributed storage systems are employed to reduce data access latency for clients all over the world. We present GlobLease, an elastic, globally-distributed and consistent key-value store. It is organised as multiple distributed hash tables (DHTs) storing replicated data and namespace. Across DHTs, data lookups and accesses are processed with respect to the locality of DHT deployments. We explore the use of leases in GlobLease to maintain data consistency across DHTs. The leases enable GlobLease to provide fast and consistent read access in a global scale with reduced global communications. The write accesses are optimized by migrating the master copy to the locations, where most of the writes take place. The elasticity of GlobLease is provided in a fine-grained manner in order to precisely and efficiently handle spiky and skewed read workloads. In our evaluation, GlobLease has demonstrated its optimized global performance, in comparison with Cassandra, with read and write latency less than 10 ms in most of the cases. Furthermore, our evaluation shows that GlobLease is able to bring down the request latency under an instant 4.5 times workload increase with skewed key distribution (a zipfian distribution with an exponent factor of 4) in less than 20 seconds.

8.1 Introduction

With the increasing popularity of Cloud computing, as an essential component of it, distributed storage systems have been extensively used as backend storages by most of the cutting-edge IT companies, including Microsoft, Google, Amazon, Facebook, LinkedIn, etc. The rising popularity of distributed storage systems is mainly because of their potentials to achieve a set of desired properties, including high performance, data availability, system scalability and elasticity. However, achieving these properties is not trivial. The performance of a distributed storage system depends on many factors including load balancing, replica distribution, replica synchronization and caching. To achieve high data availability without compromising data consistency and system performance, a set of algorithms needs to be carefully designed, in order to efficiently synchronize data replicas. The scalability of a distributed storage system is achieved through the proper design of the system architecture and the coherent management of all the factors mentioned above. Some of the state of the art systems achieving most of the above desire properties are presented in [1, 2, 3, 4].

System performance can be largely leveraged when using replication. Replication provides a system to handle workload simultaneously using multiple replicas, thus achieving higher system throughput. Furthermore, intuitively, the availability of data is increased by maintaining multiple copies in the system. However, replication also brings a side-effect, which is the maintenance of replica consistency. Consistency maintenance among replicas imposes an extra communication overhead in the storage system that can cause the degradation of the system performance and scalability. This side-effect is even more obvious when the system is geo-replicated, where the communications among replicas might experience relatively long latency. Furthermore, since the storage service is a stateful service, the elasticity of a distributed storage system is extremely hard to achieve. Specifically, the elasticity of a storage system cannot be achieved only by adding or removing storage servers. The state (data) need to be replicated or reallocated, which introduces a significant data movement overhead.

We consider the following scenario. Assume a large scale service with clients distributed in a global scale, where the majority of them are readers. It is often the case that the popular contents attract significant percentage of readers. Typically, the workload increase caused by the popular contents is usually spiky and not long-lasting. Typical applications include Wikis, WEB 2.0 and social network applications. A well-known incident was the death of Michael Jackson, when his profile page attracted a vast amount of readers in a short interval, causing a sudden spiky workload. In order to efficiently and effectively handle such skewed and spiky workload, we propose GlobLease, a consistent and elastic storage system that can be deployed in a global scale. It achieves low latency read accesses in a global scale and efficient write accesses in one area with sequential consistency guarantees. Fine-grained elasticity with reduced data movement overhead is integrated in GlobLease to handle the popular contents (spiky and skewed workload).

The contributions of this work are as follows.

- We explore the use of multiple DHTs for geo-replication, which implements geo-aware routing.
- We propose a lease-based consistency protocol, that shows high performance for global read and regional write accesses by reducing the global communication.
- We provide the fine-grained elasticity of GlobLease using affiliated nodes, that enables the efficient handling of spiky and skewed workload.
- We evaluate the geo-performance and fine-grained elasticity of GlobLease in comparison with Cassandra in Amazon EC2.

8.2 System Architecture

We assume that readers have some knowledge of DHTs and are familiar with the concepts of availability, consistency and scalability in a distributed storage system. The background knowledge can be obtained in [5, 6, 7].

GlobLease is constructed with a configurable number of replicated DHTs shown in Fig. 8.1. Each DHT maintains a complete replication of the whole data. This design provides flexibility in replica distribution and management at a global scale. Specifically, GlobLease forms up replication groups across the DHT rings, which scales out the limitation of successor list replication [8]. Multiple replicated DHTs can be deployed in different geographical regions in order to improve data access latency. Building GlobLease with DHT-based overlay provides it with a set of desirable properties, including self-organization, linear scalability, and efficient lookups. The self-organizing property of DHTs allows GlobLease to efficiently and automatically handle node join, leave and failure

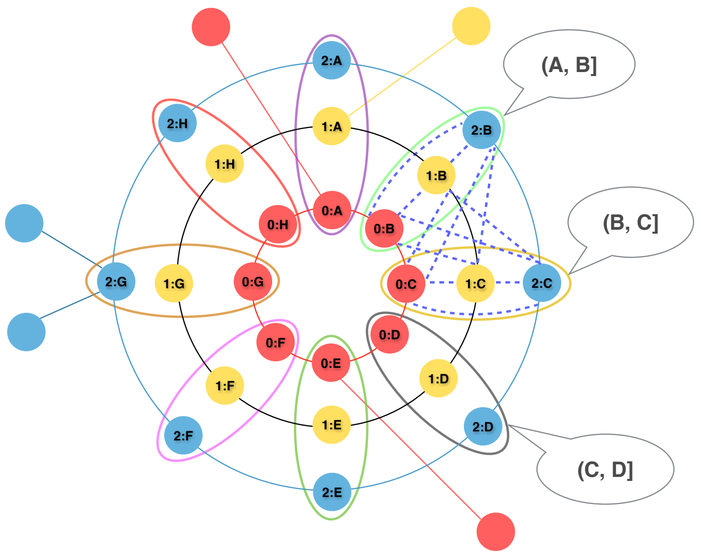

Figure 8.1. GlobLease system structure having three replicated DHTs

events using pre-defined algorithms in each node to stabilize the overlay [5, 9]. The peer-to-peer (P2P) paradigm of DHTs enables GlobLease to achieve linear scalability by adding/removing nodes in the ring. One-hop routing can be implemented for efficient lookups [2].

8.2.1 Nodes

Each DHT Ring is given a unique ring ID shown as numbers in Fig. 8.1. Nodes illustrated in the figure are virtual nodes, which can be placed on physical servers with different configurations. Each node participating in the DHTs is called a standard node, which is assigned a node ID shown as letters in Fig. 8.1. Each node is responsible for a specific key range starting from its predecessor's ID to its own ID. The ranges can be further divided online by adding new nodes. Nodes that replicate the same keys in different DHTs form *the replication group*. For simple illustration, the nodes form the replication group shown within the ellipse in Fig. 8.1 are responsible for the same key range. However, because of possible failures, the nodes in each DHT ring may have different range configurations. Nodes that stretch outside from the DHT rings in Fig. 8.1 are called *affiliated nodes*. They are used for fine-grained management of replicas, which are explained in Section 8.4.1. GlobLease stores key-value pairs. The mappings and lookups of keys are handled by consistent hashing of DHTs. The values associated with the keys are stored in the memory of each node.

8.2.2 Links

Basic Links

Links connecting a node's predecessor and successor within the same DHT are called *local neighbour links* shown as solid lines in Fig. 8.1. Links that connect a node's predecessors and successors across DHTs are called *cross-ring neighbour links* shown as dashed lines. Links within a replication group are called *group links* shown as dashed lines. Normally, routings of requests are conducted with priority choosing local neighbour links. A desired deployment of GlobLease assumes that different rings are placed in different locations. In such case, communications using local neighbour links are much faster than using cross-ring neighbour links. Cross-ring neighbour is selected for routing when there is failure in the next hop local neighbour.

The basic links are established when a standard node or a group of standard nodes join GlobLease. The bootstrapping is similar to other DHTs [5, 9] except that GlobLease needs to update cross-ring neighbour links and group links.

Routing Links

With basic links, GlobLease is able to conduct basic lookups and routings by approaching the requested key hop by hop. In order to achieve efficient lookups, we introduce the routing links, which are used to reduce the message routing hops to reach the responsible node of the requested data. In contrast to basic links, routing links are established gradually with the processing of requests. For example, when node A receives a data request for the first time, which needs to be forwarded to node B, the request is routed to node B hop by hop using basic links. When the request reaches node B, node A will get an echo message regarding the routing information of node B including its responsible key range and ip address. Finally, the routing information is kept in node A's routing table maintained in its memory. As a consequence, a direct routing link is established from node A to node B, which can be used for the routings of future requests. In this way, all nodes in the overlay will eventually be connected with one-hop routing. In failure scenarios, if some links are not reachable, they will be removed from the routing table.

8.3 Lease-based Consistency Protocol

In order to guarantee data consistency in replication groups across DHTs, a lease-based consistency protocol is designed. Our lease-based consistency protocol implements sequential consistency model and is optimized for handling global read-dominant and regional write-dominant workload.

8.3.1 Lease

A lease is an authorization token for serving read accesses within a time interval. A lease is issued on a key basis. There are two essential properties in the lease implementation. First is authorization, which means each replica of the data that has a valid lease is able to serve the read access for the clients. Second is the time bound, which allows the lease itself to expire when the valid time period has passed. The time bound of lease is essential in handling possible failures on non-masters. Specifically, if an operation requires the update or invalidate of leases on non-masters, which cannot be completed due to failures, the operation waits and can proceed when the leases are expired naturally.

8.3.2 Lease-based Consistency

We assign a master on a key basis in each replication group to coordinate the lease-based consistency protocol among replicas. The lease-based protocol handles read and write requests as follows. Read requests can be served by either the master or any non-masters with a valid lease of the requested key. Write requests have to be routed to and only handled by the master of the key. To complete a write request, a master needs to guarantee that leases associated with the written key are either invalid or properly updated together with the data in all the replicas. The validities of leases are checked based on lease records, which are created on masters whenever a lease is issued to a non-master. The above process ensures the serialization of write requests in masters and no stale data will be provided by non-masters, which complies the sequential consistency guarantee.

8.3.3 Lease Maintenance

The maintenance of the lease protocol consists of two operations. One is lease renewals from non-masters to masters. The other one is lease updates issued by masters to non-masters. Both lease renewals and updates need cross-ring communications, which are associated with high latency in a global deployment of GlobLease. Thus, we try to minimize both operations in the protocol design.

A lease renewal is triggered when a non-master receives a read request while not having a valid lease of the requested key. The master creates a lease record and sends the renewed lease with updated data to the non-master upon receiving a lease renewal request. The new lease enables the non-master to serve future reads of the key in the leasing period.

Lease update of a key is issued by the master to its replication group when there is a write to the key. We currently provide two approaches in GlobLease to proceed with lease updates. The first approach is *active update*. In this approach, a master updates leases along with the data of a specific key in its replication group whenever it receives a write on that key. The write is returned when the majority of the nodes in the replication group are updated. This majority should include all the non-masters that still hold valid leases of the key. Write to the majority in a replication group guarantees the high availability of the data. The other approach is *passive update*. It allows a master to reply to a write request faster when a local write is completed. The updated data and leases are propagated to the non-masters asynchronously. The local write is applicable only when there are no valid leases of the written key in the replication group. In case of existing valid leases in the replication group, the master works as the active update. In this way, passive update also keeps sequential consistency guarantee.

Active update provides the system with higher data availability, however, it results in worse write performance because of cross-ring communication. Passive update provides the system with better write performance when the workload is write dominant. However, the data availability is compromised in this case. Both passive and active updates are implemented in separate APIs in GlobLease and can be used by different applications with different requirements.

8.3.4 Leasing Period

The length of a lease is configurable in our system design. At the moment, the reconfiguration of the length of the lease is implemented with node granularity. Further, we plan to extend it to key granularity. The flexibility of lease length allows GlobLease to efficiently handle workload with different access patterns. Specifically, read dominant workload works better with longer leases (less overhead of lease renewals) and write dominant workloads cooperate better with shorter leases (less overhead of lease updates if the passive update mode is chosen).

Another essential issue of leasing is the synchronization of the leasing period on a master and its replication group. Every update from the master should correctly check the validity of all the leases on the non-masters according to the lease records and update them if necessary. This indicates that the record of the leasing period on the master should be the same with or last longer than the corresponding lease on the non-masters. Since it is extremely hard to synchronize the timings in a distributed system [6], we ensure that the record of the leasing periods on the master starts later than the leasing periods on the non-masters. The leases on the non-masters start when the messages of issuing the leases arrive. On the other hand, the records of the leases on the master start when the acknowledgement messages of the successful starting of the leases on the non-masters are received. With the assumption that the latency of message delivery in the network is much more significant than the clock drifts in each participating nodes. The above algorithm guarantees that the records of the leases on the master last longer than the leases on the non-masters and assures the correctness of the consistency guarantee.

8.3.5 Master Migration and Failure Recovery

Master migration is implemented based on a two-phase commit protocol. Master failure is handled by using the replication group as a Paxos group [10] to elect a new master. In order to keep the sequential consistency guarantee in our protocol, we need to ensure that either no master or only one correct master of a key exists in GlobLease.

The two phase commit master migration algorithm works as follows. In the prepare phase, the old master acts as the coordination node, which broadcasts new master proposal message in the replication group. The process will only move forward when an agreement is received from all the nodes from the replication group. In the commit phase, the old master broadcasts the commit message to all the nodes and changes its own state to recognize the new master. Notice that message loss or node failures may happen in this commit phase. If nodes in the replication group, which are not the new master, fail to commit to this message, the recognition of correct mastership is further fixed through an echo message gradually triggered by write requests. Specifically, if the mastership is not correctly changed, the write requests will be forwarded to the old master from the replication group. Since write messages should only be forwarded to the master, when the old master receives a write message from a node in its replication group, it assumes that this node does not know the correct master in the system. An echo message with the correct master information is sent to this node. And the write request is forwarded to the new master. If the new master fails to acknowledge in the commit phase, a roll-back operation will be issued from the old master to its replication group.

Master failure recovery is implemented based on the assumption of fail stop model [11]. There are periodical heartbeat messages from the non-master nodes in the replication group to check the liveness of the current master. If a master node cannot receive the majority of the heartbeat message within a timeout interval, it will give up its mastership to guarantee our previous assumption that there is no more than one master in the system. In the meantime, any non-master node can propose a master election process in the replication group if it cannot receive the response of the heartbeat messages from the master within sufficient continuous period. The master election process follows the two-phase Paxos algorithm. A non-master node in the replication group proposes its own ring ID as well as node ID as values. Only non-master nodes that have passed the heartbeat timeout interval may propose values and vote for the others. The node with the smallest ring ID gets more than the majority of the promises wins the election. Any non-master node that fails to recognize the new master will be guided through the write echo message described above.

8.3.6 Handling Read and Write Requests

With the lease consistency protocol, GlobLease is able to handle read and write requests with respect to the requirement of sequential consistency model. Read requests can be handled by the master of the key as well as the non-masters with valid leases. In contrast, write requests will eventually be routed to the responsible masters. The first time write and future updates of a key are handled differently by master nodes. Specifically, the first time, a write always uses the active update approach, because it creates a record of the written key on non-master nodes, which ensures the correct lookup of the data when clients contact the non-master nodes for read accesses. In contrast, future updates of a key can be handled either using the active or passive approach. After updating the data and lease on non-master nodes, lease records, which store the information of the leasing periods, the associated keys, and the associated non-master nodes, are maintained in master nodes. The lease records are referred when a write request is handled on the master node to decide whether the updates of the leases are required to the non-master nodes if the passive write mode is chosen. Algorithm 1 and Algorithm 2 present the pseudo codes for processing read and write requests.

Algorithm 1 Pseudo Code for Read Request

```
n.receiveReadRequest(msg)
if n.isResponsibleFor(msg.to) then
    if n.isMaster(msg.key) then
        value = n.getValue(msg.key)  n.returnValue(msg.src, value)
    end
    if n.isExpired(lease) then
        n.forwardRequestToMaster(msg)  n.renewLeaseRequest(msg.key)
    else
        value = n.getValue(msg.key)  n.returnValue(msg.src, value)
    end
else
    nextHop = n.getNextHopOfReadRequest(msg.to)  n.forwardRequestToNextNode(nextHop)
end
```

8.4 Scalability and Elasticity

The architecture of GlobLease enables its scalability in two forms. First, the scalable structure of DHTs allows GlobLease to achieve elasticity by adding or removing nodes to the ring overlay. With this property, GlobLease can easily expand to a larger scale in order to handle generally larger workload or scale down to save cost. However, this form of scalability is associated with large overhead, including reconfiguration of multiple ring overlays, key range divisions and the data rebalancing associated, and the churn of the routing table stored in each node's memory. Furthermore, this approach is feasible only when the workload is growing in a uniform manner. Thus, when confronting intensively changing workloads or with skewed distribution, this form of elasticity might not be enough. We have extended the system with fine-grained elasticity by using affiliated nodes.

8.4.1 Affiliated Nodes

Affiliated nodes are used to leverage the elasticity of the system. Specifically, the application of affiliated nodes allows configurable replication degrees for each key. This is achieved by attaching affiliated nodes to any standard nodes, which are called host standard nodes in this case. Then, a configurable subset of the keys served in the host standard node can be replicated at attached affiliated nodes. The affiliated nodes attached to the same host standard node can have different configurations on the set of the replicated keys. The host standard node is responsible to issue and maintain leases of the keys replicated at each affiliated node. The routing links to the affiliated nodes are established in other standard nodes' routing tables respect to a specific key after the first access forwarded by the host standard node. If multiple affiliated nodes hold the same key, the host standard node forwards requests in a round-robin fashion.

Affiliated nodes are designed as lightweight processes that can join/leave system overlay by only interacting with a standard node. Thus, addition and removal of affiliate nodes introduce very little overhead. Rapid deployment of affiliated nodes allows GlobLease to

Algorithm 2 Pseudo Code for Write Request

```
n.receiveWriteRequest(msg, MODE)
%Check whether it is a key update with passive update mode
if n.contains(msg.key) & MODE == PASSIVE then
    leaseRec = n.getLeaseRecord(msg.key)
    if leaseRec == ALLEXPIRE then
        n.writeValue(msg.key, msg.value)
        lazyUpdate(replicationGroup, msg)
        return SUCCESS
    end
else
    lease = n.generatorLease()
    for $server \in replicationGroup$ do
        checkResult = n.issueLease(server, msg.key, msg.value, lease)
    end
    while retries do
        ACKServer = getACKs(checkResult)
        noACKServer = replicationGroup-ACKServer
        leaseExpired = getLeaseExp(leaseRec)
        if $noACKServer \in leaseExpired$ & $sizeOf(noACKServer) < sizeOf(replicationGroup)/2$ then
            lazyUpdate(noACKServer, msg)
            n.writeValue(msg.key, msg.value)
            for $server \in ACKServer$ do
                n.putLeaseRecord(server, msg.key, lease)
            end
            return SUCCESS;
        else
            for $server \in noACKServer$ do
                checkResult += n.issueLease(server, msg.key, msg.value, lease)
            end
            retries -= retries
        end
    end
    return FAIL;
end
```

swiftly handle workload spikes. Furthermore, the dynamic configuration of the replication degrees on a key basis allows skewed (popular) keys to be highly replicated on the affiliated nodes on demand. This key-based extra replication not only allows GlobLease to handle skewed workloads, but also further leverage the fast deployment of affiliated nodes, which requires less data movement overhead by precisely replicating the highly demanded keys. In this way, GlobLease is also able to handle spiky and skewed workload in a swift fashion. There is no theoretical limit on the number of the affiliated nodes in the system, the only concern is the overhead to maintain data consistency on them, which is explained in the next section.

Consistency Issues

In order to guarantee data consistency in affiliated nodes, a secondary lease is established between an affiliated node and a host standard node. The secondary lease works in a similar way as the lease protocol introduced in Section 8.3. An affiliated node holding a valid lease of a specific key is able to serve the read requests of that key. The host standard node is regarded as the master to the affiliated node and maintains the secondary lease. The principle of issuing a secondary leases on an affiliated node is that it should be a sub-period of a valid lease of a specific key holding on the host standard node. The invalidation of a key's lease on a host standard node involves the invalidation of all the valid secondary leases of this key issued by this host standard node to its affiliated nodes.

8.5 Evaluation

We evaluate the performance of GlobLease under different intensities of read/write workloads in comparison with Cassandra [1]. Furtermore, the evaluation of GlobLease goes through its performance with different read/write ratios in workloads and different configurations of lease lengths. The fine-grained elasticity of GlobLease is also evaluated through handling spiky and skewed workloads.

8.5.1 Experiment Setup

We use Amazon Elastic Compute Cloud (EC2) to evaluate the performance of GlobLease. The choice of Amazon EC2 allows us to deploy GlobLease in a global scale. We evaluate GlobLease with four DHT rings deployed in the U.S. west (California), U.S. East, Ireland, and Japan. Each DHT ring consists of 15 standard nodes and a configurable number of affiliated nodes according to different experiments. We use the same Amazon EC2 instance to deploy standard nodes and affiliated nodes. One standard or affiliated node is deployed on one Amazon EC2 instance. The configuration of the nodes are described in Table 8.1.

As a baseline experiment, Cassandra is deployed using the same EC2 instance type and amount in each region as GlobLease. We configure read and write quorums in Cassandra in favor of its performance. Specifically, for read dominant workload, Cassandra reads from one replica and writes to all replicas. For write dominant workload, Cassandra writes to one replica and reads from all replicas. Note that with this configuration, Cassandra gains more performance since only one replica from one region is needed to process a request. However, Cassandra only achieves casual consistency in read-dominant experiment and sequential consistency in write dominant experiment, which is less stringent than GlobLease. More replicas (overhead) are needed to achieve sequential consistency with read dominant workload in Cassandra. Even though, GlobLease outperforms Cassandra as shown in our evaluations.

We have modified Yahoo! Cloud Serving Benchmark (YCSB) [12] to generate either uniform random or skewed workloads to GlobLease and Cassandra. YCSB clients are deployed in an environment described in Table 8.1 and parameters for generating workloads are presented in Table 8.2.

Table 8.1. Node Setups

Specifications	Nodes in GlobLease	YCSB client
Instance Type	m1.medium	m1.xlarge
CPUs	Intel Xeon 2.0 GHz	Intel Xeon 2.0 GHz*4
Memory	3.75 GiB	15 GiB
OS	Ubuntu Server 12.04.2	Ubuntu Server 12.04.2
Location	U.S. West, U.S. East, Ireland, Japan	U.S. West, U.S. East, Ireland, Japan

Table 8.2. Workload Parameters

Total clients	50
Request per client	Maximum 500 (best effort)
Request rate	100 to 2500 requests per second (2 to 50 requests/sec/client)
Read dominant workload	95% reads and 5% writes
Write dominant workload	5% reads and 95% writes
Read skewed workload	Zipfian distribution with exponent factor set to 4
Length of the lease	60 seconds
Size of the namespace	10000 keys
Size of the value	10 KB

8.5.2 Varying Load

Fig. 8.2 presents read performance of GlobLease with comparison to Cassandra using the read dominant workload. The workloads are evenly distributed to all the locations according to GlobLease and Cassandra deployments. The two line plots describe the average latency of GlobLease and Cassandra under different intensities of workloads. In GlobLease, the average latency slightly decreases with the increase of workload intensity because of the efficient usage of lease. Specifically, each renewal of the lease involves the interaction between master and non-master nodes, which introduces high cross region communication latency. When the intensity of the read dominant workload increases, within a leasing period, data with valid leases are more frequently accessed, which results in a large portion of requests are served with low latency. This leads to the decrease of the average latency in GlobLease. In Contrast, as the workload increases, the contention for routing and the access to data on each node are increased, which causes the slight increase of average latency in Cassandra.

The boxplot in Fig. 8.2 shows the read latency distribution of GlobLease (left box) and Cassandra (right box). The outliers, which are high latency requests, are excluded from the boxplot. The high latency requests are discussed in Fig. 8.4 and Fig. 8.5. The boxes in the boxplots are increasing slowly since the load on each node is increasing. The performance of GlobLease is slightly better than Cassandra in terms of the latency of local operations

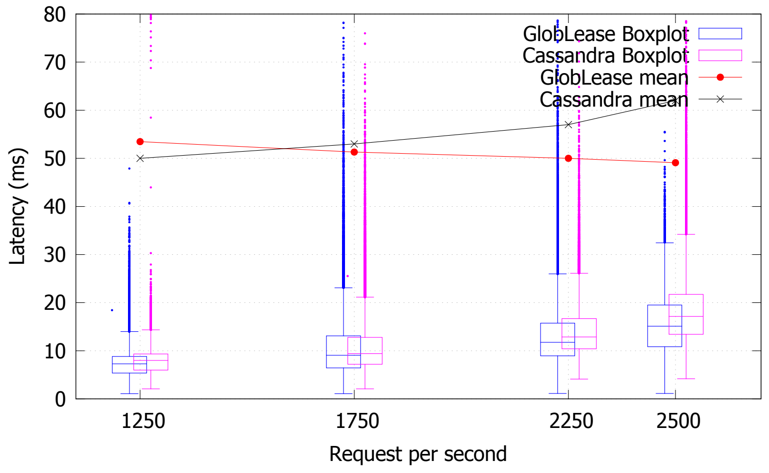

Figure 8.2. Impact of varying intensity of read dominant workload on the request latency

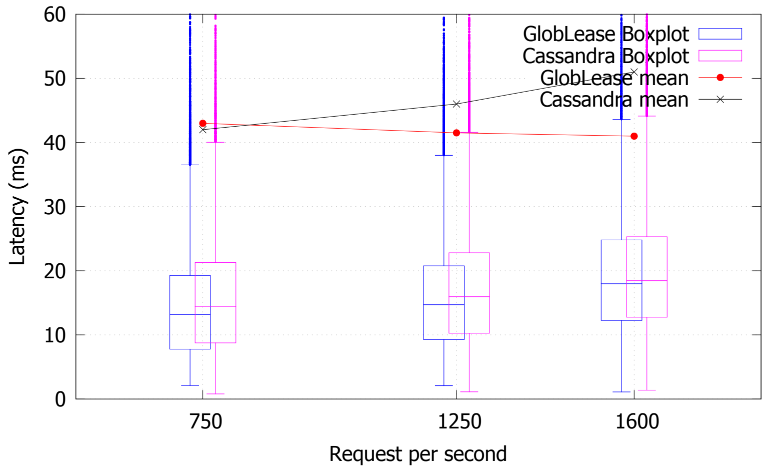

Figure 8.3. Impact of varying intensity of write dominant workload on the request latency

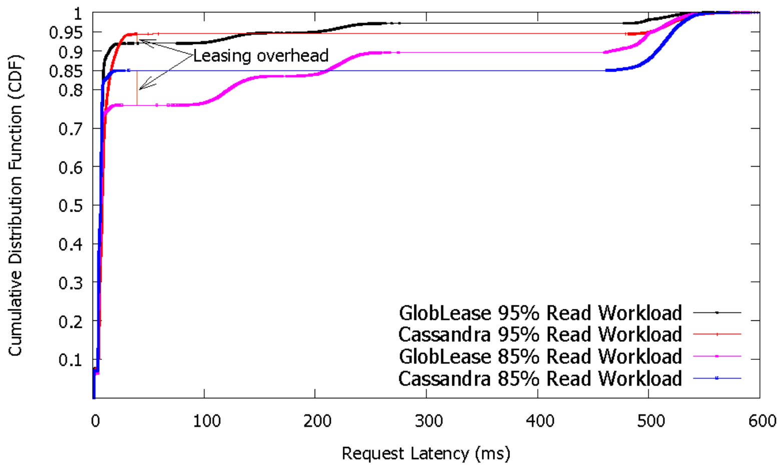

Figure 8.4. Latency distribution of GlobLease and Cassandra under two read dominant workloads

(operations that do not require cross region communication) shown in the boxplots and the average latency shown in line plots. There are several techniques that contribute to the high performance of GlobLease, including one-hop routing (lookup), effective load balancing (key range/mastership assignment) and efficient key-value data structure stored in memory.

For the evaluation of write dominant workload, we enable master migration in GlobLease. We assume that a unique key is only written in one region and the master of the key is assigned to the corresponding region. This assumption obeys the fact that users do not frequently change their locations. With master migration, more requests can be processed locally if the leases on the requested keys are expired and passive write mode is chosen. For the moment, the master migration is not automated, it is achieved by calling the master migration API from a script by analyzing the incoming workload (offline).

An evaluation using write-dominant workload on GlobLease and Cassandra is presented in Fig. 8.3. GlobLease achieves better performance in local write latency and overall average latency than Cassandra. The results can be explained in the same way as the previous read experiment.

Fig. 8.4 shows the performance of GlobLease and Cassandra using two read dominant workload (85% and 95%) in CDF plot. The CDF gives a more complete view of two systems' performance including the cross region communications. Under 85% and 95% read dominant workload, Cassandra experience 15% and 5% cross region communications, which are more than 500 ms latency. These cross region communications are triggered by write operations because Cassandra is configured to read from one replica and write to all replicas, which in favor of its performance under the read dominant workload. In contrast, GlobLease pays around 5% to 15% overhead in maintaining leases (cross region communication) in 85% and 95% read dominant workloads as shown in the figure. From the CDF, around 1/3 of the cross region communication in GlobLease are around 100 ms, another 1/3 are around 200 ms and the rest are, like Cassandra, around 500 ms. This is

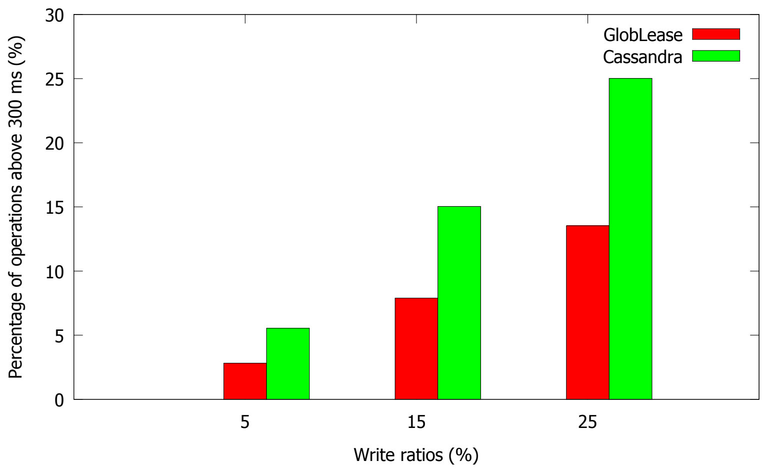

Figure 8.5. High latency requests

because renewing/invalidating leases do not require all the replicas to participate. Respect to the consistency algorithm in GlobLease, only master and non-masters with valid lease of the requested key are involved. So master of a requested key in GlobLease might need to interact with 0 to 3 non-masters to process a write request. Latency connecting data centers varies from 50 ms to 250 ms, which result in 100 ms to 500 ms round trip. In GlobLease, write request are processed with global communication latency ranging from 0 ms to 500 ms depending on the number of non-master replicas with valid lease. On the other hand, Cassandra always needs to wait for the longest latency among servers in different data centers to process a write operation which requires the whole quorum to agree. As a result, GlobLease outperforms Cassandra after 200 ms as shown in Fig. 8.4. Fig. 8.5 zooms in on the high latency requests (above 300 ms) in Fig. 8.4 under three read dominant workloads (75%, 85% and 95%). GlobLease significantly reduces (around 50%) high latency requests comparing to Cassandra. This improvement is crucial to the applications that are latency sensitive or having stringent SLO requirements.

8.5.3 Lease Maintenance Overhead

In Fig. 8.6, we evaluate lease maintenance overhead in GlobLease. The increasing portion of write request imposes more lease maintenance overhead on GlobLease since writes trigger lease invalidation and cause future lease renewals. The y-axis in Fig. 8.6 shows the extra lease maintenance messages comparing to Cassandra under throughput of 1000 request per second and 60 second lease. The overhead of lease maintenance is bounded by the following formula:

$$\frac{WriteThroughput}{ReadThroughput} + \frac{NumberOfKeys}{LeaseLength * ReadThroughput}$$

The first part of the formula represents the overheads introduced by writes that inval-

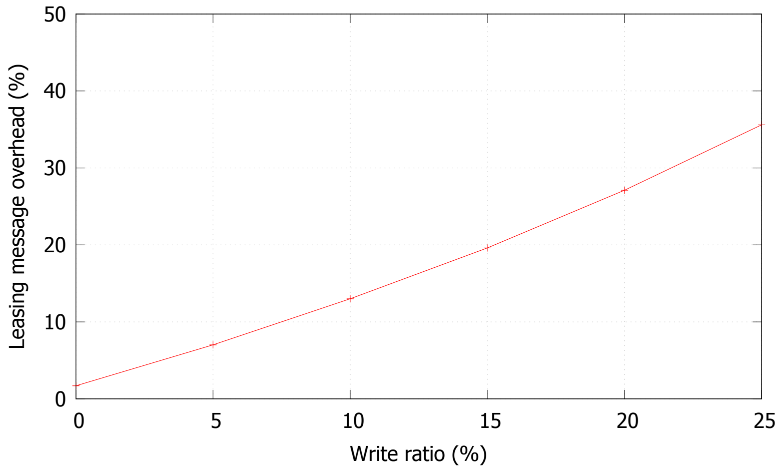

Figure 8.6. Impact of varying read:write ratio on the leasing overhead

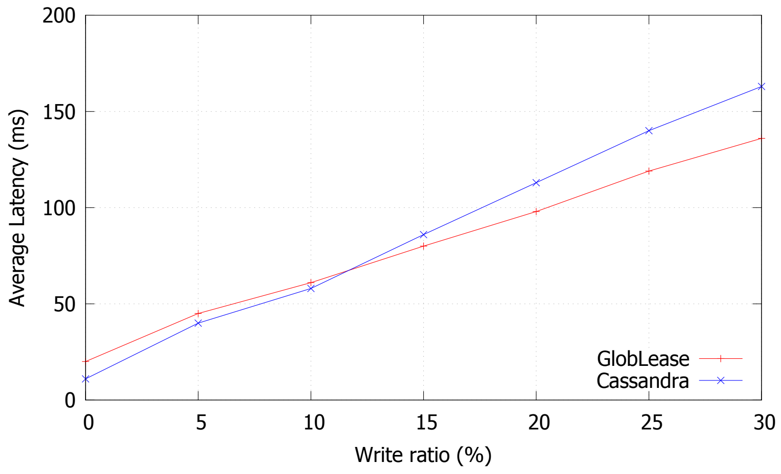

Figure 8.7. Impact of varying read:write ratio on the average latency

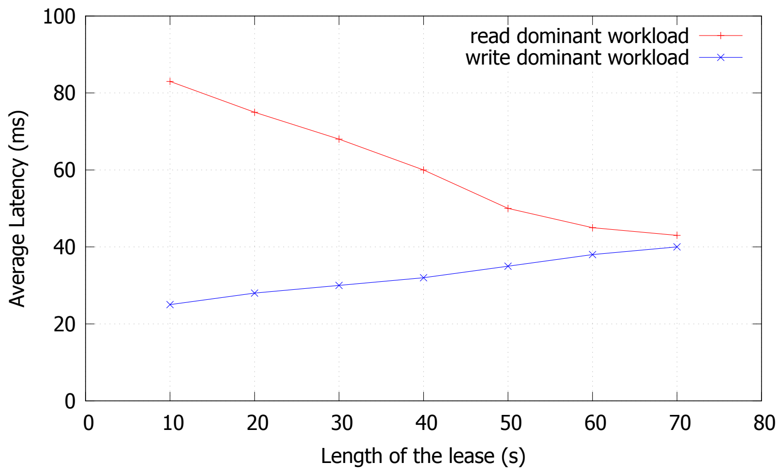

Figure 8.8. Impact of varying lengths of leases on the average request latency

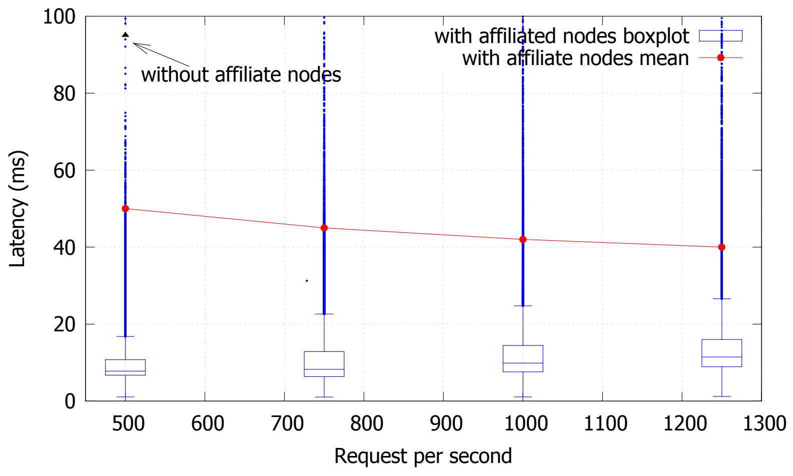

Figure 8.9. Impact of varying intensity of skewed workload on the request latency

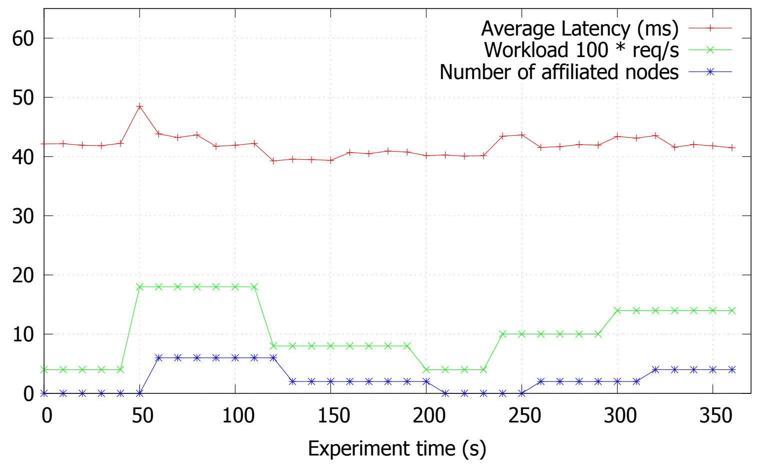

Figure 8.10. Elasticity experiment of GlobLease

idate leases. The second part of the formula stands for the overheads for reads to renew leases. Even though lease maintenance introduces some overhead, GlobLease can outperform quorum-based storage systems, such as Cassandra, when latency between data centers vary. GlobLease benefits from smaller latency among close data centers as shown in Fig. 8.4 and Fig. 8.5.

8.5.4 Varying Read/Write Ratio

In Fig. 8.7, we vary the read/write ratio of the workload. The workload intensity is fixed to 1000 request per second for both GlobLease and Cassandra. As shown in Fig. 8.7, GlobLease has larger average latency comparing to Cassandra when the write ratio is low. This is because that GlobLease pays overhead to maintain leases as evaluated in Fig. 8.6. However, GlobLease outperforms Cassandra when the write ratio grows. This is explained in Fig. 8.5 where GlobLease reduces the percentage of high latency requests significantly comparing to Cassandra. The improvement on the high latency requests compensate the overhead of lease maintenance leading better average latency in GlobLease.

8.5.5 Varying Lease Length

We vary the length of leases to examine its impact on access latency for read-dominant and write-dominant workloads. The workload intensity is set to 1000 requests per sec. Fig. 8.8 shows that, with the increasing length of the lease, average read latency improves significantly since, in a valid leasing time, more read accesses can be completed locally. In contrast, average write latency increases since more cross-region updates are needed if there are valid leases in non-master nodes. Since the percentage of the mixture of reads and writes in read and write dominant workload are the same (95%), with the increasing length of the lease, they approximate the same steady value. Specifically, this steady value, which is around 60s in our case, is also influenced by the throughput of the system and the number of key entries.

8.5.6 Skewed Read Workload

In this experiment, we measure the performance of GlobLease under highly skewed read-dominant workload, which is common in the application domain of social networks, wikis, and news where most of the clients are readers and the popular contents attract most of the clients. We have extended YCSB to generate highly skewed read workload following the Zipfian distribution with the exponent factor of 4. Fig. 8.9 shows that, when GlobLease has sufficient number of affiliated nodes (6 in this case), it can handle skewed workload by coping the highly skewed keys in the affiliated nodes. The point in the top-left corner of the plot shows the performance of the system without affiliated nodes, which is the case of a system without fine-grain replica management. This scenario cannot expand to higher load because of the limit of high latency and the number of clients.

8.5.7 Elasticity with Spiky and Skewed Workload

Fig. 8.10 shows GlobLease's fine-grained elasticity under highly spiky and skewed workload, which follows a Zipfian distribution with the exponent factor of 4. The workload is spread evenly in three geographical locations, where GlobLease is deployed. The intensity of the workload changes from 400 req/s to 1800 req/s immediately at 50s point in the x-axis. Based on the key ranks of the Zipfian distribution, the most popular 10% of keys are arranged to be replicated in the affiliated nodes in three geographical locations. Based on our observation, it takes only tens of milliseconds for an affiliated node to join the overlay and several seconds to transfer the data to it. The system stabilizes with affiliated nodes serving the read workloads in less than 10 sec. Fig. 8.10 shows that GlobLease is able to handle highly spiky and skewed workload with stable request latency, using fine-grained replica management in the affiliated nodes. For now, the process of workload monitoring, key pattern recognition, and keys distribution in affiliated nodes are conducted with pre-programmed scripts. However, this can be automated using control theory and machine learning as discussed in [13, 14, 15].

8.6 Related Work

8.6.1 Distributed Hash Tables

DHTs have been widely used in many storage systems because of their P2P paradigm, which enables reliable routing and replication in the presence of node failures. Selected studies of DHTs are presented in Chord [5], Pastry [16], Symphony [9]. The most common replication schema implemented on top of DHTs are successor-lists, multiple hash functions or leaf-sets. Besides, ID-replication [8, 17] and symmetric replication [18] are also discussed in literature. Our approach takes advantage of DHT's reliable routing and self-organizing structure and is different from the existing approaches in two aspects. First, we have implemented our own replication schema across multiple DHT overlays, which aims at fine-grained replica placement in the scenario of geographical replication. Our replication schema is similar to [8] but differs from it in the granularity of replica management and the routing across replication groups. Second, when GlobLease is deployed in a global scale, request routing is handled by selecting link with low latency according to the deployment.

8.6.2 Data Consistency

Consistency protocols in geo-replicated scenarios have gained great interests recently. Recent proposed solutions [7, 19, 20] use the classical Paxos algorithm [10], which requires a quorum to agree on operations and transactions. In contrast, we implement a strong consistency protocol inspired by the cache coherency protocol [21]. Our consistency schema distinguishes the read and write performance. We expect to have better performance in reads comparing to the previous approaches, since, most of the times, no global synchronization is needed. We have masters on key basis to handle the write accesses. With the flexibility to migrate the masters to the most intensive written location, the write accesses are also improved in GlobLease.

8.6.3 Lease-based Consistency

There are many usage scenarios of leases in distributed systems. Leases are first proposed to deal with distributed cache consistency issues in [22]. Later, the idea of using leases to maintain cache consistency is extended in [23]. Leases are also used to improve the performance of classic Paxos algorithm [24]. Furthermore, leases were explored to preserve consistency in transactions [25, 26, 27]. In sum, leases are used to guarantee the correctness of a resources in a time interval. Because leases are time-bounded assertions of resources, leases are fault tolerant in a distributed environment. In our paper, we explore the usage of leases in maintaining data consistency in a geo-replicated key-value store. Leases are used to reduce the overhead of consistency maintenance across geographical areas where communications among nodes observe significant latency. Evaluation shows that lease is effective in reducing high latency requests by only paying a reasonable overhead.

8.6.4 Elasticity Issues

Elasticity is a property of a system, which allows it to scale up and down, i.e., to grow and shrink, in order to offer satisfactory service with reduced cost in the presence of changing workloads. In particular, elasticity of a storage service, which requires data to be properly allocated before serving the clients, is well studied in [15, 14, 13], etc. However, to our best knowledge, most of these works tackle with elasticity in a coarse-grained fashion under the assumption that the changing of workloads are uniformly distributed on each participating node. In this way, the elasticity is achieved by adding/removing nodes based on workload intensity without transparently managing data skewness. In contrast, GlobLease focuses on fine-grained elasticity. Skewed or spiky keys are efficiently and precisely replicated with higher replication degree and start serving the workload with reduced overhead in affiliated nodes.

8.6.5 Storage Systems

Many successful distributed storage systems have been built by cutting-edge IT companies recently, including Google's Spanner [4], Facebook's Cassandra [1], Microsoft's Azure storage [28], Linkedin'a Voldemort [29], Yahoo!'s PNUTS [3], and Amazon's Dynamo [2]. GlobLease, as well as Voldemort, Cassandra and Dynamo, employs DHTs for namespace division, request routing, and fault tolerance. GlobLease differs from them mainly in two aspects. First, they will not scale to a global scale because of the successor-list replication, which, to some extent, tightly bounds system deployment. GlobLease solves this issue by replicating multiple DHTs with integration of geo-aware routing. Second, to our best knowledge, none of these systems is able to change the replication degree of a specific key swiftly on the fly to handle highly skewed and spiky workload.

8.7 Conclusion

GlobLease aims at achieving low latency data accesses and fine-grain replica management in a global scale. It employs multiple DHT overlays, each of which is a replicated data store in order to reduce access latency in a global scale. Geo-aware routing among DHTs is implemented to reduce data lookup cost. A lease-based consistency protocol is designed and implemented, which is optimized for keeping sequential data consistency with reduced global communication. Comparing to Cassandra, GlobLease achieves faster read and write accesses in a global scale with less than 10 ms latency in most of the cases. The overhead of maintaining leases and the influence of lease length are also studied in our work. Furthermore, the secondary leasing protocol allows us to efficiently control data consistency in affiliated nodes, which are used to serve spiky and skewed read workloads. Our evaluation shows that GlobLease is able to react to an instant 4.5 times workload increase with skewed key distribution swiftly and brings down the request latency in less than 20 seconds. In sum, GlobLease has demonstrated its optimized performance and fine-grained elasticity under sequential consistency guarantee in a global scale.

Acknowledgment

This work was supported by the Erasmus Mundus Joint Doctorate in Distributed Computing funded by EACEA of the European Commission under FPA 2012-0030, and by the FP7 project CLOMMUNITY funded by the European Commission under EU FP7 GA number 317879, and by the End-to-End Clouds project funded by the Swedish Foundation for Strategic Research under the contract RIT10-0043.

Bibliography

[1] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 2010.

[2] Giuseppe DeCandia, Deniz Hastorun, and et al. Dynamo: Amazon's highly available key-value store. In *Proc. SOSP*, 2007.

[3] Brian F. Cooper, Raghu Ramakrishnan, and et al. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB*, 2008.

[4] James C. Corbett, Jeffrey Dean, and et al. Spanner: Google's globally-distributed database. In *Proc. OSDI*, 2012.

[5] Ion Stoica, Robert Morris, and et al. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. SIGCOMM*, 2001.

[6] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 1978.

[7] Tim Kraska, Gene Pang, and et al. Mdcc: multi-data center consistency. In *Proc. EuroSys*, 2013.

[8] Tallat M. Shafaat, Bilal Ahmad, and Seif Haridi. Id-replication for structured peer-to-peer systems. In *Proc. Euro-Par*, 2012.

[9] Gurmeet Singh Manku, Mayank Bawa, and Prabhakar Raghavan. Symphony: distributed hashing in a small world. In *Proc. USITS*, 2003.

[10] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 2001.

[11] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst.*, 1983.

[12] Brian F Cooper, Adam Silberstein, and et al. Benchmarking cloud serving systems with ycsb. In *Proc. SOCC*, 2010.

[13] Ahmad Al-Shishtawy and Vladimir Vlassov. Elastman: elasticity manager for elastic key-value stores in the cloud. In *Proc. CAC*, 2013.

[14] Beth Trushkowsky, Peter Bodík, and et al. The scads director: scaling a distributed storage system under stringent performance requirements. In *Proc. FAST*, 2011.

[15] Harold C Lim, Shivnath Babu, and Jeffrey S Chase. Automated control for elastic storage. In *Proc. ICAC*, 2010.

[16] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proc. Middleware*, 2001.

[17] Lisa Glendenning, Ivan Beschastnikh, and et al. Scalable consistency in scatter. In *Proc. SOSP*, 2011.

[18] Ali Ghodsi, LucOnana Alima, and Seif Haridi. Symmetric replication for structured peer-to-peer systems. In *Databases, Information Systems, and Peer-to-Peer Computing*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007.

[19] Stacy Patterson, Aaron J. Elmore, and et al. Serializability, not serial: concurrency control and availability in multi-datacenter datastores. *Proc. VLDB*, 2012.

[20] Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 2006.

[21] James Archibald and Jean-Loup Baer. Cache coherence protocols: evaluation using a multiprocessor simulation model. *ACM Trans. Comput. Syst.*, 1986.

[22] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *SIGOPS Oper. Syst. Rev.*, 1989.

[23] Jian Yin, Lorenzo Alvisi, Michael Dahlin, and Calvin Lin. Volume leases for consistency in large-scale systems. *IEEE Trans. on Knowl. and Data Eng.*

[24] Felix Hupfeld, BjÃ¶rn Kolbeck, and et al. Fatlease: scalable fault-tolerant lease negotiation with paxos. *Cluster Computing*, 2009.

[25] Jed Liu, Tom Magrino, and et al. Warranties for faster strong consistency. In *Proc. NSDI*, 2014.

[26] Nuno Carvalho, Paolo Romano, and Luís Rodrigues. Asynchronous lease-based replication of software transactional memory. In *Proc. Middleware*, 2010.

[27] Danny Hendler, Alex Naiman, and et al. Exploiting locality in lease-based replicated transactional memory via task migration. In *Distributed Computing*, Lecture Notes in Computer Science. 2013.

[28] Brad Calder, Ju Wang, and et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proc. SOSP*, 2011.

[29] Roshan Sumbaly, Jay Kreps, and et al. Serving large-scale batch computed data with project voldemort. In *Proc. FAST*, 2012.

Chapter 9

BwMan: Bandwidth Manager for Elastic Services in the Cloud

Ying Liu, Vamis Xhagjika, Vladimir Vlassov and Ahmad Al-Shishtawy

In *12th IEEE International Symposium on Parallel and Distributed Processing with Applications* (ISPA), 2014

Abstract

The flexibility of Cloud computing allows elastic services to adapt to changes in workload patterns in order to achieve desired Service Level Objectives (SLOs) at a reduced cost. Typically, the service adapts to changes in workload by adding or removing service instances (VMs), which for stateful services will require moving data among instances. The SLOs of a distributed Cloud-based service are sensitive to the available network bandwidth, which is usually shared by multiple activities in a single service without being explicitly allocated and managed as a resource. We present the design and evaluation of BwMan, a network bandwidth manager for elastic services in the Cloud. BwMan predicts and performs the bandwidth allocation and tradeoffs between multiple service activities in order to meet service specific SLOs and policies. To make management decisions, BwMan uses statistical machine learning (SML) to build predictive models. This allows BwMan to arbitrate and allocate bandwidth dynamically among different activities to satisfy specified SLOs. We have implemented and evaluated BwMan for the OpenStack Swift store. Our evaluation shows the feasibility and effectiveness of our approach to bandwidth management in an elastic service. The experiments show that network bandwidth management by BwMan can reduce SLO violations in Swift by a factor of two or more.

9.1 Introduction

Cloud computing with its pay-as-you-go pricing model and illusion of the infinite amount of resources drives our vision on the Internet industry, in part because it allows providing elastic services where resources are dynamically provisioned and reclaimed in response to fluctuations in workload while satisfying SLO requirements at a reduced cost. When the scale and complexity of Cloud-based applications and services increase, it is essential and challenging to automate the resource provisioning in order to handle dynamic workload without violating SLOs. Issues to be considered when building systems to be automatically scalable in terms of server capabilities, CPU and memory, are fairly well understood by the research community and discussed in literature, e.g., [1, 2, 3]. There are open issues to be solved, such as efficient and effective network resource management.

In Cloud-based systems, services, and applications, network bandwidth is usually not explicitly allocated and managed as a shared resource. Sharing bandwidth by multiple physical servers, virtual machines (VMs), or service threads communicating over the same network, may lead to SLO violations. Furthermore, network bandwidth can also be presented as a first class managed resource in the context of Internet Service Provider (ISP), inter-ISP communication, Clouds as well as community networks [4], where the network bandwidth is the major resource.

In our work, we demonstrate the necessity of managing the network bandwidth shared by services running on the same platform, especially when the services are bandwidth intensive. The sharing of network bandwidth can happen among multiple individual applications or within one application of multiple services deployed in the same platform. In essence, both cases can be solved using the same bandwidth management approach. The difference is in the granularity in which bandwidth allocation is conducted, for example, on

VMs, applications or threads. In our work, we have implemented the finest bandwidth control granularity, i.e., network port level, which can be easily adapted in the usage scenario of VMs, applications, or services. Specifically, our approach is able to distinguish bandwidth allocations to different ports used by different services within the same application. In fact, this fine-grained control is needed in many distributed applications, where there are multiple concurrent threads creating workloads competing for bandwidth resources. A widely used application in such scenario is distributed storage service.

A distributed storage system provides a service that integrates physically separated and distributed storages into one logical storage unit, with which the client can interoperate as if it is one entity. There are two kinds of workload in a storage service. First, the system handles dynamic workload generated by the clients, that we call *user-centric workload*. Second, the system tackles with the workload related to system maintenance including load rebalancing, data migration, failure recovery, and dynamic reconfiguration (e.g., elasticity). We call this workload *system-centric workload*.

In a distributed storage service, the user-centric workload includes access requests issued by clients; whereas the system-centric workload includes the data replication, recovery, and rebalance activities performed to achieve and to ensure system availability and consistency. Typically the system-centric workload is triggered in the following situations. At runtime, when the system scales up, the number of servers and the storage capacity is increased, that leads to data transfer to the newly added servers. Similarly, when the system scales down, data need to be migrated before the servers are removed. In another situation, the system-centric workload is triggered in response to server failures or data corruptions. In this case, the failure recovery process replicates the under-replicated data or recover corrupted data. Rebalance and failure recovery workloads consume system resources including network bandwidth, thus may interfere with user-centric workload and affect SLOs.

From our experimental observations, in a distributed storage system, both user-centric and system-centric workloads are network bandwidth intensive. To arbitrate the allocation of bandwidth between these two kinds of workload is challenging. On the one hand, insufficient bandwidth allocation to user-centric workload might lead to the violation of SLOs. On the other hand, the system may fail when insufficient bandwidth is allocated for data rebalance and failure recovery [1]. To tackle this problem, we arbitrate network bandwidth between user-centric workload and system-centric workload in a way to minimize SLO violations and keep the system operational.

We propose the design of BwMan, a network bandwidth manager for elastic Cloud services. BwMan arbitrates the bandwidth allocation among individual services and different service activities sharing the same Cloud infrastructure. Our control model is built using machine learning techniques [5]. A control loop is designed to continuously monitor the system status and dynamically allocate different bandwidth quotas to services depending on changing workloads. The bandwidth allocation is fine-grained to ports used by different services. Thus, each service can have a demanded and dedicated amount of bandwidth allocation without interfering among each other, when the total bandwidth in the shared platform is sufficient. Dynamic and dedicated bandwidth allocation to services supports their elasticity properties with reduced resource consumption and better performance guarantees. From our evaluation, we show that more than half of the SLO violations is prevented by using BwMan for an elastic distributed storage deployed in the Cloud. Furthermore, since BwMan controls bandwidth in port granularity, it can be easily extended to adapt to other usage scenarios where network bandwidth is a sharing resource and creates potential bottlenecks.

In this work, we build and evaluate BwMan for the case of a data center LAN topology deployment. BwMan assumes that bandwidth quotas for each application is given by data center policies. Within a limited bandwidth quota, BwMan tries to utilize it in the best way, by dividing it to workloads inside the applications. Specifically, BwMan arbitrates the available inbound and outbound bandwidth of servers , i.e., bandwidth at the network edges, to multiple hosted services; whereas the bandwidth allocation of particular network flows in switches is not under the BwMan control. In most of the deployments, control of the bandwidth allocation in the network by services might not be supported.

The contributions of this work are as follows.

- First, we propose a bandwidth manager for distributed Cloud-based services using predictive models to better guarantee SLOs.
- Second, we describe the BwMan design including the techniques and metrics of building predictive models for system performance under user-centric and system-centric workloads as a function of allocated bandwidth.
- Finally, we evaluate the effectiveness of BwMan using the OpenStack Swift Object Storage.

The rest of the paper is organized as follows. In Section 9.2, we describe the background for this work. Section 9.3 presents the control model built for BwMan. In Section 9.4, we describe the design, architecture, and work-flow of BwMan. Section 9.5 shows the performance evaluation of the bandwidth manager. We conclude in Section 9.7.

9.2 OpenStack Swift

A distributed storage service provides an illusion of a storage with infinite capacity by aggregating and managing a large number of storage servers. Storage solutions [6, 7, 8, 9] include relational databases, NoSQL databases, distributed file systems, array storages, and key-value stores. In this paper, we consider an object store, namely OpenStack Swift, as a use case for our bandwidth management mechanism. Swift follows a key-value storage style, which offers a simple interface that allows to put, get, and delete data identified by keys. Such simple interface enables efficient partitioning and distribution of data among multiple servers and thus scaling well to a large number of servers. Examples of key-value storages are Amazon S3, OpenStack Swift, Cassandra [6] and Voldemort [7]. OpenStack Swift, considered in this work, is one of the storage services of OpenStack Cloud platform [8]. In Swift, there are one or many Name Nodes (representing a data entry point to the distributed storage) that are responsible for the management of the Data Nodes. Name Nodes may store the metadata describing the system or just be used as access hubs to the

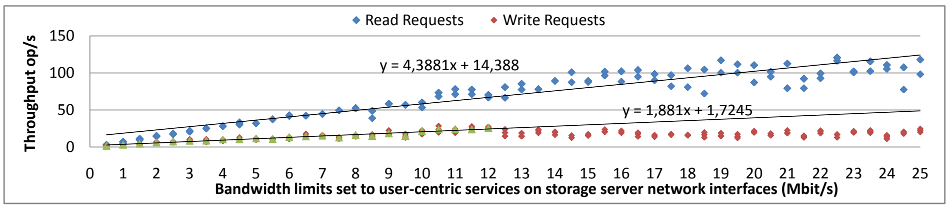

Figure 9.1. Regression Model for System Throughput vs. Available Bandwidth

distributed storage. The Name Nodes may also be responsible for managing data replication, but leave actual data transfer to the Data Nodes themselves. Clients access the distributed storage through the Name Nodes using a mutually agreed upon protocol and the result of the operation is also returned by the same Name Node. Despite Name Nodes and Data Nodes, Swift consists of a number of other components, including Auditors, Updators and Replicators, together providing functionalities such as highly available storage, lookup service, and failure recovery. In our evaluation, we consider bandwidth allocation tradeoffs among these components.

9.3 Predictive Models of the Target System

BwMan bandwidth manager uses easily-computable predictive models to foresee system performance under a given workload in correlation to bandwidth allocation. As there are two types of workloads in the system, namely user-centric and system-centric, we show how to build two predictive models. The first model defines correlation between the user-oriented performance metrics under user-centric workload and the available bandwidth. The second model defines correlation between system-oriented performance metrics under system-centric workload and the available bandwidth.

We define user-oriented performance metrics as the system throughput measured in read/write operations per second (op/s). As a use case, we consider the system-centric workload associated with failure recovery, that is triggered in response to server failures or data corruptions. The failure recovery process is responsible to replicate the under-replicated data or recover corrupted data. Thus, we define the system-oriented performance metrics as the recovery speed of the corrupted data in megabyte per second (MB/s). Due to the fine-grained control of network traffic on different service ports, the bandwidth arbitration by BwMan will not interfere with other background services in the application, such as services for failure detection and garbage collection.

The mathematical models we have used are regression models. The simplest case of such an approach is a one variable approximation, but for more complex scenarios, the number of features of the model can be extended to provide also higher order approximations. In the following subsections, we show the two derived models.

9.3.1 User-oriented Performance versus Available Bandwidth

First, we analyze the read/write (user-centric) performance of the system under a given network bandwidth allocation. In order to conduct decisions on bandwidth allocation against read/write performance, BwMan uses a regression model [2, 3, 10] of performance as a function of available bandwidth. The model can be built either off-line by conducting experiments on a rather wide (if not complete) operational region; or on-line by measuring performance at runtime. In this work, we present the model trained off-line for the OpenStack Swift store by varying the bandwidth allocation and measuring system throughput as shown in Fig. 9.1. The model is set up in each individual storage node. Based on the incoming workload monitoring, each storage node is assigned with demanded bandwidth accordingly by BwMan in one control loop. The simplest computable model that fits the gathered data is a linear regression of the following form:

$$Throughput[op/s] = \alpha_1 * Bandwidth + \alpha_2 \tag{9.1}$$

For example, in our experiments, we have identified the weights of the model for read throughput to be $\alpha_1 = 4.388$ and $\alpha_2 = 14.38$. As shown in Fig. 9.1, this model approximates with a relatively good precision the predictive control function. Note that the second half of the plot for write operations is not taken into consideration, since the write throughput in this region does not depend on the available bandwidth since there are other factors, which might become the bottlenecks, such as disk write access.

9.3.2 Data Recovery Speed versus Available Bandwidth

Next, we analyse the correlation between system-centric performance and available bandwidth, namely, data recovery speed under a given network bandwidth allocation. By analogy to the first model, the second model was trained off-line by varying the bandwidth allocation and measuring the recovery speed under a fixed failure rate. The difference is that the model predictive process is centrally conducted based on the monitored system data integrity and bandwidth are allocated homogeneously to all storage servers. For the moment, we do not consider the fine-grained monitor of data integrity on each storage node. We treat data integrity at the system level.

The model that fits the collected data and correlates the recovery speed with the available bandwidth is a regression model where the main feature is of logarithmic nature as shown in Fig. 9.2. The concise mathematical model is

$$RecoverySpeed[MB/s] = \alpha_1 * ln(Bandwidth) + \alpha_2 \tag{9.2}$$

Fig. 9.2 shows the collected data and the model that fits the data. Specifically, in our case, the weights in the logarithmic regression model are $\alpha_1 = 441.6$ and $\alpha_2 = -2609$.

9.4 BwMan: Bandwidth Manager

In this section, we describe the architecture of BwMan, a bandwidth manager which arbitrates bandwidth between user-centric workload and system-centric workload of the target

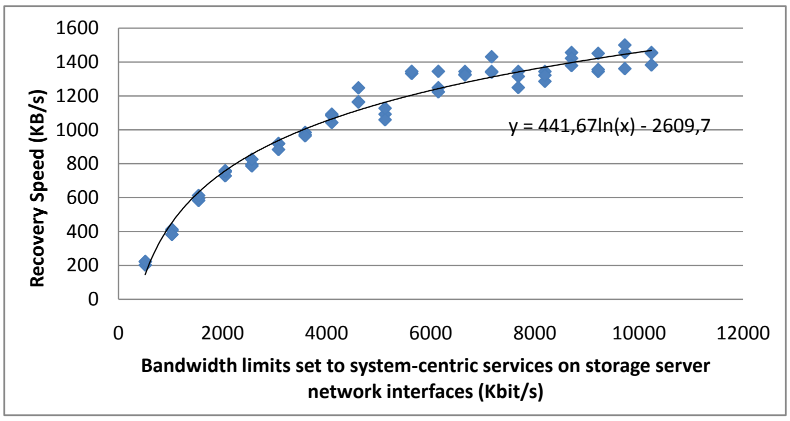

Figure 9.2. Regression Model for Recovery Speed vs. Available Bandwidth

distributed system. BwMan operates according to the MAPE-K loop [11] (Fig. 9.3) passing the following phases:

- Monitor: monitor user-defined SLOs, incoming workloads to each storage server and system data integrity;
- Analyze: feed monitored data to the regression models;
- Plan: use the predictive regression model of the target system to plan the bandwidth allocation including tradeoffs. In the case when the total network bandwidth has been exhausted and cannot satisfy all the workloads, the allocation decisions are made based on specified tradeoff policies (explained in Section 9.4.2);
- Execute: allocate bandwidth to sub-services (storage server performance and system failure recovery) according to the plan.

Control decisions are made by finding correlations through data using two regression models (Section 9.3). Each model defines correlations between a specific workload (user-centric or system-centric) and bandwidth.

9.4.1 BwMan Control Work-flow

The flowchart of BwMan is shown in Fig. 9.4. BwMan monitors three signals, namely, user-centric throughput (defined in SLO), the workload to each storage server and data integrity in the system. At given time intervals, the gathered data are averaged and fed to analysis modules. Then the results of the analysis based on our regression model are passed to the planning phase to decide on actions based on SLOs and potentially make tradeoff decision. The results from the planning phase are executed by the actuators in the execution phase. Fig. 9.4 depicts the MAPE phases as designed for BwMan. For the Monitor phase, we have two separate monitor ports, one for user-centric throughput (M1) and the other one for data failure rates (M2). The outputs of these stages are passed to the

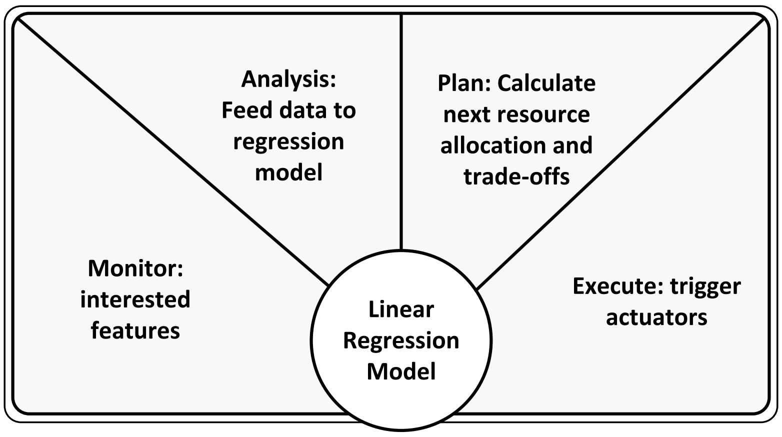
Analysis:
Feed data to regression model
Plan: Calculate next resource allocation and trade-offs
Monitor: interested features
Linear Regression Model
Execute: trigger actuators

Figure 9.3. MAPE Control Loop of Bandwidth Manager

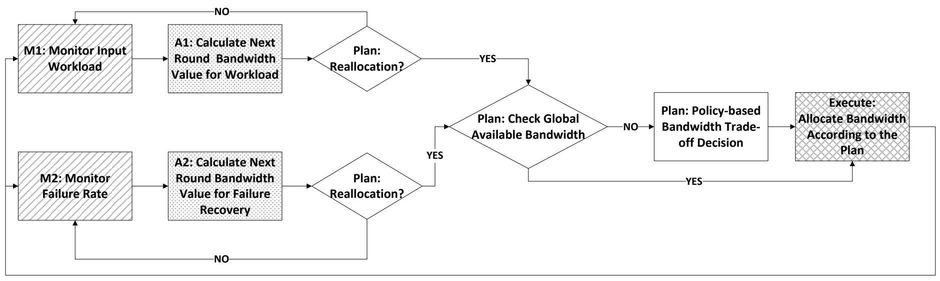
M1: Monitor Input Workload
A1: Calculate Next Round Bandwidth Value for Workload
Plan: Reallocation?
NO
YES
Plan: Check Global Available Bandwidth
NO
Plan: Policy-based Bandwidth Trade-off Decision
Execute: Allocate Bandwidth According to the Plan
YES
M2: Monitor Failure Rate
A2: Calculate Next Round Bandwidth Value for Failure Recovery
Plan: Reallocation?
YES
NO

Figure 9.4. Control Workflow

Analysis phase represented by two calculation units, namely A1 and A2, that aggregate and calculate new bandwidth availability, allocation and metrics to be used during the Planning phase according to the trained models (Section 9.3). The best course of action to take during the Execution phase is chosen based on the calculated bandwidth necessary for user-centric workload (SLO) and the current data failure rate, estimated from system data integrity in the Planning phase. The execution plan may include also the tradeoff decision in the case of bandwidth saturation. Finally, during the Execution phase, the actuators are employed to modify the current state of the system, which is the new bandwidth allocations for the user-centric workload and for the system-centric (failure recovery) workload to each storage server.

9.4.2 Tradeoff Scenario

BwMan is designed to manage a finite resource (bandwidth), so the resource may not always be available. We describe a tradeoff scenario where the bandwidth is shared among user-centric and system-centric workloads.

In order to meet specified SLOs, BwMan needs to tune the allocation of system resources in the distributed storage. In our case, we observe that the network bandwidth available for user-centric workload directly impact user-centric performance (request throughput). Thus, enough bandwidth allocation to the user-centric workload is essential to meet SLOs. On the other hand, system-centric workload, such as failure recovery and data rebalance, are executed in order to provide better reliability for data in a distributed storage. The rebalance and replication process moves copies of the data to other nodes in order to have more copies for availability and self-healing purposes. This activity indirectly limits user-centric performance by impacting the internal bandwidth of the storage system. While moving the data, the available bandwidth for user-centric workload is lowered as system-centric workload competes for the network bandwidth with user-centric workload.

By arbitrating the bandwidth allocated to user-centric and system-centric workloads, we can enforce more user-centric performance while penalizing system-centric functionalities or vice versa. This tradeoff decision is based on policies specified in the controller design.

The system can limit the bandwidth usage of an application by selecting the requests to process and those to ignore. This method is usually referred as admission control, which we do not consider here. Instead we employ actuators to arbitrate the bandwidth between user-centric workload and system-centric workload.

9.5 Evaluation

In this section, we present the evaluation of BwMan in OpenStack Swift. The storage service was deployed in an OpenStack Cloud in order to ensure complete isolation and sufficiently enough computational, memory, and storage resources.

9.5.1 OpenStack Swift Storage

As a case study, we evaluate our control system in OpenStack Swift, which is a widely used open source distributed object storage started from Rackspace [12]. We identify that, in Swift, user-centric workload (system throughput) and system-centric workload (data rebalance and recovery) are not explicitly managed. We observe that data rebalance and failure recovery mechanisms in Swift are essentially the same. These two services adopt a set of replicator processes using the "rsync" Linux utility. In particular, we decide to focus on one of these two services: failure recovery.

9.5.2 Experiment Scenarios

The evaluation of BwMan in OpenStack Swift has been conducted under two scenarios. First, we evaluate the effectiveness of BwMan in Swift with specified throughput SLO for the user-centric workload, and failure rates that correspond to system-centric workload (failure recovery), under the condition that there is enough bandwidth to handle both workloads. These experiments demonstrate the ability of BwMan to manage bandwidth in a way that ensures user-centric and system-centric workloads with maximum fidelity.

Second, a policy-based decision making is performed by BwMan to tradeoff in the case of insufficient network bandwidth to handle both user-centric and system-centric workloads. In our experiments, we give higher priority to the user-centric workload compared to system-centric workload. We show that BwMan adapts Swift effectively by satisfying the user-defined SLO (desired throughput) with relatively stable performance.

9.5.3 Experiment Setup

Swift Setup

We have deployed a Swift cluster with a ratio of 1 proxy server to 8 storage servers as recommended in the OpenStack Swift documentation [13]. Under the assumption of uniform workload, the storage servers are equally loaded. This implies that the Swift cluster can scale linearly by adding more proxy servers and storage servers following the ratio of 1 to 8.

Workload Setup

We modified the Yahoo! Cloud Service Benchmark (YCSB) [14] to be able to generate workloads for a Swift cluster. Specifically, our modification allows YCSB to issue read, write, and delete operations to a Swift cluster with best effort or a specified steady throughput. The steady throughput is generated in a queue-based fashion, where the request incoming rate can be specified and generated on demand. If the rate cannot be met by the system, requests are queued for later execution. The Swift cluster is populated using randomly generated files with predefined sizes. The file sizes in our experiments are chosen based on one of the largest production Swift cluster configured by Wikipedia [15] to store static images, texts, and links. YCSB generates requests with file sizes of 100KB as like an average size in the Wikipedia scenario. YCSB is given 16 concurrent client threads and generates uniformly random read and write operations to the Swift cluster.

Failure Generator and Monitor

The injected file loss in the system is used to trigger the Swift's failure recovery process. We have developed a failure generator script that uniformly at random chooses a data node, in which it deletes a specific number of files within a defined period of time. This procedure is repeated until the requested failure rate is reached.

To conduct failure recovery experiments, we customized the swift-dispersion tool in order to populate and monitor the integrity of the whole data space. This customized tool functions also as our failure recovery monitor in BwMan by providing real-time metrics on data integrity.

The Actuator: Network Bandwidth Control

We apply NetEm's tc tools [16] in the token buffer mode to control the inbound and outbound network bandwidth associated with the network interfaces and service ports. In this

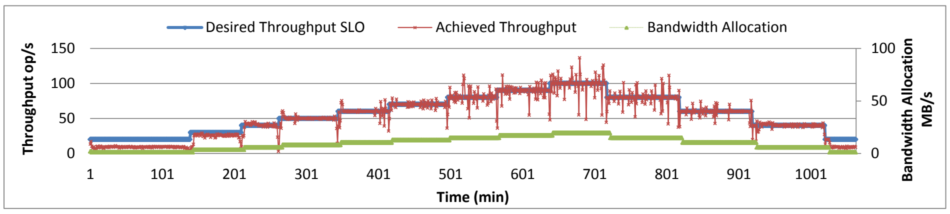

Figure 9.5. Throughput under Dynamic Bandwidth Allocation using BwMan

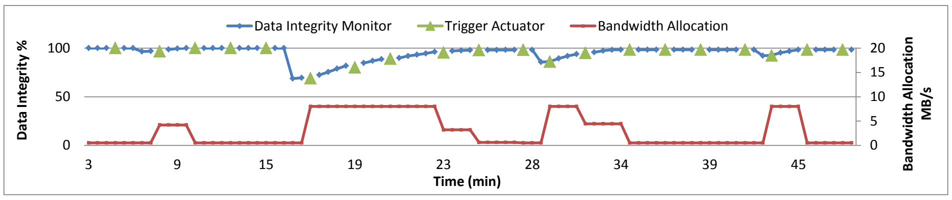

Figure 9.6. Data Recovery under Dynamic Bandwidth Allocation using BwMan

way, we are able to manage the bandwidth quotas for different activities in the controlled system. In our deployment, all the services run on different ports, and thus, we can apply different network management policies to each of the services.

9.5.4 User-centric Workload Experiment

Fig. 9.5 presents the effectiveness of using BwMan in Swift with dynamic user-centric SLOs. The x-axis of the plot shows the experiment timeline, whereas the left y-axis corresponds to throughput in op/s, and the right y-axis corresponds to allocated bandwidth in MB/s.

In these experiments, the user-centric workload is a mix of 80% read requests and 20% write requests, that, in our view, represents a typical workload in a read-dominant application.

Fig. 9.5 shows the desired throughput specified as SLO, the bandwidth allocation calculated using the linear regression model of the user-centric workload (Section 9.3), and achieved throughput. Results demonstrate that BwMan is able to reconfigure the bandwidth allocated to dynamic user-centric workloads in order to achieve the requested SLOs.

9.5.5 System-centric Workload Experiment

Fig. 9.6 presents the results of the data recovery process, the system-centric workloads, conducted by Swift background process when there are data corruption and data loss in the system. The dotted curve sums up the monitoring results, which constitute the 1% random sample of the whole data space. The sample represents data integrity in the system with max value at 100%. The control cycle activation is illustrated as triangles. The solid curve stands for the bandwidth allocation by BwMan after each control cycle. The calculation of

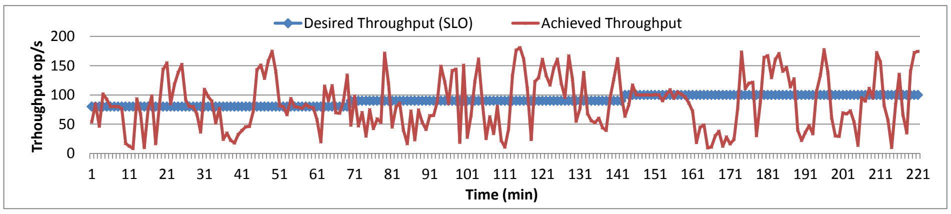

Figure 9.7. Throughput of Swift without BwMan

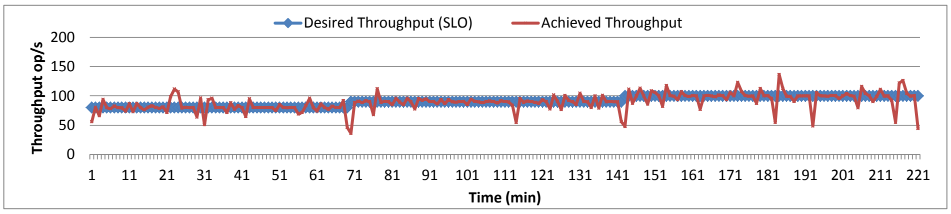

Figure 9.8. Throughput of Swift with BwMan

bandwidth allocation is based on a logarithmic regression model obtained from Fig. 9.2 in Section 9.3.

9.5.6 Policy-based Tradeoff Scenario

In this section, we demonstrate that BwMan allows meeting the SLO according to specified policies in tradeoff decisions when the total available bandwidth is saturated by user-centric and system-centric workloads. In our experiments, we have chosen to give preference to user-centric workload, namely system throughput, instead of system-centric workload, namely data recovery. Thus, bandwidth allocation to data recovery may be sacrificed to ensure conformance to system throughput in case of tradeoffs.

In order to simulate the tradeoff scenario, the workload generator is configured to generate 80 op/s, 90 op/s, and 100 op/s. The generator applies a queue-based model, where requests that are not served are queued for later execution. The bandwidth is dynamically allocated to meet the throughput SLO for user-centric workload.

Fig. 9.7 and Fig. 9.8 depict the results of our experiments conducted simultaneously in the same time frame; the x-axis shares the same timeline. The failure scenario introduced by our failure simulator is the same as in the first series of experiments (see data integrity experiment in Fig. 9.6).

Fig. 9.7 presents the achieved throughput executing user-centric workload without bandwidth management, i.e., without BwMan. In these experiments, the desired throughput starts at 80 op/s, then increases to 90 op/s at about 70 min, and then to 100 op/s at about 140 min. Results indicate high presence of SLO violations (about 37.1%) with relatively high fluctuations of achieved throughput.

Fig. 9.8 shows the achieved throughput in Swift with BwMan. In contrast to Swift without bandwidth management, the use of BwMan in Swift allows the service to achieve

Table 9.1. Percentage of SLO Violations in Swift with/out BwMan

SLO confidence interval	Percentage of SLO violation	
	With BwMan	Without BwMan
5%	19.5%	43.2%
10%	13.6%	40.6%
15%	8.5%	37.1%

required throughput (meet SLO) most of the time (about 8.5% of violation) with relatively low fluctuations of achieved throughput.

Table 9.1 summarizes the percentage of SLO violations within three given confidence intervals (5%, 10%, and 15%) in Swift with/out bandwidth management, i.e., with/out BwMan. The results demonstrate the benefits of BwMan in reducing the SLO violations with at least a factor of 2 given a 5% interval and a factor of 4 given a 15% interval.

9.6 Related Work

The benefits of network bandwidth allocation and management is well understood as it allows improving performance of distributed services, effectively and efficiently meeting SLOs and, as consequence, improving end-users' experience with the services. There are different approaches to allocate and control network bandwidths, including controlling bandwidth at the network edges (e.g., of server interfaces); controlling bandwidth allocations in the network (e.g., of particular network flows in switches) using the software defined networking (SDN) approach; and a combination of those. A bandwidth manager in the SDN layer can be used to control the bandwidth allocation on a per-flow basis directly on the topology achieving the same goal as the BwMan controlling bandwidth at the network edges. Extensive work and research has been done by the community in the SDN field, such as SDN using the OpenFlow interface [17].

A typical work of controlling bandwidth allocation in the network is presented in Seawall [18]. Seawall uses reconfigurable administrator-specified policies to share network bandwidth among services and enforces the bandwidth allocation by tunnelling traffic through congestion controlled, point to multipoint, edge to edge tunnels. In contrast, we propose a simpler yet effective solution. We let the controller itself dynamically decide the bandwidth quotas allocated to each services through a machine learning model. Administrator-specified policies are only used for tradeoffs when the bandwidth quota is not enough to support all the services on the same host. Using machine learning techniques for bandwidth allocation to different services allows BwMan to support the hosting of elastic services in the cloud, whose demand on the network bandwidth varies depending on the incoming workload.

A recent work of controlling the bandwidth on the edge of the network is presented in EyeQ [19]. EyeQ is implemented using virtual NICs to provide interfaces for clients to specify dedicated network bandwidth quotas to each service in a shared Cloud environment. Our work differs from EyeQ in a way that clients do not need to specify a dedicated bandwidth quota, instead, BwMan will manage the bandwidth allocation according to the desired SLO at a minimum bandwidth consumption.

The theoretical study of the tradeoffs in the network bandwidth allocation is presented in [20]. It has revealed the challenges in providing bandwidth guarantees in a Cloud environment and identified a set of properties, including min-guarantee, proportionality and high utilization to guide the design of bandwidth allocation policies.

9.7 Conclusion and Future Work

We have presented the design and evaluation of BwMan, a network bandwidth manager providing model-predictive policy-based bandwidth allocation for elastic services in the Cloud. For dynamic bandwidth allocation, BwMan uses predictive models, built from statistical machine learning, to decide bandwidth quotas for each service with respect to specified SLOs and policies. Tradeoffs need to be handled among services sharing the same network resource. Specific tradeoff policies can be easily integrated in BwMan.

We have implemented and evaluated BwMan for the OpenStack Swift store. Our evaluation has shown that by controlling the bandwidth in Swift, we can assure that the network bandwidth is effectively arbitrated and allocated for user-centric and system-centric workloads according to specified SLOs and policies. Our experiments show that network bandwidth management by BwMan can reduce SLO violations in Swift by a factor of two or more.

In our future work, we will focus on possible alternative control models and methodology of controller designs for multiple Cloud-based services sharing the network infrastructure in Clouds and Cloud federations. In addition, we will investigate impact of network topology and link capacities on the network bottlenecks within or between data centers, and how to integrate controlling bandwidth on edges of the network with bandwidth allocation and with allocation in the network topology using SDN approach.

Acknowledgement

This work was supported by the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC) funded by the Education, Audiovisual and Culture Executive Agency of the European Commission under the FPA 2012-0030, by the End-to-End Clouds project funded by the Swedish Foundation for Strategic Research under the contract RIT10-0043, and by the FP7 project CLOMMUNITY funded by the European Commission under EU FP7 GA number 317879.

Bibliography

[1] Harold C Lim, Shivnath Babu, and Jeffrey S Chase. Automated control for elastic storage. In *Proc. ICAC*, 2010.

[2] Beth Trushkowsky, Peter Bodík, and et al. The scads director: scaling a distributed storage system under stringent performance requirements. In *Proc. FAST*, 2011.

[3] Ahmad Al-Shishtawy and Vladimir Vlassov. ElastMan: Elasticity manager for elastic key-value stores in the cloud. In *Proc. CAC*, 2013.

[4] Bart Braem, Chris Blondia, and et al. A case for research with and on community networks. *ACM SIGCOMM Computer Communication Review*, 2013.

[5] I.H. Witten, E. Frank, and M.A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques: Practical Machine Learning Tools and Techniques*. The Morgan Kaufmann Series in Data Management Systems. Elsevier Science, 2011.

[6] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 2010.

[7] Roshan Sumbaly, Jay Kreps, and et al. Serving large-scale batch computed data with project voldemort. In *Proc. FAST*, 2012.

[8] Openstack cloud software, June 2013.

[9] Yi Wang, Arnab Nandi, and Gagan Agrawal. SAGA: Array Storage as a DB with Support for Structural Aggregations. In *Proceedings of SSDBM*, June 2014.

[10] Peter Bodık, Rean Griffith, and et al. Statistical machine learning makes automatic control practical for internet datacenters. In *Proc. HotCloud*, 2009.

[11] IBM Corp. *An architectural blueprint for autonomic computing*. IBM Corp., 2004.

[12] Ken Pepple. *Deploying OpenStack*. O'Reilly Media, 2011.

[13] Openstack swift's documentation, June 2013.

[14] Brian F Cooper, Adam Silberstein, and et al. Benchmarking cloud serving systems with ycsb. In *Proc. SOCC*, 2010.

[15] Scaling media storage at wikimedia with swift, June 2013.

[16] Stephen Hemminger et al. Network emulation with netem. In *Linux Conf Au*. Citeseer, 2005.

[17] Nick McKeown, Tom Anderson, and et al. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 2008.

[18] Alan Shieh, Srikanth Kandula, and et al. Sharing the data center network. In *Proc. NSDI*, 2011.

[19] Vimalkumar Jeyakumar, Mohammad Alizadeh, and et al. Eyeq: Practical network performance isolation at the edge. In *Proc. NSDI*, 2013.

[20] Lucian Popa, Gautam Kumar, and et al. Faircloud: Sharing the network in cloud computing. In *Proc. SIGCOMM*, 2012.

Chapter 10

ProRenaTa: Proactive and Reactive Tuning to Scale a Distributed Storage System

Ying Liu, Navaneeth Rameshan, Enric Monte, Vladimir Vlassov and Leandro Navarro

To appear *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (CCGrid), 2015

Abstract

Provisioning stateful services in the Cloud that guarantees high quality of service with reduced hosting cost is challenging to achieve. There are two typical auto-scaling approaches: predictive and reactive. A prediction based controller leaves the system enough time to react to workload changes while a feedback based controller scales the system with better accuracy. In this paper, we show the limitations of using a proactive or reactive approach in isolation to scale a stateful system and the overhead involved. To overcome the limitations, we implement an elasticity controller, ProRenaTa, which combines both reactive and proactive approaches to leverage on their respective advantages and also implements a data migration model to handle the scaling overhead. We show that the combination of reactive and proactive approaches outperforms the state of the art approaches. Our experiments with Wikipedia workload trace indicate that ProRenaTa guarantees a high level of SLA commitments while improving the overall resource utilization.

10.1 Introduction

Hosting services in the Cloud are becoming more and more popular because of a set of desired properties provided by the platform, which include low setup cost, professional maintenance and elastic provisioning. Services that are elastically provisioned in the Cloud are able to use platform resources on demand, thus saving hosting costs by appropriate provisioning. Specifically, instances are spawned when they are needed for handling an increasing workload and removed when the workload drops. Enabling elastic provisioning saves the cost of hosting services in the Cloud, since users only pay for the resources that are used to serve their workload.

A well-designed elasticity controller helps reducing the cost of hosting services using dynamic resource provisioning and, in the meantime, does not compromise service quality. Levels of service quality are usually defined in SLAs (Service Level Agreements), which are negotiated and agreed between service consumers and the service providers. A violation of SLA affects both the provider and consumer. When a service provider is unable to uphold the agreed level of service, they typically pay penalties to the consumers. From the consumers perspective, a SLA violation can result in degraded service to their clients and consequently lead to loss in profits. Hence, SLA commitment is essential to the profit of both Cloud service providers and consumers.

In general, Cloud services can be coarsely characterized in two categories: state-based and stateless. Scaling stateless services is easier since no overhead of state migration is involved. However, scaling state-based services often requires state-transfer/replication, which introduces additional overhead during the scaling. In this paper, we investigate the elastic scaling of distributed storage systems, which provide indispensable services in the Cloud and are typical state-based systems. One of the most commonly defined SLAs in a distributed storage system is its service latency. Guaranteeing latency SLAs in back-end distributed storage systems is desirable in supporting many latency sensitive services, such as Web 2.0 services. However, it is a challenging task for an elasticity controller since it needs to achieve the following properties:

1. Accurate resource allocation that satisfy both constraints: minimize provisioning cost and SLA violations.

2. Swift adaptation to workload changes without causing resource oscillation.

3. Be aware of scaling overheads, including the consumption of system resources and time, and prevent them from causing SLA violations.

4. Efficient use of resources under SLA constraints during scaling. Specifically, when scaling up, it is preferable to add instances at the very last possible moment. In contrast, during scaling down, it is better to remove instances as soon as they are not needed anymore. The timings are challenging to control.

To the best of our knowledge, none of the state of the art systems achieve all these properties in their controller designs. Broadly speaking, elasticity in a distributed storage system is achieved in two ways. One solution relies on reacting to real-time system metrics, including CPU usage, incoming load, I/O operations, and etc. It is often referred as *reactive control*. Another approach is to explore historic access patterns of a system in order to conduct workload prediction and controls for the future. It is called *proactive control*.

The first approach can scale the system with a good accuracy since scaling is based on observed workload characteristics. However, a major disadvantage of this approach is that the system reacts to workload changes only after it is observed. As a result, SLA violations are observed in the initial phase of scaling because of data/state migration in order to add/remove instances in a distributed storage system and causes a period of disrupted service. The latter approach, on the other hand, is able to prepare the instances in advance and avoid any disruption in the service. However, the accuracy of workload prediction largely depends on application-specific access patterns. Worse, in some cases workload patterns are not even predictable. Thus, proper methods need to be designed and applied to deal with the workload prediction inaccuracies, which directly influences the accuracy of scaling that in turn impacts SLA guarantees and the provisioning costs.

In essence, proactive and reactive approach complement each other. Proactive approach provides an estimation of future workloads giving a controller enough time to prepare and react to the changes but having the problem of prediction inaccuracy. Reactive approach brings an accurate reaction based on current state of the system but without leaving enough time for the controller to execute scaling decisions.

We present ProRenaTa, which is an elasticity controller for distributed storage systems combining both proactive and reactive scaling techniques. ProRenaTa contributes in achieving the previously identified properties using the following techniques:

- A study of the prediction methods for a typical type of web application (Wikipedia) focusing on pattern characterizations, recognitions and accurate predictions (satisfying property 2).

- A cost model and scheduler for data/state migration is integrated to evaluate the scaling overhead and generate premium scaling plan to execute the predicted scaling decision (satisfying property 3 and 4).

- A reactive module that continuously improves the scaling outcomes of prediction-based scaling by comparing the predicted workload with the observed workload (satisfying property 1).

We show in our evaluations that ProRenaTa outperforms the state of the art approaches in terms of resource utilization (saving hosting cost) and keeping SLA commitment (achieving quality of service).

10.2 Observations and background

It is challenging to achieve elasticity in stateful systems especially under the constraint of SLA. There are two major reasons. One reason is that scaling stateful systems require state migrations, which often introduces an additional overhead. The other reason is that the scaling of such systems are often associated with delays. To be specific, adding or removing instances cannot be completed immediately because of the waiting time for state transfer. These are the two major reasons that cause SLA violations while scaling stateful systems.

In this section, we briefly introduce the concepts of distributed storage systems. Then, we justify the above arguments with experimental observations.

10.2.1 Distributed storage systems

A distributed storage system provides an unified storage service to its clients by integrating and managing a large number of backend storage instances. Compared to traditional storage systems, distributed storage systems usually have the advantages of high scalability and high availability. Distributed storage systems can be organized using many different approaches. For example, Hadoop Distributed File System [1] organizes its storage instances using a centralized naming service provided by a single NameNode; Cassandra [2] and Dynamo [3] like systems employ distributed hash tables (DHTs) to decentralize the maintenance of the namespace and request routing; Spanner [4] and PNUTs [5] adopts multiple name service components to manage the namespace and request routing. In our work, we take a kind of distributed storage system that is organized similar to Cassandra. Background of such storage systems can be obtained in [2, 3]. Specifically, we use GlobLease [6] as the underlying storage system serving the workload. GlobLease is a key-value store that uses DHT for request routing and namespace maintenance similar to Cassandra [2]. Virtual tokens are implemented to even the workload distribution and overhead on addition or removal of nodes in the overlay. We setup GlobLease very similar to Cassandra using the read/write consistency level "ONE". More details of GlobLease is presented in [6].

10.2.2 Observations

We setup experiments to investigate the scaling of GlobLease with respect to a simple workload pattern described in Figure 10.1 (a). The experiment is designed as simple as possible

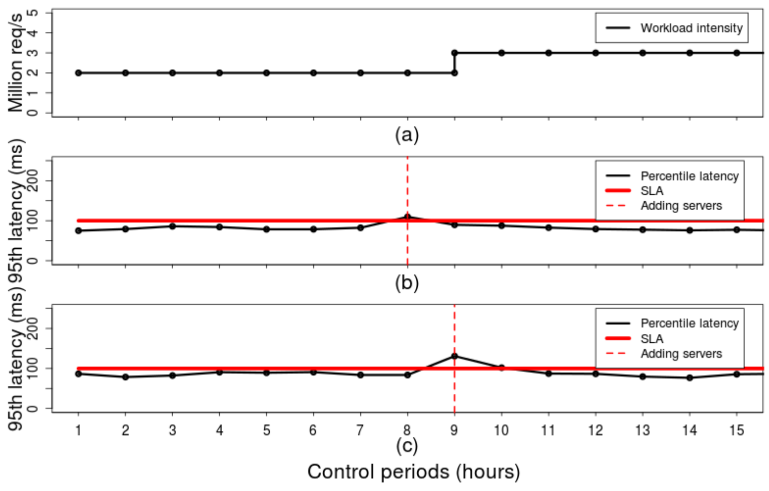

Figure 10.1. Observation of SLA violations during scaling up. (a) denotes a simple increasing workload pattern; (b) scales up the system using a proactive approach; (c) scales up the system using a reactive approach

to demonstrate the idea. Specifically, we assume a perfect prediction of the workload patterns in a prediction based elasticity controller and a perfect monitor of the workload in a feedback based elasticity controller. The elasticity controllers try to add storage instances to cope with the workload increase in Figure 10.1 (a) to keep the low latency of requests defined in SLAs. Figure 10.1 (b) and Figure 10.1 (c) present the latency outcome using naive prediction and feedback based elasticity controller respectively. Several essential observations can be formalized from the experiments.

It is not always the workload that causes SLA violations. Typically, a prediction based elasticity control tries to bring up the capacity of the storage cluster before the actual workload increase. In Figure 10.1 (b), a prediction based controller tries to add instances at control period 8. We observe the SLA violation during this period because of the extra overhead, i,e, data redistribution, imposed on the system when adding storage instances. The violation is caused by the data transfer process, which competes with client requests in terms of servers' CPUs, I/Os, and bandwidths. We solve this problem by presenting a data migration model that controls the data transfer process based on the spare capacity of the server with respect to latency SLA commitment.

Another interesting observation can be seen from Figure 10.1 (c), which simulate the scaling of the system using a feedback approach. It shows that scaling up after seeing a workload peak (at control period 9) is too late. The SLA violation is observed because the newly added instances cannot serve the increased workload immediately. Specifically, proper portion of data needs to be copied to the newly added instances before they can serve the workload. Worse, adding instances at the last moment will even aggravate the SLA violation because of the scaling overhead like in the previous case. Thus, it is necessary to scale the system before the workload changes like using a prediction based approach.

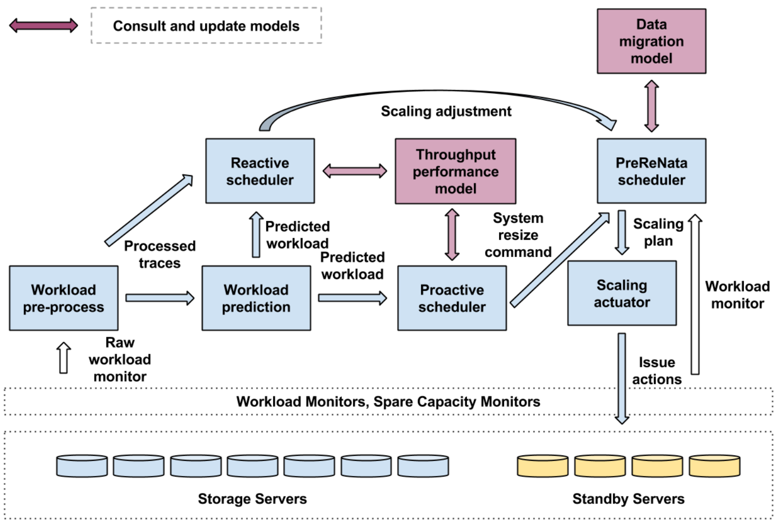

Figure 10.2. ProRenaTa control framework

However, only using prediction based approach is not enough even though we can handle the data transfer overhead using a data migration model. It is because that the prediction is not always accurate. Even using Wikipedia workload [7] where the pattern is very predictable, a small amount of prediction errors are expected. We adjust those errors using a feedback approach. Combing the usage of prediction based approach and the feedback approach yields much better performance in terms of SLA commitments and resource utilization shown in our later evaluations.

10.3 System design

In this section, we present the design of ProRenaTa, which is an elasticity controller for distributed storage systems that combines both reactive and proactive control in order to achieve high system utilization and less SLA violations.

Figure 10.2 is the system architecture of ProRenaTa. It follows the idea of MAPE-K (Monitor, Analysis, Plan, Execute - Knowledge) control loop, but has many customizations and improvements.

10.3.1 Monitor

The arrival rate of reads and writes on each node is monitored and defined as input workload in ProRenaTa. Then, the workload is fed to two modules: workload pre-processing and ProRenaTa scheduler.

Workload pre-process: The workload pre-processing module aggregates the monitored workload in a predefined window interval. We define this interval as smoothing window (SW). The granularity of SW depends on workload patterns. Very large SW will smooth out transient/sudden workload changes while very small SW will cause oscillation in scaling. The size of SW in ProRenaTa can be configured in order to adjust to different workload patterns.

The monitored workload is also fed to ProRenaTa scheduler to estimate the utilization of the system and calculate the spare capacity that can be used to handle scaling overhead. Detailed design of ProRenaTa is explained in Section 10.3.3.

10.3.2 Analysis

Workload prediction : The pre-processed workload is forwarded to the workload prediction module for workload forecasting. The prediction methods will be explained in Section 10.4. Workload prediction is conducted every prediction window (PW). Specifically, at the beginning of each PW, the prediction module forecasts the workload intensity at the end of the current PW. Workload pre-processing provides an aggregated workload intensity at the beginning of each SW. In our setup, SW and PW have the same size and are synchronized. The output of the prediction module is an aggregated workload intensity marked with a time stamp that indicates the deadline for the scaling to match such workload. Workload aggregations and predictions are conducted at a key granularity. The aggregation of the predicted workload intensity on all the keys is the total workload, which is forwarded to the proactive scheduler and the reactive scheduler.

10.3.3 Plan

Proactive scheduler : The Proactive scheduler calculates the number of instances needed in the next PW using the performance model in Section 10.5.1. Then, it sends the number of instances that need to be added/removed to the ProRenaTa scheduler.
Reactive scheduler : The reactive scheduler in ProRenaTa is different from those that reacts on monitored system metrics. Our reactive scheduler is used to correct the inaccurate scaling of the system caused by the inaccuracy of the workload prediction. It takes in the pre-processed workload and the predicted workload. The pre-processed workload represents the current system status while the predicted workload is a forecast of workload in a PW. The reactive scheduler stores the predicted workload at the beginning of each PW and compares the predicted workload with the observed workload at the end of each PW. The difference from the predicted value and the observed value represents the scaling inaccuracy. Using the differences of the predicted value and the observed value as an input signal instead of monitored system metrics guarantees that the reactive scheduler can operate along with the proactive scheduler and not get biased because of the scaling activities from the proactive scheduler. The scaling inaccuracy, i,e, workload difference between prediction and reality, needs to be amended when it exceeds a threshold calculated by the throughput performance model. If scaling adjustments are needed, the number of instances that need to be added/removed is sent to the ProRenaTa scheduler.
ProRenaTa scheduler : The major task for ProRenaTa scheduler is to effectively and efficiently conduct the scaling plan for the future (provided by the proactive scheduler) and the scaling adjustment for now (provided by the reactive scheduler). It is possible that the

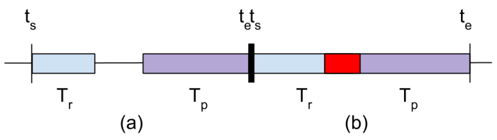

Figure 10.3. Scheduling of reactive and proactive scaling plans

scaling decision from the proactive scheduler and the reactive scheduler are contradictory. ProRenaTa scheduler solves this problem by consulting the data migration model, which quantifies the spare system capacity that can be used to handle the scaling overhead. The data migration model estimates the time needed to finish a scaling decision taking into account the current system status and SLA constraints explained in Section 10.5.2. Assume that the start time of a PW is t_s and the end time of a PW is t_e. The scaling plan from the reactive controller needs to be carried out at t_s while the scaling plan from the proactive controller needs to be finished before t_e. Assume the workload intensity at t_s and t_e is W_s and W_e respectively. We assume a linear evolving model between current workload intensity and the future workload intensity. Thus, workload intensity at time t in a PW can be calculated by $W(t) = \gamma * t + W_s$ where $\gamma = (W_e - W_s)/(t_e - t_s)$. let $Plan_r$ and $Plan_p$ represent the scaling plan from the reactive controller and the proactive controller respectively. Specifically, a $Plan$ is a integer number that denotes the number of instances that needs to be added or removed. Instances are added when $Plan$ is positive, or removed when $Plan$ is negative. Note that the plan of the proactive controller needs to be conducted based on the completion of the reactive controller. It means that the actual plan that needs to be carried out by the proactive plan is $Plan_p' = |Plan_p - Plan_r|$. Given workload intensity and a scaling plan to the data migration model, it needs T_r and T_p to finish the scaling plan from the reactive controller and the proactive controller respectively.

We assume that $T_r < (t_e - t_s) \&\& T_p < (t_e - t_s)$, i,e, the scaling decision by either of the controller alone can be carried out within a PW. This can be guaranteed by understanding the applications' workload patterns and tuning the size of PW accordingly. However, it is not guaranteed that $(T_r + T_p) < (t_e - t_s)$, i,e, the scaling plan from both controllers may not get finished without having an overlapping period within a PW. This interference needs to be prevented because having two controllers being active during an overlapping period violates the assumption, which defines only current system workload influence data migration time, in the data migration model.

In order to achieve the efficient usage of resources, ProRenaTa conduct the scaling plan from the proactive controller at the very last possible moment. In contrast, the scaling plan of the reactive controller needs to be conducted immediately. The scaling process of the two controllers are illustrated in Figure 10.3. Figure 10.3(a) illustrates the case when the reactive and proactive scaling do not interfere with each other. Then, both plans are carried out by the ProRenaTa scheduler. Figure 10.3(b) shows the case when the system cannot support the scaling decisions of both reactive and proactive controller. Then, only the difference of the two plans ($|Plan_r - |Plan_p - Plan_r||$) is carried out. And this plan is regarded as a proactive plan and scheduled to be finished at the end of this PW.

10.3.4 Execute

Scaling actuator : Execution of the scaling plan from ProRenaTa scheduler is carried out by the scaling actuator, which interacts with the underlying storage system. Specifically, it calls add server or remove server APIs exposed by the storage system and controls the data migration among storage servers. The quota used for data migration among servers are calculated by Prerenata scheduler and indicated to the actuator. The actuator limits the quota for data migration on each storage servers using BwMan [8], which is a bandwidth manager that allocates bandwidth quotas to different services running on different ports. In essence, BwMan uses Netem tc tools to control the traffic on each storage server's network interface.

10.3.5 Knowledge

To facilitate the decision making to achieve elasticity in ProRenaTa, there are three knowledge bases. The first one is the throughput model presented in Section 10.5.1, which correlates the server's capability of serving read and write requests under the constraint of SLA latency. The second one is the migration overhead model presented in Section 10.5.2, which quantify the spare capacity that can be used to handle data migration overhead while performing system reconfiguration. The last one is the monitoring, which provides real-time workload information, including composition and intensity, in the system to facilitate the decision making in ProRenaTa.

10.4 Workload Prediction

The prediction of wikipedia workload is a specific problem that does not exactly fit the common prediction techniques found in literature. This is due to the special characteristics of the workload. On the one hand, the workload associated can be highly periodic, which means that the use of the context (past samples), will be effective for making an estimation of the demand. On the other hand the workload time series may have components that are difficult to model with demand peaks that are random. Although the demand peaks might have a periodic component (for instance a week), the fact that the amplitude is random, makes the use of linear combination seperated by week intervals unreliable. The classical methods are based on linear combinations of inputs and old outputs with a random residual noise, and are known as ARIMA, (Autoregressive-Integrated-Moving-Average) or Box-Jenkins models [9].

ARIMA assumes that the future observation depends on values observed a few lags in the past, and a linear combination of a set of inputs. These inputs could be of different origin, and the coefficients of the ARIMA model takes care of both, the importance of the observation to the forecast, and also the scaling in case that the input has different units than the output. However, an important limitation of the ARIMA framework is that it assumes that the random component of the forecasting model is limited to the residual noise. This is a strong limitation because the randomness in the forecasting of workload, is also present in the amplitude/height of the peaks. Other prediction methodologies are based on hybrid methods that combine the ideas from the ARIMA framework, with non-linear methods such as Neural Networks, which do not make hypothesis about the input-output relationships of the functions to be estimated. See for instance [10] The hybrid time series prediction methods, use Neural Netwoks or similar techniques for modeling possible non-linear relationships between the past and input samples and the sample to be predicted. Both methods, the ARIMA and a hybrid method assume that the time series is stationary, and that the random component is a residual error, which is not the case of the workload time series.

10.4.1 Representative workload types

In this subsection we categorize the workload to a few generic representative types. These categories are important because they justify the architecture of the prediction algorithm we propose:
Stable load and cyclic behavior : This behaviour corresponds to an waveform that can be understood as the sum of a few (i.e. 3 or 4) sinusoids plus a random component which can be modeled as random noise. The stable load and cyclic behavior category models keywords that have a clear daily structure, with a repetitive structure of maxima and minima. This category will be dealt with a short time forecast model.
Periodic peaks : This behaviour corresponds to peaks that appear at certain intervals, which need not be harmonics. The defining characteristic is the sudden appearance of the peaks, which run on top of the stable load. The periodic peaks category models keywords that have a structure that depends on a memory longer than a day, and is somehow independent of the near past. This is the case of keywords that for instance, might be associated to a regular event, such as chapters of a TV series that happen certain days of the week.This category will be dealt with a long time forecast model.
Random peaks and background noise : This behaviour corresponds to either to rarely sought keywords which have a random behaviour of low amplitude or keywords that get popular suddenly and for a short time. As this category is inherently unpredictable, unless there is outside information available, we deal with his category using the short term forecasting model, which accounts for a small percentage of the residual error.

10.4.2 Prediction methodology

The forecasting method consists of two modules that take into account the two kind of dependencies in the past: short term for **s**table load, cyclic behavior and background noise and long term for **p**eriodic peaks . Below is a brief summary of each module

- The short term module will make an estimate of the actual value by using a Wiener filter [11] which combines linearly a set of past samples in order to estimate a given value, in this case, the forecasted sample. In order to make the forecast the short term module uses information in a window of several hours. The coefficients of the linear combination are obtained by minimizing the Mean Square Error (MSE) between the forecast and the reference sample. The short term prediction is denoted as $\tilde{x}_{Shrt}[n]$.

The actual structure of the filter is the following:

$$\tilde{x}_{Shrt}[n + N_{FrHr}] = \sum_{i=0}^{L_{Shrt}} w_i x[n - i]$$

where; L_{Shrt}: is the length of the Wiener filter, N_{FrHr}: is the forecasting horizon, $x[n]$: is the n-th sample of the time series, and w_i: is the i-th coefficient of the Wiener filter. Also, as the behaviour of the time series is not stationary, we will recompute the weights of the Wiener filter forecaster when the prediction error increases (MSE) increases for certain length of time [11].

- The long term module $\tilde{x}_{Lng}[n]$ takes into account the fact that there are periodic and sudden rises in the value to be forecasted. These sudden rises depends on the past values by a number of samples much higher than the number of past samples of the short term predictor L_{Shrt}. These rises in demand have an amplitude higher than the rest of the time series, and take a random value with a variance that empirically has been found to be variable in time. These periodicities will be denoted as a set $\{P_0 \ldots P_{N_p}\}$, where P_i indicates the i-th periodicity in the sampling frequency and N_p the total number of periodicities. Empirically, in a window of one month, the periodicities of a given time series were found to be stable in most cases, i.e. although the variance of the peaks changed, the values of P_i were stable. In order to make this forecast, we generated a train of periodic peaks, with an amplitude determined by the mean value taken by the time series at different past periods. This assumes a prediction model with a structure similar to the auto-regressive (AR), which combines linearly past values at given lags. The structure of this filter is the following:

$$\tilde{x}_{Lng}[n + N_{FrHr}] = \sum_{i=0}^{N_p} \sum_{j=0}^{L_j} h_{i,j} x[n - jP_i]$$

  where, N_{FrHr}: is the forecasting horizon, N_p: is the total number of periodicities, L_j: is the number of weighted samples of the i-th periodicity, $h_{i,j}$: is the weight assigned to each sample used in the estimation, $x[n]$: is the n-th sample of the time series. Note we will not use the moving average (MA) component, which presupposes external inputs. A model that takes into account external features, should incorporate a MA component.

- The final estimation is as follows:

$$\tilde{x}[n + N_{FrHr}] = \begin{cases} \tilde{x}_{Lng}[n + N_{FrHr}] & \text{if n+}N_{FrHr} = k_0 P_i \\ \tilde{x}_{Shrt}[n + N_{FrHr}] & \text{if n+}N_{FrHr} \neq k_0 P_i \end{cases}$$

where the decision on the forecast to use is based on testing if $n + N_{FrHr}$ is a multiple of any of the periodicities P_i . Specific details on the implementation of each of the predictors can be found in the Appendix 10.8.

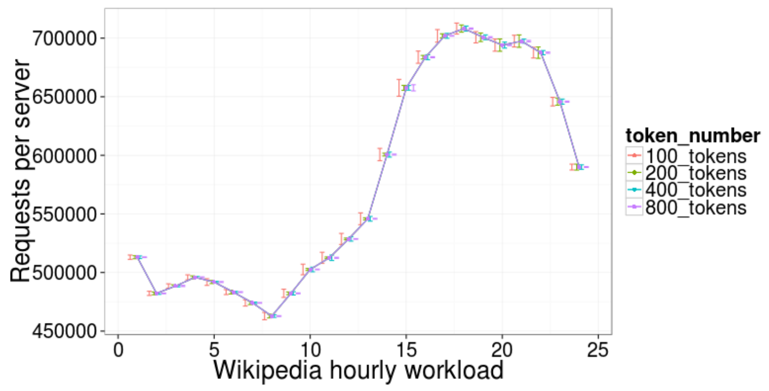

Figure 10.4. Standard deviation of load on 20 servers running a 24 hours Wikipedia workload trace. With larger number of virtual tokens assigned to each server, the standard deviation of load among servers decreases.

10.5 Controller models

In this section, we present the models that we use in ProRenaTa controllers. In general, there are two models that are constantly consulted and updated in every control period. One is a throughput performance model, which defines the correlation between the throughput of the system and the number of participating instances. The other one is a data migration model, which quantifies the available capacity that can be used to handle the scaling overhead under the current system load.

10.5.1 Throughput performance model

The throughput performance model determines the minimum number of servers that is needed to meet the SLA requirements with respect to a specific workload. In order to build a suitable performance model for a distributed storage system, the first thing that needs to be understood is how requests are distributed to servers.

Load balance in distributed storage systems

We target a kind of distributed storage system that is organized using DHTs as introduced in Section 10.2. We assume that virtual tokens are implemented in the target distributed storage system. Enabling virtual tokens allows a physical server to host multiple discrete virtual tokens in DHT namespace and storing the corresponding portions of data. The number of virtual tokens on a physical server is proportional to the server's capability. The virtual tokens assigned to a physical server are randomly selected in our case. Figure 10.4 shows that with sufficient number of virtual tokens, requests tend to evenly distributed among physical servers with different workload intensities replayed using a 24 hour Wikipedia access trace. Thus, we can derive a performance model for the distributed storage system by modeling its individual servers. Specifically, under the assumption of evenly distributed workload, if

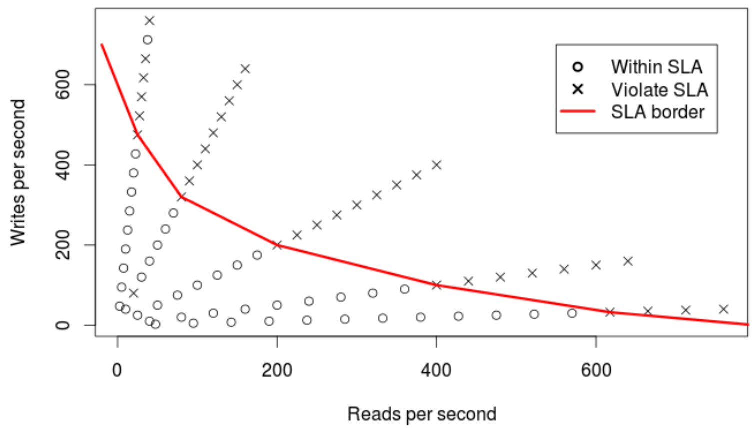

Figure 10.5. Throughput performance model

any server in the system does not violate the SLA constraints, the system as a whole does not violate the SLA constraints proposed and argued in [12].

Performance model based on throughput

Figure 10.5 shows an off-line trained throughput-based performance model for a physical server. Different models can be build for different server flavors using the same profiling method. The workload is represented by the request rate of read and write operations. Under a specified SLA latency constraint, a server can be in 2 states: satisfy SLA (under the SLA border) or violate SLA (beyond the SLA border). We would like to arrange servers to be utilized just under the SLA border. This translates to having a high resource utilization while guaranteeing the SLA requirements. The performance model takes a specific workload intensity as the input and outputs the minimum number of instances that is needed to handle the workload under SLA. It is calculated by finding the minimum number of servers that results in the load on each server ($Workload/NumberOfServers$) closest to and under the SLA border in Figure 10.5. In the real experiment, we have setup a small margin for over-provisioning. This can not only better achieve SLA, but also allows more spare capacity for data transfer during scaling up/down. This margin is set as 2 servers in our experiment and can be configured differently case to case to tradeoff the scaling speed and SLA commitment with resource utilization.

10.5.2 Data migration model

The data migration model in ProRenaTa monitors the load in the system and outputs the maximum data transfer rate that can be used for scaling activities without compromising the SLA. By using the data migration model, ProRenaTa is able to obtain the time that is

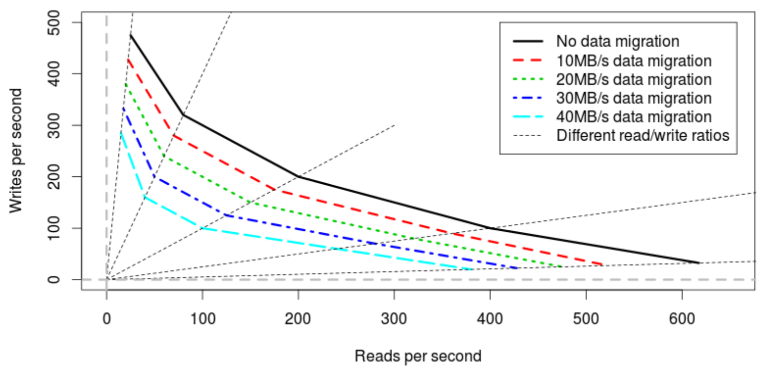

Figure 10.6. Data migration model under throughput and SLA constraints

needed to conduct a scaling plan, i,e, adding/removing n instances, under current system status and SLA constraint. A detailed analytical model can be found in Appendix 10.8

Statistical model

In this section, we describe the profiling of a storage instance under three parameters: read request rate, write request rate and data migration speed. The data migration is manually triggered in the system by adding new storage instances. We observe that during data migration, network bandwidth is the major resource that compete between data transfer and client request handling in storage servers. Thus, the bandwidth used for data migration is controlled as a parameter. Under the constraint of SLA latency, we have multiple SLA borders under different data transfer rate shown in Figure 10.6. Using this statistical model, given current workload and servers in the system, we are able to find the spare capacity that can be used for data migration. Specifically, a given total workload consisting of read and write request rate is uniformly mapped to each server and expressed as a data point in Figure 10.6. The closest border below this data point indicates the data migration speed that can be offered.

10.6 Evaluation

In this section, we present the evaluation of ProRenaTa elasticity controller using a workload synthesized from Wikipedia access logs during 2009/03/08 to 2009/03/22. The access traces are available here [7]. We first present the setup of the storage system (GlobLease) and the implementation of the workload generator. Then, we present the evaluation results of ProRenaTa and compare its SLA commitments and resource utilization with feedback and prediction based elasticity controller.

Table 10.1. GlobLease and workload generator setups

Specifications	GlobLease VMs	Workload VMs
Instance Type	m1.medium	m1.xlarge
CPUs	Intel Xeon 2.8 GHz*2	Intel Xeon 2.0 GHz*8
Memory	4 GiB	16 GiB
OS	Ubuntu 14.04	Ubuntu 14.04
Instance Number	5 to 20	5 to 10

10.6.1 Implementation

Deployment of the storage system

GlobLease [6] is deployed on a private OpenStack Cloud platform hosted at out university (KTH). Homogeneous virtual machine instance types are used in the experiment for simplicity. It can be easily extended to heterogeneous scheduling by profiling capabilities of different instances types using the methodology described in Section 10.5.1. Table 10.1 presents the virtual machine setups for GlobLease and the workload generator.

Workload generator

We implemented a workload generator in JAVA that emulates workloads in different granularities of requests per second. To setup the workload, a couple of configuration parameters are fed to the workload generator including the workload traces, the number of client threads, and the setup of GlobLease.
Construction of the workload from raw Wikipedia access logs. The access logs from Wikepedia provides the number of accesses to each page every 1 hour. The first step to prepare a workload trace is to remove the noise in accesses. We removed non-meaningful pages such as "Main_Page", "Special:Search", "Special:Random", etc from the logs, which contributes to a large portion of accesses and skews the workload pattern. Then, we chose the 5% most accessed pages in the trace and abandoned the rest. There are two reasons for this choice: First, these 5% popular keys constructs nearly 80% of the total workload. Second, access patterns of these top 5% keys are more interesting to investigate while the remaining 95% of the keys are mostly with 1 or 2 accesses per hour and very likely remain inactive in the following hours. After fixing the composition of the workload, since Wikipedia logs only provide page views, i,e, read accesses, we randomly chose 5% of these accesses and transformed them as write operations. Then, the workload file is shuffled and provided to the workload generator. We assume that the arrivals of clients during every hour follow a Poisson distribution. This assumption is implemented in preparing the workload file by randomly placing accesses with a Poisson arrival intensity smoothed with a 1 minute window. Specifically, 1 hour workload has 60 such windows and the workload intensities

Table 10.2. Wikipedia Workload Parameters

Concurrent clients	50
Request per second	roughly 3000 to 7000
Size of the namespace	around 100,000 keys
Size of the value	10 KB

of these 60 windows form a Poisson distribution. When the workload generator reads the workload file, it reads the whole accesses in 1 window and averages the request rate in this window, then plays them against the storage system. We do not have the information regarding the size of each page from the logs, thus, we assume that the size of each page is 10 KB. We observe that the prepared workload is not able to saturate GlobLease if the trace is played in 1 hour. So, we intensify the workload by playing the trace in 10 minutes instead.

The number of client threads defines the number of concurrent requests to GlobLease in a short interval. We configured the concurrent client threads as 50 in our experiment. The size of the interval is calculated as the ratio of the number of client threads over the workload intensity.

The setup of GlobLease provides the addresses of the storage nodes to the workload generator. Note that the setup of GlobLease is dynamically adjusted by the elasticity controllers during the experiment. Our workload generator also implements a load balancer that is aware of the setup changes from a programmed notification message sent by the elasticity controllers (actuators).

Table 10.2 summaries the parameters configured in the workload generator.

Handling data transfer

Like most distributed storage systems, GlobLease implements data transfer from nodes to nodes in a greedy fashion, which puts a lot of stress on the available network bandwidth. In order to guarantee the SLA of the system, we control the network resources used for data transfer using BwMan [8]. BwMan is a previous work in our group that arbitrate network resources between user-oriented workload and system maintenance workload, data transfer in this case, under the constraint of SLAs. The amount of available network resources allocated for data transfer is calculated using the data migration model in Section 10.5.2.

10.6.2 Evaluation results

We compare ProRenaTa with two baseline approaches: feedback control and prediction-based control.

The feedback controller is built using the throughput model described in section 10.5.1. Dynamic reconfiguration of the system is performed at the beginning of each control window to match the averaged workload collected during the previous control window.

The prediction-based controller uses the prediction algorithm described in section 10.4. System reconfiguration is carried out at the beginning of the control window based on

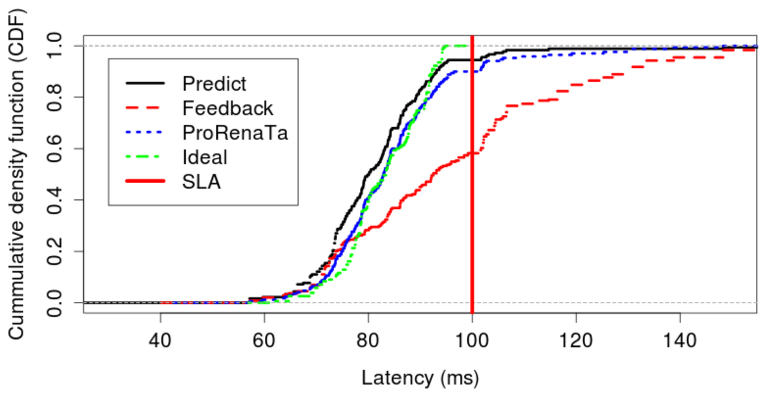

Figure 10.7. Aggregated CDF of latency for different approaches

the predicted workload intensity for the next control period. Specifically, if the workload increase warrants addition of servers, it is performed at the beginning of the current window. However, if the workload decreases, the removal of servers are performed at the beginning of the next window to ensure SLA. Conflicts may happen at the beginning of some windows because of a workload decrease followed by a workload increase. This is solved by simply adding/merging the scaling decisions.

ProRenaTa combines both feedback control and prediction-based control but with more sophisticated modeling and scheduling. Prediction-based control gives ProRenaTa enough time to schedule system reconfiguration under the constraint of SLAs. The scaling is carried out at the last possible moment in a control window under the constraint of SLA provided by the scaling overhead model described in Section 10.5.2. This model guarantees ProRenaTa with less SLA violations and better resource utilization. In the meantime, feedback control is used to adjust the prediction error at the beginning of each control window. The scheduling of predicted actions and feedback actions is handled by ProRenaTa scheduler.

In addition, we also compare ProRenaTa with an ideal case. The ideal case is implemented using a theoretically *perfect elasticity controller*, which knows the future workload, i,e, predicts the workload perfectly. The ideal also uses ProRenaTa scheduler to scale the cluster. So, comparing to the prediction based approach, the ideal case not only uses more accurate prediction results but also uses better scheduling, i,e, the ProRenaTa scheduler.

Performance overview

In this section, we present the evaluation results using the aforementioned 4 approaches with the Wikipedia workload trace from 2009/03/08 to 2009/03/22. We focus on the 95th percentile latency of requests calculated from each hour and the CPU utilization monitored every second.

SLA commitment. Figure 10.7 presents the cumulative distribution of 95 percentile latency by running the simulated Wikipedia workload from 2009/03/08 to 2009/03/22. The vertical

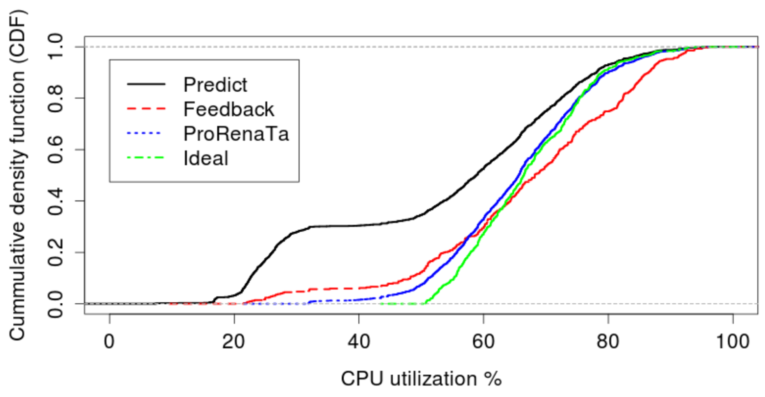

Figure 10.8. Aggregated CDF of CPU utilization for different approaches

red line demonstrates the SLA latency configured in each elasticity controller.

We observe that the feedback approach has the most SLA violations. This is because the algorithm reacts only when it observes the actual workload changes, which is usually too late for a stateful system to scale. This effect is more obvious when the workload is increasing. The scaling overhead along with the workload increases lead to a large percent of high latency requests.

ProRenaTa and the prediction-based approach achieve nearly the same SLA commitments as shown in Figure 10.7. This is because we have an accurate workload prediction algorithm presented in 10.4. Also, the prediction-based algorithm tries to reconfigure the system before the actual workload comes, leaving the system enough time and resources to scale. However, we shown in the next section that the prediction-based approach does not efficiently use the resources, i,e, CPU, which results in more provision cost.

CPU utilization. Figure 10.8 shows the cumulative distribution of the aggregated CPU utilization on all the storage servers by running the two weeks simulated Wikipedia workload.

Figure 10.8 shows that some servers in the feedback approach are under utilized (20% to 50%), which leads to high provision cost, and some are saturated (above 80%), which violates SLA latency. This CPU utilization pattern is caused by the nature of reactive approach, i,e, the system only reacts to the changing workload when it is observed. In the case of workload increase, the increased workloads usually saturate the system before it reacts. Worse, by adding storage servers at this point, the data migration overhead among servers aggravate the saturation. This scenario contributes to the saturated CPU utilization in the figure. On the other hand, in the case of workload decrease, the excess servers are removed only in the beginning of the next control period. This leads to the CPU under utilization in some scenarios.

It is shown in figure 10.8 that a large portion of servers remain under utilized when using the prediction based elasticity control. This is because of the prediction-based control algorithm. Specifically, in order to guarantee SLA, in the case of workload increase, servers

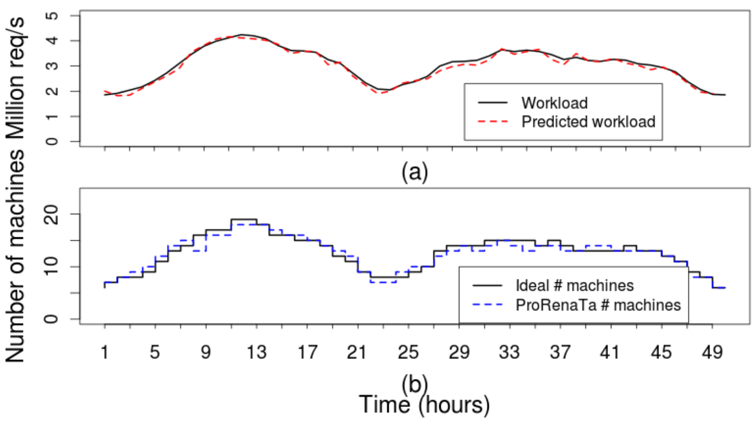

Figure 10.9. Actual workload and predicted workload and Machines dynamically allocated according to workload changes

are added in the previous control period while in the case of workload decrease, servers are removed in the next control period. Note that the CPU statistics are collected every second on all the storage servers. Thus, the provisioning margin between control periods contributes to the large portion of under utilized CPUs.

In comparison with the feedback or prediction based approach, ProRenaTa is smarter in controlling the system. Figure 10.8 shows that most servers in ProRenaTa have a CPU utilization from 50% to 80%, which results in a reasonable latency that satisfies the SLA. Under/over utilized CPUs are prevented by a feedback mechanism that corrects the prediction errors. Furthermore, there is much less over provision margins observed in the prediction based approach because of the data migration model. ProRenaTa assesses and predicts system spare capacity in the coming control period and schedules system reconfigurations (scale up/down) to an optimized time (not in the beginning or the end of the control period). This optimized scheduling is calculated based on the data migration overhead of the scaling plan as explained in Section 10.5.2. All these mechanisms in ProRenaTa leads to an optimized resource utilization with respect to SLA commitment.

Detailed performance analysis

In the previous section, we presented the aggregated statistics about SLA commitment and CPU utilization by playing a 2 weeks Wikipedia access trace using four different approaches. In this section, we zoom in the experiment by looking at the collected data during 48 hours. This 48 hours time series provides more insights into understanding the circumstances that different approaches tend to violate the SLA latency.

Workload pattern. Figure 10.9 (a) shows the workload pattern and intensity during 48 hours. The solid line presents the actual workload from the trace and the dashed line depicts the predicted workload intensity by our prediction algorithm presented in Section 10.4.

Machines allocated. Figure 10.9 (b) demonstrates the number of server instances allocated

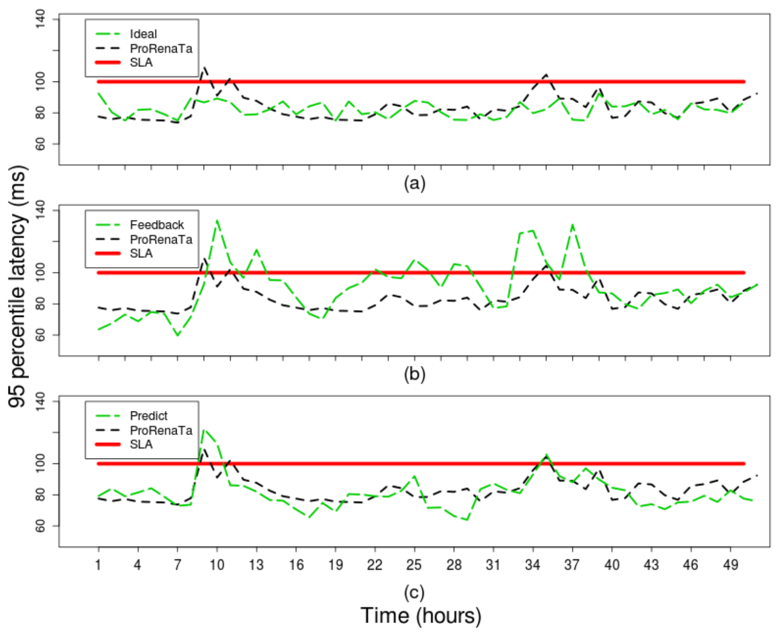

Figure 10.10. SLA commitment comparing ideal, feedback and predict approaches with ProRenaTa

to serve the workload in Figure 10.9 (a) in the ideal case and using ProRenaTa elasticity controller. The ideal provisioning is simulated by knowing the actual workload trace beforehand and feeding it to the ProRenaTa scheduler, which generates an optimized scaling plan in terms of the timing of scaling that takes into account the scaling overhead.

SLA commitment. Figure 10.10 presents the comparison of SLA achievement using the ideal approach (a), the feedback approach (b) and the prediction based approach (c) compared to ProRenaTa under the workload described in Figure 10.9 (a). Compared to the ideal case, ProRenaTa violates SLA when the workload increases sharply. The SLA commitments are met in the next control period. The feedback approach on the other hand causes severe SLA violation when the workload increases. ProRenaTa takes into account the scaling overhead and takes actions in advance with the help of workload prediction, which gives it advantages in reducing the violation in terms of extend and period. In comparison with the prediction based approach, both approaches achieve more or less the same SLA commitment because of the pre-allocation of servers before the workload occurs. However, it is shown in Figure 10.8 that the prediction based approach cannot use CPU resource efficiently.

Utility Measure

An efficient controller must be able to achieve high resource utilization and at the same time guarantee SLA commitments. In order to measure the efficiency of different approaches, we define a simple utility measure that allows us to compose resource utilization and la-

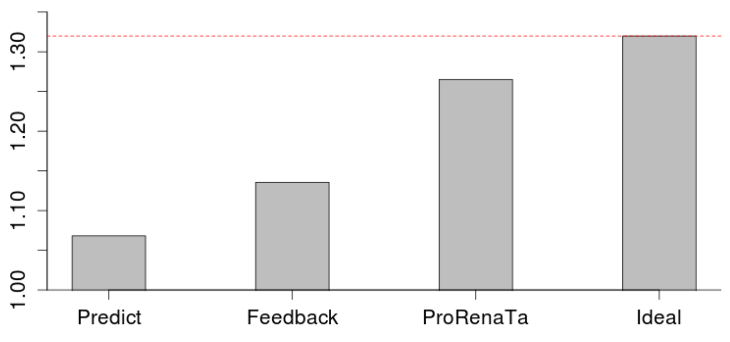

Figure 10.11. Utility for different approaches

tency SLA achievements as a simple function. We define the utility measure as the ratio of normalized CPU utilization over normalized latencies. The efficiency of the controller is directly proportional to the utility measure. We analyze the results obtained by running a 3 weeks Wikipedia workload trace using different auto-scaling controllers. Figure 10.11 shows the utility measure for 4 different scaling approaches. An ideal controller is the best case because it does perfect prediction and also perfect data migration. Predictive approach achieves the lowest utility measure because of the scaling margins that causes resource under-utilization as explained in Section 10.6.2. While the feedback based approach increases both latency and CPU utilization, the increase in CPU dominates over latency and hence results in a better utility measure. ProRenaTa achieves a utility measure closest to the ideal case because it improves CPU utilization while achieving low latencies.

10.7 Related work

There are numerous works done in the field of auto-scaling recently under the context of Cloud computing. Broadly speaking, they can be characterized as scaling stateless services [13, 14, 15] and stateful service [12, 16, 17, 18]. We characterize the approaches into two categories: reactive and proactive. Typical methods used for auto-scaling are threshold-based rules, reinforcement learning or Q-learning (RL), queuing theory, control theory and time series analysis.

The representative systems that use threshold-based rules to scale a service are Amazon Cloud Watch [19] and RightScale [20]. Simply speaking, this approach defines a set of thresholds or rules in advance. Violating the thresholds or rules to some extent will trigger the action of scaling. Threshold-based rule is a typical implementation of reactive scaling.

Reinforcement learning are usually used to understand the application behaviors by building empirical models. Simon [21] presents an elasticity controller that integrates several empirical models and switches among them to obtain better performance predictions.

The elasticity controller built in [18] uses analytical modeling and machine-learning. They argued that by combining both approaches, it results in better controller accuracy.

[22] uses the queueing theory to model a Cloud service and estimates the incoming load. It builds proactive controllers based on the assumption of a queueing model. It presents an elasticity controller that incorporates a reactive controller for scale up and proactive controllers for scale down.

Recent influential works that use control theory to achieve elasticity are [12, 16, 17]. The reactive module in ProRenaTa uses similar technique to achieve auto-scaling as the ones applied in Scads director [12] and ElastMan [16]. Specifically, both approaches build performance models on system throughput using model predictive control. Scads director tries to minimize the data migration overhead associated with the scaling by arranging data into small data bins. However, this only alleviate the SLA violations. Lim [17] uses CPU utilization as the monitored metrics in a classic feedback loop to achieve auto-scaling. A data migration controller is also modeled in this work. However, it is only used to tradeoff the SLA violation with the scaling speed.

Recent approaches using time-series analysis to achieve auto-scaling are [23, 15]. [23] adapts second order ARMA for workload forecasting under the World Cup 98 workload. [15] proves that it is accurate to use wavelets to provide a medium-term resource demand prediction. With the help of the prediction results, VMs can be spawned/migrated before it is needed in order to avoid SLA violations. [13] uses on-line resource demand prediction with prediction errors corrected.

ProRenaTa differs from the previous approaches in two aspects. First, most of the previous approaches either use reactive or proactive scaling techniques, ProRenaTa combines both approaches. Reactive controller gives ProRenaTa better scale accucacy while proactive controller provides ProRenaTa enough time to handle the scaling overhead (data migration). The complimentary nature of both approaches provide ProRenaTa with better SLA guarantees and higher resource utilizations. Second, to our best knowledge, when scaling a stateful system, e,g, a storage system, none of the previous systems explicitly model the cost of data migration when scaling. Specifically, ProRenaTa assesses the scaling cost under the scaling goal with continuous monitoring of the spare capacity in the system. In essense, ProRenaTa employs time-series analysis to realize a proactive scaling controller. This is because of the workload characteristics discussed in Section 10.4 and analyzed in [24]. A reactive module is applied to correct prediction errors, which further guarantees the SLA and boosts the utilization of resources.

10.8 Conclusion

In this paper, we investigate the efficiency of an elasticity controller that combines both proactive and reactive approaches for auto-scaling a distributed storage system. We show the limitations of using proactive or reactive approach in isolation to scale a stateful system. We design ProRenaTa that combines both proactive and reactive approaches. ProRenaTa improves the classic prediction based scaling approach by taking into account the scaling overhead, i.e., data/state migration. Moreover, the reactive controller helps ProRenaTa to achieve better scaling accuracy, i.e., better resource utilization and less SLA violations, without causing interference with the scaling activities scheduled by the proactive controller. Our results indicate that ProRenaTa outperforms the state of the art approaches by guaranteeing a high level of SLA commitments while also improving the overall resource utilization.

Bibliography

[1] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[2] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[3] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

[4] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[5] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

[6] Ying Liu, Xiaxi Li, and Vladimir Vlassov. Globlease: A globally consistent and elastic storage system using leases. `http://dx.doi.org/10.13140/2.1.2183.7763`. (To appear in) Proceedings of the 2014 IEEE International Conference on Parallel and Distributed Systems (ICPADS '14).

[7] Wikipedia traffic statistics v2. `http://aws.amazon.com/datasets/4182`.

[8] Ying Liu, Vamis Xhagjika, Vladimir Vlassov, and Ahmad Al Shishtawy. Bwman: Bandwidth manager for elastic services in the cloud. In *Parallel and Distributed*

Processing with Applications (ISPA), 2014 IEEE International Symposium on, pages 217–224, Aug 2014.

[9] George EP Box, Gwilym M Jenkins, and Gregory C Reinsel. *Time series analysis: forecasting and control*. John Wiley & Sons, 2013.

[10] Timothy Masters. *Neural, novel and hybrid algorithms for time series prediction*. John Wiley & Sons, Inc., 1995.

[11] Thomas Kailath, Ali H Sayed, and Babak Hassibi. Linear estimation. 2000.

[12] Beth Trushkowsky, Peter Bodík, Armando Fox, Michael J. Franklin, Michael I. Jordan, and David A. Patterson. The scads director: Scaling a distributed storage system under stringent performance requirements. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association.

[13] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: Elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 5:1–5:14, New York, NY, USA, 2011. ACM.

[14] Jing Jiang, Jie Lu, Guangquan Zhang, and Guodong Long. Optimal cloud resource auto-scaling for web applications. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 58–65, May 2013.

[15] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *Proc. of the USENIX International Conference on Automated Computing (ICAC'13). San Jose, CA*, 2013.

[16] Ahmad Al-Shishtawy and Vladimir Vlassov. Elastman: Elasticity manager for elastic key-value stores in the cloud. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, CAC '13, pages 7:1–7:10, New York, NY, USA, 2013. ACM.

[17] Harold C. Lim, Shivnath Babu, and Jeffrey S. Chase. Automated control for elastic storage. In *Proceedings of the 7th International Conference on Autonomic Computing*, ICAC '10, pages 1–10, New York, NY, USA, 2010. ACM.

[18] Diego Didona, Paolo Romano, Sebastiano Peluso, and Francesco Quaglia. Transactional auto scaler: Elastic scaling of in-memory transactional data grids. In *Proceedings of the 9th International Conference on Autonomic Computing*, ICAC '12, pages 125–134, New York, NY, USA, 2012. ACM.

[19] Amazon cloudwatch. `http://aws.amazon.com/cloudwatch/`.

[20] Right scale. `http://www.rightscale.com/`.

[21] Simon J. Malkowski, Markus Hedwig, Jack Li, Calton Pu, and Dirk Neumann. Automated control for elastic n-tier workloads based on empirical modeling. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, pages 131–140, New York, NY, USA, 2011. ACM.

[22] A. Ali-Eldin, J. Tordsson, and E. Elmroth. An adaptive hybrid elasticity controller for cloud infrastructures. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pages 204–212, April 2012.

[23] N. Roy, A. Dubey, and A. Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 500–507, July 2011.

[24] Ahmed Ali Eldin, Ali Rezaie, Amardeep Mehta, Stanislav Razroev, Sara Sjostedt de Luna, Oleg Seleznjev, Johan Tordsson, and Erik Elmroth. How will your workload look like in 6 years? analyzing wikimedia's workload. In *Proceedings of the 2014 IEEE International Conference on Cloud Engineering*, IC2E '14, pages 349–354, Washington, DC, USA, 2014. IEEE Computer Society.

Predictor Implementation

Short term forecast the short term component is initially computed using as data the estimation segment, that is the same initial segment used in order to determine the set of periods P_i of the long term forecaster. On the forecasting component of the data, the values of the weights w_i of the Wiener filter are updated when the forecasting error increases for a certain length of time. This assumes a time series with a statistical properties that vary with time. The procedure for determining the update policy of the Wiener filter is the following: first the forecasting error at a given moment

$$Error[n] = |\tilde{x}[n] - x[n]|^2$$

note that this is computed taking into account a delay equal to the forecasting horizon N_{FrHr}, that is $\tilde{x}[n]$ is compute form the set of samples: $\{x[n - N_{FrHr}] \ldots x[n - N_{FrHr} - L_{Shrt}]\}$. In order to decide when to update the coefficients of the Wiener filter, we compute a long term MSE and a short term MSE by means of an exponential window. Computing the mean value by means of an exponential window is justified because it gives more weight to the near past. The actual computation of the MSE at moment n, weights the instantaneous error $Error[n]$, with the preceding MSE at $n - 1$. The decision variable $Des[n]$ is the ratio between the long term MSE at moment n $MSE_{lng}[n]$ and the the short term MSE at moment n $MSE_{srt}[n]$:

$$MSE_{lng}[n] = (1 - \alpha_{lng})Error[n] + \alpha_{lng}MSE_{lng}[n-1]$$

$$MSE_{srt}[n] = (1 - \alpha_{shrt})Error[n] + \alpha_{srt}MSE_{shrt}[n-1]$$

where α is the memory parameter of the exponential window, with $0 < \alpha < 1$ and for our experiment α_{lng} was set to 0.98, which means that the sample $n - 100$ is given 10 times less weight that the actual sample and α_{shrt} was set to 0.9, which means that the sample $n - 20$ is given 10 times less weight that the actual sample. The desition value is defined as:

$$Des[n] = MSE_{srt}[n]/max(1, MSE_{srt}[n])$$

if $Des[n] > Thrhld$ it is assumed that the statistics of the time series has changed and a new set of coefficients w_i are computed for the Wiener filter. The training data sample consists of the near past and are taken as $\{x[n] \ldots x[n - MemL_{Shrt}]\}$. For our experiments we took as threshold $Thrhld = 10$ and $Mem = 10$. Empirically we have found that the performance does not change much when these values are slightly perturbed. Note that the max() operator in the denominator of the expression that computes $Des[n]$ prevents a division by zero in the case of keywords with low activity.

Long term forecast In order to compute the parameters P_i of the term $\tilde{x}_{Lng}[n]$ we reserved a first segment (estimation segment) of the time series and we computed the auto-correlation function on this segment. The autocorrelation function measures the similarity of the time series to itself as a function of temporal shifts and the maxima of the autocorrelation function indicates it's periodic components denoted by P_i. These long term periodicities are computed from the lags of the positive side of the auto-correlation function with a value above a threshold. Also, we selected periodicities corresponding to periods greater than 24 hours. The amplitude threshold was defined as a percentage of the auto correlation at lag zero (i.e. the energy of the time series). Empirically we found that the 0.9 percent of the energy allowed to model the periods of interest. The weighting value $h_{i,j}$ was taken as $1/L_j$ which gives the same weight to each of the periods used for the estimation. The number of weighted periods L_j was selected to be two, which empirically gave good results.

Analytical model for Data Migration

We consider a distributed storage system that runs in a Cloud environment and uses ProReNaTa to achieve elasticity while respecting the latency SLA. The storage system is organized using DHT and virtual tokens are implemented. Using virtual tokens in a distributed storage system provides it with the following properties:

- The amount of data stored in each physical server is proportional to its capability.
- The amount of workload distributed to each physical server is proportional to its capability.
- If enough bandwidth is given, the data migration speed from/to a instance is proportional to its capability.

At time t, Let D be the amount of data stored in the storage system. We consider that the amount of data is very large so that reads and writes in the storage system during a small period do not significantly influence the data amount stored in the system. We assume that, at time t, there are N storage instances. For simplicity, here we consider that the storage instances are homogeneous. Let C represents the capability of each storage instance. Specifically, the maximum read capacity in requests per second and write capacity in requests per second is represented by $\alpha * C$ and $\beta * C$ respectively under the SLA latency constraint. The value of α and β can be obtained from our performance model.

Let L denotes the current workload in the system. Therefore, $\alpha' * L$ are read requests and $\beta' * L$ are write requests. Under the assumption of uniform workload distribution, the read and write workload served by each physical server is $\alpha' * L/N$ and $\beta' * L/N$ respectively. We define function f to be our data migration model. It outputs the maximum data migration rate that can be obtained under the current system load without compromising SLAs. Thus, function f depends on system load ($\alpha' * L/N, \beta' * L/N$), server capability ($\alpha * C, \beta * C$) and the SLA ($SLA_{latency}$).

We assume the predicted workload is $L_{predicted}$. According to the performance model introduced in the previous section, we know that a scaling plan in terms of adding or removing instances can be given. Let us consider a scaling plan that needs to add or remove n instances. When adding instances, n is a positive value and when removing instances, n is a negative value.

First, we calculate the amount of data that needs to be reallocated. It can be expressed by the difference of the amount of data hosted on each storage instance before scaling and after scaling. Since all the storage instances are homogeneous, the amount of data stored in each storage instance before scaling is D/N. And the amount of data stored in each storage instance after scaling is $D/(N+n)$. Thus, the amount of data that needs to be migrated can be calculated as $|D/N - D/(N+n)| * N$, where $|D/N - D/(N+n)|$ is for a single instance. Given the maximum speed that can be used for data migration ($f()$) on each instance, the time needed to carry out the scaling plan can be calculated.

$Time_{scale} = \frac{|D/N - D/(N+n)|}{f(\alpha * C, \beta * C, \alpha' * L/N, \beta' * L/N, SLA_{latency})}$

The workload intensity during the scaling in the above formula is assumed to be constant L. However, it is not the case in the real system. The evolving pattern of the workload during scaling is application specific and sometimes hard to predict. For simplicity, we assume a linear evolving pattern of the workload between before scaling and after scaling. However, any workload evolving pattern during scaling can be given to the data migration controller with little adjustment. Remind that the foreseeing workload is $L_{predicted}$ and the current workload is L. If a linear changing of workload is assumed from L to $L_{predicted}$, using basic calculus, it is easy to know that the effective workload during the scaling time is the average workload $L_{effective} = (L + L_{predicted})/2$. The time needed to conduct a scaling plan can be calculated using the above formula with the effective workload $L_{effective}$.

We can obtain α and β from the performance model for any instance flavor. α' and β' are obtained from workload monitors. Then, the problem left is to fine a proper function f that defines the data migration speed under certain system setup and workload condition with respect to SLA constraint. The function f is obtained using the statistical model explained in section 10.5.2.