Co-funded by the
Erasmus+ Programme
of the European Union

Towards Elastic High-Performance Geo-Distributed Storage in the Cloud

YING LIU

School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden 2016
and
Institute of Information and Communication Technologies,
Electronics and Applied Mathematics
Université catholique de Louvain
Louvain-la-Neuve, Belgium 2016

KTH School of Information and
Communication Technology
SE-164 40 Kista
SWEDEN

TRITA-ICT 2016:24
ISBN 978-91-7729-092-6

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av doktorsexamen i Informations- och kommunikationsteknik måndagen den 3 oktober 2016 klockan 10:00 i Sal C, Electrum, Kungl Tekniska högskolan, Kistagången 16, Kista.

Tryck: Universitetsservice US AB

To my beloved Yi

and my parents Xianchun and Aiping.

Abstract

In this thesis, we have investigated two aspects towards improving the performance of distributed storage systems. In one direction, we present techniques and algorithms to reduce request latency of distributed storage services that are deployed geographically. In another direction, we propose and design elasticity controllers to maintain predictable/stable performance of distributed storage systems under dynamic workloads and platform uncertainties.

On the research path towards the first direction, we have proposed a lease-based data consistency algorithm that allows a distributed storage system to serve read-dominant workload efficiently in a global scale. Essentially, leases are used to assert the correctness/freshness of data within a time interval. The leasing algorithm allows replicas with valid leases to serve read requests locally. As a result, most of the read requests are served with little latency. Furthermore, leases' time-based assertion guarantees the liveness of the consistency algorithm even in the presence of failures. Then, we have investigated the efficiency of quorum-based data consistency algorithms when deployed globally. We have proposed MeteorShower framework, which is based on replicated logs and loosely synchronized clocks, to augment quorum-based data consistency algorithms. In core, MeteorShower allows the algorithms to maintain data consistency using slightly old replica values provided in the replicated logs. As a result, the quorum-based data consistency algorithms no longer need to query for updates from remote replicas, which significantly reduces request latency. Based on similar insights, we build a transaction framework, Catenae, for geo-distributed data stores. It employs replicated logs to distribute transactions and aggregate the execution results. Transactions are distributed in order to accomplish a speculative execution phase, which is coordinated using a transaction chain algorithm. The algorithm orders transactions based on their execution speed with respect to each data partition, which maximizes the concurrency and determinism of transaction executions. As a result, most of the execution results on replicas in different data centers are consistent when examined in a validation phase. This allows Catenae to commit a serializable read-write transaction experiencing only a single inter-DC RTT delay in most of the cases.

Following the second research path, we examine and control the factors that cause performance degradation when scaling a distributed storage system. First, we have proposed BwMan, which is a model-based network bandwidth manager. It alleviates performance degradation caused by data migration activities when scaling a distributed storage system. It is achieved by dynamically throttling the network bandwidth allocated to these activities. As a result, the performance of the storage system is more predictable/stable, i.e., satisfying latency-based service level objective (SLO), even in the presence of data migration. As a step forward, we have systematically modeled the impact of data migrations. Using this model, we have built an elasticity controller, namely, ProRenaTa, which combines proactive and reactive controls to achieve better control accuracy. With the help of workload prediction and the data migration model, ProRenaTa is able to calculate the best possible scaling plan to resize a distributed storage system under the constraint of achieving scaling deadlines, reducing latency SLO violations and minimizing VM provisioning cost. As a result, ProRenaTa yields much higher resource utilization and less latency SLO violations comparing to state-of-the-art approaches while provisioning a distributed storage system. Based on ProRenaTa, we have built an elasticity controller named Hubbub-scale, which adopts a control model that generalizes the data migration overhead to the impact of performance interference caused by multi-tenancy in the Cloud.

Keywords: *Geo-distributed Storage Systems; Data Replication; Data Consistency; Transaction; Elastic Computing; Elasticity Controllers; Service Latency; Service Level Objective; Performance Interference*

Sammanfattning

I denna avhandling har vi undersökt två aspekter att förbättra prestanda för distribuerade lagringssystem. I ett spår, presenterar vi metoder och algoritmer för att minska latensen vid begäran av distribuerade lagringstjänster som distribueras geografiskt. I ett annat spår föreslår vi och designar kontrollmekanismer för elasticitet ansvariga för att upprätthålla förutsägbara/stabila prestanda hos distribuerade lagringssystem under dynamisk arbetsbelastning och osäkerheter hos plattformen.

Inom forskningen för att realisera det första spåret, har vi föreslagit en lease-baserad datakonsistensalgoritm som tillåter ett distribuerat lagringssystem att effektivt hantera läsningsdominerad arbetsbelastning i global skala. Väsentligen används "leasing" för att garantera riktigheten/aktualiteten hos data inom ett tidsintervall. Leasingalgoritmen tillåter repliker med giltiga "avtal" (leases) att betjäna läsförfrågningar lokalt. Som ett resultat av detta, betjänas de flesta av de lästa förfrågningarna med liten latens. Dessutom garanterar den tidsbaserade naturen av leasingpåståendena liveness hos konsistensalgoritmen även i närvaro av misslyckanden. Sedan har vi undersökt effektiviteten av kvorumbaserade datakonsistensalgoritmer när de används globalt. Vi har föreslagit ramverket MeteorShower, som är baserat på replikerade loggar och löst synkroniserade klockor, för att förstärka kvorumbaserade datakonsistensalgoritmer. Väsentligen tillåter MeteorShower algoritmerna att upprätthålla överensstämmelse mellan uppgifter genom att använda något äldre replikerade värden i de replikerade loggarna. Som ett resultat av detta behöver datakonsistensalgoritmerna inte längre fråga efter uppdateringar från avlägsna repliker, vilket avsevärt minskar latensen hos begärandena. Baserat på liknande insikter, bygger vi ett transaktionsramverk, Catenae, för geo-distribuerade datalager. Den använder replikerade loggar för att distribuera transaktioner och aggregera resultaten. Transaktioner fördelas i syfte att åstadkomma en spekulativ genomförandefas, som samordnas med hjälp av en algoritm baserad på transaktionskedjor. Algoritmen beställer transaktioner baserat på deras hastighet i förhållande till varje datapartition, vilket maximerar samtidigheten och determinismen hos transaktionerna. Som ett resultat är de flesta av exekveringsresultaten på kopior i olika datacenter förenliga när de undersöks i en valideringsfas. Detta gör det möjligt för Catenae att genomföra en läs- och skrivtransaktion inom ett enda fördröjningssteg mellan DC RTT i de flesta fall.

Inom det andra forskningsspåret, undersöker vi och kontrollerar de faktorer som orsakar prestandaförsämring vid skalning av ett distribuerat lagringssystem. Först har vi föreslagit BwMan, som är en modellbaserad manager av nätverksbandbredd. Denna lindrar prestandaförsämringen som orsakas av datamigreringsaktiviteter vid skalningen av ett distribuerat lagringssystem genom att dynamiskt begränsa den bandbredd som tilldelas denna verksamhet. Som ett resultat blir prestanda hos lagringssystemet mycket mer förutsägbar/stabil, dvs systemet uppfyller latensbaserade servicenivåmål (SLO), även i närvaro av datamigrering. Som ett steg framåt, har vi systematiskt modellerat effekterna av datamigreringar. Med hjälp av denna modell, har vi byggt en styrenhet för elasticitet, nämligen ProRenaTa, som kombinerar proaktiva och reaktiva kontrollmekanismer för att uppnå bättre noggrannhet i kontrollen. Med hjälp av förutsägelser av arbetsbelastningen och datamigreringsmodellen kan ProRenaTa beräkna bästa möjliga skalningsplan för att ändra storlek på ett distribuerat lagringssystem under begränsningen att uppnå tidsfrister för skalningen, minska brott mot SLO-latenser och minimera kostnaden för tillhandahållande av VM. Som ett resultat, ger ProRenaTa ett mycket högre resursutnyttjande och mindre brott mot SLO-latenser jämfört med state-of-the-art-metoder samtidigt som ett distribuerat lagringssystem tillhandahålls. Baserat på ProRenaTa har vi byggt styrenheter för elasticitet, vilka använder styrmodeller som generaliserar overheaden för datamigrering för effekterna av prestandastörningar som orsakas av "multi-tenancy" i molnet.

Résumé

Dans cette thèse, nous avons étudié deux approches pour améliorer la performance des systèmes de stockage distribués. Premièrement, nous présentons des techniques et des algorithmes pour réduire la latence des requêtes à des services de stockage géo-distribués. Deuxièmement, nous avons conçu des contrôleurs d'élasticité pour maintenir des performances prévisibles et stables des systèmes de stockage distribués soumis à des charges de travail dynamiques et aux incertitudes de plate-forme.

Selon le premier axe de cette recherche, nous avons proposé un algorithme de cohérence des données basé sur des bails (contrats limités dans le temps) qui permet à un système de stockage distribué de fournir efficacement une charge de travail principalement en lecture à une échelle globale. Essentiellement, les bails sont utilisés pour faire valoir la justesse et la fraîcheur des données pendant un intervalle de temps. L'algorithme permet à des répliques avec des bails valides de répondre à des requêtes locales de lecture. Par conséquent, la plupart des demandes de lecture sont servies avec peu de latence. En outre, la durée des bails garantit la vivacité de l'algorithme de cohérence, même en présence de défaillances. Ensuite, nous avons étudié l'efficacité des algorithmes de cohérence des données basés sur des quorums - lorsqu'ils sont déployés à l'échelle globale. Nous avons proposé le système MeteorShower, qui est basé sur des journaux répliqués et des horloges faiblement synchronisées, pour améliorer les algorithmes de cohérence des données basés sur des quorums. Fondamentalement, MeteorShower permet aux algorithmes de maintenir la cohérence des données en utilisant des valeurs de répliques légèrement anciennes fournies par les journaux répliqués. En conséquence, les algorithmes de cohérence des données basés sur des quorums n'ont plus besoin de demander les mises à jour à partir de répliques à distance, ce qui réduit considérablement la latence des requêtes. Basé sur des idées semblables, nous avons construit un système transactionnel, Catenae, pour les systèmes de stockage géo-distribués. Il emploie des journaux répliqués pour distribuer les transactions et agréger les résultats d'exécution. Les transactions sont distribuées afin de réaliser une phase d'exécution spéculative, qui est coordonnée à l'aide d'un algorithme de chaîne de transaction. L'algorithme ordonne les transactions en fonction de leur vitesse d'exécution par rapport à chaque partition de données, ce qui maximise la simultanéité et le déterminisme des exécutions de transaction. Par conséquent, la plupart des résultats d'exécution des répliques dans les différents centres de données sont cohérents lorsqu'ils sont examinés lors d'une phase de validation. Cela permet à Catenae d'exécuter une transaction de lecture-écriture avec dans la plupart des cas un seul délai aller-retour inter-DC.

En ce qui concerne la deuxième voie de recherche, nous avons examiné et contrôlé les facteurs qui causent une dégradation des performances lors du redimensionnement d'un système de stockage distribué. Premièrement, nous avons proposé BwMan, qui est un gestionnaire de bande passante de réseau basé sur des modèles. Il atténue la dégradation des performances causée par les activités de migration de données lors du redimensionnement d'un système de stockage distribué par régulation dynamiquement de la bande passante allouée à ces activités. Ainsi, les performances du système de stockage sont beaucoup plus prévisibles et stables, c'est-à-dire qu'elles répondent à un objectif de qualité de service basé sur la latence, même en présence de migration des données. Ensuite, nous avons modélisé de manière systématique l'impact des migrations de données. En utilisant ce modèle, nous avons construit un contrôleur d'élasticité, ProRenaTa, qui combine des contrôles proactifs et réactifs pour obtenir une meilleure précision de contrôle. Grâce à la prévision de la charge de travail et au modèle de migration de données, ProRenaTa est capable de calculer le meilleur plan possible afin de redimensionner un système de stockage distribué sous la contrainte de respect des délais, en réduisant les violations de l'objectif de qualité de service basé sur la latence et en minimisant les coûts de création de machines virtuelles. Par conséquent, ProRenaTa conduit à une bien meilleure utilisation des ressources et à moins de violations en qualité de service basé sur la latence, en comparaison avec les approches constituant actuellement à l'état de l'art en matière de fourniture d'un système de stockage distribué. Sur la base de ProRenaTa, nous avons construit des contrôleurs d'élasticité, qui adoptent des modèles de contrôle qui généralisent le surcoût de migration de données à l'impact de l'interférence de la performance causée par la colocation dans le Cloud.

Acknowledgments

This work has been supported 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 GA number 317879, and in part by the End-to-End Clouds project funded by the Swedish Foundation for Strategic Research (SSF) under the contract RIT10-0043.

I am deeply thankful to the following people, without the help and support of whom, I would not have managed to complete this thesis.

First and foremost, my primary supervisor Vladimir Vlassov, for his patient guidance, constant feedback, indispensable encouragement and unconditional support throughout this work. My secondary supervisors, Peter Van Roy, Jim Dowling and Seif Haridi, for their valuable advice and insights. My advisors at Ericsson Research, Fetahi Wuhib and Azimeh Sefidcon. Their visions have profoundly impacted my research.

My sincere thanks to Marco Canini for his invaluable and visionary research guidance.

My wholehearted thanks to two special teachers, Johan Montelius and Leandro Navarro.

Lydia Y. Chen, Olivier Bonaventure, Charles Pecheur, Peter Van Roy, Vladimir Vlassov, Jim Dowling, and Seif Haridi, for assessing and grading my thesis at the private defense. Their constructive comments remarkably helped in delivering the final version of this work.

Dejan Kostic for agreeing to serve as the internal reviewer of my thesis. His experience and valuable feedback significantly helped in improving the final version of this thesis.

Gregory Chockler for agreeing to serve as the thesis opponent. Erik Elmroth, Maria Kihl, and Ivan Rodero for accepting to be on the thesis grading committee. And, Sarunas Girdzijauskas for chairing my thesis defense.

Special thanks to my co-authors, Vamis Xhagjika, Navaneeth Rameshan, Ahmad Al-Shishtawy, Enric Monte, and my master students. Working with them has been a great pleasure and indispensable learning experience.

My EMJD-DC colleagues and colleagues at KTH and UCL, namely Vasiliki Kalavri, Manuel Bravo, Zhongmiao Li, Hooman Peiro Sajjad, Paris Carbone, Kamal Hakimzadeh, Anis Nasir, Jingna Zeng, Ahsan Javed Awan and Ruma Paul for always being available to discuss ideas, for being supportive and for the good times we spent hanging out together.

Thomas Sjöland, Sandra Gustavsson Nylén, Susy Mathew, and Vanessa Maons for providing unconditional support throughout my PhD studies at KTH and UCL.

Ying Liu

Contents

List of Figures

Chapter 1

Introduction

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

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

Availability is another desired property for a storage system. Availability means that data stored in the system is safe and always (or most of the time) available to its clients. Replication is usually implemented in a distributed storage system 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 functioning and accessible servers that have copies of the data.

Maintaining multiple copies of the same data brings the challenge of **data consistency**. Based on application requirements and usage scenarios, a storage solution is expected to provide some level of consistency guarantees. For example, strong consistency ensures that all the data copies act synchronously like one single copy and it is desired because of its predictability. Other consistency models, such as eventual consistency, allow data copies to diverge within a short period of time. In the general case, stricter consistency models necessitate more overhead for a system.

Multiple data replicas in multiple servers also need to survive **network partitions**. Network partitions block communication between data copies. In this scenario, either inconsistent results or no results can be returned to clients. The availability, consistency

and partition tolerance together make up the three essential aspects in a distributed storage system known as the CAP theorem [1, 2]. The theorem states that only two of the three properties can be achieved in one system.

Elasticity describes a property of a storage system, which is able to scale up/down according to the incoming workload to maintain a desired quality of service (QoS). The elasticity of a storage system is usually achieved with the help of an autonomic elasticity controller, which monitors several metrics that reflect the realtime status of the managed system and issues corresponding scaling operations to maintain a desired QoS.

1.1 In the Cloud

In this thesis, we study the performance of distributed 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 with 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)"; where X can be infrastructure, hardware, platform, or software. In this thesis, I assume that a Cloud supports 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 possibility for companies to host their services without operating their own data center. Moreover, the pay-as-you-go business model allows companies to use Cloud resources on a short-term basis as needed. On the 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.

Geo-distribution

Another advantage of using Cloud services is their wide geographical coverage. For the moment, dominant Cloud providers, such as Amazon Web Services, Google Cloud Platforms, Microsoft Azure, and IBM Bluemix, allow users to host their services in data centers across multiple continents. It facilitates companies to start their business on a global scale, which enables them to provide services closer to their clients, thus reducing service latency. However, maintaining services across multiple data centers also brings new challenges. In this thesis, we will investigate some of them.

1.2 Research Objectives

The objective of the thesis is to optimize service latency/throughput 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 dynamic workload that needs to be handled by the system.

Naturally, a less efficient storage solution slows down the processing of requests, whereas an intensive workload might saturate the system and cause performance degradation. Thus, in order to achieve a low latency/high throughput distributed storage solution, we define two main goals:

1. to improve the efficiency of distributed storage algorithms, and
2. to enable storage systems adapt to workload changes

Our vision towards the first goal is to make storage systems deployed in a larger scale, so that requests can be served by servers that are close to clients. This will significantly reduce request latency, especially the portion of high latency requests, if clients are distributed across a large geographical area. However, when the system is deployed across a large geographical area, the communication overhead within the system dramatically increases. This increased overhead significantly influences service latency when requests need to access several data replicas, which are separated by large physical distances. And this is usually the case when a system maintains strong consistency guarantees, e.g., sequential consistency. We specify our optimizations in this scenario with the objective of reducing replica communication overhead under the requirements of sequential data consistency while not compromising a system's scalability and availability.

The core challenge to achieve the second goal is introduced by the complexity of workload patterns, which can be dynamic in intensity, concurrency, and locality. We propose smart middleware, i.e., elasticity controllers, to effectively and efficiently provision resources allocated to a distributed storage system. The resource allocation considers the workload characteristics when optimizing to low service latency and reduced provisioning cost. Specifically, an increase in the workload typically results in an increase in the allocated resources to maintain the low service latency. On the other hand, a decrease in the workload leads to the removal of surplus resources to save the provisioning cost. Due to the characteristics of a distributed storage service, the addition and removal of resources is non-trivial. This is because storage services are stateful, which means that data needs to be allocated to newly added resources before they can serve requests and data needs to be migrated before resources can be safely removed from the system. Thus, we focus our research on the data migration challenge while designing an efficient and effective elasticity controller for a distributed storage system.

1.3 Research Methodology and Road Path

In this section, we describe the methods and road paths that I followed during my PhD studies.

1.3.1 Design of Efficient Storage Solutions

The methodology

The work on this matter does not follow analytical, mathematical optimization methods, but is rather based on an empirical approach. I approach the problem by first studying tech-

niques/algorithms used in the design of distributed storage solutions. This process provides me the knowledge base for inventing new algorithms to improve the efficiency of storage solutions. After understanding the state of the art solutions, I start investigating the usage scenarios of distributed storage solutions. The efficiency of a storage solution varies on different usage scenarios. I 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. My research focuses on designing efficient storage solutions for a newly emerged usage scenario, i.e., providing services in a global scale. To accomplish my research, I first investigate the causes of inefficiency when applying current storage solutions to this usage scenario. After examining a sufficient number of leading storage solutions, I choose the most suitable system architecture for my usage case. I tailor algorithms by avoiding the known performance bottlenecks. Finally, I evaluate my design and implementation by comparing it with several leading storage solutions. I use request latency as the performance measure and also discuss computational complexity of the algorithm, when applicable.

The road path

By understanding the state-of-the-art approaches in the design of distributed storage systems [3, 4, 5, 6] and the current trends in Cloud computing, I have identified a gap between efficient storage system designs and an emerging usage scenario, i.e., geo-distribution. Specifically, there is insufficient research on achieving low latency when a distributed storage system is deployed in a large geographical area. I have conducted my research by first designing and implementing a globally-distributed and consistent key-value store, which is named GlobLease. It organizes multiple distributed hash tables (DHTs) to store the replicated data and namespace, which allows different DHTs to be placed in different locations. Specifically, data lookups and accesses are processed with respect to the locality of DHT deployments, which improves request latency. Moreover, GlobLease uses leases to maintain data consistency among replicas. It enables GlobLease to provide fast and consistent read accesses with reduced replica communications. Write accesses are also optimized by migrating the master copy of data to the locations where most of the writes take place. With the experience of GlobLease, I have continued my research with dominant open-source storage solutions, which have large user groups. Thus, more people could benefit from the research results. Specifically, I have designed and implemented a middleware called MeteorShower on top of Cassandra [3], which minimizes the latency overhead to maintain data consistency when data are replicated in multiple data centers. MeteorShower employs a novel message propagation mechanism, which allows data replicas to converge faster. As a result, it significantly reduces request latency. Furthermore, the technique applied in MeteorShower does not compromise the existing fault tolerance guarantees provided by the underlying storage solution, i.e., Cassandra. To leverage more usage scenarios, I have implemented a similar message propagation mechanism to achieve low latency serializable transactions across a large geographical area. In this matter, I have proposed, Catenae, which is a framework that provides transaction support for Cassandra when data is replicated across multiple data centers. It leverages similar message propaga-

tion principles proposed in MeteorShower for interactions among transactions. Moreover, Catenae employs and extends a transaction chain concurrency control algorithm to speculatively execute transactions in each data center to maximize the execution concurrency. As a result, Catenae significantly reduces transaction execution latency and abort rates.

1.3.2 Design of an Elasticity Controller

The methodology

The work on the elasticity controller also follows an empirical approach. My 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 the service latency of the controlled systems. Then, I study the technologies that are used in building performance models and the frameworks that are applied in implementing the controller. The results of the studies allow me to discover the unconsidered elements/parameters that influence the effectiveness and accuracy of an elasticity controller. I experimentally verify my assumptions on the performance degradation of elasticity controllers when the elements/parameters are not considered. Once I have confirmed the space for improving the effectiveness and accuracy of elasticity controllers, I innovate by designing new performance models that consider those environmental and system elements/parameters. After implementing my controller using the novel performance model, I evaluate it by comparing it to the original implementation. For the evaluation, I deploy my system in real platforms and test it with real-world workload, where possible. I use service latency and resource utilization as performance measures.

The road path

Studies on distributed storage systems have shown that data needs to be allocated/deallocated to storage nodes before they start serving client requests. It mainly causes two challenges when scaling a distributed storage system. First, migrating data consumes system resources in terms of network bandwidth, CPU time, and disk I/Os. This means that scaling up/down a storage system will hurt the performance of the system during the data migration phase. On the other hand, migrating data consumes time, which means that a scaling decision cannot have immediate effect on the current system status. There is a delay before new resources start alleviating system loads. For the first challenge, I have conducted my research towards regulating the resources that are used for data migration when the system resizes. Specifically, I have designed and implemented a bandwidth arbitrator named BwMan, which regulates the bandwidth allocation among client requests and data migration load. It throttles the bandwidth consumed by data migration when it starts affecting the QoS. BwMan guarantees that the system operates with a predictable request latency and throughput. For the second challenge, I have proposed a novel data migration performance model, which can analytically calculate the time that is needed to scale a distributed storage system under the current workload. A workload prediction module is integrated to facilitate the resizing of the system with the data migration performance model. These techniques are implemented in an elasticity controller called ProRenaTa. With the

successful experience of ProRenaTa, I have continued my research in the generalization of data migration overhead. It is generalized to include interference from the platforms hosting the storage system. Using a similar modeling technique, I have broadened the research to consider the performance interference caused by co-runners sharing the same platform when scaling storage systems. The elasticity controller, namely Hubbub-scale, is implemented to scale distributed storage systems in a multi-tenant environment.

1.4 Contributions

This thesis presents techniques that optimize the service latency of distributed storage systems. On one hand, it focuses on designing efficient storage solutions to be deployed geographically. On the other hand, it presents research about the design of elasticity controllers in order to achieve predictable performance of distributed storage systems under dynamic workloads. In the following paragraphs, I present my contributions that address research problems in these two domains. All of the works included in this thesis are written as research papers, and most of them have been published in international peer-reviewed publications. I was the main contributor of all the works/research papers included in this thesis. I was the initiator and the main contributor in coming up with the research ideas and formalizing them. I motivated and approaches the research challenges from unique angles. I implemented all the parts of the mentioned research works. I was also the main conductor in evaluating all the research works proposed.

Efficient Storage Solutions

GlobLease. I address the problem of high read request latency when a distributed storage system serves requests from multiple data centers while data consistency is preserved. I approach the problem from the idea of cache coherency. Essentially, I have adapted the idea of leasing resources to maintain data consistency. Then, I have implemented the idea of leasing data in GlobLease, which is a globally-distributed and consistent key-value store. It differs from the state of the art works and contributes in three ways. First, my system is organized as multiple distributed hash tables storing replicated data and namespace, which allows different DHTs to be placed in different locations. Specifically, I have implemented several optimizations in the routing of requests. Firstly, data lookups and accesses are processed with respect to the locality of DHT deployments, which gives priority to data located in the same data center. Second, I have applied leases to maintain data consistency among replicas. Leases enable a data replica to be read consistently within a specific time bound. It allows GlobLease to provide fast and consistent read accesses without inter-replica communication. I have also optimized writes in GlobLease by migrating the master copy of the data to the locations where most of the writes take place. Third, I have designed GlobLease to be highly adaptable to different workload patterns. Specifically, fine-grained elasticity is achieved in GlobLease using key-based multi-level lease management. It allows GlobLease to precisely and efficiently handle spiky and skewed read workloads. These three aspects of GlobLease enable it to have predictable performance in a geographical setup. As a re-

sult, GlobLease reduces more than 50% of high latency requests that contribute to the tail latency.

MeteorShower. I reduce the request latency of distributed storage systems which rely on majority quorum based data consistency algorithms. First, I have identified that pulling updates from replicas causes high latency. I propose that replicas actively exchange their updates using periodic status messages, which halves the delay of receiving the updates. Based on this insight, I allow each replica to maintain a cache of other replicas. The cached values are updated with the periodic status messages from other replicas. With the slightly stale caches of replicas, I try to reason about the data consistency levels based on different applications of the cached values. Taking sequential consistency as an example, I have proved that we are able to achieve this data consistency with significantly less delays using the cached values. Another advantage of my algorithm is that it does not compromise any existing properties of the system, for example, fault tolerance, since it is incremental to the existing algorithm. To validate my algorithm, I have implemented it with Cassandra in a system called MeteorShower. The performance of MeteorShower is compared against Cassandra. I have confirmed that my algorithm is able to out-performance traditional majority quorum operations significantly in the context of geo-distributed storage systems.

Catenae. I have designed and implemented Catenae, which provides low latency transactions when data are replicated and stored in different data centers. I approach the idea from using periodic replicated epoch messages among replicas to distribute transactions and aggregate transaction results. My idea is instead of pulling transactions and transaction results from replicas, Catenae uses periodic epoch messages to push the payload to all replicas. This approach halves the delay for replicas to acknowledge the updates from the other replicas. As a result, the commit latencies of transactions are reduced. Furthermore, in order to boost the transaction execution concurrency, I have employed and extended a transaction chain algorithm [7]. With the information from epoch messages, the transaction chain algorithm is able to speculatively execute transactions upon replicas with maximized concurrency and determinism of transaction ordering. I have experimentally proved that following my algorithm most of the speculative executions will produce valid results. Inconsistent results from speculative executions can be amended through a voting process among data centers (breaking ties using data center ids).

Evaluations have shown that my system, Catenae, is able to commit a transaction with half a RTT to a single RTT in most of the cases. Evaluations with the TPC-C benchmark show that Catenae significantly outperforms Paxos Commit [8] over 2-Phase Lock [9] and Optimistic Concurrency Control [10]. My system achieves more than twice the throughput of the other two approaches with 50% less commit latency.

Elasticity on Storage Systems

BwMan. I address the issue of performance degradation of a distributed storage system when it conducts data migration because of resizing activities. I have identified that there are mainly two types of workloads in a distributed storage system: user-centric workloads and system-centric workloads. A user-centric workload is the load that is created by client requests. Workloads that are caused by data migration, which can be initiated because of

load rebalancing, failure recovery, or system resizing (scaling up/down), are called system-centric workloads. Obviously, both workloads are network bandwidth intensive. I demonstrate that without explicitly managing the network bandwidth resources among these two different workloads leads to unpredictable performance of the system.

I have designed and implemented BwMan, a network bandwidth manager for distributed storage systems. BwMan dynamically arbitrates bandwidth allocations to different services within a virtual machine. Dedicated bandwidth allocation to user-centric workloads guarantees the predictable performance of the storage system. Without hurting user-centric performance, dynamic bandwidth allocation to system-centric workloads allows system maintenance tasks to finish as fast as possible. Evaluations demonstrate that the performance of the storage system under the provisioning of BwMan appears to be more than twice as predictable and stable as its counterpart without BwMan during system resizing.

ProRenaTa. I have identified that data migration in distributed storage systems not only consumes resources, but also delays the service of the newly added resources. Then, I have designed and implemented ProRenaTa, which is an elasticity controller that addressed the above issue while scaling a distributed storage system.

Experimentally, I demonstrate that there are limitations, caused by data migration, while relying solely on proactive or reactive tuning to auto-scale a distributed storage system. Specifically, a reactive controller can scale the system with 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, performance degradation is observed in the initial phase of scaling because of data/state migration in order to add/remove resources. Proactive controllers, on the other hand, are able to prepare resources in advance and avoid performance degradation. However, the performance of the proactive controller largely depends on the accuracy of workload prediction, which varies for different workload patterns. Worse, in some cases workload patterns are not even predictable. Thus, proper methods need to be designed and applied to deal with the inaccuracies of workload prediction, which directly influences the accuracy of scaling that in turn impacts system performance.

I have also identified that, in essence, proactive and reactive approaches complement each other. A 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. A 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, I have designed ProRenaTa, which is an elasticity controller that combines both proactive and reactive insights. I have built a data migration model to quantify the overhead to finish a scale-in or scale-out plan in a distributed system, which is not explicitly considered or modelled in the state of the art works. The data migration model is able to guarantee stable/predictable performance while scaling the system. By consulting the data migration model, the ProRenaTa scheduler is able to arbitrate the resources that are allocated to system resizing without sacrificing system performance. Experimental results indicate that ProRenaTa outperforms state of the art approaches by guaranteeing a higher level of performance commitments while also maintaining efficient resource utilization.

1.5 Research Limitations

There are certain limitations in the application of research results from this thesis.

The storage solutions that we have proposed in this thesis, i,e,. **GlobLease**, **MeteorShower** and **Catenae**, will out-performance the state-of-the-art approaches only when they are deployed in multiple data centers. And the replicas of data items need to be hosted in different data centers, which means that each data center maintains and replicates a full storage namespace.

With respect to each individual system, GlobLease sacrifices a small portion of the low latency read requests to reduce a large portion of extremely high latency write requests. It is not designed to completely remove high latency requests, which limits its application to latency sensitive use cases. Furthermore, GlobLease does not tolerate network partitions. We tradeoff the tolerance of network partitions for data availability and consistency. Thus, in the presence of network partitions, some of GlobLease servers might not be able to serve requests in order to preserve data consistency. Another limitation of GlobLease is its tolerance to server failure. Slave server failures will significantly influence the write performance of GlobLease depending on the length of leases since writes can only proceed when all leases are updated or expired. Master server failures influence the performance of both reads and writes, which need to wait for a Paxos election for the new master.

MeteorShower behaves better than GlobLease in the presence of server failures since it adopts majority quorums instead of master-slave paradigm. Essentially, there is no overhead when a minority of replicas fail. However, MeteorShower, like all the storage systems relying on majority quorums, does not tolerate the failure of a majority of servers. Since MeteorShower extensively utilizes the network resources among servers, its performance depends on the network connectivity of those servers. It is observed in our evaluations that we have a significantly shorter tail in latency when MeteorShower uses a intra-DC network than a inter-DC network. The effect is more prominent under a more intensive workload. Thus, MeteorShower is not suitable for platforms where the performance of network is limited. Lastly, the data consistency algorithm in MeteorShower relies on the physical clocks of all the servers. The correctness of MeteorShower depends on the assumption of bounded clocks, which means the clock of each server can be represented by the real-time clock within a bounded error. Thus, significant drifts in clocks interfere with the correctness and performance of MeteorShower.

Similar to MeteorShower, Catenae also extensively exploits network bandwidth available to servers. Thus, the same limitation applies to Catenae as well. Furthermore, the performance of Catenae depends on the predictability of transaction execution time on each data partition. Thus, when most of the transactions have the same processing time on most of the data partitions, Catenae will not perform better than the state-of-the-art approaches. Also, the rollback operations in Catenae may trigger cascading aborts. It significantly degrades the performance of Catenae under a highly skewed workload. Another possible limitation of Catenae is its application scenario. Essentially, Catenae can only process transactions that are chainable. It means that data items that are accessed by a transaction should be known before its execution. And the accesses of these data items should follow a deterministic order. Thus, Catenae cannot execute any type of transactions.

There are also limitations when applying the controllers proposed in this thesis. Specifically, **BwMan** manages network bandwidth uniformly in each storage server. It relies on the mechanisms in the storage system to balance the workload in each storage server. Thus, the coarse grained management of BwMan is not applicable to systems where workloads are not well-balanced among servers. Furthermore, BwMan does not scale the bandwidth allocated to a storage service horizontally. It conducts the scaling vertically, which means that BwMan only manages the network bandwidth within a single host to individual services. BwMan is not able to scale a distribute storage system when the bandwidth is not sufficient. In addition, the empirical model of BwMan is trained offline, which makes it impossible to adapt to changes of the execution environment that are not considered in the model.

With respect to **ProRenaTa**, it integrates both proactive and reactive controllers. However, the accuracy of workload prediction plays an essential role in the performance of ProRenaTa. Specifically, a poorly predicted workload causes possibly wrong actions from the proactive controller. As a result, severe SLO violations happen. In other words, ProRenaTa is not able to perform effectively without an accurate model for predicting the workload. Furthermore, ProRenaTa sets up a provisioning margin for data migration during the scaling of a distributed storage system. The margin is used to guarantee a specific scaling speed of the system. But, it leads to an extra provisioning cost. Thus, it is not recommended to provision a storage system that does not scale frequently or does not need to migrate a significant amount of data during scaling. In addition, the control models in ProRenaTa are trained offline, which makes them vulnerable to unmonitored execution environment changes. Besides, the data migration model and the bandwidth actuator, namely BwMan, assume a well-balanced workload on each storage server. The imbalance of workload on each server will influence the performance of ProRenaTa.

1.6 Research Ethics

The research conducted in this thesis does not have any human participants involved. Thus, it is exempted from the discussions of most of the ethic issues. The possible ethic concerns of my research are the applications of the proposed storage systems and the privacy of the data stored. However, it is the users responsibility to guarantee that my storage solutions are not used for storing and serving illegal contents. Furthermore, the studies on the security and privacy of the stored data are orthogonal to the research presented in this thesis.

On the other hand, the research results from this thesis can be applied to reduce the consumption of energy. Specifically, the application of elastic provisioning in a storage system improves the utilization of the underlying resources, which are fundamentally computers. Essentially, redundant computer resources are removed from the system when the incoming workload drops. And the removed computers can be shut down to save energy.

1.7 List of Publications

Most of the content in this thesis are based on material previously published in peer reviewed conferences.

Chapter 4 is based on the following papers.

1. 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

2. Y. Liu, X. Guan, and V. Vlassov, *MeteorShower: Minimizing Request Latency for Geo-replicated Peer to Peer Data Stores*, under submission, 2016

3. Y. Liu, Q. Wang, and V. Vlassov, *Catenae: Low Latency Transactions across Multiple Data Centers*, accepted for publication in 22nd IEEE International Conference on Parallel and Distributed Systems (ICPADS), 2016

Chapter 5 includes researches presented in the following papers.

1. 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

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

3. N. Rameshan, Y. Liu, L. Navarro and V. Vlassov, *Hubbub-Scale: Towards Reliable Elastic Scaling under Multi-Tenancy*, 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2016

Research works that have been conducted during my PhD studies but not included in this thesis are the following.

1. 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

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

3. Y. Liu, D. Gureya, A. Al-Shishtawy and V. Vlassov, *OnlineElastMan: Self-Trained Proactive Elasticity Manager for Cloud-Based Storage Services*, IEEE International Conference on Cloud and Autonomic Computing (ICCAC), 2016

4. N. Rameshan, Y. Liu, L. Navarro and V. Vlassov, *Augmenting Elasticity Controllers for Improved Accuracy*, accepted for publication in 13th IEEE International Conference on Autonomic Computing (ICAC), 2016

5. N. Rameshan, Y. Liu, L. Navarro and V. Vlassov, *Reliable Elastic Scaling for Multi-Tenant Environments*, accepted for publication in 6th international workshop on Big Data and Cloud Performance (DCPerf), 2016

1.8 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 gives the necessary background and describes the systems used in this research work. Chapter 3 provides an overview of related techniques in achieving high performance geo-distributed storage systems hosted in the Cloud. Chapter 4 focuses on improving the efficiency of distributed storage systems deployed across a large geographical area. Chapter 5 discusses using elasticity controllers to guarantee predictable performance of distributed storage systems under dynamic workloads. Chapter 6 contains conclusions and future work.

Chapter 2

Background

Hosting services in the Cloud is becoming more and more popular because of a set of desired properties provided by the platform, that include a 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 and static web servers. Distributed storage service is a stateful service, where state/data needs to be properly maintained. 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 [11, 12]. 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.

In the rest of this chapter, we present the concepts and techniques used in this thesis.

- Cloud, a visualized environment to effectively and economically host services;
- Elastic computing, a cost-efficient technique to host services in the Cloud;
- Distributed storage systems, storage systems that are organized in a decentralized fashion

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 [13]." A Cloud is the integration of data center hardware and software that provides "X as a service

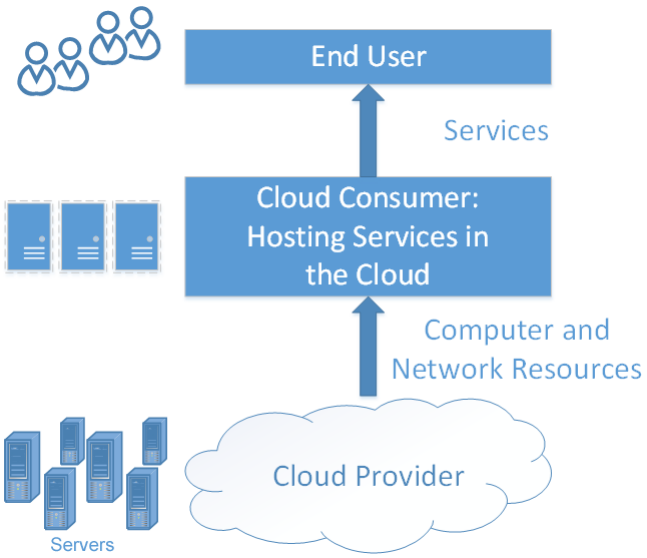


Figure 2.1 – The Roles of Cloud Provider, Cloud Consumer, and End User in Cloud Computing

(XaaS)" to clients; where X can be infrastructure, hardware, platform, and software. These services in the Cloud are made available in pay-as-you-go manner to users. The advantages of Cloud computing to Cloud providers, consumers, and end users are well understood. Cloud providers make profits in renting out 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 the 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 available whenever requested without building or provisioning their own data centers. End users are 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 [13], 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.

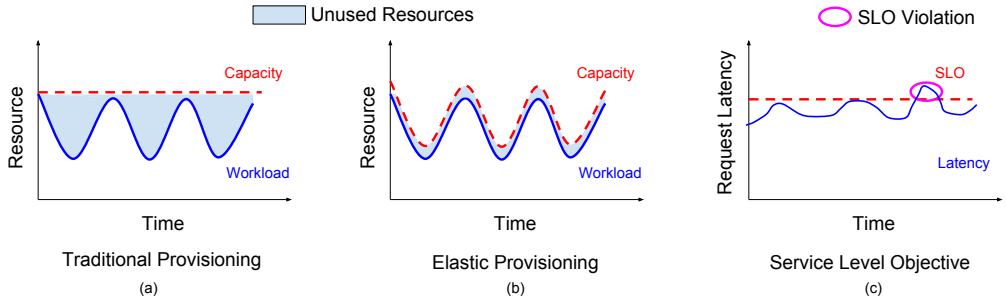


Figure 2.2 – (a) traditional provisioning of services; (b) elastic provisioning of services; (c) compliance of latency-based service level objective

2.1.1 Service Level Agreement

Service Level Agreements (SLA) define the quality of service that is expected from the service provider. SLAs are usually negotiated and agreed between Cloud service providers and Cloud service consumers. An 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 the consumer. When the service provider is unable to uphold the agreed level of the service, penalties are paid to the consumers. From the consumers perspective, an SLA violation can result in degraded service to their clients and consequently lead to loss in profits. Hence, the SLA commitment is essential to the profit of both Cloud service providers and consumers. In practice, an SLA is divided into multiple Service Level Objectives (SLOs). Each SLO focuses on the guarantee of one aspect of the service quality, e.g., service up time or service latency.

2.2 Elastic Computing

Cloud computing provides unlimited resources on demand, which facilitates the application of elastic computing [14]. Essentially, elastic computing means that a service is elastically provisioned according to its needs. Figure 2.2 illustrates the process of traditional and elastic provisioning of a service under a diurnal workload pattern. Specifically, Figure 2.2 (a) shows that a service is constantly provisioned with the amount of resources according to its peak load in order to achieve a specific level of QoS all the time. It is obvious that this traditional service provisioning approach wastes a significant amount of resources when the workload decreases from its peak level. On the other hand, Figure 2.2 (b) presents the elastic provisioning approach where the amount of the provisioned resources follows the changes in the workload. It is intuitive that elastic provisioning saves a significant amount of resources comparing to the traditional approach.

Both traditional and elastic provisioning approach aim to guarantee a specific level of quality of service or a service level objective (SLO). For example, Figure 2.2 (c) shows a commitment of a latency-based SLO most of the times. The goal of a well-designed provisioning strategy is to prevent SLO violations with the minimum amount of provisioned

resources, achieving the minimum provisioning cost. In order to maintain an SLO and reduce the provisioning cost, the correct amount of resources needs to be provisioned. Insufficient provisioning of resources, with respect to the workload, leads to an increase in the request latency and violates the SLO. However, over-provisioning of resources causes inefficiency in utilizing the resources and results in a higher provisioning cost.

In sum, 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 workload changes. Typically, an elastic system is able to add or remove service instances (with proper configurations) according to increasing or decreasing workloads in order to meet the SLOs, 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. The unlimited amount of resources on demand in Cloud is a perfect suit for elastic computing.

Elasticity of a system is usually achieved with elasticity controllers. A core requirement of an elasticity controller is that it should be able to help saving the provisioning cost of an application without sacrificing its performance. In order to achieve this requirement, an elasticity controller should satisfy the following properties:

1. Accurate resource allocation that minimizes the provisioning cost and SLO violations.

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

3. Efficient use of resources under SLO requirement 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.

In addition, the services hosted in the Cloud can be 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 the scaling activities are not properly handled. Thus, it is challenging to scale a stateful system. We have identified an additional requirement, which needs to be added when designing an elasticity controller for stateful services.

4. Be aware of the scaling overhead, including the consumption of system resources and time, and prevent it from causing SLO violations.

2.2.1 Auto-scaling Techniques

There are different techniques that can be applied to implement an elasticity controller. Typical methods are threshold-based rules, reinforcement learning or Q-learning (RL), queuing theory, control theory and time series analysis. We have used reinforcement learning, control theory and time series analysis to develop elasticity controllers described in Chapter 5 of this thesis.

Threshold-based Rules

The representative systems that use threshold-based rules to scale a service are Amazon Cloud Watch [15] and RightScale [16]. This approach defines a set of thresholds or rules in advance. Violating the thresholds or rules will trigger the action of scaling.

Reinforcement Learning or Q-learning (RL)

Reinforcement learning is 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 depends on the consulted value from the model. The accuracy of the model depends on the metrics and the selected model, as well as the amount of data trained to derive the model. For example, [17] 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.

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, [19] 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.

Control Theory

Elasticity controllers built using control theory to scale systems are mainly reactive feedback controllers, 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

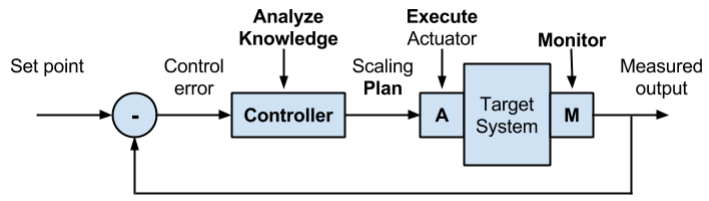


Figure 2.3 – Block Diagram of a Feedback Control System

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 2.3 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 methods 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.

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 a predicted value of an interested input metric, such as the CPU utilization or the workload intensity, in order to facilitate the decision making of an elasticity controller. It is easier to scale a system if the incoming workload is known beforehand. Prior knowledge of workload allows the controller to have enough time and resources to handle reconfiguration overhead of the system.

2.3 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 provides background on the three main aspects of a distributed storage system, namely, organiz-

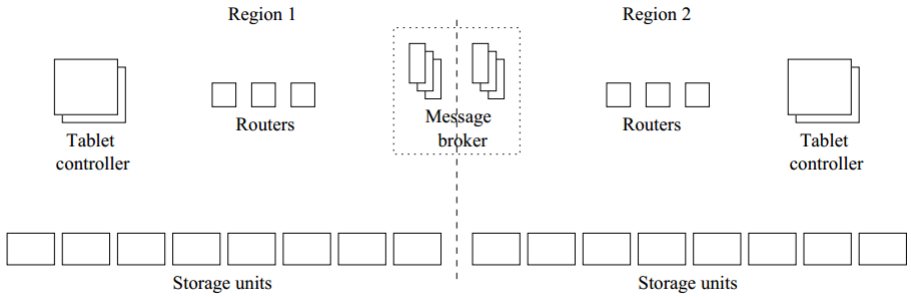


Figure 2.4 – Storage Structure of Yahoo! PNUTS

ing structure (in Section 2.3.1), data replication (in Section 2.3.2) and data consistency (in Section 2.3.3).

2.3.1 Structures of Distributed Storage Systems

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

Hierarchical Structure

A hierarchical distributed storage system is constructed with multiple components responsible for different functionalities. For example, special 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 [20], Hadoop File System [21], Google Spanner [4] and Yahoo! PNUTS [5]. An example is given in Figure 2.4, which shows the storage structure of Yahoo! PNUTS. It uses tablet controller to maintain the storage namespace, router components to route requests to responsible tablet controllers, message brokers to asynchronously deliver messages 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 changes whenever nodes join or leave the system. This structure also enables scalability of the system since all nodes function the same way and are organized in a decentralized fashion, i.e., there is no potential bottleneck. Availability is achieved by having data redundancies in multiple peer servers 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

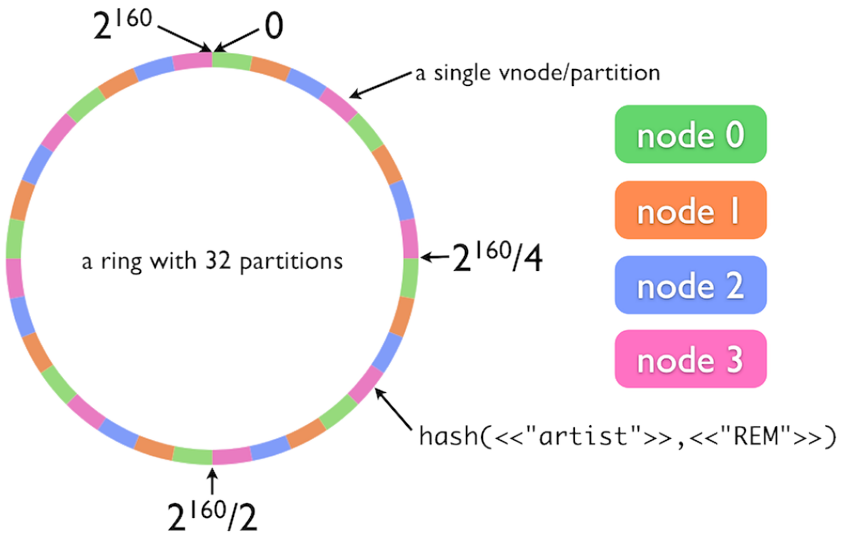


Figure 2.5 – Distributed Hash Table with Virtual Nodes

a centralized namespace service for searching resources, which is proved to be a bottleneck. An elegant solution to this issue is using a distributed hash table (DHT). It uses the hashes of object names to locate the objects. Different routing strategies and heuristics are proposed to improve the routing efficiency.

Distributed Hash Table

Distributed Hash Table (DHT) is widely used in the design of distributed storage systems [6, 3, 12]. 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. Each node maintains the data that falls into the range between its ID and its predecessor's ID. As an improvement, 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 2.5 presents a DHT namespace distributed among four nodes with virtual nodes enabled.

Request routing in a DHT is handled by forwarding requests through predecessor links, successor links or finger links. Finger links are established among nodes based on some criteria/heuristics for efficient routing [22, 23]. 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.

2.3.2 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. It allows the storage system to continuously serve data to its clients. The system scalability is also improved by using replication techniques. Concurrent clients are 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 2.3.3.

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 extremely 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: *scaling out without replication* or *scaling out with replication*. For scaling out without replication, 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 as a whole is capable to handle larger workloads. However, this solution requires expertise on service logic, based on which, data partitioning and distribution need to be performed in order to achieve scalability. Consequently, after scaling out, 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. On the other hand, scaling out with replication copies data from one server to multiple 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 [4, 12]. 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.

2.3.3 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 models are proposed based on different usage scenarios and application requirements. Typical data consistency models include atomic, sequential, causal, FIFO, bounded staleness, monotonic reads, read my writes, and etc. In this thesis, we focus on the application of sequential data consistency model. 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 up-to-date 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 [3], different data consistency models can be implemented by specifying different values of R and W, which denote the acknowledges required to return a read or write operation. For example, in order to achieve sequential consistency, the minimum requirement is that $R + W > N$.

2.3.4 Paxos

Paxos, the de-facto distributed consensus protocol that can handle $\frac{N}{2} - 1$ node failures for a system that has N nodes. It is widely used for achieving consensus in distributed systems.

Consensus

Generally, there are two requirements for a consensus protocol: safety and liveness. Safety states the correctness of the protocol: only a value that has been proposed can be chosen, only a single value is chosen and a process never learns a value unless it actually has been chosen. Liveness says that the protocol eventually behaves as we expected: some proposed value is eventually chosen and if a value is chosen, a process will eventually learn it.

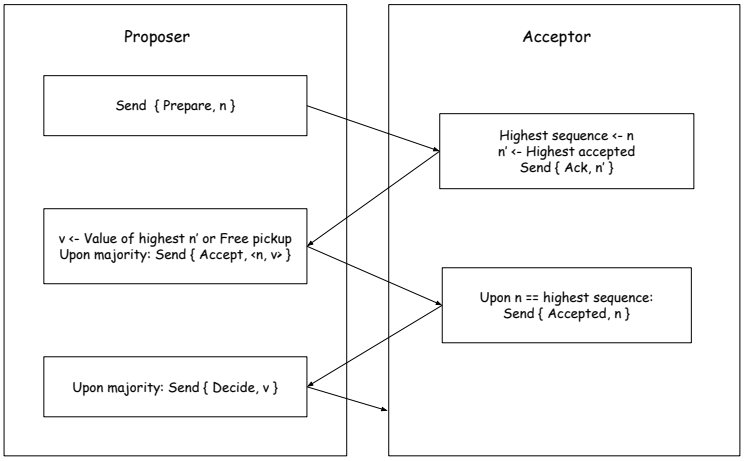


Figure 2.6 – Paxos Algorithm

Paxos protocol usually makes the following assumptions: there is a set of nodes that can propose values, any node can crash and recover, node has access to the stable storage, the messages are passed among nodes asynchronously, and messages can be lost or duplicated but never corrupted. A naive approach to achieve consensus is to assign a single accepter node who will choose the first value it receives from other proposals. This is an easy solution to implement but it has drawbacks such as single point of failure and high load on the single acceptor.

Paxos algorithm

In Paxos (Fig 2.6), there are three roles: proposers, acceptors, and learners. The consensus procedure starts with a proposer picking up a unique sequence (say n), and sends a prepare message to all acceptors. Whenever an acceptor receives a prepare message with the unique sequence n, it promises not to accept proposals with sequence number smaller than n. This is the first phase of Paxos. It proposes the value and gets the promises that the proposed value is the one being agreed. It is called the propose phase.

The second phase starts when the proposer receives the promises from the majority of the acceptors. The proposer picks up a value v which has the highest proposal number in the received promises and issues accept message for the sequence n and value v to all acceptors. Note that if there is no such value existing yet, the proposer could freely pick up a new value. Whenever an acceptor receives the accept message, it sends back accepted response if it has not responded to any proposals whose sequence is bigger than n. Otherwise, it will send a reject to the proposer. When the proposer receives the responses from a majority of the acceptors, it decides the consensus value v and broadcasts the decision to all learners.

Otherwise, the protocol aborts and restarts. The second phase of Paxos is about to write the highest proposed value to all nodes, which we call it the commit phase.

Paxos algorithm could guarantee that the consensus proceeds even if a minority of nodes fails, which is a perfect solution for the crash-recover failure model. However, with the basic Paxos protocol, liveness might not be achieved since multiple proposer could lead to endless proposals, e.g., proposers compete with each other by proposing a bigger sequence number. A unique leader solves the problem by guaranteeing that a proposal is proposed at a time.

2.3.5 Transactions

Transaction concept was derived from contract law: when a contract is signed, it needs the joint signature to make a deal [24]. In database systems, transactions are abstractions of procedures that the operations are guaranteed by the database to be with all done or nothing done in the presence of failures.

Atomicity, consistency, isolation and durability are four properties of modern transaction systems:

- ***Atomicity*** specifies all or nothing property of a transaction. The successful execution of a transaction will guarantee that all actions of the transaction will be executed. Or if the transaction is aborted, the system would behave as if the transaction was never happened.
- ***Consistency*** specifies that the database is moved from one consistent state to another with respect to the constraint of the database.
- ***Isolation*** describes that the intermediate results among concurrent transactions are not observable.
- ***Durability*** specifies that once a transaction is committed, the result is permanent and can be seen by all other transactions.

Concurrent transactions

Transactions in a database system should be executed as if they were happened sequentially. There are several isolation levels defined correspond to different concurrency levels:

- ***Read uncommitted*** is the lowest isolation level. With read uncommitted isolation, one transaction could see another transaction's uncommitted result. However, a transaction might be aborted when its uncommitted data is read, which is known as 'dirty read'.
- ***Read committed*** level avoids 'dirty read' since it does not allow a transaction to read uncommitted data from another transaction. It, however, could have 'non-repeatable read' exists. Since read committed isolation does not guarantee that the value should not be updated by another transaction prior to the read transaction commits.

- ***Repeatable read*** isolation level avoids 'non-repeatable read' as the read lock is held until the read transaction is committed, so that the value could not be updated by another transaction unless the read transaction is committed. However, repeatable read does not guarantee the result set is static with the same selection criteria. A transaction read twice with the same criteria might get different result sets. This is known as 'phantom reads'.
- ***Serializable*** isolation level guarantees that all interleaved transactions in the system have the equivalent execution results as they are executed in serial.

Distributed transactions

Transactions in distributed systems are far more complicated than transactions in a traditional database system. The atomicity property will not be guaranteed if two or more servers can not reach a joint decision. Two phase commit is the most commonly used commit protocol in distributed transactions, which helps to achieve all or nothing property in distributed transaction systems. Typical concurrency handling for distributed transaction includes pessimistic locking and optimistic concurrency control, which have different pros and cons. And they will be discussed in this section.

Two phase commit

There are two phases in two phase commit protocol: proposal phase and commit phase. There is a transaction manager in the system that gathers and broadcasts the commit decisions. There are also resource managers who propose transaction commits and decide whether the received commit decisions from the transaction manager should be committed or aborted.

In proposal phase, a resource manager, which could also be a transaction manager, proposes a value to commit to the transaction manager. Upon receiving the proposal, the transaction manger broadcasts the proposal to all the resource managers and waits for their replies. The resource managers reply the transaction manager with prepared or not prepared. When the transaction manager has received prepared messages from all of the resource managers, the protocol proceeds to the commit phase. The transaction manager broadcasts the commit message to all resource managers. Then, all the resource managers commit and move to the committed stage.

Normally, $3N - 1$ messages, where N is the number of resource managers, need to be exchanged for a successful execution of the two phase commit protocol. Specifically, a proposal message is sent from one of the resource managers to the transaction manager. Then, the transaction manager sends $N-1$ preparation messages to the resource managers. $N - 1$ replies are received from the resource managers. Lastly, the transaction manager sends the commit message to the N resource managers. The number of messages can be reduced to $3N - 3$ when one of the resource managers acts as the transaction manager.

The protocol aborts when the transaction manager does not receive all the replies from the resource managers. This can be caused by several reasons. For example, the transaction manager can fail, one or more resource managers can fail or messages in the network can

be delayed or lost. When the transaction manager fails, the resource managers which have replied the prepared message are not able to know whether the transaction is committed or aborted. In such case, two phase commit is a blocking protocol. Some other protocols, such as three phase commit[25], solved the blocking in two phase commit.

Concurrency in distributed transactions

Two phase locking

Two phase locking (2PL) utilizes locks to guaranteed the serializability of transactions. There are two types of locks: write-lock and read-lock. The former is associated to resources before performing write on them and the latter is associated to resources before performing read on them. A write lock could block a resource being read or written by other transactions until the lock is released. While a read lock could block a resource being written but will not block a concurrent read from other transactions.

2PL also involves two phases: the expanding phase and the shrinking phase. In the expanding phase, locks are acquired and no locks are released. In the shrinking phase, Locks are released and no locks are acquired. There are also some variants of 2PL. Strict two phase locking states that transactions should be strictly applied with 2PL, and will not release write locks until it is committed. On another hand, read locks could be released in the shrinking phase before the transaction commits. Strong strict 2PL will not release both write and read locks until the transaction commits. Dead lock is an issue with 2PL and needs to be carefully handled.

Optimistic concurrency control

Optimistic concurrency control (OCC) handles the concurrency in distributed transactions from another perspective. In OCC, transactions proceed without locking on resources. Before committing, a transaction validates whether there are other transactions that have modified the resources it has read/written. If so, the transaction rolls back.

In order to efficiently implement the validation phase before transactions commit, timestamps and vector clocks are adopted to record the versions of resources. The nature of OCC provides an improvement on throughput of concurrent transactions when conflicts are not frequent. With the increasing number of conflicts, the abort rate in OCC increases and the system throughput decreases dramatically.

2.3.6 Use Case Storage Systems

OpenStack Swift

OpenStack Swift is a distributed object storage system, which is part of OpenStack Cloud Software [26]. 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 [20] and HDFS [21]. 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.

Cassandra

Cassandra [3] is open sourced under Apache licence. It is a distributed storage system which is highly available and scalable. It stores column-structured data records and provides the following key features:

- **Distributed and decentralized architecture:** Cassandra is organized in a peer-to-peer fashion. Specifically, each node performs the same functionality in a Cassandra cluster. However, each node manages a different namespace, which is decided by the hash function in the DHT. Comparing to Master-slave, the design of Cassandra avoids single point of failure and maximizes its scalability.

- **Horizontal scalability:** The peer to peer structure enables Cassandra to scale linearly. The consistent hashing implemented in Cassandra allows it to swiftly and efficiently locate a queried data record. Virtual node techniques are applied to balance the load on each Cassandra node.

- **Tunable data consistency level:** Cassandra provides tunable data consistency options, which is realized through using different combinations of read/write APIs. These APIs use *ALL, EACH_QUORUM, QUORUM, LOCAL_QUORUM, ONE, TWO, THREE, LOCAL_ONE, ANY, SERIAL, LOCAL_SERIAL* to describe read/write calls. For example, the *ALL* option means the Cassandra reads/writes all the replicas before returning to clients. The explanation of each read/write option can be easily found on Apache Cassandra website.

- **An SQL like query tools - CQL:** the common access interface in Cassandra is exposed using Cassandra Query Language (CQL). CQL is similar to SQL in its semantics. For example, a query to get a record whose id equals to 100 results the same statement in both of CQL and SQL (*SELECT * FROM USER_TABLE WHERE ID= 100*). It reduces the learning curve for developers to use CQLs and get started with Cassandra.

Chapter 3

Related Works

3.1 Distributed Storage Systems

The goal of the thesis is to improve the performance of geo-distributed storage systems, specifically the service latency. We investigate storage systems that store data in a replicated fashion. Data replication guarantees high availability of data and increases system throughput when replicas can be used to serve clients concurrently. However, replicas need to be synchronized to provide a certain level of data consistency, e.g., sequential consistency. In general, the overhead to synchronize replicated data can significantly increase the service latency. It is even worse when the communication costs among replicas increase, which is expected when replicas are deployed globally. We contribute in the design and implementation of replica synchronization mechanisms that minimize replica communication overhead while achieving sequential data consistency. As a result, the service latency of geo-replicated storage systems is improved. We first discuss our related works in general under the topic of data replication and data consistency. Then, we present the related works and compare them with the systems designed in this thesis, i.e., GlobLease, MeteorShower, and Catenae.

3.1.1 Data Replication and Data Consistency

Many successful distributed storage systems have been built by cutting-edge IT companies recently, including Google's Spanner [4], Facebook's Cassandra [3], Microsoft's Azure storage [27], Linkedin'a Voldemort [28], Yahoo!'s PNUTS [5], Hadoop File System [21] and Amazon's Dynamo [6]. In these systems, data are stored in a replicated fashion. Data replication not only provides the systems with higher availability, but also improves the performance of the systems by allowing replicas to serve requests concurrently.

With the expanding of their businesses to a global scale, these large enterprises start to deploy their storage services across a large geographical area. On one hand, this approach improves the availability of the services with the tolerance of even data center failures. On the other hand, it allows data to be served close to its clients, who are located all over the world. Practically, it is realized by replicating data in multiple data centers to obtain a wider geographical coverage. For example, Google Spanner [4] is one of the representative

systems designed for serving data geographically. It serves requests with low latency when the requests can be returned locally. There are also techniques built upon data replication, which are used to improve service latency [29, 30], especially tail latency [31], since the fastest response from any replica can be returned to clients.

However, maintaining data consistency among replicas is challenging, especially when replicas are deployed across such a large area, where communications involve significant delays. There is a large body of work on designing distributed storage systems with different consistency models. In general, a stronger data consistency model associates with a larger replica synchronization overhead. The weakest data consistency model is eventual consistency, where it allows replicas to be inconsistently stored in the system. The only guarantee is that the replicas will converge eventually and the convergence time is not bounded. Typical systems that implement this consistency model are Cassandra [3], Dynamo [6], MongoDB [32], and Riak [33]. Another widely studied data consistency model is casual consistency, which has a stronger semantics than eventual consistency. It ensures that casually related operations are guaranteed to be executed in a total order in all replicas [34, 35, 36]. Stronger than causal consistency, there is sequential consistency. It guarantees that all operations appear to have the same total order on all replicas [37, 38, 12, 39]. There are also works [12, 6, 3, 34, 35] that tradeoff system performance with data consistency guarantees according to different usage scenarios. Under the scenario of geo-replication, there are works [40, 41, 4, 12, 42] that optimize the efficiency of using cross data center communication while keeping data consistent.

3.1.2 Related Works for GlobLease

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 [22], Pastry [43], Symphony [23]. The most common replication schema implemented on top of DHTs are successor-lists, multiple hash functions or leaf-sets. Besides, ID-replication [44, 45] and symmetric replication [46] are also discussed in literature.

GlobLease takes advantage of DHT's reliable routing and self-organizing structure. It 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 [44] 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 prioritized by judiciously selecting links with low latency according to the system deployment.

Lease-based Consistency

There are many applications of leases in distributed systems. Leases are first proposed to deal with distributed cache consistency issues in [47]. The performance of lease-based con-

sistency is improved in [48]. Furthermore, leases are also used to improve the performance of classic Paxos algorithm [49]. In addition, they are also studied to be applied in preserving ACID properties in transactions [50, 51, 52]. In sum, leases are used to guarantee the correctness of a resource in a time interval. Since leases are time-bounded assertions of resources, they facilitate the handling of failures, which is desired in a distributed environment.

In GlobLease, we explore the usage of leases in maintaining data consistency in a geo-replicated key-value store. GlobLease tradeoffs a small portion of low latency read requests to reduce a large portion of high latency write requests under a read dominant workload. Comparing to master-based data consistency algorithm, GlobLease provides a higher level of fault tolerance.

Asynchronous Data Propagation

GlobLease employs an asynchronous data propagation layer to achieve robustness and scalability while reducing the communication overhead of synchronizing the replicas across multiple geographical areas. Similar approach can be found in the message broker of Yahoo! Pnuts [5]. Master-slave replication is the canonical paradigm [53, 54]. GlobLease extends the master-slave paradigm with the per key mastership granularity. This allows GlobLease to migrate the master of the keys close to their most written places in order to reduce most of the write latencies.

A typical asynchronous update approach can be found in Dynamo [6] with epidemic replication. It allows updates to be committed in any replicas with any orders. The divergence of the replicas will be eventually reconciled using a vector clock system. However, irreconcilable updates and roll backs may happen in this replication mechanism, which exposes high logic complexity for the upper applications.

3.1.3 Related Works for MeteorShower

Global Time

Having a global knowledge of time helps to reduce the synchronization among replicas since operations can be naturally ordered based on global timestamps. However, synchronizing time in distributed systems is extremely challenging [55], which leads us to the application of loosely synchronized clocks, e.g., NTP [56]. Loosely synchronized clocks are applied in many recent works to build distributed storage systems that achieve different consistency models from casual consistency [36, 57, 58] to linearizability [4]. Specifically, GentleRain [36] uses loosely synchronized timestamp to causally order operations, which eliminates the need for dependency check messages. Clock-SI [57] exploits loosely synchronized clocks to provide timestamps for snapshots and commits in partitioned data stores. Spanner [4] employs bounded clocks to execute transactions with reduced delays while maintaining the ACID property.

MeteorShower assumes a bounded loosely synchronized time on each server. It exploits the loosely synchronized time in a different manner. Specifically, a total order of write requests is produced using the loosely synchronized timestamp from each server. Then, read

requests are judiciously served by choosing slightly stale values but satisfying the sequential consistency constraint. It is novel and different from the state of the art approaches by exploiting slightly stale values in the global time-line. Essentially, we use bounded loosely synchronized time to push the boundary of serving read requests while preserving data consistency.

Replicated Log

Replicated logs are first proposed by G.T.Wuu et al. [59] to achieve data availability and consistency in an unreliable network. The concept of replicated log is still widely adopted in the design of modern distributed storage systems [38, 42, 3] or algorithms [60, 61]. For example, Megastore [38] applies replicated log to ensure that a replica can participate in a write quorum even as it recovers from previous outages. Helios [42] uses replicated log to perceive the status of remote nodes, based on which transactions are scheduled efficiently. Chubby [60] can be implemented using replicated logs as its message passing layer.

MeteorShower employs replicated logs for the similar reason: perceiving the status of remote replicas. However, MeteorShower exploits the information contained in the replicated logs differently. The information captured in the logs are the updates of replicas in remote MeteorShower servers. MeteorShower uses this information to construct a slight stale history of replicas stored in remote servers marked with loosely synchronized timestamp. Then, MeteorShower is able to judiciously serve requests with slightly stale values while preserving sequential data consistency, which significantly improves request latency.

Catenae also employs replicated logs in its backend. The logs are used for two purposes: transaction distribution and transaction validation. The transaction distribution payload is exploited similarly to MeteorShower. It provides an aggregated and consistent input sequence of transactions received from all replicas, which facilitates the execution of transactions. On the other hand, the transaction validation payload is used to reach a consensus on the transaction execution results among data centers. It is similar to the usage scenario in [60], where a consensus is reached among replicas (processes).

3.1.4 Related Works for Catenae

Geo-distributed Transactions

Previously, transactions are supported by traditional database systems and usually data is not replicated. To support transactions in a large scale on top of a storage system where data is geographically replicated is challenging. There are geo-distributed transaction frameworks that are built on replicated commit [40], paxos commit [41, 4], parallel snapshot isolation [62], and deterministic total ordering based on prior analysis of transactions [63, 64].

Catenae supports serializable transactions for geo-distributed data stores. It differs from the existing approaches in two ways. First, it extends transaction chains [7] to achieve deterministic execution of transactions without prior analysis of transactions. This improves transaction execution concurrency, removes bottleneck and single point of failure. Second, a replicated log style protocol is designed and implemented to coordinate transaction executions in multiple DCs with reduced RTT rounds to commit a transaction.

Comparing to the original transaction chain algorithm proposed in [7], Catenae extends the algorithm for replicated data stores. Specifically, Catenae allows multiple versions of a record in chain servers to enable read-only transactions and support transaction catch ups in case of replica divergence. The extended transaction chain algorithm manages the concurrency among transactions in the same DC while an epoch boundary protocol, which is based on loosely synchronized clocks, controls the execution of transactions among DCs.

3.2 Elasticity Controllers

Elasticity allows a system to scale up (out) and down (in) in order to offer predictable (stable) performance with reduced provisioned resources in the presence of changing workloads. Usually, elasticity is discussed under two perspectives, i.e., vertical elasticity and horizontal elasticity. The former case scales a system within a host, i.e., a physical machine, by changing the allocated resources using a hypervisor [65, 66, 67]. On the other hand, the latter method resizes a system by adding or removing VMs or physical machines [30, 68, 69, 70, 71]. In this work, we focus on the study of elasticity controllers for horizontal scaling.

We first provide an overview of achieving elasticity, especially on storage systems, from the perspective of industry and research. Then, we discuss the related techniques to build an elasticity controller. Lastly, we compare the approaches presented in this thesis with the state-of-the-art approaches in some specific aspects.

3.2.1 Overview of Practical Approaches

Most of the elasticity controllers available in public Cloud services and used nowadays in production systems are policy based and rely on simple if-then threshold based triggers. Examples of such systems include Amazon Auto Scaling [72], Rightscale [16], and Google Compute Engine Autoscaling [73].

The wide adoption of this approach is mainly due to its simplicity in practice as it does not require pre-training or expertise to get it up and running. Policy based approaches are suitable for small-scale systems in which adding/removing a VM when a threshold is reached (e.g., CPU utilization) is sufficient to maintain the desired SLO. For larger systems, it might be non-trivial for users to set the thresholds and the correct number of VMs to add/remove.

3.2.2 Overview of Research Approaches

Most of the advanced elasticity controllers, which go beyond a simple threshold based triggers, require a model of the target system in order to be able to reason about the status of the system and decide on control actions needed to improve the system. The research focus in this domain is on developing advanced control/performance models or novel procedures during control flows.

Researches in this realm can be broadly characterized as designing elasticity controller for scaling stateless services [74, 75, 76, 19] and for scaling stateful services, such as dis-

tributed storage systems [30, 70, 68, 18]. The major difference that distinguishes an elasticity controller for scaling stateful services from its counterpart is the consideration of state transfer overhead during scaling. It makes the design of such an elasticity controller more challenging. Works in this area include ElastMan [70], SCADS Director [30], scaling HDFS [68], ProRenata [69], and Hubbub-scale [71]. SCADS Director [30] is tailord for a specific storage service with pre-requisites that are not common in storage systems, which is fine-grained monitoring and migration of storage buckets. ElastMan [70], uses two controllers in order to efficiently handle diurnal and spiky workloads, but it does not consider the data migration overhead during scaling storage systems. Lim et al. [68] have designed a controller to scale Hadoop Distributed File System (HDFS), which uses CPU utilization as input metric. They have shown that CPU utilization highly correlates request latency and it is easier for monitoring. Concerning data migration, they only rely on the data migration API integrated in HDFS, which only manages the data migration speed in a coarse-grained manner. ProRenaTa [69] minimizes the SLO violation during scaling by combining both proactive and reactive control approaches but it requires a specific prediction algorithm and the control model needs to be trained offline. Hubbub-Scale [71] and Augment Scaling [77] argue that platform interference can mislead an elasticity controller during its decision making, however, the interference measurement needs the access of many low level metrics, e.g. cache counters, of the platform.

3.2.3 Overview of Control Techniques

Recent works on designing elasticity controllers can be also categorized by the control techniques applied in the controllers. 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 [15] and RightScale [16]. 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 is usually used to understand the application behaviors by building empirical models. Simon et al. [17] 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. Scads director [30] presents a performance model, which is obtained empirically, that correlates the percentile request latency with the observed workload in terms of read/write request intensity.

Ali-Eldin et al. [19] uses the queueing theory to model a Cloud service and estimate 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 [70, 68]. ElastMan [70] employs two control models to tackle with two different patterns in the workload.

Specifically, a feed-forward module is designed to incorporate workload spikes while a feedback module is implemented to process regular diurnal workload. Lim et al. [68] uses CPU utilization as the monitored metrics in a classic feedback loop to achieve auto-scaling.

Recent approaches using time-series analysis to achieve auto-scaling are [78, 76, 74]. Predictions allow elasticity controllers to react to future workload changes in advance, which leaves more time to reconfigure the provisioned systems. Specifically, Agile [76] proves that it is accurate to use wavelets to provide a medium-term resource demand prediction. Nilabja et al. [78] adapts second order ARMA for workload forecasting under the World Cup 98 workload. CloudScale [74] presents on-line resource demand prediction with prediction errors corrected.

3.2.4 Related Works for BwMan

Controlling Network Bandwidth

The dominant resource consumed by data migration process is the network bandwidth. There are different approaches to allocate and control network bandwidth, 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) approaches [79]; and a combination of both. 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 [80].

Recent works have investigated the correlation between performance and network bandwidth allocated to an application. For example, a recent work of controlling the bandwidth on the edge of the network is presented in EyeQ [81]. 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. Another work of controlling bandwidth allocation is presented in Seawall [82]. Seawall uses reconfigurable administrator-specified policies to share network bandwidth among services and enforces the bandwidth allocation by tunnelling traffic through congestion control, point to multi-points, edge to edge tunnels. A theoretical study of the challenges regarding network bandwidth arbitration in the Cloud is presented in [83]. It has revealed the needs and obstacles in providing bandwidth guarantees in a Cloud environment. Specifically, it has identified of a set of properties, including min-guarantee, proportionality and high utilization, in order to pioneer the design of bandwidth allocation policies in the Cloud.

In contrast, BwMan is a simpler yet effective solution. We let the controller itself dynamically decide the bandwidth quotas allocated to each services through statistically learnt models. These models correlate the desired service level objective (QoS) with the minimum bandwidth requirement. Administrator-specified policies are only used for trade-offs when the bandwidth quota is not enough to support all the services on the same host. Dynamic bandwidth allocation allows BwMan to support the hosting of elastic services, whose demand on the network bandwidth varies depending on the incoming workload.

3.2.5 Related Works for ProRenaTa

Modelling Data Migration

Elasticity of storage systems requires data to be properly migrated while scaling up (out) and down (in). The closest works that concern this specific issue are presented in [68, 30, 84, 85]. To be concise, Scads director [30] tries to minimize the data migration overhead associated with the scaling by arranging data into small data bins. However, this only alleviates the SLO violations instead of eliminating them. In Lim's work [68], a data migration controller is designed. However, it only uses APIs limited to HDFS to coarsely arbitrate between SLO violations and system scaling speed. FRAPPE [84, 85] alleviates service interruption during system reconfigurations by speculatively executing requests.

ProRenaTa differs from the previous approaches in two aspects. First, ProRenaTa combines both reactive or proactive scaling techniques. Reactive controller gives ProRenaTa better scale accucacy while proactive controller provides ProRenaTa enough time to handle the data migration. The complementary nature of both approaches provide ProRenaTa with stricter SLO commitment and higher resource utilization. Second, to our best knowledge, when scaling a storage system, the previous approaches do not explicitly model the cost of data migration. Instead, ProRenaTa explicitly manages the scaling cost (data migration overhead) and the scaling goal (deadline to scale). Specifically, it first calculates the data that need to be migrated in order to accomplish a scaling decision. Then, based on the monitoring of the spare capacity in the system, ProRenaTa determines the maximum data migration speed without compromising the SLO. Thus, it knows the time to accomplish a scaling decision under the current system status. And this information is judiciously applied to schedule scaling activities to minimize the provisioning cost.

3.2.6 Related Works for Hubbub-scale

Performance Interference

DejaVu [86] relies on an online-clustering algorithm to adapt to load variations by comparing the performance of a production VM and a replica of it that runs in a sand-box to detect interference and learns from previous allocations the number of machines for scaling. A similar system, DeepDive [87], first relies on a warning system running in the VM to conduct early interference analysis. When the system suspects that one or more VMs are subjected to interference, it clones the VM on-demand and executes it in a sandboxed environment to detect interference. If interference does exist, the most aggressive VM is migrated on to another physical machine. Both these approaches require a sand boxed environment to detect interference as they do not consider the behaviour of the co-runners. Stay-Away [88] is a dynamic reconfiguration technique that throttles batch application proactively to minimize the impact of performance interference and guarantee QoS of latency critical services.

Hubbub-scale models contention from the behaviour of the co-runners. Our solution instead shows ways to quantify the interference-index and how this can be used to perform reliable elastic scaling.

Another class of work has also investigated providing QoS management for different applications on multicore [89, 90, 91]. While demonstrating promising results, resource partitioning typically requires changes to the hardware design, which is not feasible for existing systems. Recent efforts [92, 93, 94] demonstrate that it is possible to accurately predict the degradation caused by interference with prior analysis of workload. In [95] the application is profiled statically to predict interference and identify safe co-locations for VMs. It mainly focuses on predicting which applications can be co-run with a given application without degrading its QoS beyond a certain threshold. The limitation of static profiling introduces a lack of ability to adapt to changes in application dynamic behaviour. Paragon [96] tries to overcome the problem of complete static profiling by profiling only a part of the application and relies on a recommendation system, based on the knowledge of previous execution, to identify the best placement for applications with respect to interference. Since only a part of the application is profiled, dynamic behaviours such as phase changes and workload changes are not captured and can lead to a suboptimal schedule resulting in performance degradation.

Hubbub-scale, in contrast, relies on quantifying contention in real time, allowing it to adapt to workload and phase changes.

Chapter 4

Achieving High Performance on Geographically Distributed Storage Systems

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 some of the above desire properties are presented in [3, 6, 5, 4].

Usage Scenario

Performance of storage systems can be largely leveraged using data replication. Replication provides a system to handle workload simultaneously using multiple replicas, thus achieving higher system throughput. Furthermore, the availability of data is increased when maintaining multiple copies of data 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. The overhead of maintaining data consistency is even more obvious when the system is geo-replicated, where the communications among replicas experience relatively long latency.

4.1 GlobLease

We approach the design of high performance geographically distributed storage system with the handling of read dominant workload, which is one of the most common access patterns in WEB 2.0 services. A read dominant workload has some characteristics. For example, it is often the case that popular contents attract significant percentage of readers. It causes skewness in access patterns. Moreover, the workload increase caused by the popular contents is usually spiky and not long-lasting. 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. Under the scenario of skewed and spiky read dominant access pattern, we propose our geographically distributed storage solution, namely, **GlobLease**. It is designed to be a consistent and elastic storage system under the usage of read dominant workload. Specifically, it achieves low latency read accesses in a global scale and efficient write accesses in one area with sequential consistency guarantees.

4.1.1 GlobLease at a glance

GlobLease assumes that data replicas are deployed in different data centers and the communication among them involve significant latency. Read dominant workloads are initiated from each data center while write workloads regarding specific data items are initiated in one of the data centers. In other words, GlobLease targets the usage scenario where there are multiple readers and a single writer for a specific data item.

Under this usage scenario, GlobLease implements sequential data consistency. It extends the paradigm of master-based replication, which is designed to efficiently handle read dominant workload. In GlobLease, masters do not actively keep the replicated data up to date, which significantly reduces the latency of writes comparing to the traditional master-based approach. On the other hand, masters issue leases along with the updates to replicas when they need to serve read requests. Leases give time-bounded rights to slave replicas to handle reads. Considering the skewed pattern of read requests, GlobLease yields excellent performance. Furthermore, the time-bounded assertion in lease provides a higher level of fault tolerance comparing to the tradition master-based approach. Specifically, lease mechanism allows write requests to proceed after the expiration of the lease in a failed replica. Evaluation of GlobLease in a multiple data center setup indicates that GlobLease tradeoff a small portion of low latency read requests, as the leasing overhead, to reduce a large portion of high latency write requests. As a result, GlobLease improves the average latency of read and write requests while providing a higher level of fault tolerance.

The rest of this section presents the detailed design of GlobLease.

4.1.2 System Architecture of GlobLease

Background knowledge regarding DHTs, data availability, data consistency, and system scalability of a distributed storage system can be obtained in Chapter 2 or research works [22, 97, 41].

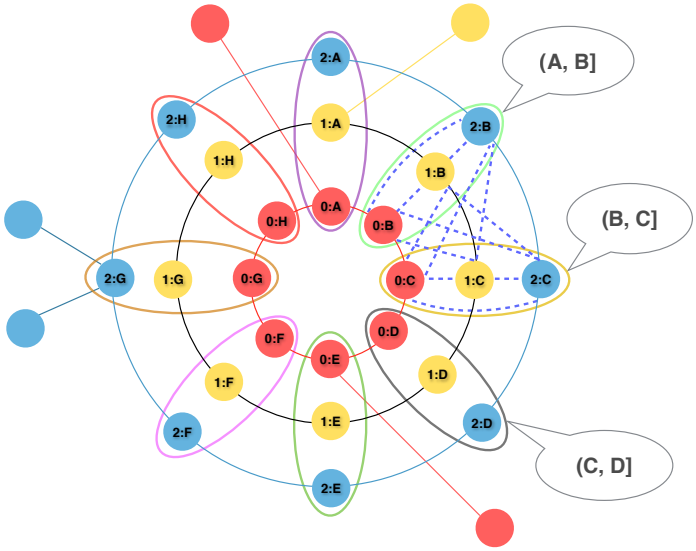


Figure 4.1 – GlobLease system structure having three replicated DHTs

GlobLease is constructed with a configurable number of replicated DHTs shown in Fig. 4.1. Each DHT maintains a complete replication of the whole namespace and data. Specifically, GlobLease forms up replication groups across the DHT rings, which scales out the limitation of successor list replication [44]. 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 events using pre-defined algorithms in each node to stabilize the overlay [22, 23]. 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 [6].

GlobLease Nodes

Each DHT Ring is given a unique ring ID shown as numbers in Fig. 4.1. 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. 4.1. 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. 4.1 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. 4.1 are called *affiliated nodes*. They are used for fine-grained management of replicas, which are explained in Section 4.1.4.

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.

GlobLease Links

Basic Links: The basic links include three kinds of links. Links connecting a node's predecessor and successor within the same DHT are called *local neighbour links* shown as solid lines in Fig. 4.1. 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 [22, 23] 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. The number of routing links maintained in each node is configurable depending on the node's memory size. When reaching the maximum number of routing links, the least recently used link is replaced.

4.1.3 Lease-based Consistency Protocol in GlobLease

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.

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 replicas of the data that have valid leases are able to

serve read accesses. Second is the time bound, which allows a lease to expire when the valid time period has passed. The time bound of lease is essential in handling possible failures on servers storing slave replicas. Specifically, if an operation requires the update or invalidation of leases on slave replicas, which cannot be completed due to failures, the operation waits until those leases expire naturally.

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 valid leases 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 validity of leases are checked based on lease records, which are created on masters whenever leases are issued to non-masters. 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.

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 follows the active update.

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, data availability is compromised in this case. Both passive and active updates are implemented with separate APIs in GlobLease. It can be tuned by applications encountering different workload patterns or having different requirements.

Leasing Period

The length of a lease is configurable in our system design. At the moment, the length of a lease is implemented with per node granularity. Further, it can be extended to per 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 workload cooperates better with shorter leases (less overhead of lease updates, especially when 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 leasing records and update them if necessary. This indicates that the leasing period recorded on the master should be the same with or last longer than the corresponding leasing period on the non-masters. Since it is extremely hard to synchronize the time in a distributed system [97], 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 sequential data consistency guarantee.

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 [98] 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 non-master nodes in the replication group 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 on a non-master node is not correctly changed, any

message from this node sent to the old master will trigger an echo message, which contains the information regarding the correct master. If the new master forwards a write to the old master, it means that the new master fails to acknowledge its mastership. In this case, the old master restarts the two phase master migration protocol.

Master failure recovery is implemented based on the assumption of fail stop model [99]. There are periodical heartbeat messages from the non-master nodes in the replication group to check the status 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. If the node that proposes a master election is able to collect a majority of promises from other nodes, it runs the second phase to Paxos to change its status to a master. We use node ids to break ties during the majority votes of the Paxos process. Any non-master node that fails to recognize the new master will be guided through the echo message described above.

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, a new write is always processed with the active update approach in order to create a record of the written key on non-master nodes, which ensures the correctness in the lookup of the data when clients contact the non-master nodes for read accesses. Conversely, updates of a key can be handled either using the active or passive approach. In either case, a write or an update on a non-master node associates with a lease, and the information regarding the lease is maintained in the master node. The information of the lease is referred, when another write arrives at the master node, to decide whether the lease is still valid. Algorithm 1 and Algorithm 2 present the pseudo codes for processing read and write requests.

4.1.4 Scalability and Elasticity of GlobLease

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 resources. However, this form of scalability is associated with large overhead, including reconfiguration of multiple ring overlays, division of the namespace, rebalancing of the data, and the churn of the rout-

Algorithm 1 Pseudo Code for Read Request

```
Data: Payload of a read request, msg
Result: A value of the requested key is replied
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
```

ing table cached in each node's memory. Furthermore, this approach is feasible when the growing workload is long lasting and preferably in a uniform manner. Thus, when confronting intensive and transient increase of workloads, especially when the access pattern is skewed, this form of elasticity might not be enough. We have extended the system with a fine-grained elasticity with the usage of affiliated nodes.

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. We do not distinguish the affiliated nodes at the moment, since they are deployed in the same location as their host standard node. In the future, we would like to enable the deployment of affiliated nodes in different locations to serve small scale transient workloads.

Affiliated nodes are designed as lightweight processes that can join/leave system overlay by only interacting with a standard node. In addition, since only highly requested data

Algorithm 2 Pseudo Code for Write Request

```
Data: Payload of a write request and write mode, msg, MODE
Result: An acknowledgement for the write request
// 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 ∈ replicationGroup do
        checkResult = n.issueLease(server, msg.key, msg.value, lease)
    end
    while retries do
        // ACKServer:  server lists that have acknowledged the update.
        // leaseExpired:  server lists that do not have valid leases
        ACKServer = getACKs(checkResult)
         noACKServer = replicationGroup-ACKServer
         leaseExpired = getLeaseExp(leaseRec)
         if noACKServer ∈ leaseExpired &&
         sizeOf(noACKServer) < sizeOf(replicationGroup)/2 then
            lazyUpdate(noACKServer, msg)
             n.writeValue(msg.key, msg.value)
             for server ∈ ACKServer do
                n.putLeaseRecord(server, msg.key, lease)
            end
            return SUCCESS;
        else
            for server ∈ noACKServer do
                checkResult += n.issueLease(server, msg.key, msg.value, lease)
            end
            retries -= retries
        end
    end
    return FAIL;
end
```

items are replicated in affiliated nodes, the data migration overhead is negligible. Thus, the addition and removal of affiliate nodes introduce very little overhead and can be completed in seconds. 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.

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 4.1.3. 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.

4.1.5 Evaluation of GlobLease

We evaluate the performance of GlobLease under different intensities of read/write workloads in comparison with Cassandra [3]. 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.

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 4.1.

As a baseline experiment, Cassandra is deployed using the same amount of EC2 instances with the same instance type 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 (READ-ONE-WRITE-ALL), which is essentially the same as traditional master-based approach. For write dominant workload, Cassandra writes to one replica and reads from all replicas (READ-ALL-WRITE-ONE). With this setup, Cassandra is able to achieve its best performance since only one replica is needed to process a read or write request in read-dominant or write-dominant workload. Note that Cassandra only achieves casual consistency using the naive implementation of READ-ONE-WRITE-ALL for a read-dominant workload,

Table 4.1 – 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 4.2 – 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

which is less stringent than GlobLease. Even so, GlobLease outperforms Cassandra as shown in our evaluations.

We have modified Yahoo! Cloud Serving Benchmark (YCSB) [100] to generate either uniform random or skewed workloads to GlobLease and Cassandra. YCSB clients are deployed in an environment described in Table 4.1 and parameters for generating workloads are presented in Table 4.2.

Varying Load

Fig. 4.2 presents read performance of GlobLease with comparison to Cassandra under a 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 increasing usage efficiency of each 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

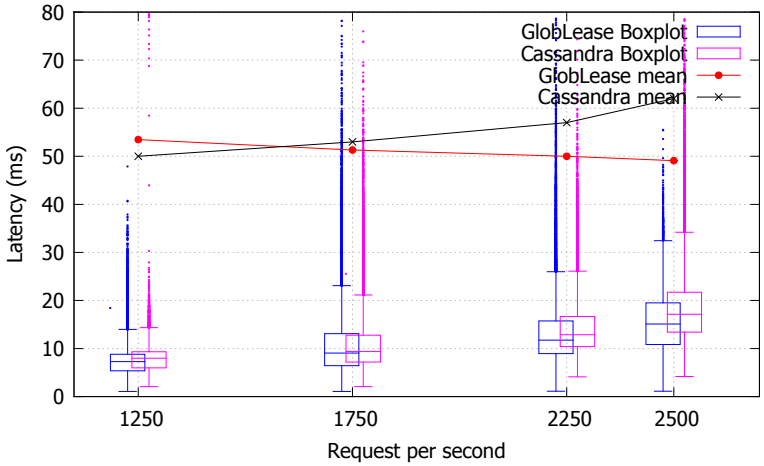


Figure 4.2 – Impact of varying intensity of read dominant workload on the request latency

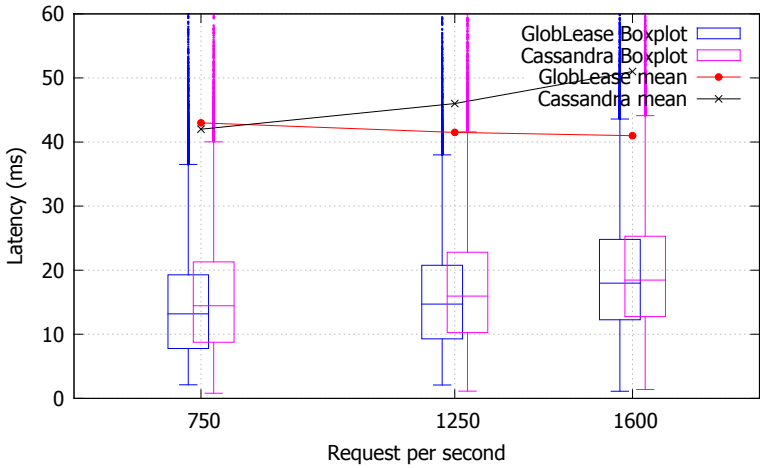


Figure 4.3 – Impact of varying intensity of write dominant workload on the request latency

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. 4.2 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. These high latency requests constitute 5% to 10% of the total requests in our evaluations. We discuss these outliers in Fig. 4.4 and Fig. 4.5. 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 (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

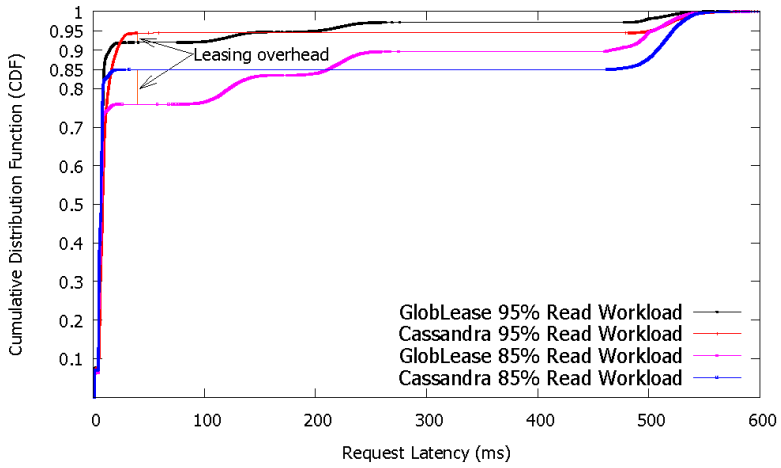


Figure 4.4 – Latency distribution of GlobLease and Cassandra under two read dominant workloads

(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. 4.3. 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. 4.4 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 experiences 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 95% and 85% 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 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

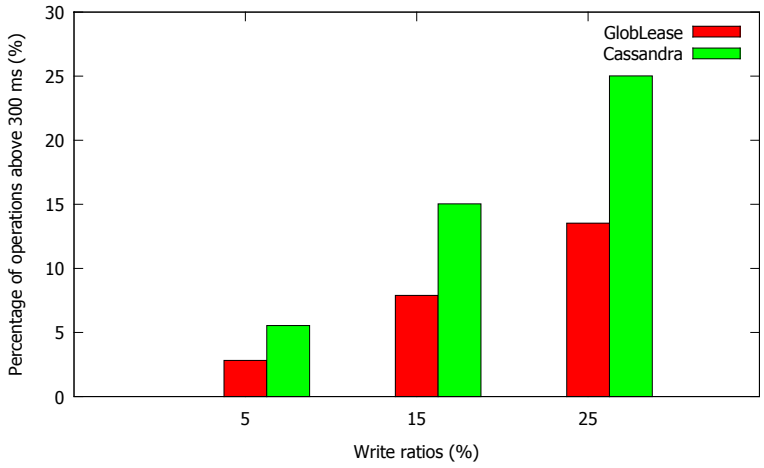


Figure 4.5 – High latency requests

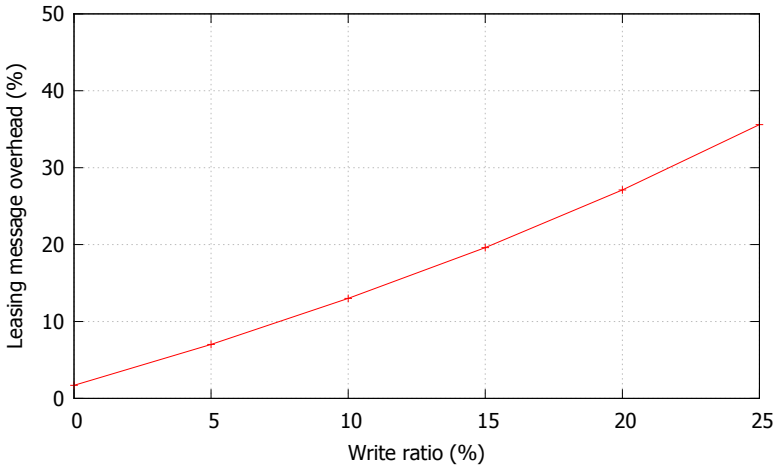


Figure 4.6 – Impact of varying read:write ratio on the leasing overhead

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 requests are processed with global communication latency ranging from 0 ms to 500 ms depending on the number of non-master replicas with valid leases. 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. 4.4. Fig. 4.5 zooms in on the high latency requests (above 300 ms) 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 sensitive to a large portion of high latency requests.

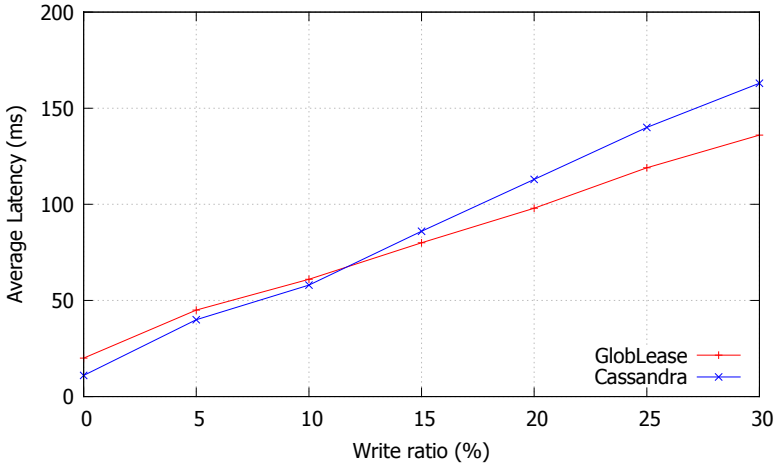


Figure 4.7 – Impact of varying read:write ratio on the average latency

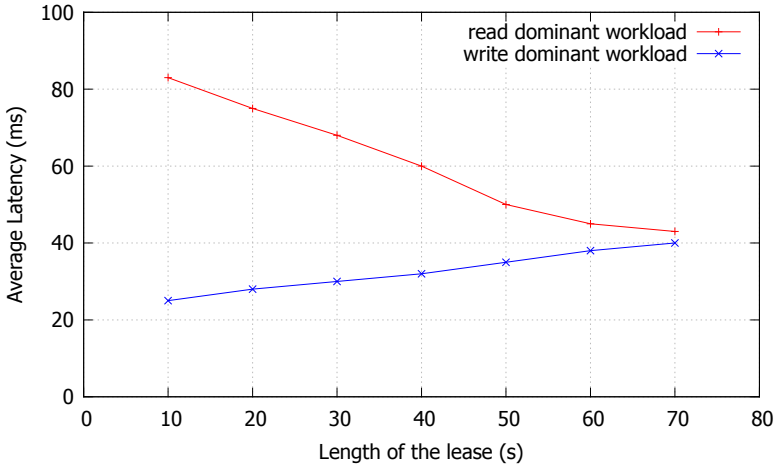


Figure 4.8 – Impact of varying lengths of leases on the average request latency

Lease Maintenance Overhead

In Fig. 4.6, we evaluate lease maintenance overhead in GlobLease. The increasing portion of write requests impose more lease maintenance overhead on GlobLease since writes trigger lease invalidations and cause future lease renewals. The y-axis in Fig. 4.6 shows the extra lease maintenance messages comparing to Cassandra under throughput of 1000 request per second and 60 second leasing period. 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-

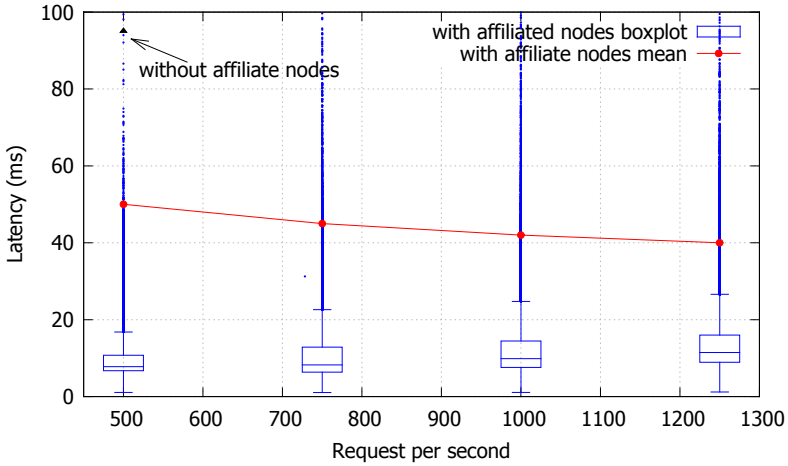


Figure 4.9 – Impact of varying intensity of skewed workload on the request latency

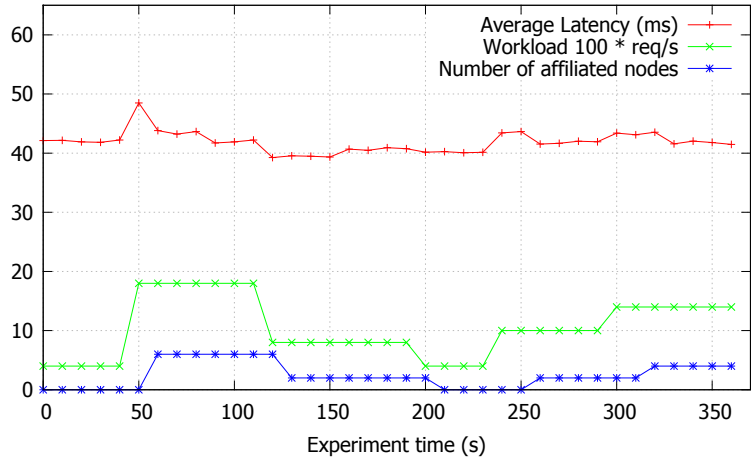


Figure 4.10 – Elasticity experiment of GlobLease

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 Cassandra when latency between data centers vary. GlobLease benefits from smaller latency among close data centers as shown in Fig. 4.4 and Fig. 4.5.

Varying Read/Write Ratio

In Fig. 4.7, 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. 4.7, 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. 4.6. However, GlobLease outperforms Cassandra when the write ratio grows. This is explained

in Fig. 4.5 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.

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 second. Fig. 4.8 shows that, with the increasing length of leases, 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.

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. 4.9 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.

Elasticity with Spiky and Skewed Workload

Fig. 4.10 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. 4.10 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 [70, 30, 68].

4.1.6 Summary and Discussions of GlobLease

Lease enables cache-style low latency read operations from multiple geographical locations. The expiration feature of lease enables non-blocking write operations even in the scenario of non-master node failures. Failure of a master node are handled by a two-phase election within the replication group. Even though all the failure cases are handled in GlobLease, they are not tackled efficiently in case of frequent failures. In other words, GlobLease works efficiently when there is no failures in the system. In the presence of failures, a read request could be delayed up to 3 RTTs among all nodes. Specifically, 2 RTTs are used to elect a new master while another RTT is required to request a lease from the new master. Similarly, a write request could suffer the same as the read request with a delay up to 3 RTTs. Essentially, master nodes become the single point of failures that significantly affect the performance of GlobLease in the presence of failures. This is a well-known drawback of master-based replicated storage systems.

On the other hand, GlobLease is not symmetric in handling read and write workloads. As a result, the benefit of GlobLease could be suppressed when there is a significant amount of writes and they are uniformly distributed in all locations. It means that writes on a specific key are not always initiated from the same geographical location. As a result, migrations of master nodes no longer improve the performance of writes. Furthermore, frequent writes invalidate/update leases within short intervals, leaving little time for leases to serve read requests locally. Consequently, the performance of reads also suffer.

With the consideration of the above two aspects, we proceed our research on majority quorum based distributed storage systems. Examples of such systems include Cassandra [3], Voldemort [28], Dynamo [6], Riak [33] and ElasticSearch [101]. The handling of requests does not involve the concept of masters, which eliminates the single point of failure. Furthermore, reads and writes are usually accomplished with the involvement of all the replicas. And, not all the responses of the replicas are needed to return a request. It provides more robustness in the presence of failures. Since these systems treat read and write requests symmetrically, they are more suitable for handling a read/write mixed workload.

4.2 MeteorShower

A classic approach to handle read/write mixed workload in a distributed storage system with consideration of data consistency and service availability starts with the concept of majority quorum. A quorum consists of all the replicas of a data item. The total number of replicas of a data item is also known as the replication degree (n). Let us assume that a read request on data item X is served by r number of replicas and a write request on X is served by w number of replicas, then the minimum requirement to achieve sequential data consistency is $r + w > n$.

Typically, a read/write request is sent to all replicas in order to obtain a sufficient amount of replies to satisfy the requirement. This approach yields adequate performance

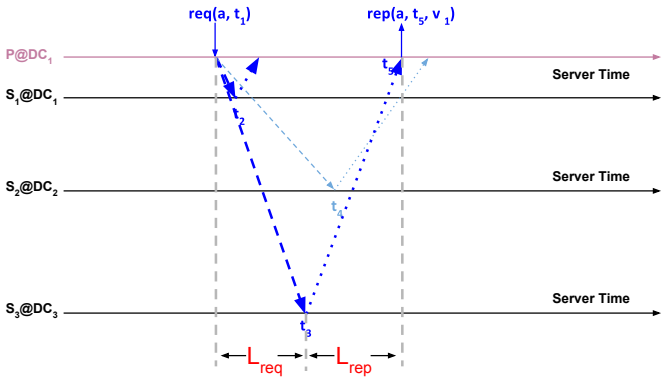


Figure 4.11 – Typical Quorum Operation

when replicas are relatively close to each other, e.g., in the same data center. However, this is not the case in a geo-replicated storage system, where replicas are hosted in different data centers for performance and availability purposes. Communications with highly distributed replicas incur significant delays. Thus, using majority quorums to achieve sequential data consistency when replicas are deployed geographically leads to significant increase in request latency.

4.2.1 The Insight

In this section, we investigate the cause and provide insights on high request latency while using majority quorums in a geo-distributed data store.

The setup: We assume a replicated data store deployed in multiple data centers and replicas are not hosted in the same data center. A data center hosts many storage instances, which are responsible of hosting a specific part of the total namespace. A storage instance manages various data items, which are replicated among different storage instances hosted in different data centers. Client requests are received and returned by storage instances from the closest data center.

A scenario: Figure 4.11 illustrates a scenario where the requested data item is replicated in three data centers, i.e. DC_1, DC_2 and DC_3. A client close to DC_1 has issued a read request, which has been received by one of the storage instances in DC_1, which then acts as a proxy for the request. Figure 4.11 illustrates a typical scenario of processing a majority quorum read. It means that the read request can be returned when at least two of the three replicas acknowledge. In our case, DC_1 and DC_3 are faster in returning the read request. After receiving the replies from DC_1 and DC_3, the proxy returns the read request. We use this representative scenario to explain the causes and insights of high request latency.

The cause: The essential cause for high request latency is the network delay on requesting item values from all replicas (e.g., L_{req} in Figure 4.11) and replying the requests (shown as L_{rep} in Figure 4.11). Essentially, L_{req} is paid to ask all the replicas, involved in a specific request, to report their current status. L_{rep} is paid to deliver the status of all

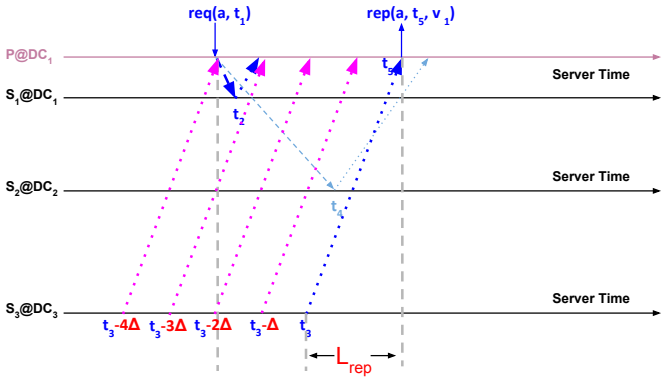


Figure 4.12 – Typical Quorum Operation

replicas to the requester, i.e. the proxy.

The insight: L_{rep} is hard to remove since we would like to read an updated copy of data and preserve data consistency. However, L_{req} can be removed if all the replicas actively exchange their status. In this case, the requester just needs to wait for status reports from replicas. Figure 4.12 continues the above scenario with the removal of the request message from the proxy to DC_3 while adding periodic status messages from DC_3 to the proxy. If the proxy waits for the status message sent from DC_3 at t_3, the reply of the read request would be the same as the previous scenario.

Research questions: Given Figure 4.12, we would raise the question that whether the proxy can return the read request when receiving the status message sent from DC_3 earlier than t_3. What would happen if the read request is returned after receiving the status message from DC_3 sent at $t_3 - \Delta t$, or $t_3 - 2 * \Delta t$ as shown in Figure 4.12? Are we still preserving the same data consistency level as the previous algorithm? If so, what is the constraint on the status messages for the proxy to return the request.

A promising result: If the proxy is able to preserve the same data consistency level by analysing the status message from DC3 sent at $t_3 - 4 * \Delta$, then the request is able to be returned without any cross data center delays.

In the following section, we focus on solving the research question and explaining the constraints to achieve the promising result.

4.2.2 Distributed Time and Data Consistency

System Model

We assume a system consisting of a set of storage instances connected via a communication network. Messages in this communication network can be lost or delayed, but cannot be corrupted. Each storage instance replicates a portion of the total data in the system. There is an application process running at each storage instance. The application process has the whole namespace mapping and addresses of replicas. The process is able to access the data stored locally.

Furthermore, each process has the access to its local physical clock. Formally, a clock is a monotonically increasing function of real time [102]. We assume that the time drift of all the clocks in all the servers are bounded to an error ε. It means that the clock difference of any two instances' clock times t_1 and t_2 is bounded to ε, i.e., $|t_1 - t_2| \leq \varepsilon$.

To sum up, the application process at each instance is able to perform the following operations:

- readRequest $read(key, timestamp, source)$: the application process initiates a read request on a data item;
- readLocal $readLocal(key, timestamp, source)$: the application process performs a local read on key;
- writeRequest $write(key, value, timestamp, source)$: the application process initiates a write request on a data item;
- writeLocal $writeLocal(key, value, timestamp, source)$: the application process updates the value of data item key to $value$ locally if $timestamp$ is larger than the timestamp associated with the locally stored data item key;
- broadcastStatus: the application process broadcasts its local updates to application processes that manage peer replicas periodically;
- updateStatus: the application process receives broadcasts from remote replicas and updates its view on remote replicas;
- time: the application process is able to access its local clock.

Sequential Consistency

In order to discuss the research question presented in the previous section, we choose to implement sequentil data consistency, which is a widely used data consistency model. We provide the formal definition of sequential consistency (SC) based on the definition given by Hagit Attiya et. al [103].

Definition 1. *An execution δ is sequentially consistent if there exists a legal sequence γ of operations such that γ is a permutation of ops(δ) and, for each process p, ops$(\delta)|p$ is equal to $\gamma|p$.*

Intuitively, SC preserves the real-time order of operations by the same client [104]. In contrast, linearizability guarantees the real-time order for all operations. SC requires that every operation takes effect at some point and occurs somewhere in the permutation γ. This ensures that every write can be eventually observed by all clients. In other words, if v is written to a register X, there cannot be an infinite number of subsequent read operations from register X that returns a value written prior to v.

Algorithm Description

We propose our read/write algorithm to minimize the latency of majority quorum reads described above. Then, we prove its SC guarantee while solving the constraint for status messages in order to be used for read requests. We refer this constraint as the staleness of the status message θt.

Replicas exchange their updates using the algorithm illustrated in Figure 4.13. Essentially, the algorithm has a sender function and a receiver function. The sender function periodically broadcasts local updates packed in status messages to remote replicas. A status message contains the updates of data items happened after the previous status message. Each status message is associated with a timestamp when it is sent to remote replicas based on the server's local clock. The receiver function aggregates status messages and maintains them in a component called $statusMap$. The aggregation of historical status messages provide a slightly outdated caches of remote replicas with respect to local present. New status messages on specific data items will wake up the corresponding read requests maintained in the $readSubscriptionMap$, which maintains local reads that are blocked because of lacking up-to-date status messages.

When a proxy node receives a read request, it forwards the read request to the closest application process that manages a replica of the requested data item as shown in Figure 4.15 (a). Then, a local read request is processed by the application process as illustrated by the algorithm in Figure 4.14 (a). Specifically, a local read request does not initiate communications among remote replicas. Instead, it checks the updates of remote replicas by analyzing the status messages periodically reported from all replicas in the $statusMap$. A read request is returned when it can obtain a consensus value of the requested data item from a majority of replicas' status messages with staleness bounded by θt. Otherwise, it is kept in the $readSubscriptionMap$. This process can be regarded as a query to a majority of replicas. In case there are multiple updates of a requested item in a status message that satisfies the above constraint, the newer update is returned. Another question is the upper bound of θt, which means the maximum tolerance of the staleness of replica values while preserving data consistency. We will investigate this issue when we prove the sequential consistency of the algorithm.

A write request is initiated by an originator process/proxy sending a write to all the replicas as illustrated in Figure 4.15 (b). The $timestamp$ of the write is the local time when the originator process/proxy invokes the request. The write request is returned when the originator has received a **majority** of *Ack* from all the replicas. When an application process receives the write from the proxy, it executes the request following the algorithm described in Figure 4.14 (b). Specifically, it checks whether the $timestamp$ in the request is larger than the timestamp of the requested data item stored locally. If the result is positive, an update on the data item is performed locally, a status message is registered and an *Ack* is sent to the proxy. Otherwise, a *Rej* is sent to the proxy. In this way, all writes are ordered deterministically based on the local invocation timestamps, breaking ties with node IDs.

Data: $broadcastStatus$
Result: Broadcast updates to peer replicas periodically
while *dispatchInterval* **do**
  foreach $key \in namespace$ **do**
    foreach $replica \in replicaList[key]$ **do**
      Send $statusMessage[key]$ to $replica$
    end
  end
end
/* $statusMessage[key]$ contains the updates of key in the current dispatchInterval. */
/* In implementation, status messages are sent to replicas in an aggregated manner. */

(a) Broadcast Local Updates/Status

Data: $updateStatus[key][replica]$
Result: Maintain a local version of remote replicas
Upon Receiving $statusMessage[key][replica]$
  Add $statusMessage[key][replica]$ to $statusMap[key][replica]$
  /* $statusMap[key]$ is used to keep track of updates of key from all peer replicas. */
if key *exists in* $readSubscriptionMap$ **then**
  Awake the pending reads concerning key.
end

(b) Update Remote Replicas' Status

Figure 4.13 – Status Messages

Data: $readLocal(key, timestamp, source)$
Result: Return the $value$ of key
$validStatus = []$
for $status \in statusMap[key]$ **do**
  if $status.timestamp + \theta t > timestamp$ **then**
    $validStatus.append(status)$
  end
  /* θt is the maximum staleness of status messages in the consistency model proved */
end
$validStatus.append(localStatus)$ // add local status of *key*
if $validStatus.size > \frac{replicaList[key].size}{2}$ **then**
  Find majority $value$ in $status \in validStatus$
  if *the majority size* $> \frac{replicaList[key].size}{2}$ **then**
    return $value$
  end
end
$readSubscriptionMap[key].append(read(key, timestamp, source))$

(a) Handle Read Requests - Server Side

Data: $writeLocal(key, value, timestamp, source)$
Result: Write to local storage
if $timestamp > localStatus.timestap$ **then**
  Perform $write(key, value, timestamp)$ in local storage
    $statusMessage[key].append(write(key, value, timestamp, self))$
    /* add this write to statusMessage queue for broadcasting to peer replicas. */
  Return Ack
end
Return Rej

(b) Handle Write Requests - Server Side

Figure 4.14 – Server Side Read/Write Protocol

Data: $read(key, timestamp, source)$
Result: Return the $value$ of key to the caller
Send $read(key, timestamp, self)$ to the nearest replica of key
while $NotTimeout$ **do**
  if *Reply from the nearest replica* **then**
    Return $value$
  end
end
Return $Timeout$
/* The result from the nearest replica already considers the updates of remote replicas. */

(a) Handle Read Requests - Proxy Side

Data: $write(key, value, timestamp, source)$
Result: Majority write to the storage system
foreach $replica \in replicaList[key]$ **do**
  Send $write(key, value, timestamp, self)$ to $replica$
end
while $NotTimeout$ **do**
  if *number of* $Ack > \frac{replicaList[key].size}{2}$ **then**
    Return $Success$
  end
  if *number of* $Rej > \frac{replicaList[key].size}{2}$ **then**
    Return $Fail$
  end
end
Return $Timeout$
/* A majority write to all replicas to ensure data availability. */

(b) Handle Write Requests - Proxy Side

Figure 4.15 – Proxy Side Read/Write Protocol

Proofs:

From the algorithm we provided, we prove that it satisfies sequential consistency under a specific constraint of θt.

Lemma 1. *For each admissible execution and every process p, p's read operations reflect all the values successfully applied by its previous write operations, all updates occur in the same order at each process, and this order preserves the order of write operations on a per-process basis.*

Proof. Regarding process p. A write returns success only when a majority of replicas accept the write proposed by p. And this indicates that the write timestamp proposed by p is larger than the timestamp of the previous writes. Since any read is served by querying a majority of replicas, a read originated by process p after the successful write will at least reflect the value written (intersect with the write majority quorum) or a newer value proposed by other processes in between the write and read from p. Regarding multiple processes, write operations are applied based on the local invocation timestamps. Since write operations are acknowledged by majority, any two writes proposed by p and q can be deterministically ordered based on their timestamps, breaking ties with node identifiers. Because of the monotonicity of physical clocks, this deterministic order is the same order at each process, and this order preserves the order of write operations on a per-process basis.

When using status messages instead of querying a majority of replicas, we solve the upper bound of θt, in order to preserve the above constraint. We would like to refer to the concept of consistent cuts [97, 105]. Briefly, it is a view of events in a distributed system that obeys a logical happen before order. In Figure 4.16, a read operation R of object X follows a write operation W of object X with a delay of Δr. We assume that Δr is close to zero. The read request R from DC_1 observes a consistent view of object X if it fetches the value of X follows the consistent cuts C_3 and C_4, where object X is applied and consistent in all data centers/replicas. We regard the cut C_2 legal as well since a majority of the replicas is consistent at this point. However, the cut C_1 is not acceptable since a majority of the replicas of X have not applied the update by W yet. Given the causal order that W happens before R, R should reflect the updates proposed by W. Thus, R is legal when a majority of replicas are consistent after the updates of W. In order to calculate the upper bound of θt, let us assume the latencies between $P@DC_1$ and $S_1@DC_1$, $S_2@DC_2$, and $S_3@DC_3$ are L_1, L_2, and L_3 respectively. Imagine $L_1 < L_3 < L_2$. When a write request requires a majority of replicas to acknowledge, in order to guarantee this update is observed when reading a majority of replicas at time $T - \theta t$, the upper bound of θt should be less than L_3, which is the median value of L_1, L_3, and L_2. To extend the application of θt, when a write request only requires the reply from one of the replicas, possibly the closest one, the upper bound of θt is L_1, which is the minimum value of L_1, L_3, and L_2, for a read "ALL" operation to observe the update. Similarly, if a write request needs to wait for the acknowledgement from all the replicas, the upper bound of θt is L_2, which is the maximum value of L_1, L_3, and L_2, for any replica to observe the update (read "ONE").

Lemma 2. *For each admissible execution and every process p, p's read operations cannot reflect the updates applied by its following write operations.*

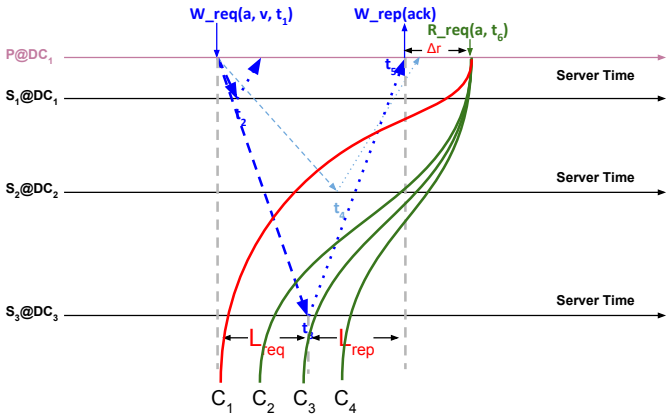


Figure 4.16 – Upper bound of the staleness of reads

Proof. For simplicity, we constraint θt to be a positive value for now, which means a read originated by process p will never read any updates newer than its local present. Thus, a read by process p at present T will never reflect any writes from p after local time T, since the write cannot be applied before time T. In sum, a read R of object X happens before a write W of object X cannot observe W's update on X when the R and W are originated from the same process p.

Lemma 3. *For each admissible execution, every process p, and all data items X and Y, if read R of object Y follows write W of object X in $ops(\delta)|p$, then R's read from p of Y follows W's write from p of X.*

Proof. Imagine that a read $R^{'}$ of object X happens the same time as R's read of object Y. Since R follows W in $ops(\delta)|p$ and according to Lemma 1, $R^{'}$ reflects the updated value of X by W. It means that $R^{'}$ fetches a value of X from a majority of the replicas with maximum staleness of θt from present. Since $R^{'}$ happens the same time as R, p fetches a Y value with a maximum staleness of θt from present. From Lemma 1, we know that the maximum staleness θt guarantees that the updates of Y, if any, is already propagated at a majority of the replicas. Thus, R's read from p of Y follows W's write from p of X.

Theorem 1. *The algorithm proposed provides sequentially consistent reads and writes from multiple processes.*

Proof. We follow the sequential consistency proof procedures provided in [103]. Fix an admissible execution δ. Define the sequence of operations γ as follows. Order the writes in δ with the timestamps associated with each write operations, breaking ties using process ids. Then, we explain the places to insert reads. We start from the beginning of δ. Read operations $[Read_p(X), Ret_p(X, v)]$ are inserted immediately after the latest of (1) the previous operation for p (either read or write on any object), and (2) the write that spawned the latest update of p regarding X preceding the generation of the $Ret_p(X, v)$.

We must show $ops(\delta)|p = \gamma|p$. Fix some process p. We show $ops(\delta)|p = \gamma|p$ in the following four scenarios.

1. The relative ordering of two reads in $ops(\delta)|p$ is the same in $\gamma|p$ by the construction rules of γ.

2. The relative ordering of two writes in $ops(\delta)|p$ is the same in $\gamma|p$ given by the ordering of write by monotonically increasing physical timestamps and Lemma 1.

3. Suppose a relative ordering of read R follows write W in $ops(\delta)|p$. By definition of γ, R comes after W in γ.

4. Suppose a relative ordering of read R precedes write W in $ops(\delta)|p$. Suppose in contradiction that R comes after W in γ. Then, in δ, there is some read $R^{'}$ and some write $W^{'}$ such that (1) $R^{'}$ equals R or happens before R in δ; (2) $W^{'}$ equals W or happens after W; (3) $W^{'}$'s update on object X is applied before $R^{'}$ś read on object X, $R^{'}$ is able to read $W^{'}$'s update according to Lemma 1. However, in $ops(\delta)$, the relative ordering of R precedes W, i.e., R is not able to read W's update according to Lemma 2. Thus, $R^{'}$ cannot see $W^{'}$'s update, a contradiction.

Summary of θt

The lower bound of θt is easy to calculate. Imagine that all the operations are ordered by their invocation timestamps, breaking ties using node IDs. In this case, the value of θt is 0, when all the operations use the present timestamp T.

On the other hand, the upper bound of θt depends on the latencies between the proxy and storage servers as well as the read/write mode used, which can be write "ONE", "QUORUM", or "ALL". We assume that the read/write modes are used correspondingly to achieve sequential consistency, i.e., $r + w > n$ as introduced in the beginning of Section 4.2. Under this scenario, when writes only require one replica to acknowledge, the upper bound of θt is the minimum latency between the proxy and storage servers. On the other hand, when writes require all replicas to respond, the upper bound of θt is the maximum latency between the proxy and storage servers. In the case of quorum writes, the upper bound of θt is the median value of the latency between the proxy and storage servers. The application of the upper bound of θt provides the best performance of read requests.

In practice, the clock time among servers are not perfectly synchronized. Under our assumptions that the clock difference of any two servers' clock times t_1 and t_2 is bounded to ε, i.e., $|t_1 - t_2| \leq \varepsilon$, the upper bound of θt shall not only concern the latency among proxy and storage servers. The calculation should also consider the worst case clock drift among servers, which is ε. Thus, the calculation of the upper bound of θt in all three cases need to subtract ε.

4.2.3 Messages in MeteorShower

Using the algorithm described above, we propose **MeteorShower**. It improves request latency for majority quorum based storage systems when they are deployed geographically. The major insight is that instead of pulling the status of remote replicas, MeteorShower enables replicas to report their status periodically through status messages. Then, MeteorShower judiciously use the updates in the status messages to serve read requests.

In the design, we have separated the delivery of the actual updates and their metadata. This is because that an update is usually propagated immediately among replica servers while the metadata can be buffered in intervals. The actual updates are propagated using write notifications and the metadata are encapsulated in status messages.

Write notifications are used to propagate writes among storage servers. Specifically, when a write is applied upon a storage server, a corresponding write notification regarding the write is sent out to its replica servers. A write notification is constructed using the following format:

$$WriteNotification = \{Key, \quad Update, \quad Ts, \quad Vn\}$$

It records the identification of the record (Key), the updated value of the record ($Update$), the physical local timestamp of the update (Ts) and the version number of the record (Vn).

Here is an intuition of write notification in Cassandra. When a write request is received by one of the storage servers, i.e., the proxy/leader, it propagates the write to all the storage servers that store the corresponding record. If it is a majority write, the proxy/leader collects confirmations from a majority of the storage servers and returns to the client. The propagation of writes is encapsulated as write notification in MeteorShower. It conveys more information, such as timestamps and versions, in order to implement sequentially consistent reads (as explained in Section 4.2.2).

A **status message** is an accumulation of write notifications initiated and received in a MeteorShower server in an interval. The interval defines the frequency of exchanging status messages and it is configurable. A status message records the writes applied on a MeteorShower server using the following format.

$$StatusMessage = \{Payload, MsgTs, Seq, Redundant\}$$

A timestamp ($MsgTs$) is included when the status message is sent out. It is the timestamp that we use to identify the staleness of a status message from replicas. A status message is sequenced (Seq) using a universal MeteorShower server ID as prefix. Thus, the sequence number allows recipients to group and order status messages with respect to senders. $Redundant$ is a history of status messages in previous intervals. It is configurable, i.e., the number of previous status messages to be included, to tolerate status message lost. The $Payload$ is the accumulation of write notifications but without the value of the record to minimize the communication overhead.

$$payload = \sum\{key, \quad ts, \quad vn\}$$

At the end of each interval, a status message is propagated to all MeteorShower peers.

4.2.4 Implementation of MeteorShower

MeteorShower is completely decentralized. It is a peer to peer middleware, which is designed to be deployed on top of majority quorum based distributed storage systems. MeteorShwoer consists of six primary components as shown in Figure 4.17. They are status message dispatcher, message receiver, status map, read subscription map and read/write workers.

Figure 4.17 – Interaction between components

Status Message Dispatcher

Status message dispatcher is the component that periodically sends out status messages to MeteorShower peers. A write operation received and processed by the write worker creates an entry in the status message poll. Entries are aggregated to construct a status message when the dispatching interval is reached. A status message is sent out to all the MeteorShower peers every interval even when there is no entry during an interval.

Message receiver

Message receiver is a component that receives and processes write notifications and status messages. A write notification triggers the write worker to update the corresponding record to the underlying storage. A status message is used to update the status map, which decides whether a read request could be served. Both write notifications and status messages awake pending reads in read subscription map.

Status map

Status map keeps track of status messages sent from MeteorShower peers. It is used to check whether a read request can be served with respect to the constraint of maximum staleness described in Section 4.2.2.

Read subscription map

A read request cannot be served if the required write notification or status message are not received. In this case, the read request is preserved in the read subscription map. The read subscription map is iterated when receiving new status messages or write notifications.

Write worker

The first responsibility of a write worker is to persist record to the underlying storage if the update has a timestamp larger than the local timestamp of the affected data item. Then, it sends out write notifications to the MeteorShower peers to synchronize the update. In the meantime, an entry is preserved in the status message dispatcher.

Read worker

This component is designed to handle read requests. Using status messages, a read worker decides the version of writes to be returned for a read request w.r.t. the maximum staleness. Then, the read worker finds the correct version or a newer version stored locally and returns the request. The read request is kept in the read subscription map when the required status messages or write notifications are not received.

4.2.5 Evaluation of MeteorShower

We evaluate the performance of MeteorShower on Google Cloud Platform(GCP). It enables us to deploy MeteorShower in a real geo-distributed data center setting. We first present the evaluation results of MeteorShower in a controlled environment, where we simulate multiple "data centers" inside one data center. It enables us to manipulate latency among different "data centers". Then, we evaluation MeteorShower with a real multiple data center setup in GCP.

MeteorShower on Cassandra

We have integrated MeteorShower algorithm with Cassandra, which is a widely applied distributed storage system. Specifically, we have integrated MeteorShower write worker, reader worker and message receiver components in the corresponding functions in Cassandra. Status message dispatcher, status map and read subscription map are implemented as standalone components. During our evaluation, we bundle the deployment of Cassandra and MeteorShower services together on the same VM. We adopt the proxy implementation in Cassandra, where the first node that receives a request acts as the proxy and coordinates the request.

The Baseline

We compare the performance of MeteorShower with the performance of Cassandra using different read/write APIs. Specifically, read "QUORUM" and "ALL" APIs are used as baseline for read requests while write "ONE" and "QUORUM" APIs are employed as baseline for write requests. The choice of these sets of APIs takes into the consideration of achieving sequential consistency. For example, the application of read "QUORUM" ("ALL") API together with write "QUORUM" ("ONE") API provides sequential consistency in Cassandra.

The Choice of θt in MeteorShower

We have implemented the MeteorShower algorithm with the lower bound and upper bound of θt, namely MeteorShower$_1$ and MeteorShower$_2$. The lower bound and upper bound of θt are presented in Section 4.2.2. Specifically, in case of read "QUORUM" ("ALL") operation in MeteorShower, it requires that the proxy server receives the status messages from a majority (all) of the replicas with timestamp larger than $T - \theta t$. The write operations in MeteorShower are essentially the same as they are in Cassandra.

NTP setup

To reduce the time skew among MeteorShower servers, NTP servers are setup on each server. Briefly, NTP protocol does not modify system time arbitrarily. Time in each server still ticks monotonically. NTP minimizes the time differences among servers by changing the length of a time unit, e.g., the length of one second, in its provisioned server. We configure NTP servers to first synchronize within a data center, since the communication links observe less latency, which improves the accuracy of NTP protocol. Thus, there is one coordinator NTP server in a data center. Then, we have chosen a middle point, where observes the least latency to all the data centers, to host a global NTP coordinator. In this way, NTP servers inside one data center periodically synchronize with the local coordinator while local coordinators synchronize with the global coordinator.

NTP is used to guarantee the bound of time drifts (ε) among servers. Empirically, we observe that NTP is able to keep the clock drifts of all servers within 1 ms most of the time. To be on the safe side, we evaluate our system under the maximum drift $\varepsilon = 2ms$.

Workload Benchmark

We use Yahoo! Cloud Serving Benchmark (YCSB) [100] to generate workload to MeteorShower. YCSB is an open source framework used to test the performance and scalability of distributed databases. It is implemented in Java and has excellent extensibility, where users can customize YCSB client interface to connect to their databases. YCSB provides a configuration file, using which users are able to manipulate the generated workload pattern, including read/write ratio, data record size, concurrent client thread number, and etc.

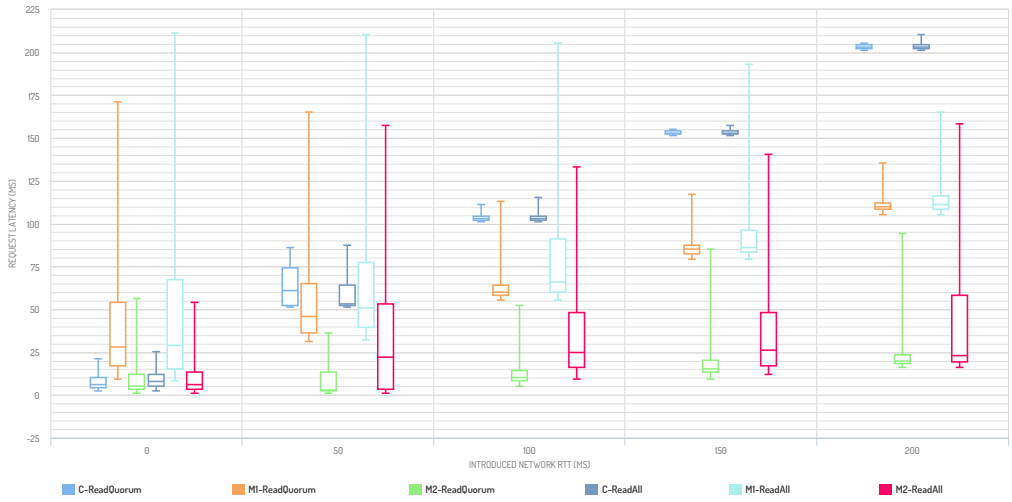


Figure 4.18 – Cassandra read latency using different APIs under manipulated network RTTs among DCs

Evaluation of MeteorShower

Evaluation in Controlled Environment

In this experiment, we evaluate the performance of MeteorShower$_1$ and MeteorShower$_2$ under different cross data center network latencies in comparison with the original Cassandra baseline approach. Since latency cannot not be manipulated in a real multi-DC setup, this experiment is conducted in a single data center with simulated multiple "data centers". Specifically, communications of VMs in different simulated "data centers" are introduced with a static latency. The latency is manipulated using NetEM tc commands [106].

For the cluster setup, we have initialized MeteorShower and Cassandra servers with the medium instances in GCP, which has two virtual cores. We have setup the cluster with 9 instances using 3 instances simulating a data center, which results in 3 data centers. We have spawned another 3 medium instances hosting YCSB in each simulated "data center". Each YCSB instance runs 24 client threads and connects to a local Cassandra/MeteorShower server to generate workloads. The composition of the workload is 50% reads and 50% updates.

Figure 4.18 and Figure 4.19 present the read and write latency of MeteorShower and Cassandra under different simulated cross data center delays, which are shown in the x-axis. We have run the workload with different combinations of read/write APIs in MeteorShower and Cassandra. Specifically, the workload is run against Cassandra, MeteorShower$_1$ and MeteorShower$_2$ with read QUORUM v.s. write Quorum and read ALL v.s. write ONE. We use write QUORUM instead of write ONE in combination with read ALL in the evaluation of MeteorShower$_2$, which enables it to use the upper bound of θt. The latency of each approach is aggregated from all the "data centers" and ploted as boxplot.

Figure 4.18 shows that the latency of read QUORUM and read ALL operations in Cas-

Figure 4.19 – Cassandra write latency using different APIs under manipulated network RTTs among DCs

sandra, MeteorShower$_1$ and MeteorShower$_2$ are similar. This is because that "data centers" are equally distanced among each other, i.e., having the same network latency. Thus, waiting for a majority of replies requires more or less the same time as waiting for all the replies. As for Cassandra, the latency of its read operations increase with the increase of network latency introduced among "data centers". The reason is that these operations need to actively request the updates from remote replicas before returning, which leads to a round trip latency. On the other hand, MeteorShower$_1$ only needs a single trip delay to complete read QUORUM/ALL operations in this case. This is because that a read at time t can be returned when it has received the status messages from a majority/all of the replicas with timestamp t. These status messages require a single trip latency to travel to the originator of the read plus the delay waiting for a status message dispatch interval of remote servers. MeteorShower$_1$ is not suitable to be deployed when the latency among data centers is less than 50 ms since it has non-negligible overhead in sending and receiving status messages. Despite the consumption of system resources, there is also a delay when waiting for a status message, which is sent every 10 ms in our experiment. It is reflected in Figure 4.18 when the introduced network latency is 0. Furthermore, this message exchanging overhead also causes a long tail in the latency of MeteorShower operations. Thus, MeteorShower is not suitable for applications that require stringent percentile latency guarantees. As for MeteorShower$_2$, which is configured with the upper bound of θt. It can be easily calculated that the upper bound of θt is a single trip latency introduced minus ε. It essentially allows read operations to return immediately since a read at time t only needs the status messages from a majority/all of the replicas with timestamp $t - \theta t$. Ideally, the status message with timestamp $t - \theta t$ should arrive at any "data center" no later than t plus a status message dispatch interval $10ms$. Thus, the latency of read QUORUM/ALL operations in MeteorShower$_2$ remain stable in the presence of increasing network latency among "data

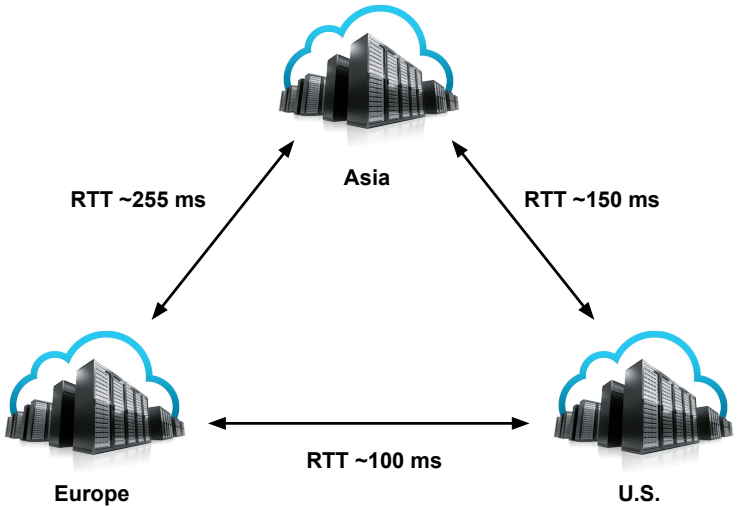


Figure 4.20 – Multiple data center setup

centers".

The writes of MeteorShower$_1$ and MeteorShower$_2$ are the same. So, we only show the writes of MeteorShower$_1$ in Figure 4.19. Since we have not changed the writes in MeteorShower comparing to the original implementation in Cassandra, the performance of both approaches are similar. However, we do observed a slightly long tail of write latency in MeteorShower, which is caused by the frequently exchanged status messages among servers in different "data centers".

Evaluation with Multiple Data Centers

We move on to evaluate the performance of MeteorShower in a multiple data center setup using GCP, which is shown in Figure 4.20. Specifically, we have used three data centers located in Europe, the U.S. and Asia. The latency between data centers are presented in the figure. To make it consistent, we have used the same instance type and configuration as the previous experiment to setup the MeteorShower and Cassandra cluster as well as YCSB. We focus our analysis on the latency of read requests, since the writes are essentially the same in Cassandra and MeteorShower.

Figure 4.21 presents the aggregated read latency from the three data centers. As explained in the previous evaluation, the read requests of Cassandra experience a round trip latency. However, read QUORUM operations experience the round trip latency from the closer remote data center while read ALL operations need to wait for the replies from the further remote data center. Similarly, read QUORUM/ALL operations in MeteorShower$_1$ observe a single trip latency from the closer/further remote data center. MeteorShower$_2$ performs the best since it only requires status messages that is θt earlier than MeteorShower$_1$. And in this setup, the upper bound of θt is around $50ms - 2ms$ for requests generated from Europe and U.S. data centers and around $75ms - 2ms$ for requests originated in Asia.

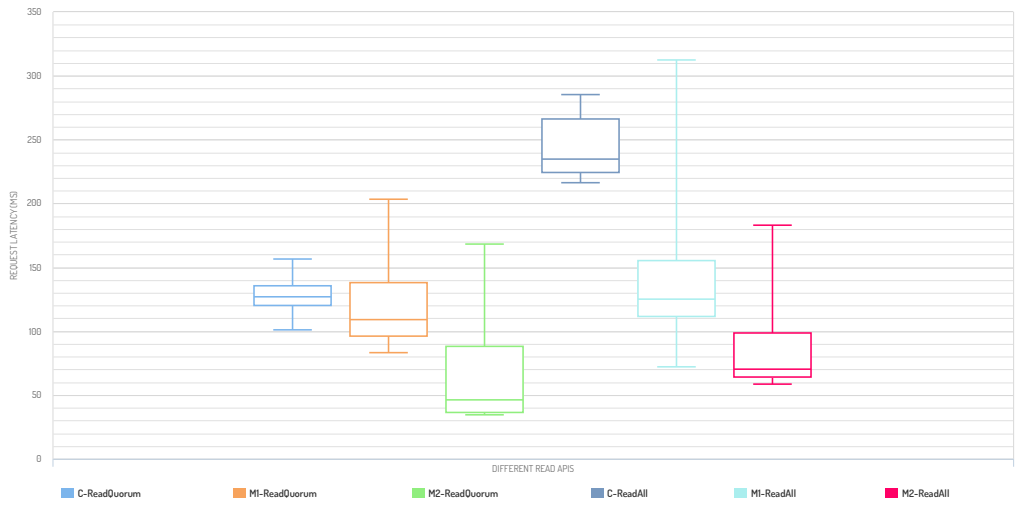


Figure 4.21 – Aggregated read latency from 3 data centers using different APIs

We present the fine-grained results of the evaluation in Figure 4.22 and Figure 4.23. These two figures present the request latency from each data center. Specifically, Figure 4.22 shows the results grouped by different approaches while Figure 4.23 describes the latency grouped by data centers. We focus our explanation on the impact of different delays between data centers.

As we can see from Figure 4.22 and Figure 4.23, except of MeteorShower$_2$, request latency in Asia is higher than request latency in Europe and U.S.. This is because that the data center in Asia experience high latency to both Europe and U.S. data centers, especially Europe. On the other hand, MeteorShower$_2$ allows read QUORUM requests to return with the same latency as the requests in Europe and U.S.. Specifically, MeteorShower$_2$ will expect a status message from U.S. data center instead of Europe, which is further. And a single trip communication from U.S. to Asia costs around $75ms$, which is compensated by a higher upper bound of θt of requests originated in Asia. Thus, the read QUORUM requests perform the same in Asia as the requests in Europe and U.S. even though Asia has the worst network connection. Similar conclusion can be drawn from the performance of read ALL, where requests perform even better in Asia than in Europe. Because requests from both data centers need to wait for status messages from the furthest data center. However, requests originated from Asia has a larger upper bound of θt ($75ms - 2ms$) than the requests initiated from Europe (upper bound of θt equals $50ms - 2ms$). So, the read ALL latency of requests in Asia is less than the request latency in Europe. As for MeteorShower$_1$, the performance of read ALL requests in Europe are similar to the performance of read ALL requests in Asia, since all requests need to pay for the highest latency and Asia data center does not have the advantage of a larger upper bound in θt. Obviously, the requests from the U.S. data center experience the least latency in all the cases. This is because that U.S. has the best connection to the other two data centers.

In sum, MeteorShower$_1$ needs a little more than single trip delay to return a read re-

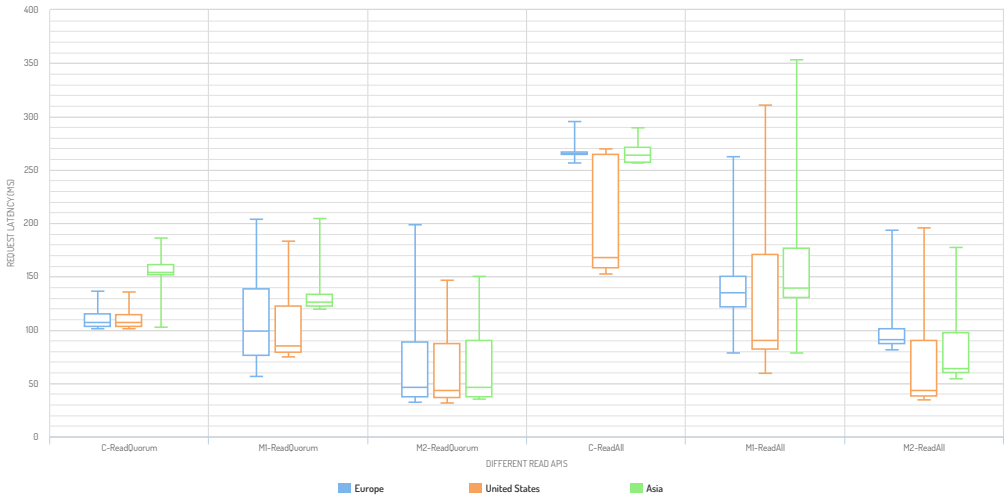


Figure 4.22 – Read latency from each data center using different APIs grouped by APIs

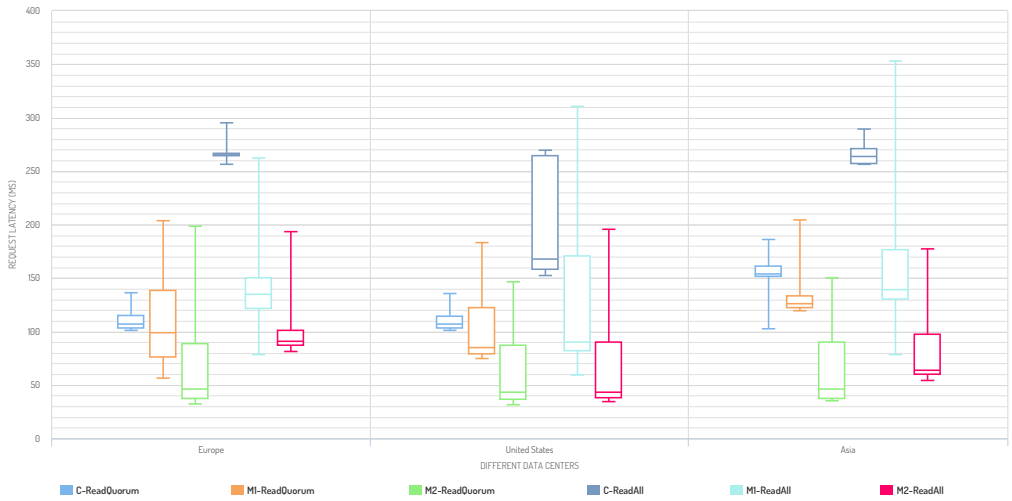


Figure 4.23 – Read latency from each data center using different APIs grouped by DCs

quest, which is significantly faster than Cassandra in most of the requests. MeteorShower$_2$ is even faster than MeteorShower$_1$. It is able to return a read requests immediately in most of the cases taken into account the reasonable overhead consumed to exchange status messages among data centers. Furthermore, MeteorShower$_2$ has a big advantage that it allows requests originated from a not well connected data center (Asia) to be returned with improved latency. To some extent, the performance of MeteorShower$_2$ is irrelevant to the connectivity, in terms of latency, of a data center. Overall, the latency of MeteorShower has a longer tail than Cassandra, which makes it not suitable for percentile latency sensitive applications.

4.2.6 Summary and Discussions of MeteorShower

MeteorShower presents a novel read write protocol for majority quorum-based storage systems. It allows replica quorums to function more efficiently when replicas are deployed in multiple data centers. Essentially, MeteorShower exhausts the exploration of a global timeline, constructed using loosely synchronized clocks, in order to judiciously serve read requests under the requirement of sequential consistency. The algorithm allows MeteorShower to serve a read request without cross data center communication delays in most of the cases. As a result, MeteorShower achieves significantly less average and mean read latency comparing to Cassandra majority quorum operations. It is also worth to mention that MeteorShower keeps all the desirable properties of a majority quorum-based system, such as fault tolerance, balanced load, etc. This is because that MeteorShower algorithm only augments the existing majority quorum-based operations. However, MeteorShower does observe some overhead. It scarifies the tail latency of requests because of the extensive exchanging of messages among remote replicas, which saturates the network resources to some extent.

4.3 Catenae

Serving requests with low latency while data are replicated and maintained consistently across large geographical areas is challenging. With the proposal of GlobLease and MeteorShower in the previous sections, we are able to achieve efficient key value accesses globally. In the section, we investigate mechanisms in order to support efficient transactional accesses across multiple DCs.

The high latency communications among DCs causes significant communication overhead to maintain ACID properties in transactions using traditional concurrency control algorithms, such as two phase lock (2PL) [9] and optimistic concurrency control (OCC) [10]. On the other hand, maintaining data consistency among replicas in multiple DCs also involves a large amount of cross DC communications.

In order to address these two challenges, we investigate the triggers of cross DC communications. In essence, these communications are used for synchronizations in two scenarios. First, algorithm maintains the total ordering of different transactions with respect to different data partitions. Second, algorithm tackles with the ordering of operations on replicas for maintaining replica consistency.

The second cause is discussed in the previous sections. We investigate in details the first cause. The goal is to reduce latency of achieving linearizable transactions in a geo-distributed environment. The time spent from receiving a transaction until it is committed and returned to the client defines the latency of a transaction. This process involves message **transmission delays** among DCs and **concurrency delays** to reach a consensus on a linearizable execution order of conflicting transactions in all DCs. When consensus is reached in all DCs, a transaction is executed with **execution delays**.

The transmission delay depends on the locations and connectivity of DCs and usually cannot be optimized. Reducing the message exchanging rounds among DCs to reach a consensus on the execution of transactions are studied in recent works [40, 41]. It is

theoretically proved that the lower bound for two conflicting transactions to commit and maintain serializability in two DCs is the RTT between them [42].

The concurrency delay is the time spent for a transaction to be allowed to commit in all DCs. The concurrency delay is caused by conflicting transactions. Examples of the concurrency delay can be the waiting time for locks in 2PL or the time spent to abort and retry in OCC.

The execution delay is subjective to the specification of the hosting machines and the efficiency of the underlying storage system while performing read and write requests.

Proposal

We propose a framework, **Catenae**, which provides serializable transactions on top of data stores deployed in multiple DCs. It manages the **concurrency among transactions** and maintains **data consistency among replicas**. Catenae executes transactions with low latency by improving **the transmission and concurrency delays**. In order to reduce cross DC synchronizations, Catenae leverages the insight of using speculative executions of transactions in each DC, which expects a coherent total ordering of transactions in all DCs and eliminates the need for synchronizing replicas.

However, achieving speculative executions of transactions with a deterministic order on a global scale is not trivial. Static analysis of transactions before execution is able to produce a deterministic ordering among transactions. Nevertheless, this approach has the disadvantage of high static analysis overhead and potentially inefficient scheduling among transactions. To be specific, a complete set of transactions needs to be analyzed and ordered at a single site in the system, which is a scalability bottleneck and a single point of failure. Moreover, static ordering of transactions cannot guarantee efficient executions of transactions w.r.t. concurrency. Because it is impossible to efficiently order conflicting transactions when their access time on each data partition is unknown before execution. Approaches, such as ordering transactions by comparing the receiving timestamps [42], lead to inefficient execution of transactions for the same reason.

Many other works [64, 63] achieve this by analyzing transactions before execution and giving priority to some transactions while aborting or suspending conflicting transactions, in order to have only non-conflicting transactions to be executed in parallel on data replicas. In essence, the concurrency control in those approaches is similar to two phase locking (2PL), which increases the transaction execution time and limits the throughput. For example, a transaction T_1 arrives at t_1 and writes on data partition a and b while another transaction T_2 arrives later at $t_2(t_2 > t_1)$ and writes on data partition b. T_1 and T_2 are conflicting with each other and a total order needs to be preserve on all replicas of data partition a and b in order to maintain serializability. Usually, a static analysis before the execution is hard to know which transaction should have the priority to be executed first. Typically, such priority is given based on the arrival time of transactions. Thus, T_1 is ordered before T_2. Assuming the time spent on writing each data partition is constant Δt, then, the execution time of T_1 and T_2 is $2 * \Delta t + \Delta t = 3 * \Delta t$. Obviously, this type of concurrency controls can potentially block the concurrency of transaction executions.

Catenae pushes transaction execution concurrency to the limit by delaying the decision

on transaction execution orders until they are conflicting a shared data partition. This allows transactions to be ordered naturally by their execution speed rather than their arrival time. Back to the example, assuming T_2 arrives slightly behind T_1, which gives $t_2 - t_1 < \Delta t$, T_2 is able to access data partition b before T_1 since T_1 has not finished writing on data partition a. When T_1 has finished writing on a and continues to write on b at time $t_1 + \Delta t$, it observes that T_2 is in the middle of writing on b. Then, T_1 is naturally ordered behind T_2 and will write on b until T_2 finishes. The total execution time of T_1 and T_2 is $\Delta t + t_2 - t_1 + \Delta t = 2 * \Delta t + t_2 - t_1 < 3 * \Delta t$. A formalization of this concurrency control is a **transaction chain concurrency control** algorithm, which will be explained in details in Section 4.3.3.

The insight in Catenae is that the execution speed of transactions on each record is unique and deterministic. Ideally, Catenae believes that given the same set of transactions to multiple **fully replicated DCs**, the execution order of the conflicting transactions in these DCs are likely to be the same using the transaction chain concurrency control. Experimental validations (Section 4.3) and evaluations (Section 4.3.4) of Catenae under a symmetric cluster setup, i.e. the same VM instance type in multiple DCs on top of Google Cloud Platform, shows that most of the conflicting transactions are ordered identically in all DCs. Thus, Catenae first speculatively executes the same set of transactions in each DC. Then, inconsistent executions will be corrected by a validation phase.

Validations of the insight

We validate the success rate of speculative executions of Catenae in three DCs of Google Cloud Platform. Specifically, we have randomly generated 10000 records and replicated them on 4 storage servers in each DC. These random records have different data size, which leads to different processing times when reading or writing on the record. Then, we have a coordinator in each DC that generates transactions with specific throughput to the 4 storage servers in the same DC. We guarantee that coordinators generate the same transaction sequence with Poisson arrivals. Each transaction will read/write 1 to 4 data records out of 10000 records. The distribution of the record accessed is configured to be uniform random, zipfian with exponent 1 or zipfian with exponent 2. Figure 4.24 presents the evaluation of running 100000 transactions in each DC. Those transactions are generated to the storage servers with different rates, which are from 1000 to 11000 request per second as shown on the x axis. Storage servers execute transactions using transaction chains concurrency control algorithm. In short, the algorithm orders transactions based on the access order on the first shared data record. A simple example of the algorithm is presented in the previous section and the detailed explanation of the algorithm will be discussed in Section 4.3.3 The execution dependencies of each transaction in each DC are compared. If the execution dependency of a transaction is the same in all three DCs, it means a success in speculative execution. Otherwise, the speculative execution is invalid. The y-axis in Figure 4.24 illustrates the success rate of speculative execution under 3 workload access patterns, i.e., uniform random, zipfian with exponent 1 and zipfian with exponent 2. The results indicate that the transaction chain algorithm is able to allow transactions to be executed on record replicas without coordination but still yields a very high (above 80%) result consistency rate (success rate of speculative execution) when the access pattern of the workload

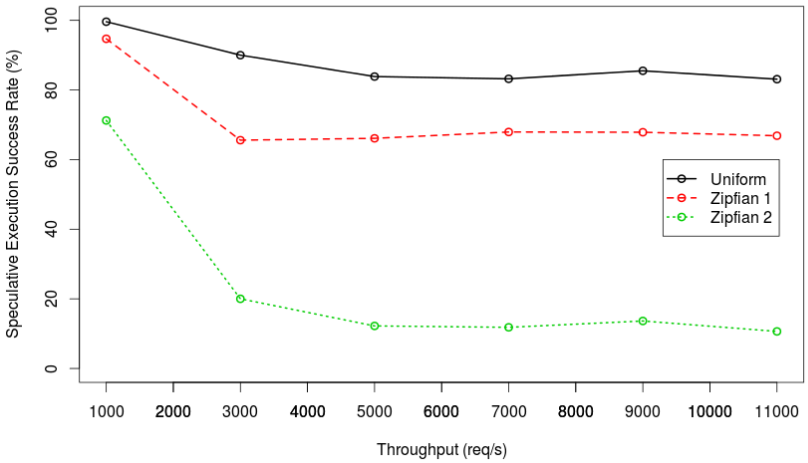


Figure 4.24 – Success rate of speculative execution using transaction chain

is uniform. Even when the access pattern is zipfian with exponent 1, Catenae is able to obtain a reasonable success rate (above 60%) on speculative execution using transaction chains. However, the evaluation results also show that the speculative execution will fail with extremely contended access pattern (zipfian with exponent 2).

4.3.1 The Catenae Framework

Transaction Client. Catenae has a transaction client library for receiving and pre-processing transactions. Transaction clients are the entries of Catenae in each DC. They pre-process transactions from standard query languages, such as SQL, and chop them to sequences of key-value read/write operations.Then, pre-processed transactions are forwarded to coordinators for scheduled execution among DCs.

Coordinator. There is one coordinator in each DC, which is responsible for the speculative executions and validations of transactions among DCs. It is achieved through the exchange of epoch messages among coordinators in different DCs in a fixed time interval. We defer the explanation of epoch messages in Section 4.3.2 Coordinators are designed to be stateless in each DC, thus Catenae can have multiple coordinators in one DC by partitioning the responsible namespace range.

Secondary Coordinator. There is an optional secondary coordinator for each DC that stands by the coordinator in that DC. Secondary coordinator receives duplicated epoch messages from other coordinators. It becomes primary coordinator when the coordinator fails.

Chain Servers. Transaction chain servers are hosted together with storage nodes of a NOSQL data store. Transaction chain servers are responsible for traversing of a transaction chain by passing through its forward and backward pass phase, during which, it maintains

and resolves transaction dependencies and conflicts, record temporary copies of transaction execution results, and issues corresponding NOSQL operations to the underlying storage servers when the transaction commits. The transaction chain algorithm is explained in Section 4.3.3

Transaction Resolver. There is a transaction resolver in each DC. It maintains implicit dependencies to avoid cyclic dependency among transactions. It is queried by transaction chain servers when they suspect a formation of a circle during transaction execution. Transaction resolvers perform a topology sorting among transactions with respect to the existing explicit dependencies. Then, a circle-free implicit dependency is returned to the chain server and stored in the transaction resolver for further queries until the involved transactions have committed or aborted.

4.3.2 Epoch Boundary

Epoch boundary is a concept similar to logical clock proposed by Lamport, but using the real time from the system. It separates continuous time into discrete time slices. The start or end of a discrete time slice is a boundary. Time boundaries are used as synchronization barriers among replicated servers deployed in different DCs. In Catenae, synchronizations of the status of replicated servers are not triggered by events, such as a transaction is received by one server or a consensus is needed to validate an execution result, but rather is conducted periodically at each boundary. The advantage of actively synchronizing server states among DCs is that it reduces the delay for a DC to realize the updates from other DCs. Specifically, when a DC needs additional information to proceed an operation, for example, to validate whether it holds the most recent data copy, it does not need to send a request and wait for a response to/from another DC, but rather wait for the next epoch boundary. It optimizes the communication latency among DCs from a RTT to a single trip plus the delay of an epoch. Epoch boundaries are not suitable to be implemented in low latency networks, such as intra DC networks, when inter-server latency is low. In this case, a RTT is rather short comparing to an epoch. Furthermore, periodically sending and receiving epoch messages also involves non-trivial overhead. However, this approach prevails when servers need to communicate through high latency links, such as inter DC links, when an epoch delay is negligible comparing to a single message trip. Specifically, the typical RTTs among DCs are from 50ms to 400ms, which can be easily measured through [107]. In contrast, the typical setup of the epoch interval is from 5ms to 30ms.

As shown in Figure 4.25, $DC2$ is able to aware an event happened at t in $DC1$ with delay less than $C + E$. However, with active queries, $DC2$ will know the status of $DC1$ after a delay of $2 * C$, which is significantly larger than $C + E$.

In order to ease the maintenance of server membership and reduce the overhead of sending epoch messages, there is one coordinator server in each DC to maintain epoch boundaries. System time on each coordinator server is synchronized using NTP to minimize time difference on coordinators. The length of the epochs is a globally configurable parameter. Epochs are associated with monotonically increasing epoch IDs that is coherent on each coordinator. At the end of each epoch, a synchronization boundary is placed with the dispatching of status updates (payload) from/to all coordinators using epoch messages.

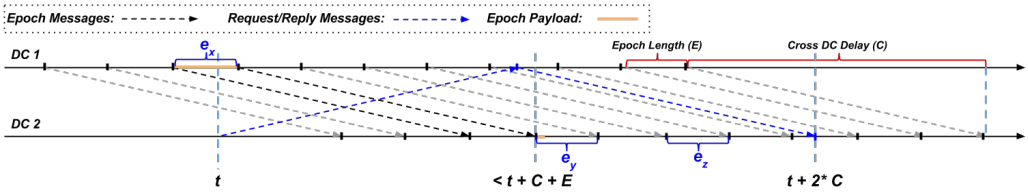


Figure 4.25 – Epoch Messages among Data Centers

Transaction Distribution Payload

The first part of epoch payload relates to transaction distribution. Ideally, with the application of epoch boundaries, each DC is able to acquire the transactions received from other DCs with a single cross DC message delay plus an epoch. By knowing the complete set of input transactions in an epoch, Catenae is able to speculatively execute transactions using the TC algorithm (Section 4.3.3) and maximize the possibility to obtain a coherent execution order of conflicting transactions in all DCs.

Transaction Validation Payload

The second part of epoch payload concerns about transaction validation. The speculative executions need to be validated on the execution order of conflicting transactions in all DCs since they could possibly be executed differently. Specifically, the execution order of transactions could be different due to the heterogeneity of the execution environment, platform as well as possible exceptions. Catenae leverages a light-weight static analysis of input transactions to create different transactions sets. The transaction set with conflicting transactions needs a validation phase to confirm their execution results. We defer the explanation of the multiple DC transaction chain algorithm in Section 4.3.3.

Batching and Dispatching of Payloads

For transaction distribution payload, all coordinators batch transactions received in each epoch. These transactions are associated with a local physical timestamp when it is received by Catenae. For transaction validation payload, coordinator batches conflicting transactions that have finished in each epoch along with their execution dependencies. The batched payloads are sent among coordinators at the end of each epoch. Instead of simply exchanging the payload of the current epoch, coordinators also attach the payload from the previous two epochs. According to our experiments, the redundancy in epoch payloads effectively handles message losses and delays during transmission among coordinators.

4.3.3 Multi-DC Transaction Chain

The life cycle of a transaction in Catenae includes received, scheduled, executed, finished, and committed (returned). We present the multi-DC transaction chain algorithm with the explanation of the life cycle of an transaction.

Receive Transactions

Coordinators receive transactions from other coordinators in epochs. Due to the transmission delays, transactions received in the current epoch are transactions sent by other coordinators in a past epoch. For example, in Figure 4.25, transactions received by $DC2$ at e_y are transactions sent from $DC1$ at e_x. The epoch ID (EID) is used to identify an epoch message. Coordinators continuously receive and keep track of epoch messages from other DCs and aggregate them by EIDs. With the complete receipt of epoch messages from all the coordinators concerning the same EID, the transactions in the epoch messages are grouped together and moved to transaction schedule phase. Transactions in lower EIDs are scheduled before transactions in higher EIDs. This allows Catenae to have a more consistent execution of transactions in each DC. However, it also puts limitations on Catenae when there are failures, which is discussed in Section 4.3.5.

Schedule Transactions

Transactions are chopped into a set of read and write operations by Catenae client library. These operations are ordered monotonically for accessing concerned data partitions. Thus, Catenae does not support a transaction that has cyclic access on data partitions. These operations are then mapped to Catenae chain servers that are hosted with storage servers that store the corresponding data partitions. Then, the transaction is sent to the chain server that hosts the first accessed data partition.

Execute Transactions

Transaction execution in each DC is handled by a transaction chain (TC) concurrency control protocol. It allows concurrent transactions to commit freely in the natural arrive order on the storage servers unless doing so will violate serializability. This property maximizes the transaction execution concurrency by allowing transactions to execute based on their execution speed and wait only if a faster transaction already occupied the resources on a per-key granularity. This means that transaction execution is not based on a predefined order given by the prior static analysis [64, 63] or the arrival order, but the access order at a shared data partition where two transactions issue conflict operations. Since transactions are executed in DCs individually, we explain the TC algorithm from the perspective inside one DC. The algorithm needs to pass through two phases, i.e., forward pass and backward pass.

Forward Pass. The forward pass does not conduct any read/write operations, but rather leaves footprints of a transaction on accessed data partitions. These footprints are used to identify conflicting transactions. It starts with the coordinator sending a transaction to the first accessed chain server as specified in the chain. The chain server records the data partitions that the transaction reads or writes, then the transaction is forwarded to the next chain server specified in the chain until reaching the end of the transaction chain.

Backward Pass. When a transaction is on the last server of its transaction chain during the forward pass, it starts the backward pass phase. The backward pass examines whether other transactions that have left footprints and have pre-committed values on the

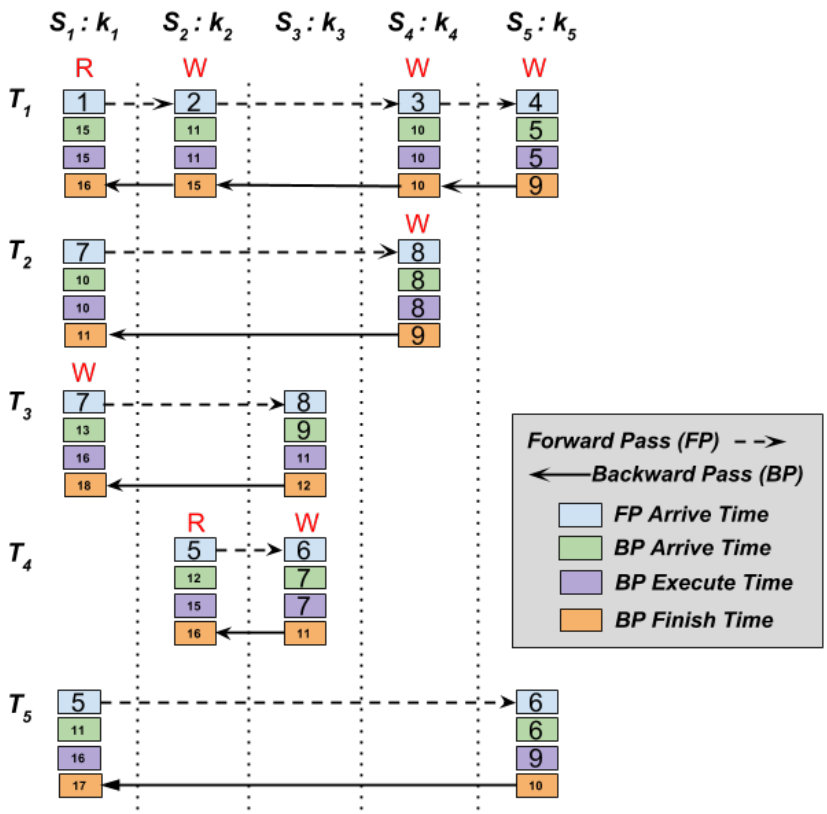


Figure 4.26 – An example execution of transactions under transaction chain concurrency control

accessed data partition. If not, the transaction may read or pre-commit on the data partition. Otherwise, the pre-committed transactions are added as dependent transactions of the current transaction. The following backward pass of the transaction needs to strictly obey the dependency, i.e. ordering behind the dependent transactions. It summarizes as the first execution rule. Specifically, for example, in Figure 4.26, T_1 has conducted backward pass and pre-committed a write on partition $S_5 : k_5$ before T_5. So, T_5 adds T_1 as its dependent transaction. Then, T_5 backward passed to $S_1 : k_1$ before T_1. T_5 knows T_1 will access $S_1 : k_1$ because it has left a footprint on $S_1 : k_1$ during its forward pass. Thus, T_5 needs to wait for T_1 on $S_1 : k_1$ even it arrives earlier in order to maintain linearizability on $S_1 : k_1$ and $S_5 : k_5$.

Execution Rule 1. *A transaction depends on another transaction if it comes later to the first shared partition in its backward pass. And the transaction is consistently ordered after transactions that it depends on regarding all the shared partitions afterwards.*

In addition to explicit dependencies added by Rule 1, a transaction also has to satisfy a set of implicit dependencies. Implicit dependencies are added to a transaction to prevent cyclic dependencies. For example, in Figure 4.26, according to Rule 1, T_3 is ordered

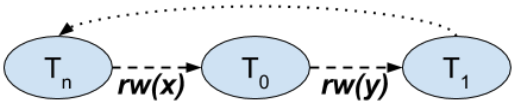


Figure 4.27 – Potential Cyclic Structure

after T_4 when accessing $S_3 : k_3$. And T_4 is ordered after T_1 when accessing $S_2 : k_2$. Transitive relation gives T_3 should be ordered after T_1, otherwise a cyclic dependency will form. However, without any hints, T_3 could be ordered before T_1 when it arrives faster on $S_1 : k_1$.

Implicit dependencies are added by a transaction resolver, which has a global view of potentially conflicting transactions in all chain servers. Detecting complete cyclic behaviors could be a NP-hard problem. Our resolver uses the pattern shown in Figure 4.27 to detect potential cyclic behaviors, which is proved to be effective and efficient in detecting cyclic dependency in transactions [108]. A topology sorting request is sent from a chain server to the resolver, when the above pattern is captured. The resolver provides a serializable sorting of the transactions that does not violate the observed constraints recorded in the transaction dependency repository, where previous dependencies are stored. Then, the sorting result is returned to the chain server and recorded in the transaction dependency repository for future queries.

Continuing the above example in Figure 4.26, T_4 knows that it is ordered before T_3 on $S_3 : k_3$. and when it knows that it is ordered after T_1 on $S_2 : k_2$, the pattern in Figure 4.27 forms. So, S_2 requests a topology sorting to the resolver. The resolver returns the only serializable topology sorting T_3 ordered after T_1. When T_3 passes to $S_1 : k_1$, the pattern also forms because it has dependency with T_4 and about to have dependency with T_1. So, $S_1 : k_1$ queries the resolver, which will return the already calculated constraint in its repository, which is $T_3 \Rightarrow T_1$. So, T_3 waits for T_1 on $S_1 : k_1$.

When a transaction has acquired both the explicit and implicit dependencies on a chain server, it attempts to read/write temporary values on the chain server, which comes as the second execution rule.

Execution Rule 2. *If all dependent transactions have already pre-committed or aborted on the particular chain server, the current transaction is able to pre-commit. Otherwise, the transaction needs to wait until the condition is satisfied.*

Validate Transactions

Since Catenae speculatively executes transactions in all DCs using TC without synchronization, some conflicting transactions can be executed in different orders. We made this design choice to tradeoff Catenae's percentile performance for its average performance. To handle the outliers, when a transaction has pre-committed on all the chain servers through its backward pass, the execution results of the transaction need to be validated.

Non-conflicting Transactions. If transactions access data partitions that are solely accessed by themselves, there is no need for transaction validations since the commit or-

ders among these transactions can be different in different DCs while serializability is still preserved. These transactions are non-conflicting. A transaction that is not conflicting with the others when the accessed data partitions are not accessed by other transactions until the end of this transaction's backward pass. A transaction can finish its backward pass at different times in different DCs and this may cause inconsistent judgement on whether the transaction is non-conflicting or conflicting. To solve this issue, a priority is given to the DC where the transaction is initiated since it will be the DC that returns the execution result to the client. If this DC decides a transaction to be non-conflicting, then it will skip the validation phase and return the results to the client directly. In this case, a dominant result will be propagated to other DCs to commit the transaction.

Conflicting Transactions. Conflicting transactions need to reach a consensus among DCs on their execution dependencies. The execution results and the execution dependencies are part of the payload in the epoch messages as described in Section 4.3.2. Upon receiving the majority execution results of a transaction from other coordinators, a second phase of the Paxos algorithm [98] is executed independently on all coordinators. Specifically, when there are a majority of DCs that have executed the transaction with the same dependency, then the transaction will be committed with this dependency. DCs that have executed the transaction with this dependency prepare to commit the temporary read/write operations from chain servers to their underlying storage servers. DCs that have performed the transaction with other dependencies will need to perform a catch up procedure explained below.

Commit Conflicting Transactions

When a transaction is allowed to commit in a DC, it checks whether its dependent transactions are committed or aborted. If all its dependent transactions are committed, the transaction is able to commit by choosing a commit timestamp from the intersect of decision periods from all DCs. The decision period is a period of epochs when all DCs are expected to received the execution result of a transaction. The lower bound of a decision period is calculated using the current epoch of a DC plus an estimated message delay among all DCs. The upper bound of the decision period is the lower bound plus an offset, which denotes the maximum delay that can be tolerated during message transmission among DCs. The decision period of a transaction from different DCs might be slightly different because of the difference of the execution environment. We deterministically choose the maximum epoch of the intersect of the decision periods from all DCs to tolerate possible delays on the arrivals of the execution results from other DCs. Then, the transaction commit message is sent to all the involved chain servers and, in the meantime, the transaction is returned to the client. Chain servers that have received commit messages from the coordinator commit the operations from the transaction to the underlying storage server and remove the corresponding dependency.

If there are uncommitted dependent transactions the pre-committed transaction has to wait until the dependent transactions are committed, caught up or aborted. In this case, the commit epoch may increase beyond the decision period and is deterministically chosen to be the next epoch of the last committed dependent transaction. If the dependent transactions

need to catch up, the transaction will need to catch up as well, since it is executed with a super-set dependency. If the dependent transactions are aborted, then the transaction is able to commit if the transaction only write-dependent on the shared key, otherwise, the transaction is aborted as well.

Transaction Catch Up. DCs that have executed a transaction with a different dependency from the majority dependency need to catch up its execution. The catch up of a transaction is executed when all its dependent transactions are committed, aborted or caught up. The catch up procedure applies update operations in a transaction with the majority voted timestamps to the underlying storage system.

Transaction Abort. Transactions can be aborted for various reasons. For example, aborts are issued by Catenae when no majority can be reached on the execution results from all DCs. Aborting a transaction removes its dependency and temporary updates on the chain servers and the resolver.

Read Only Transactions

The advantage of a geographically distributed transactional storage system is its ability to serve data close to its clients, which achieves low service latency. In order to achieve that, it is essential to support transactions that can be executed and returned locally. Catenae allows read only transactions to be executed locally while still maintaining ACID property.

Read only transactions are processed by reading the desired values concurrently from the corresponding chain servers. Read only transactions can be returned when it is not in the decision period of a transaction with uncommitted write on a read data partition. Since all transactions with write operations are committed by choosing the largest possible timestamp of the intersect of decision periods from all DCs, it is safe to read values from the underlying storage servers before the lower bound of a decision period. If the read only transaction has read a data partition during the decision period of an uncommitted transaction that has uncommitted writes, it will retry after a short delay.

4.3.4 Evaluation of Catenae

The evaluation of Catenae is performed on Google Cloud Compute with three DCs. The performance of Catenae is compared against Paxos commit [8] over Two-Phase Lock (2PL) [9] and Paxos commit over Optimistic Concurrency Control (OCC) [10]. The evaluation of Catenae focuses on performance metrics including transaction commit latency, execution concurrency (throughput) and commit rate. We measure the performance of Catenae under different workload compositions and setups to explore its most suitable usage scenarios using a microbenchmark and standard TPC-C benchmark.

Implementation

Catenae is implemented with over 15000 lines of Java code. Chain servers and coordinators are implemented as state machines. They employ JSON to serialize data and Netty sockets to communicate among chain servers and coordinators.

2PL and OCC implementation. Two-phase Lock is implemented by using Paxos commit for managing data replication among DCs and two-phase lock inside DCs to avoid conflicts. There is a coordinator in each DC to manage the lock table and synchronize data replicas when transactions commit. During transaction execution, the coordinator acquires locks and issues temporary writes of the involved data partitions to corresponding data servers (first phase of Paxos commit). The coordinator is able to lock a data partition when the majority of DCs are able to lock the data partition. During transaction commit, the coordinator issues commit messages to other DCs. A transaction is committed when a majority of DC commit (second phase of Paxos commit). Wound-wait mechanism [109] is used to avoid deadlocks.

Optimistic concurrency control also cooperates with Paxos commit to manage data replicas. There is a coordinator in each DC to validate the execution results and synchronize data replicas when transactions commit. Transactions are distributed and executed in all DCs with records on the versions of data partitions that have been read and written. Temporary values are buffered on data servers. Temporary execution results with versions of accessed partitions are voted and validated among coordinators (first phase of Paxos commit). A transaction commits when a majority of DC commit (second phase of Paxos commit). Our OCC implementation allows aborted transactions to retry one time before returning aborts to clients.

Cluster Setup

Our evaluations are conducted using Google Cloud Compute Engine. Specifically, Catenae, 2PL and OCC systems are deployed in three DCs, i.e. europe-west1-b, us-central1-a and asia-east1-a. Inside each DC, four Cassandra nodes are used as storage backend running on Google n1-standard-2 instances, which have 2 vCPUs and 7.5 GB memory. Each DC has an isolated Cassandra deployment since data replication is already handled. Catenae chain server, 2PL server daemon and OCC server daemon are deployed on the same servers as Cassandra nodes. Committed writes are propagated to Cassandra using the write-one interface. A Google Cloud n1-standard-8 instance (8 vCPUs and 30 GB memory) is initiated in each DC to serve as coordinator in all three systems. For Catenae, transaction resolver is configured together with coordinator. A Google Cloud n1-standard-16 instance (16 vCPUs and 60 GB memory) is spawned in each data to run the workload generator. The workload generator propagates workload to services deployed in the same DC. Another n1-standard-16 instance is spawned in each data to serve as frontend client server.

Configuration of Catenae. The epoch length in Catenae is configured as 10 ms, which yields reasonable tradeoff between coordinator utilization and transaction synchronization delays as later shown in Figure 4.29.

Microbenchmark

We implement a workload generator that is able to generate transactions with different number of accessed partitions, different operation types (read/update/insert) and different distribution (Uniform/Zipfian) of accessed partitions. Under different workload compo-

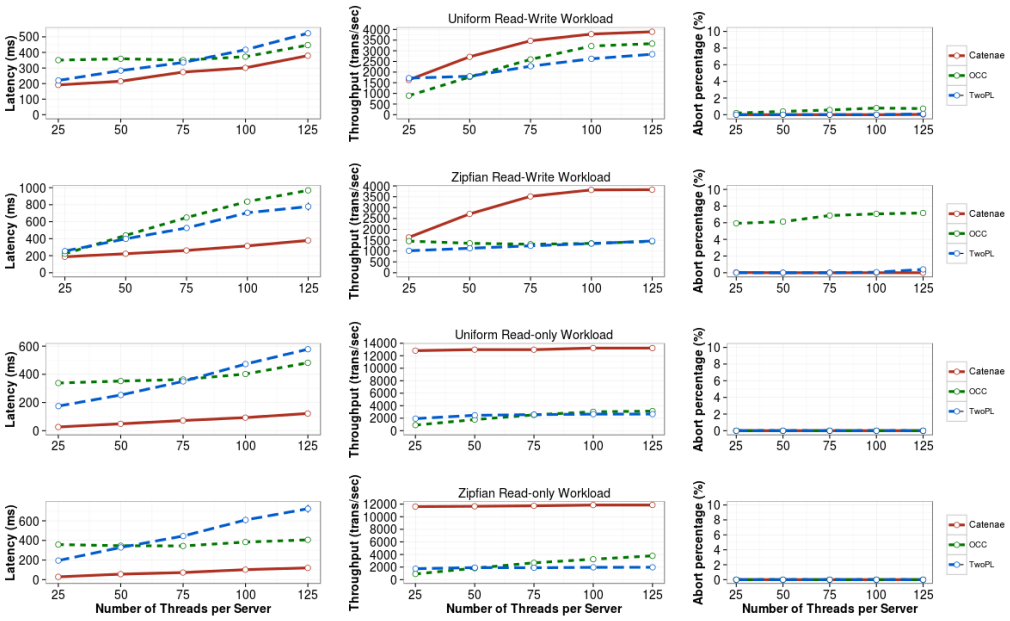


Figure 4.28 – Performance results of Catenae, 2PL and OCC using microbenchmark

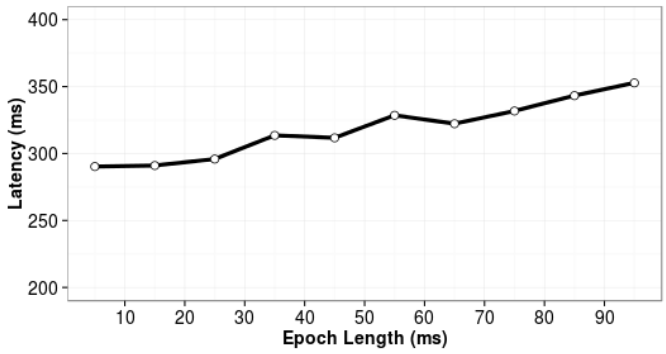


Figure 4.29 – Commit latency VS. varying epoch lengths using 75 clients/server under uniform read-write workload

sitions, we evaluate the performance of Catenae and the results are compared with 2PL and OCC. Then, an evaluation on the effect of varying epoch length in Catenae is also presented.

Workloads. We evaluate with two types of transactional workloads, i.e. read-only and read-write, with a namespace of 100000 records. Read workload is constructed with transactions that only read on data partitions. Read-write workload is formed with transactions that read, write or update data partitions. An update is translated to a read followed by a write on the same data partition. Each transaction randomly embeds one to five data partitions to be accessed. The access pattern of the involved data partitions can be uniform or zipfian with the exponent equals to one.

Results. Figure 4.28 shows the evaluation results of Catenae, 2PL and OCC under

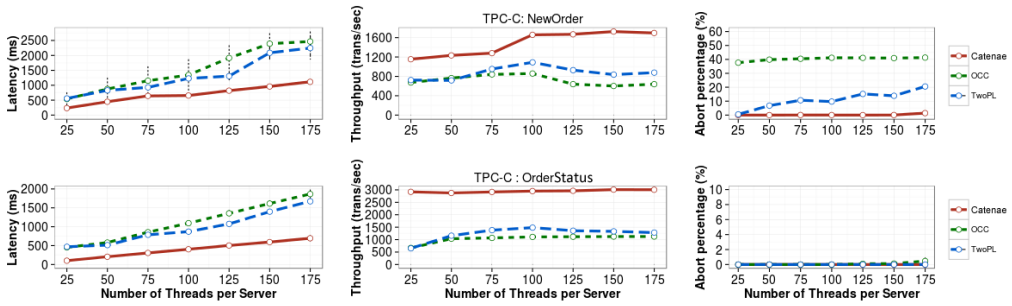


Figure 4.30 – Performance results of Catenae, 2PL and OCC under TPC-C NewOrder and OrderStatus workload

read-only and read-write transactional workloads with uniform and zipfian data access pattern. We use commit latency, throughput and abort rate as performance metrics. The results shown in Figure 4.28 are the aggregated values from three DCs.

The performance of Catenae, 2PL and OCC is comparable under uniform read-write workload. Under this workload, all three approaches need to synchronize data replicas with remote DCs but transactions are not likely to conflict with each other since the data access pattern is uniform. Catenae outperforms 2PL and OCC because of the application of epoch boundary protocol and the separation of conflicting and non-conflicting transaction sets. They have enabled Catenae to commit non-conflicting read-write transactions with a little more than a half RTT and commit conflicting read-write transactions with slightly more than a single RTT.

The throughput of 2PL and OCC start to struggle and plateau with the increasing number of clients under zipfian read-write workload, where transactions are likely to conflict with each other. As expected, OCC observes significant abort rate under this workload. On the other hand, Catenae scales nearly linearly under both uniform and zipfian workload. This is because of the efficient scheduling of concurrent transactions using the transaction chain concurrency control. Specifically, transactions are not contended until they begin accessing a shared data partition concurrently. Only at this point, the execution dependencies are established. Even so, transactions are allowed to proceed and commit given that the established dependencies are preserved. In sum, the speculative execution using transaction chains achieves very high success rate even under zipfian (with exponent=1) workload as validated in Section 4.3 (Figure 4.24).

The performance of 2PL and OCC under read-only workload is similar to their performance under uniform read-write workload since both workloads requires 2PL and OCC to synchronize replicas in remote DCs. The only different is that there is no conflict while executing and committing transactions, which leads to a higher throughput and lower commit latency in both approaches. In contrast, Catenae observes more than three times performance gains in both latency and throughput since read-only transactions can be processed locally in Catenae. It is enabled because the lower-bound of EID that a write-involved transaction is scheduled to be committed is known when it enters the validation phase, which requires a proposal of a decision period (Section 4.3.3). Thus, it is safe to return a read-only transaction locally when its timestamp is lower than the lower-bound of the decision period

of a write-involved conflicting transaction. Otherwise, the read-only transaction is retried after 50 ms.

Varying the Epoch Length. To further study the performance of Catenae, the varying size of epoch length is evaluated. As shown in Figure 4.29, transaction commit latency starts to increase steadily with epoch length more than 20 ms. This is because that transactions will only be executed after they are propagated to all DCs. The longer the epoch length, the more delay is imposed on transactions. However, too short epoch length leads to frequent exchanging of epoch messages among coordinators, which introduces performance bottleneck on the coordinators. Thus, there is a tradeoff between the length of an epoch and the overhead imposed on coordinators. So, we choose 10 ms to be the epoch length in Catenae in all the evaluations.

TPC-C

We implement TPC-C under the current specifications [110] with interfaces that propagate workload to Catenae, 2PL and OCC. Two representative operations, i.e. *NewOrder* and *OrderStatus*, are chosen to be evaluated.

Results. Figure 4.30 illustrates the evaluation results of Catenae, 2PL and OCC under extremely stressed TPC-C workload. The results are aggregated from the three operating DCs. Catenae is able to scale up from 25 clients/server to 100 clients/server under TPC-C *NewOrder* workload, after which its performance stays stable. 2PL and OCC follow similar scale up pattern. However, they only achieve roughly half of the throughput comparing to Catenae, which causes the doubling of latency. With more than 100 clients/server, there is a drop of throughput in OCC and 2PL because of high contention. The abort rate of 2PL increases when there are more conflicting transactions waiting for locks, since we have set a timeout on waiting for locks. OCC suffers from constantly significant abort rates under the *NewOrder* workload since there is extremely high read-write contention among transactions. Catenae maintains very low abort rates by efficiently scheduling concurrent transactions using transaction chain algorithm. It allows Catenae to achieve higher concurrency, which leads to a higher throughput of transaction execution. Additionally, the high success rate of speculative execution even under contended workload allows Catenae to commit transactions with low latency as shown in Figure 4.24. Thus, the faster transactions commit, the less contention is experienced in Catenae.

OrderStatus is a read-only transaction. Catenae judiciously processes read-only transactions in local DCs when the accessed records are not about to be committed to an updated value. This condition is always true when running a read-only workload against Catenae. Thus, Catenae is able to commit read-only transactions locally, which significantly reduce the commit latency and boosts the throughput. In contrast, 2PL needs to check and obtain read locks across DCs while OCC requires to validate the read values across DCs. As a result, Catenae achieves more than twice the throughput of 2PL and OCC with nearly 70% less commit latency.

4.3.5 Discussions

Speculative Execution

Catenae provides efficient transaction support on top of fully replicated data stores, such as [3, 33, 12, 37, 6]. Since Catenae relies on a deterministic duration that a transaction accesses a specific data partition on a specific chain server, it is desirable, but not mandatory, to deploy chain servers symmetrically, i.e., using the same VM flavor to host the same namespace range, in all DCs. For performance predictability and cost control, it is common and reasonable to host the instances of a specific component of an application using the same VM flavors in today's Cloud platforms. Hosting the chain servers of Catenae asymmetrically among DCs will increase the possibility to have an inconsistent transaction execution dependencies during speculative executions among DCs. This will not influence the correctness of Catenae but triggering the catch up procedure and delaying the transaction commit to two RTTs, which is the same latency overhead comparing to the classic 2PL over Paxos commit. We validated in Section 4.3, it is possible to achieve the same execution dependency in most of transactions speculatively executed using TC concurrency control in each DC. The consistent speculative execution allows transactions to be committed with a single RTT.

Liveness among Data Centers

Catenae does not pre-order transactions before execution, they are allowed to compete and concurrently execute at runtime. It maximizes the concurrency of transaction executions. However, Catenae expects DCs to execute the same set of transactions received from the epoch messages sent from all DCs, which leads to the most consistent transaction dependencies during speculative execution. The consistent dependencies during speculative execution allows Catenae to have extremely low commit latency, but comes with a tradeoff. An outage of a DC could cause other DCs to block waiting for its epoch message, which contains the transactions received in that DC. The blocking continues when the expected epoch messages are eventually delivered. This is similar to blocking scenarios in 2PL, that could be overcome by using state machine replication. Catenae applies a time-based delay tolerance technique to ascertain the state of a failed DC. Large delay tolerance may result in endless waiting for the epoch messages from a failed DC, that largely influences the performance. Small delay tolerance may neglect the transactions happened in the "suspected failed" DC and result in periodic high transaction abort rates in that DCs or a lot of transaction catch up workload across DCs. Thus, this design choice tradeoffs the high possibility to have consistent dependency during speculative execution with the complication of failure handling.

On the other hand, Catenae can be adapted to operate while receiving only majority epoch messages. Specifically, upon receiving the majority epoch messages, Catenae proceeds to transaction scheduling and execution phase. The incomplete receipt of transactions from DCs will lead to a higher possibility to have divergent transaction execution dependencies. The inconsistent execution dependency will be corrected by the selection of majority execution dependency during the validation phase with another RTT. Thus, the tradeoff

allows Catenae to operate in a failure-prone environment but with a significant overhead for catching up inconsistent transaction executions. The possibility of having inconsistent transaction executions while scheduling transactions when receiving a majority of epoch messages is evaluated in Section 4.3 and shown in Figure 4.24.

Liveness among Transactions

The transaction chain maintains serializability and liveness among transactions by ensuring that the dependencies among transactions are acyclic. A dependency is added to a pair of transaction by a chain server only when such dependency does not exist and will not generate cyclic dependency implicitly. Specifically, a chain server will not add a dependency contradictory to the dependency already embedded in the transaction. Dependency gradually propagates among chain servers with the passing of transactions and transactions are order linearly by chain servers with their observed dependency. Cyclic behavior can only happen when chain servers do not have enough information regarding some concurrent transactions, as shown in one example in Section 4.3.3. This kind of cyclic dependency is prevented by transaction resolver, who adds implicit dependency to transactions. Implicit dependencies are added when a superset of cyclic behaviors (as illustrated in Figure 4.27) are detected.

4.3.6 Summary and Discussions of Catenae

We present Catenae, a geo-distributed transaction framework that provides serializability. It leverages novel epoch boundary synchronization protocol among DCs to improve transaction commit latency and extends the transactions chain algorithm to efficiently schedule and execute transactions in multiple DCs. We show that Catenae only needs one single inter-DC communication delay to execute non-conflicting geo-distributed read/write transactions and one RTT to execute potentially conflicting geo-distributed transactions most of the time. The worst case commit latency of Catenae requires two RTTs. Catenae achieves more than twice the throughput than 2PL and OCC with more than 50% less commit latency under TPC-C workload.

However, there are certain limitations of Catenae. Similar to MeteorShower, Catenae also extensively exploits network resources of each servers. It affects the percentile execution latency of Catenae. Furthermore, the performance of Catenae depends on the determinism in transaction execution time on each data partition. Thus, when most of the transactions do not have unique processing time on data partitions, Catenae will not perform better than the state-of-the-art approaches. Also, the rollback operations in Catenae may trigger cascading aborts if the workload is highly skewed. Another possible limitation of Catenae is its application scenario. Essentially, Catenae can only process transactions that are chainable. It means that data items that are accessed by a transaction should be known before its execution. And the accesses of these data items should follow a deterministic order. Thus, Catenae cannot execute any types of transactions.

Chapter 5

Achieving Predictable Performance on Distributed Storage Systems with Dynamic Workloads

5.1 Concepts and Assumptions

Predictable Performance

In this section, we study the performance of distributed storage systems. We focus our study on using the request latency (average/percentile) of storage systems to define their quality of service. Instead of improving the request latency of storage systems, as studied in the previous section, we focus on providing predictable performance in this section. When we refer to predictable performance, we mean that the request latency of a storage system remains stable, as predicted, despite of the uncertainty of external factors, such as fluctuation in incoming workloads or disturbance from background tasks. In the context of Cloud computing, Service Level Agreements between Cloud providers and consumers sometimes cover the concept of predictable performance. Specifically, there are multiple Service Level Objectives specified in an SLA. In particular, we focus on latency based SLOs. They require that the request latency, can be average latency or percentile latency, should be maintained under a specific threshold. This is the SLO that we study in this section.

The platform

We investigate the performance of distributed storage systems hosted in the Cloud. We distinguish the notation of server and host in this context. When we refer to servers, we mean a service of an application that runs on a virtual machine spawned from a Cloud platform. A host, on the other hand, refers to a physical machine that hosts several virtual machines.

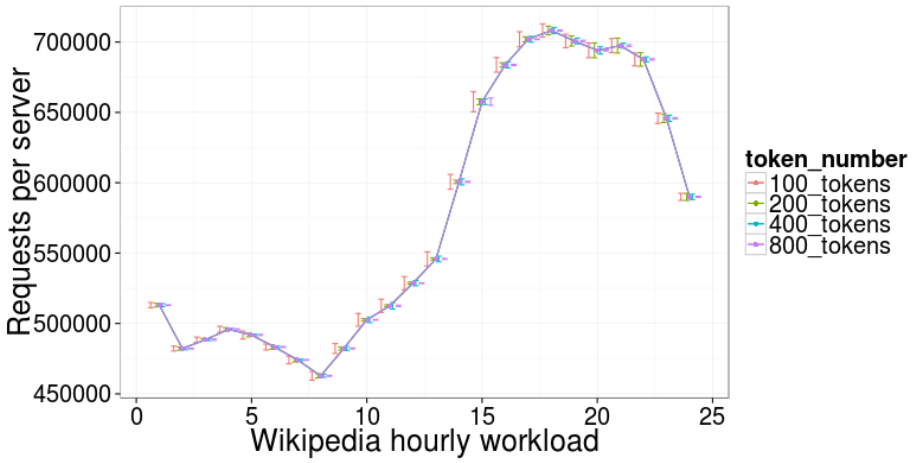


Figure 5.1 – Standard deviation of load on 20 servers running a 24 hours Wikipedia workload trace. With larger number of virtual tokens assigned to each server, the standard deviation of load among servers decreases

Load Balance among Storage Nodes

It is challenging to balance the load among storage nodes in a distributed storage system. This is because that each client request can be only served by storage nodes that host the requested data. The study of load balancing in distributed storage systems is not the focus of this thesis. We have conducted a simple evaluation on testing the load distribution on storage nodes when they are organized using a DHT (distributed hash table). We vary the number of virtual tokens in each storage node. A virtual token allows a storage node to store a specific portion of data mapped by the hash algorithm. We have simulated the workload using a 24 hour Wikipedia access trace. We demonstrate in Figure 5.1 that with a properly configured namespace and the size of virtual nodes, the Wikipedia workload roughly imposes equal loads on each storage node. We conclude that with sufficient number of virtual tokens, requests tend to be evenly distributed among storage nodes under diurnal workload patterns similar to the Wikipedia workload. And a roughly balanced load on each storage node is the scenario that we assumed for our proposed solutions in later sections.

5.2 BwMan

In order to achieve predictable performance in a distributed storage system, we demonstrate the necessity of managing resources shared by services running on the same server or host. Our first approach studies the impact of regulating network bandwidth shared by services running on the same server considering that distributed storage services are bandwidth intensive. Then, in later sections, we illustrate the effect of resource management among services sharing the same physical host.

The sharing of network bandwidth can emerge among multiple applications on the same

server or among services of one application. In essence, both cases can be solved using similar bandwidth management approaches. The difference is the granularity in which bandwidth allocation is conducted, for example, on the level of applications or service threads. We decide to achieve the finest management granularity of network bandwidth, i.e., level of service ports, since it can be easily adapted in any usage scenarios mentioned above. Essentially, our approach is able to distinguish bandwidth allocations to different ports used by different services within the same application.

We have identified that there are two kinds of workloads 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*. Typically, system-centric workload is triggered in the following situations. When the system scales out, the number of servers and the total storage capacity are increased, which triggers data migration to the newly added servers. Similarly, when the system scales in, data needs to be migrated before the servers can be 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. In sum, all system-centric workloads trigger data migration, which consume system resources including network bandwidth. The data migration workloads interfere with user-centric workloads and cause performance degradation in serving client requests.

It is intuitive and validated in our later evaluations that both user-centric and system-centric workloads are network bandwidth intensive. However, arbitrating the allocation of bandwidth between these two workloads is non-trivial. On the one hand, insufficient bandwidth allocation to user-centric workload might lead to performance degradation. On the other hand, the system may fail when insufficient bandwidth is allocated to failure recovery. Similarly, without sufficient bandwidth, the resizing of the system may take too long to finish and miss the deadline of finishing the scaling operations [68].

We have designed a bandwidth controller named **BwMan**, which uses easily-computable predictive models to foresee the performance under a given workload (user-centric or system-centric) in correlation to bandwidth allocation. It judiciously allocates network bandwidth to activities concerning user-centric and system-centric workloads. The user-centric performance model defines correlation between the incoming workload and the allocated bandwidth with respect to a performance metric. We choose to manage the request latency. The system-centric model defines correlation between the data migration speed and the allocated bandwidth. Data migration speed defines the recovery speed of data corruption or server failure or the speed of carrying out system resizing operations.

5.2.1 Bandwidth Performance Models

The mathematical models in BwMan 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.

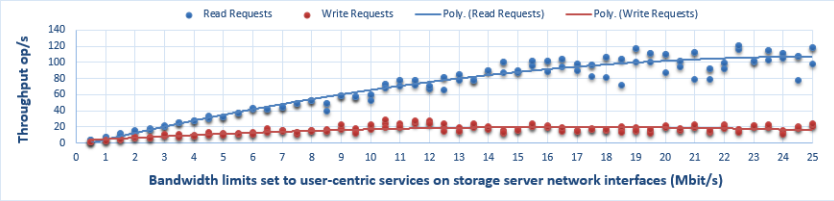


Figure 5.2 – Regression Model for System Throughput vs. Available Bandwidth

User-centric Workload versus Allocated Bandwidth

For user-centric workload, we measure the maximum throughput that can be achieved with respect to a certain request latency requirement under a specific network bandwidth allocation. Thus, BwMan is able to use the model to arbitrate bandwidth to user-centric workload when knowing the incoming workload. And this bandwidth allocation strategy satisfies a certain request latency requirement. In other words, it satisfies a latency SLO. 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 that allows average latency to be under 1s as shown in Fig. 5.2.

System-centric Workload versus Allocated Bandwidth

The correlation between system-centric performance and the allocated bandwidth is modeled in Figure 5.3. The model is trained off-line by varying the bandwidth allocation and measuring data recovery speed. The predictive process is centrally conducted based on the monitored data integrity of the whole system and bandwidth are allocated homogeneously to all storage servers. For the moment, we do not consider the fine-grained monitoring of data integrity on each storage node. We treat data integrity at the system level.

5.2.2 Architecture of BwMan

In this section, we describe the architecture of BwMan, which operates according to the MAPE-K loop [111] (Fig. 5.4) passing the following phases:

- Monitor: monitor the incoming client workload and system-centric workload;
- Analyze: feed monitored data to the regression models;
- Plan: use the predictive regression model to plan bandwidth allocations. A tradeoff has to be made when the total network bandwidth has been exhausted and cannot satisfy all workloads. The tradeoff policy is specified in Section 5.2.2;
- Execute: allocate calculated bandwidth to service ports concerning each workload according to the plan.

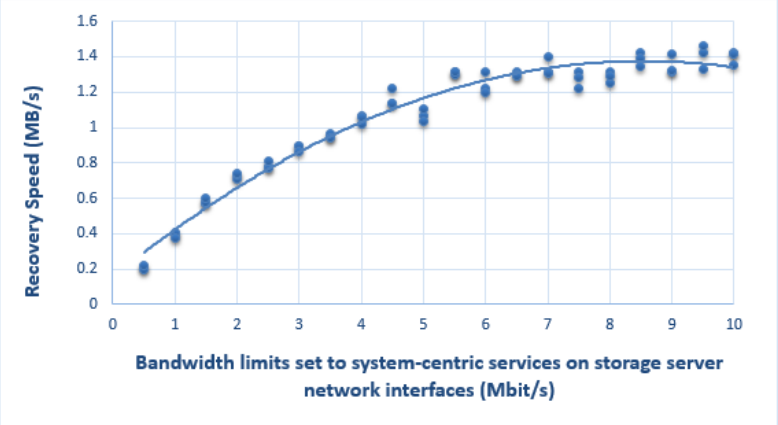


Figure 5.3 – Regression Model for Data Recovery Speed vs. Available Bandwidth

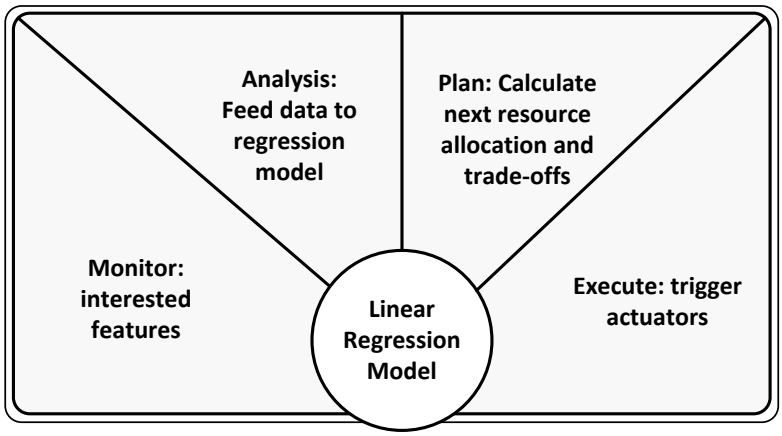


Figure 5.4 – MAPE Control Loop of Bandwidth Manager

BwMan Control Flow

The flowchart of BwMan is shown in Fig. 5.5. BwMan monitors two signals, namely, the user-centric workload and the system-centric workload. At given time intervals, the gathered data from each storage node is averaged and fed to analysis modules. Then the results of the analysis based on our regression models are passed to the planning phase to decide actions with potential tradeoffs. The results from the planning phase are executed by the actuators in the execution phase.

In details, for the Monitor phase, we have monitored two ports on each server, one for servicing user-centric workload (M1) and the other for data migration (M2). The outputs of this stage are passed to the Analysis phase represented by two calculation units, namely A1 and A2, that aggregate and calculate new bandwidth allocation according to the predictive performance models (Section 5.2.1). During the planning phase, BwMan checks whether bandwith allocations need to be updated comparing to the previous control period. In addi-

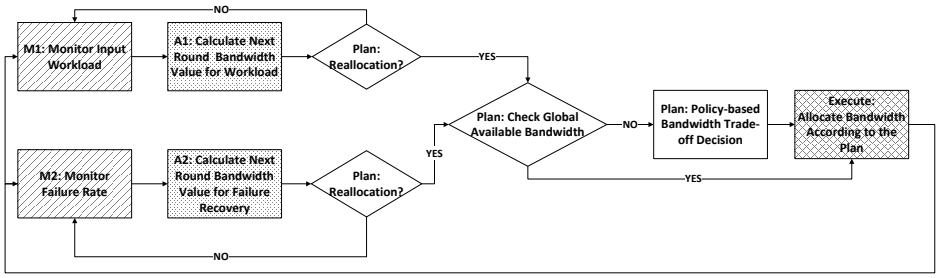


Figure 5.5 – Control Workflow

tion, it checks whether the new bandwidth allocation plan violates the maximum bandwidth available on the server. If this constraint is violated, then a tradeoff between user-centric workload and system-centric workload needs to be made. Finally, during the Execution phase, the actuators are employed to update bandwidth allocation to each service port.

Tradeoff Scenario

Since bandwidth is a finite resource on each server, so it might not be able to satisfy all workloads. We describe a tradeoff scenario where the bandwidth is shared among user-centric and system-centric workloads. For user-centric workload, a specific amount of bandwidth needs to be allocated in order to meet a specific performance requirement, i.e. request latency, under a specific workload. For system-centric workload, there might be a recovery speed constraint for data corruption or server failure to maintain data integrity of the system or a deadline to carry out a system resizing operation to meet an increasing workload. Thus, system-centric workload also requires a specific amount of bandwidth allocation for a certain data migration speed to satisfy the above activities. By arbitrating bandwidth allocated to user-centric and system-centric workloads, we can enforce more user-centric performance while penalizing system-centric operations or vice versa. The tradeoff policy in BwMan can be configured either in preference of user-centric workload or system-centric workload. As a result, the bandwidth requirement from one of these two workloads will be satisfied first.

5.2.3 Evaluation of BwMan

The evaluation of BwMan is conducted on a virtualized platform, i.e., an OpenStack Cloud. The underlying distributed storage system that BwMan manages is OpenStack Swift, which is a widely used open source distributed object storage started from Rackspace [112]. We confirm that, in Swift, bandwidth allocations for user-centric workload and system-centric workload are not explicitly managed. We observe that data migration in the case of server failure, data corruption or system resizing essentially use the same set of replicator processes based on the "rsync" Linux utility. Thus, for simplicity, we trigger data migration in Swift using a process that randomly corrupts data with a specified speed.

Swift Setup

We have deployed a Swift cluster with 1 proxy server to 8 storage servers as recommended in the OpenStack Swift documentation [113]. The proxy server is hosted on a VM with four virtual cores (2.40GHz), 8GB RAM while the storage servers are hosted on VMs with one virtual core (2.40GHz), 2GB RAM and 40GB disk size.

User-centric Workload Setup

We have modified the Yahoo! Cloud Service Benchmark (YCSB) [100] 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. If the request rate cannot be served by the storage system, requests are queued for later executions. 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 [114] 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.

Data Corruptor and Data Integrity Monitor

We have developed a script that uniformly at random chooses a storage node, in which it corrupts a specific number of files within a defined period of time. This procedure is repeated until the specified data corruption rate is reached. The process triggers Swift's failure recovery process and results in data migration in Swift.

We have customized the swift-dispersion tool in order to populate and monitor the integrity of the whole data space. This customized tool also acts as data integrity monitor in BwMan, which provides real-time metrics on the system's data integrity.

The Actuator: Network Bandwidth Control

We apply NetEm's tc tools [106] in the token buffer mode to control the inbound and outbound network bandwidth associated with the network interfaces and service ports. In this way, we are able to manage the bandwidth quotas for different activities in the controlled system. In our deployment, services run on different ports, and thus, we can apply different network management policies to them.

Evaluation Scenarios

The evaluation of BwMan in OpenStack Swift has been conducted under two scenarios. First, we evaluate the effectiveness of BwMan for the user-centric workload and system-centric workload under the condition that there is enough bandwidth to handle both work-

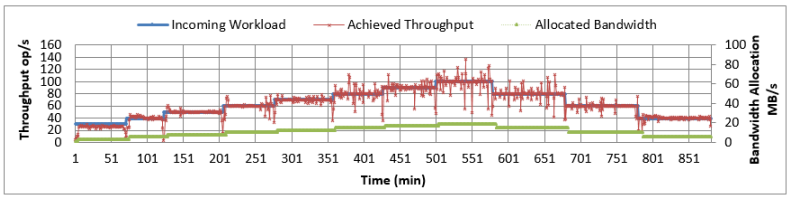


Figure 5.6 – System Throughput under Dynamic Bandwidth allocation using BwMan

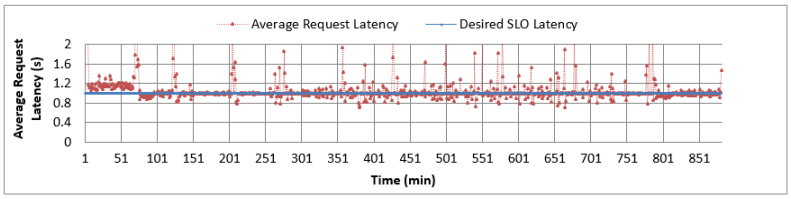


Figure 5.7 – Request Latency under Dynamic Bandwidth allocation using BwMan

loads. These experiments demonstrate the ability of BwMan to manage bandwidth 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 priority to the user-centric workload compared to the system-centric workload. We show that BwMan is able to maintain a desired average request latency for the Swift cluster.

The First Scenario

Fig. 5.6 and Fig. 5.7 present the effectiveness of using BwMan in Swift to guarantee a certain latency objective (SLO) under dynamic workloads. The x-axis of both plots show the experiment timeline, whereas the left y-axises correspond to workload intensity in Fig. 5.6 and request latency in Fig. 5.7. The right y-axis in Fig. 5.6 corresponds to allocated bandwidth by BwMan.

In this experiment, the workload that we generated using YCSB is a mix of 80% read requests and 20% write requests, that, in our view, represents a typical workload in a read-dominant application. The blue line in Fig. 5.7 shows the desired request latency. The achieved latencies in Fig. 5.7 demonstrate that BwMan is able to reconfigure bandwidth allocation during runtime according to dynamic workloads and achieves the desired request latency.

Fig. 5.8 presents the results of using BwMan to control bandwidth allocation in the scenario of data recovery. The blue curve sums up the data integrity of the whole system by examining 1% random sample of the whole data space. The control cycle activation is illustrated as green dots. The red curve stands for the bandwidth allocation by BwMan after each control cycle. The calculation of bandwidth allocation is based on an estimation of data corruption rate. The data corruption rate is calculated by the amount of corrupted

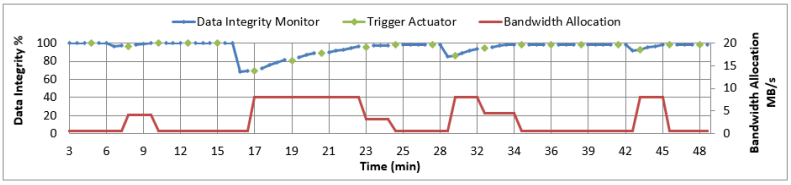


Figure 5.8 – Data Recovery under Dynamic Bandwidth allocation using BwMan

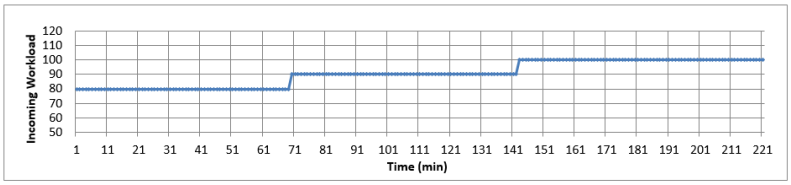


Figure 5.9 – User Workload Generated from YCSB

data over a control period. Then, the same recovery rate is mapped in Fig. 5.3 to obtain a bandwidth allocation. In this case, we assume that the system will not fail since the data recovery rate matches the data corruption rate.

The Second Scenario

In this scenario, we demonstrate that BwMan guarantees the latency SLO when the total available bandwidth is not enough for both user-centric and system-centric workloads. The tradeoff policy is set to be in favor of user-centric workload. Thus, bandwidth allocation to data recovery may be compromised to ensure user-centric performance.

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. The requests that are not served are queued for later executions. The bandwidth is dynamically allocated to meet the throughput under a specific latency SLO, i.e. 1s. The data corruptor is configured to corrupt data randomly at a specific rate, which creates bandwidth contention with the user workload.

Fig. 5.9 presents the workload generated by YCSB. Fig. 5.10 and Fig. 5.11 depict the request latency observed with/without bandwidth arbitration using BwMan.

Fig. 5.10 shows that using BwMan, the achieved request latency mostly (with only 8.5% of violation) satisfies the desired latency specified as the SLO. This is because that BwMan throttles bandwidth consumed by the data recovery process and guarantees the bandwidth allocation to user-centric workload. In contrast, Fig. 5.11 shows that without managing bandwidth between user-centric workload and system-centric workload, the desired request latency cannot be maintained. Specifically, the results indicate that there are about 37.1% latency SLO violations.

Table 5.1 summarizes the percentage of SLO violations within three given confidence intervals (5%, 10%, and 15%) with or without bandwidth management, i.e., with or without BwMan. The results demonstrate the benefits of BwMan in reducing the SLO violations

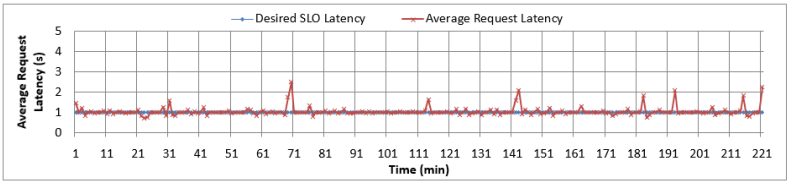


Figure 5.10 – Request Latency Maintained with BwMan

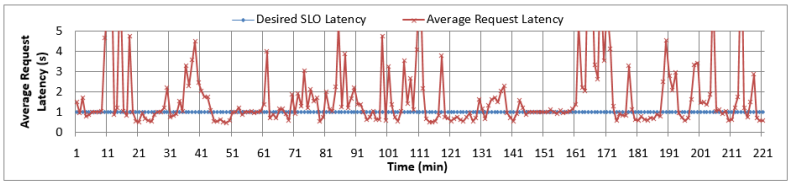


Figure 5.11 – Request Latency Maintained without BwMan

Table 5.1 – 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%

with at least a factor of 2 given a 5% interval and a factor of 4 given a 15% interval.

5.2.4 Summary and Discussions of BwMan

We present the design and evaluation of BwMan, a network bandwidth manager for services running on the same server. It allocates bandwidth to each service based on predictive models, which are built using statistical machine learning. The predictive models decide bandwidth quotas for each service with respect to specified service level objects and policies. Evaluations have shown that BwMan can reduce SLO violations for user-centric workload by a factor of two when system-centric workloads create bandwidth contention.

We show that BwMan is able to better guarantee the service level objective of user-centric workloads. However, there are limitations when applying BwMan. First of all, BwMan manages network bandwidth uniformly on each storage server. It relies on mechanisms in a storage system to balance the workload for each server. Thus, the coarse grained management of BwMan is not applicable to systems where workloads are not well-balanced among servers. Furthermore, BwMan does not scale the bandwidth allocated to a storage service horizontally. It conducts the scaling vertically, which means that BwMan only manages the network bandwidth within a single server. In other words, BwMan is not able to scale a distribute storage system when the bandwidth of storage servers are not suffi-

cient. Last but not lease, we have observed that the bandwidth quota available to a VM in a Cloud environment is not predictable. In other words, it is often not possible to know the maximum amount of bandwidth can be exclusively used by a VM. As a result, in our evaluations, BwMan arbitrates bandwidth among services in a conservative way to ensure that the assigned bandwidth are guaranteed for user-centric and system-centric workload. This is also the major reason that our evaluations are conducted with small bandwidth quotas and the system operates in an under-utilized operational region.

5.3 ProRenaTa

In the previous section, we have studied the impact of regulating network bandwidth between client-centric workload and system-centric workload, mostly data migration. It is shown that client workload and data migration compete with network bandwidth of the host. If network bandwidth is not regulated, the service quality of client requests will be affected.

Considering the scenario of scaling out of a distributed storage system, on one hand, a portion of dedicated network bandwidth need to be allocated to guarantee the service quality of the continuous client requests. On the other hand, the system needs to scale out to increase its capability to serve an increased workload in the near future. The near future is the time that the workload is expected to increase, which is usually accomplished with time-series prediction. Thus, the scaling activity has a deadline to finish, which is translated to a constraint on the minimum data migration speed. So, it also needs a portion of network bandwidth to finish the data migration with respect to the scaling deadline.

However, the amount of network bandwidth is finite in a system. Naturally, it is not wise to have the system very much over-provisioned, since we need to pay every resource that we adopted (under the scenario of Cloud computing). Thus, we consider not only the service quality achieved, but also the overall resource utilization as another factor. Then, the challenge is how to provision a distributed storage system efficiently (without too much over-provisioning) and, how to arbitrate the limited bandwidth resources between client workload and data migration to preserve system performance as well as the scaling deadline.

In this section, we present the analytical and empirical models to tackle the issue of data migration while the storage system scales out/in. Our research answers the following questions:

1. what is the role and effect of data migration during the scaling of a distributed storage system;

2. how to prevent performance degradation when scaling out/in the system involves data migration;

3. with limited amount of resources, what is the best time to start data migration in order to finish a specific scaling command on time under the current status of the system and the current/predicted workload;

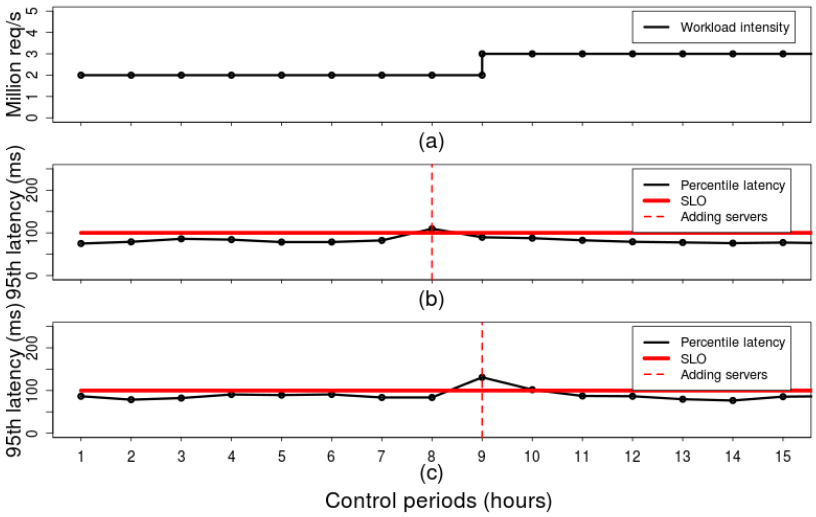


Figure 5.12 – Observation of SLO violations during scaling up. (a) denotes a simple increasing workload pattern; (b) scales up the system using a proactive approach; (c) scales up the system using a reactive approach

4. can we minimize the provisioning cost by increasing the resource utilization.

Observations

It is challenging to achieve elasticity in a distributed storage system with respect to a strict service quality guarantee (SLO) [30]. There are two major reasons. One reason is that scaling a storage system requires data migration, which introduces an additional overhead to the system. The other reason is that the scaling is often associated with delays. To be specific, adding or removing servers cannot be completed immediately because of the delay caused by data migration.

We setup experiments to validate the above argument with a simple workload pattern described in Figure 5.12 (a). The experiment is designed as simple as possible 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 5.12 (a) to keep the low latency of requests defined in SLOs. Figure 5.12 (b) and Figure 5.12 (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 SLO violations. Typically, a prediction based elasticity control tries to bring up the capacity of the storage cluster before the actual workload increase. In Figure 5.12 (b), a prediction based controller tries to add instances at control period 8. We observe the SLO violation during this period because of the extra overhead, i,e, data migration, 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 especially network bandwidths.

Another interesting observation can be seen from Figure 5.12 (c), which simulates 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 SLO 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 SLO violation because of the scaling overhead like in the previous case. Thus, it is necessary to scale the system before the workload changes.

A prediction based (proactive) elasticity controller is able to prepare the instances in advance and avoid performance degradation/SLO violations if the scaling overhead is properly handled. However, the accuracy of workload prediction largely depends on application-specific access patterns. Even using Wikipedia workload [115] where the pattern is very predictable, an certain amount of prediction errors are expected. 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 SLO guarantees and the provisioning costs.

The feedback based (reactive) approach, on the other hand, 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, SLO violations are observed in the initial phase of scaling because of data migration overhead in order to add/remove instances in the system.

In essence, it is not hard to discover that 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.

Based on these observations, we propose an elasticity controller for distributed storage systems named **ProRenaTa**, which combines proactive and reactive approaches with explicit consideration of data migration overhead during scaling.

5.3.1 Performance Models in ProRenaTa

The performance model correlates the capacity of a storage server to handle a specific amount of workload, in terms of read/write requests per second, under the requirement of latency SLO. Different models can be built for different flavors of servers using the same profiling method. Then, we use it to calculate the minimum number of servers that is needed to meet the SLO requirements under a certain workload.

The simplest case in the model is demonstrated with the black solid curve shown in Figure 5.13. It represents the scenario where there is no data migration activity in the system. The workload is transformed to the request rate of read and write operations. Under a specified latency SLO constraint, a server can be in 2 states: satisfy SLO (under the SLO

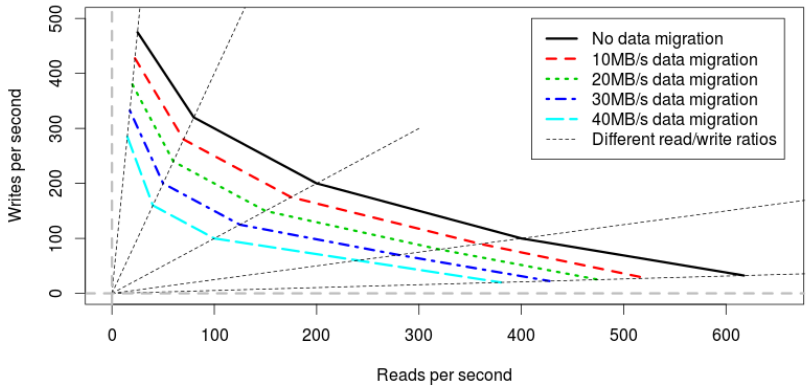


Figure 5.13 – Data migration model under throughput and SLO constraints

border in the figure) or violate SLO (beyond the SLO border in the figure). We would like to have servers to be utilized just under the SLO border to have a high resource utilization while guaranteeing the SLO requirements. The performance model takes a specific workload as input and outputs the minimum number of storage servers that is needed to handle it under the SLO. 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 SLO border in Figure 5.13.

In the real experiment, we have setup a small margin for over-provisioning. It is used to guarantee the service quality, but also leaves some spare capacity for data migration during scaling up/down. This margin is set to 2 servers in our later experiment and it can be configured differently case by case in order to tradeoff the scaling speed and the SLO commitment with resource utilization.

When data migration comes to affect, the corresponding curves in Figure 5.13 represent the maximum data migration speed that can be spared for scaling activities without compromising the SLO under the current workload. It is calculated by estimating an average workload served by each server ($Workload/NumberOfServers$) under current workload and cluster setup. Then, the workload is mapped to a point in the performance model shown in Figure 5.13. The closest border below this data point indicates the data migration speed that can be offered without sacrificing the SLO. With the maximum data migration speed obtained, the time to finish a scaling plan can be calculated.

The Analytical Model

We consider a distributed storage system that runs in a Cloud environment and uses ProRenaTa to achieve elasticity while respecting the latency SLO. 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: 1. The amount of data stored in each physical server is proportional to its capability. 2. The amount of workload distributed to each physical server is proportional to its capability. 3. 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 SLO latency constraint. The value of α and β can be obtained empirically from a trained performance model as shown in Figure 5.13.

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 the latency SLO. Thus, function f depends on system load ($\alpha^{'} * L/N, \beta^{'} * L/N$), server capability ($\alpha * C, \beta * C$) and the latency SLO ($SLO_{latency}$).

We denote the predicted workload as $L_{predicted}$. 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 calculated. 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.

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

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. $\alpha^{'}$ and $\beta^{'}$ are obtained from workload monitors. Then, the problem left is to find a proper function

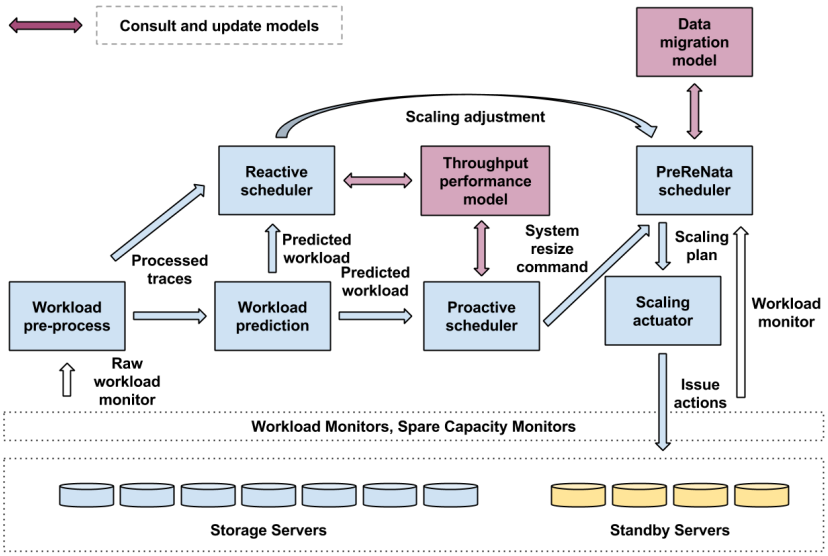


Figure 5.14 – ProRenaTa control framework

f that defines the data migration speed under certain system setup and workload condition with respect to latency SLO constraint. The function f is obtained using the empirical model explained in section 5.3.1.

5.3.2 Design of ProRenaTa

Figure 5.14 shows the architecture of ProRenaTa. It follows the idea of MAPE-K (Monitor, Analysis, Plan, Execute - Knowledge) control loop with some customizations and improvements.

Monitor

The arrival rate of reads and writes on each server 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.

Analysis

Workload prediction: The pre-processed workload is forwarded to the workload prediction module for workload forecasting. 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. The prediction methods will be explained in Section 5.3.3.

Plan

Proactive scheduler: The Proactive scheduler calculates the number of instances needed in the next PW using the performance model in Section 5.3.1. Then, it sends the number of instances 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 scaling decision from the proactive scheduler and the reactive scheduler are contradictory. ProRenaTa scheduler solves this problem by consulting the data migration model as shown in Figure 5.13, 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 SLO constraints explained in Section 5.3.1. 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

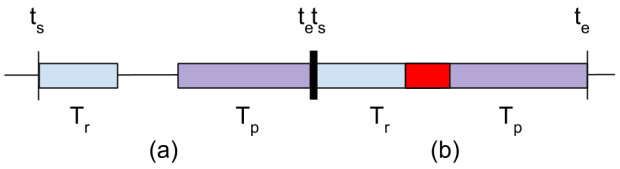


Figure 5.15 – Scheduling of reactive and proactive scaling plans

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 an 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 conducts 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 5.15. Figure 5.15(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 5.15(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.

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 [116], which is a bandwidth manager that allocates bandwidth quotas to different services running on different ports as presented in the previous Section. In essence, BwMan uses Netem tc tools to control the traffic on each storage server's network interface.

Knowledge

To facilitate the decision making to achieve elasticity in ProRenaTa, there are two knowledge bases. The first one is the performance model presented in Section 5.3.1, which correlates the server's capability of serving read and write requests under the constraint of latency SLO. In addition, the model is also able to quantify the spare capacity that can be used to handle data migration overhead while performing system resizing. 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.

Figure 5.16 illustrates the control flow of ProRenaTa. In the procedure of Proactive Control as shown in algorithm (a), $PW.T_{i+1}$ is the predicted workload at T_{i+1}, namely the start of the next control interval. The workload prediction algorithm ($workloadPrediction()$) is presented in later in Figure 5.17 and Figure 5.18. A positive value of $\Delta VMs.T_{i+1}$ indicates the number of VMs to launch (scale up). A negative value of $\Delta VMs.T_{i+1}$ indicates the number of VMs to remove (scale down). Similarly, in the procedure of Reactive Control as shown in algorithm (b), $W.T_i$ is the workload observed at T_i, using which we are able to correct the error caused by the proactive controller. ProRenaTa Scheduler as shown in algorithm (c) integrates and conducts the decisions from proactive and reactive controller. Specifically, it first calculates the resources ($RS.T_i$) currently available in the system, to reason about the maximum data rebalance speed at T_i under the constraint of maintaining the latency SLO. $T.p$ and $T.r$ are the time to finish data rebalance using the maximum possible rebalance speed for proactive and reactive scaling respectively. The decision from the proactive controller is scheduled to the latest possible time to meet the workload in the next control period while the decision from the reactive controller is scheduled immediately. Furthermore, the scheduler also calculates that whether the decisions from both controllers contradict with each other and cannot be accomplished within a control period. Then, the two scaling plans are merged.

5.3.3 Workload Prediction in ProRenaTa

We apply wikipedia workload as a use case for ProRenaTa. 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

Data: Workload trace, $Trace$
Result: Number of VMs to add or remove for the next control period
/* Program starts at time T_i */
$PW.T_{i+1}$ ←workloadPrediction($Trace$)
$VMs.T_{i+1}$ ←throughputModel(PW.T_{i+1})
$\Delta VMs.T_{i+1} \leftarrow VMs.T_{i+1} - VMs.T_i$

(a) ProRenaTa Proactive Control

Data: Observed workload, $W.T_i$
Result: Number of VMs to add or remove currently
/* Program starts at time T_i */
$\Delta W.T_i \leftarrow W.T_i - PW.T_i$
$\delta VMs.T_i \leftarrow$ throughputModel($\Delta W.T_i$)

(b) ProRenaTa Reactive Control

Data: Number of VMs to add and remove from Proactive and Reactive Controller
Result: System resizes
/* Program starts at time T_i */
$RS.T_i \leftarrow$ dataMigrationModel(T_i)
/* $RS.T_i$ is available bandwidth for data migration */
$T.p$ ←analyticalModel($\Delta VMs.T_{i+1}$, $RS.T_i$)
$T.r$ ←analyticalModel($\delta VMs.T_i$, $RS.T_i$)
if $T.p + T.r > T_{i+1} - T_i$ **then**
  $VMsToChange \leftarrow \Delta VMs.T_{i+1} + \delta VMs.T_i$
  t ←analyticalModel(VMsToChange, $RS.T_i$)
  $TimeToAct \leftarrow T_{i+1} - t$
  /* WaitUntil TimeToAct */
  ConductSystemResize($VMsToChange$)
else
  ConductSystemResize($\delta VMs.T_i$)
  $TimeToAct \leftarrow T_{i+1} - T.p$
  /* WaitUntil TimeToAct */
  ConductSystemResize($\Delta VMs.T_{i+1}$)
end

(c) ProRenaTa Scheduler

Figure 5.16 – ProRenaTa Control Flow

combination separated 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 [117].

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 take care of both, the importance of the observation to the forecast, and 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 ARIMA, 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 [118]. 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, 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.

Representative workload types

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 a 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 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.

Prediction methodology

The forecasting method consists of two modules that take into account the two kind of dependencies in the past: short term for **stable load, cyclic behavior and background noise** and long term for **periodic peaks**.

The short term module will make an estimate of the actual value by using a Wiener filter [119] 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 structure of the filter is as follows.

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

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 recompute the weights of the Wiener filter forecaster when the prediction error (MSE) increases for certain length of time [119].

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 depend 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. We denote these periodicities as a set $\{P_0 \dots P_{N_p}\}$, 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

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

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 of each sample used in the estimation, $x[n]$ is the n-th sample of the time series. We do 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 .

Implementation of Predictors

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 considering a delay equal to the forecasting horizon N_{FrHr}, that is $\tilde{x}[n]$ is compute form the set of samples: $\{x[n - N_{FrHr}] \dots x[n - N_{FrHr} - L_{Shrt}]\}$. 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 decision value is defined as:

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

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 $\{x[n] \dots x[n - MemL_{Shrt}]\}$. 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 auto-correlation function measures the similarity

Data: L_{ini}
Initialization
/* Uses $\{x[0] \dots x[L_{ini}]\}$ */
$P_i \leftarrow$ ComputeSetOfPeriodicities()
/* P_i is the set of N_p long term periodicities computed in an initial segment from the auto-correlation function. */
$L_i \leftarrow$ ValuesOfLengthFor$P_i(P_i)$
/* For this experiment $L_i = 2 \ \forall$ period P_i. */
$h_{i,j} \leftarrow$ValuesOfFilter(P_i,L_i)
/* For this experiment $h_{i,j} = \frac{1}{L_i}$ for $1 \cdots L_i$. */

(a) Initialize Long Term Module

Data: L_{ini}
Initialization
/* Uses $\{x[0] \dots x[L_{ini}]\}$ for computing w_i. */
$\{w_i\} \leftarrow$ InitalValuesOfPredictor()
/* Weights w_i are initialized by solving the Wiener equations. */
$\{N_{FrHr}, L_{Shrt}, MemL_{Shrt}\} \leftarrow$ TopologyOfShortTermPredictor() //
$\{Thrhld, \alpha_{srt}, \alpha_{lng}\} \leftarrow$ UpDatingParamOfWienerFilter()
/* Parameters $\{\alpha_{srt}, \alpha_{lng}\}$ define the memory of the filter that smooths the MSE, and $Thrhld$, is the threshold that determines the updating policy. */

(b) Initialize Short Term Module

Figure 5.17 – ProRenaTa prediction module initialization

of the time series to itself as a function of temporal shifts and the maxima of the auto-correlation 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.

Figure 5.17 and Figure 5.18 summarizes the prediction algorithms of ProRenaTa.

Data: w_i, N_{FrHr}, x

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

/* Compute $\tilde{x}_{Shrt}[n + N_{FrHr}]$ from a linear combination of the data in a window of length L_{Shrt}. */

(a) Short Term Prediction

Initialization

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

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

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

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

/* Estimation of the short term and long term value of the MSE, and the decision variable $Des[n]$. */

if $Des[n] > Thrhld$ **then**

  Compute values of the Wiener filter using data $\{x[n] \dots x[n - MemL_{Shrt}]\}$

end

(b) Update Short Term Predictor

Data: $h_{i,j}, P_i, L_i, x$

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

/* Compute $\tilde{x}_{Lng}[n + N_{FrHr}]$ from a linear combination of the data in a window of length corresponding to the periods P_i. */

(c) Long Term Prediction

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

(d) Final Estimation

Figure 5.18 – ProRenaTa prediction algorithm

Table 5.2 – 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

5.3.4 Evaluation of ProRenaTa

we present the evaluation of ProRenaTa elasticity controller using a workload synthesized from Wikipedia access logs from 2009/03/08 to 2009/03/22. The access traces are available online [115]. We first present the setup of the storage system (GlobLease as presented in Section 4.1) and the implementation of a workload generator. Then, we present the evaluation results of ProRenaTa and compare its latency SLO commitments and overall resource utilization against feedback and prediction based elasticity controllers, which act as baselines.

Deployment of the storage system

GlobLease [12] is deployed on a private OpenStack Cloud platform. Homogeneous virtual machine instance types are used in the experiment for simplicity. It can be extended to heterogeneous scheduling by profiling capabilities of different instances types using the methodology described in Section 5.3.1. Table 5.2 presents the virtual machine setups for GlobLease and the workload generator.

Workload generator

We implemented a workload generator in JAVA that generates workloads with different compositions and intensities to GlobLease. To setup the workload, a couple of configuration parameters are fed to the workload generator including the workload trace from Wikipedia, the number of client threads, and the server addresses of GlobLease.

Construction of the workload from raw Wikipedia access logs. The access logs from Wikepedia provide 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

Table 5.3 – 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

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 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/hard-coded notification message sent by the elasticity controllers (actuators). Table 5.3 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 stresses the available network bandwidth. In order to guarantee the SLO latency of the system, we control the network resources used for data transfer using BwMan, which is presented in Section 5.2. The amount of available network resources allocated for data transfer is calculated using the data migration model in ProRenaTa.

Evaluation results

We compare ProRenaTa with two baseline approaches: feedback and prediction-based elasticity controller. Most recent feedback-based auto-scaling literature on distributed storage systems are [70, 30, 68, 120]. These systems correlate monitored metrics (CPU, workload,

response time) to a target parameter (service latency or throughput). Then, they periodically evaluate the monitored metrics to verify the commitment to the SLO latency. Whenever the monitored metrics indicate a violation of the service quality or a waste of provisioned resources, the system decides to scale up/down correspondingly. Our implementation of the feedback control for comparison relies on similar approach and represents the current state of the art in feedback control. Our feedback controller is built using the throughput model described in section 5.3.1. 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.

Most recent prediction-based auto-scaling work are [74, 78, 76]. These systems predict interested metrics. With the predicted value of the metrics, they scale their target systems accordingly to match the desired performance. We implemented our prediction-based controller in a similar way by predicting the interested metric (workload) described in section 5.3.3. Then, the predicted value is mapped to system performance using an empirical performance model described in section 5.3.1. Our implementation closely represents the existing state of the art for prediction based controller. System reconfiguration is carried out at the beginning of the control window based on 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 SLO. 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 the SLO latency. The scaling is carried out at the last possible moment in a control window under the constraint of SLO latency provided by the scaling overhead model described in Section 5.3.1. This model guarantees ProRenaTa with less SLO 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

Here, we present the evaluation results using the aforementioned 4 approaches with the Wikipedia workload trace from 2009/03/08 to 2009/03/22. We select performance metric to be the 95th percentile request latency aggregated each hour. Also, we consider service provisioning cost by introducing another performance metric, which is the aggregated CPU

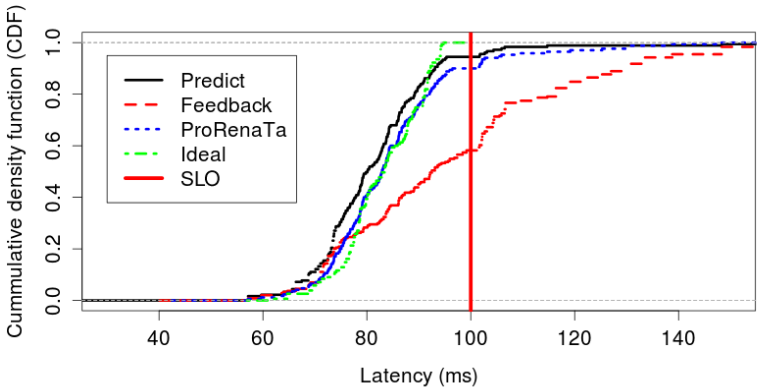


Figure 5.19 – Aggregated CDF of latency for different approaches

utilization of all GlobLease nodes.

SLO commitment. Figure 5.19 presents the cumulative distribution of 95 percentile latency by running the simulated Wikipedia workload from 2009/03/08 to 2009/03/22. The vertical red line demonstrates the SLO latency that each elasticity controller tries to maintain.

We observe that the feedback approach results in the most SLO 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 SLO commitments as shown in Figure 5.19. This is because we have an accurate workload prediction algorithm presented in 5.3.3. And, the prediction-based algorithms try 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 5.20 shows the cumulative distribution of the aggregated CPU utilization on all the storage servers by running the two weeks simulated Wikipedia workload. It 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 causes SLO violations. This CPU utilization pattern matches 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 portion of 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 causes CPU to be under utilized.

It is shown in figure 5.20 that a large portion of servers remain under utilized when

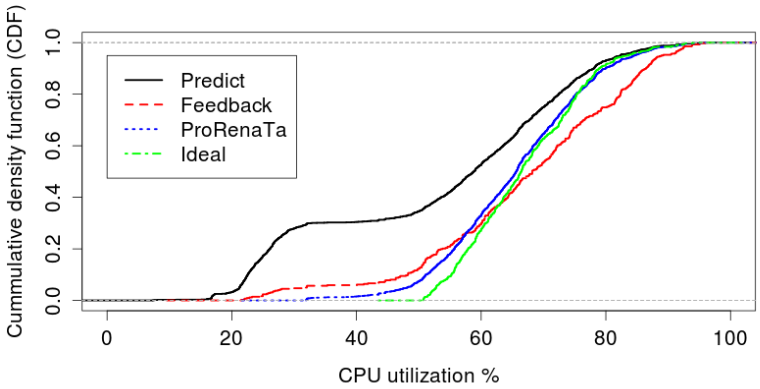


Figure 5.20 – Aggregated CDF of CPU utilization for different approaches

using the prediction based elasticity control. This is because of the prediction-based control algorithm. Specifically, in order to guarantee SLO, in the case of workload increase, servers 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 5.20 shows that most servers in ProRenaTa have a CPU utilization from 50% to 80%, which results in a reasonable request latency that satisfies the SLO. Under/over utilized CPUs are alleviated by the 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 5.3.1. All these mechanisms in ProRenaTa leads to an optimized resource utilization with respect to SLO commitment.

Detailed performance analysis

In the previous section, we presented the aggregated statistics about SLO 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 SLO latency.

Workload pattern. Figure 5.21 (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 5.3.3.

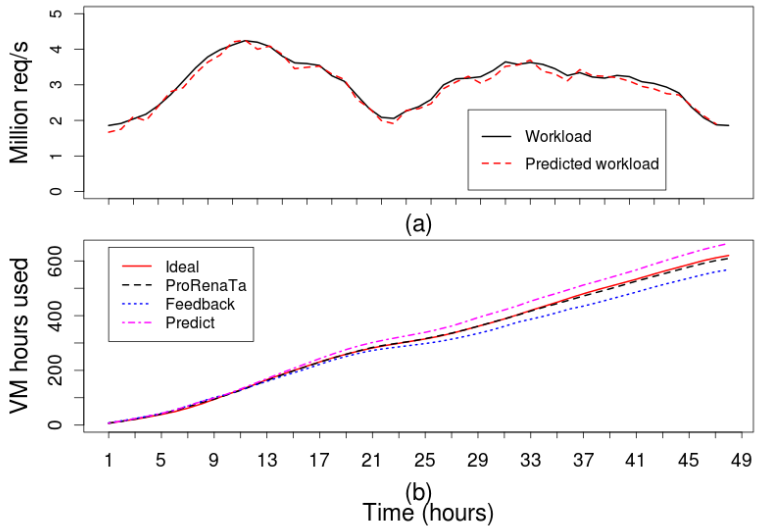


Figure 5.21 – Actual workload and predicted workload and aggregated VM hours used corresponding to the workload

Total VM hours used. Figure 5.21 (b) demonstrates the aggregated VM hours used for each approach under the workload presented in Figure 5.21 (a) during 48 hours. 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. It is shown that ProRenaTa is very close to the VM hours used by the ideal case. On the other hand, the predict approach has consumed more VMs during this 48 hours, which leads to high provisioning cost. The feedback approach has allocated too few VMs, which has caused a lot of SLO latency violations shown in Figure 5.19.

SLO commitment. Figure 5.22 presents the comparison of SLO 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 5.21 (a). Compared to the ideal case, ProRenaTa violates SLO when the workload increases sharply. The SLO commitments are met in the next control period. The feedback approach on the other hand causes severe SLO 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 SLO commitment because of the pre-allocation of servers before the workload occurs. However, it is shown in Figure 5.20 that the prediction based approach cannot use CPU resource efficiently.

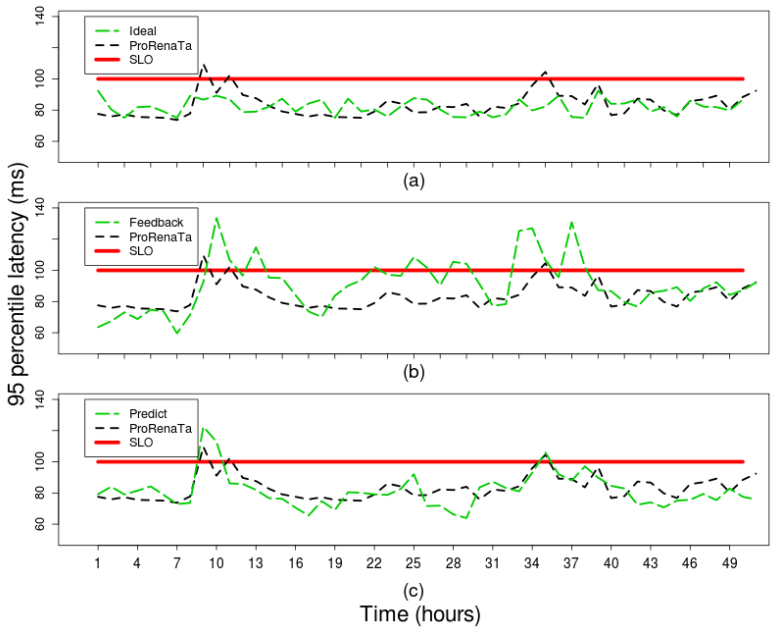


Figure 5.22 – SLO commitment comparing ideal, feedback and predict approaches with ProRenaTa

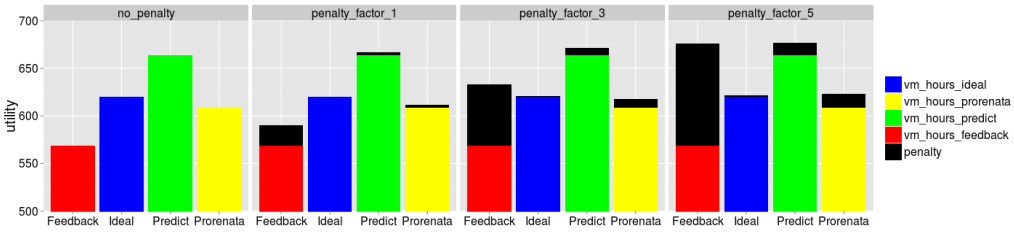


Figure 5.23 – Utility for different approaches

Utility Measure

An efficient elasticity controller must be able to achieve high CPU utilization and at the same time guarantee latency SLO commitments. Since achieving low latency and high CPU utilization are contradictory goals, the utility measure needs to capture the goodness in achieving both these properties. While a system can outperform another in any one of these properties, a fair comparison between different systems can be drawn only when both the aspects are taken into account in composition. To this order, we define the utility measure as the cost incurred:

$$U = VM\_hours + Penalty$$

$$Penalty = DurationOfSLOViolations * penalty\_factor$$

$DurationOfSLOViolations$ is the duration through the period of the experiment the SLO is violated. We vary the penalty factor which captures the different cost incurred for SLO violations. We analyze the results obtained by running a 48 hours Wikipedia workload trace using different auto-scaling controllers. Figure 5.23 shows the utility measure for 4 different scaling approaches. Without any penalty for SLO violations, feedback approach performs the best. But as the penalty for SLO violations increase, ProRenaTa and the ideal approach achieve the lowest utility (cost), which is much better than both feedback and prediction-based auto-scaling approaches.

5.3.5 Summary and Discussions of ProRenaTa

We show the limitations of using proactive or reactive approach in isolation to scale a distributed storage system. Then, we have investigated the efficiency of an elasticity controller named ProRenaTa, which combines both proactive and reactive approaches for auto-scaling a distributed storage system. It excels the classic prediction based scaling approach by taking into account the scaling overhead, i.e., data/state migration. It outperforms the traditional reactive controller by efficiently and effective scheduling scaling operations in advance, which significantly reduces SLO violations. The evaluations of ProRenaTa indicate that we are able to beats the state of the art approaches by guaranteeing a higher level of SLO commitments while also improving the overall resource utilization.

There are also limitations of ProRenaTa. First of all, like all the other prediction-based elasticity controllers, the accuracy of workload prediction plays an essential role in the performance of ProRenaTa. Specifically, a poorly predicted workload causes possibly wrong actions from the proactive controller. As a result, severe SLO violations are expected. In other words, ProRenaTa is not able to perform effectively without an accurate workload prediction. Furthermore, ProRenaTa sets up a provisioning margin for data migration during the scaling of a distributed storage system. The margin is used to guarantee a specific scaling speed of the system. But, it leads to an extra provisioning cost. Thus, it is not recommended to provision a storage system that does not scale frequently or does not need to migrate a significant amount of data during scaling. In addition, the control models in ProRenaTa are trained offline, which makes them vulnerable to unmonitored execution environment changes. Besides, the data migration model and the bandwidth actuator BwMan, assume a well-balanced workload on each storage server. The imbalance of workload on each server will influence the performance of ProRenaTa.

5.4 Hubbub-scale

In the previous sections, we have investigated the necessity of regulating network bandwidth among activities, i.e. serving client workload or migrating data, within a server in order to preserve the quality of service (satisfying a performance SLO). In this section, we study the causes of performance degradation outside the server level. Specifically, we investigate performance interference of among servers, essentially virtual machines (VMs), sharing the same host. VM performance interference happens when behavior of one VM

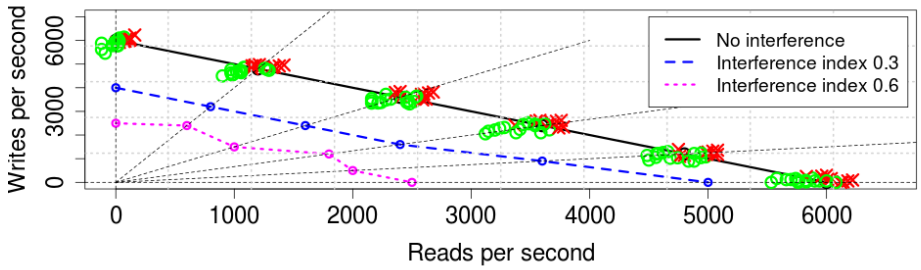


Figure 5.24 – Throughput Performance Model for different levels of Interference. Red and green points mark the detailed profiling region of SLO violation and safe operation respectively in the case of no interference.

adversely affects the performance of another due to contention in the use of shared resources in the system such as memory bandwidth, cache etc [86].

In this thesis, we skip the explanation of identifying and modeling interference from the hosting platform or VMs. For readers interested in understanding these aspects, the detailed explanations can be found in these three of my co-authored papers [71, 77]. My focus will be on applying an interference index, which captures the degree of interference that a VM is suffering and leads to performance degradation, in the scenario of elastic scaling.

In essence, data migration can be generalized as an interference imposed on the serving of client requests. Thus, similar to ProRenaTa, we have built a performance model that captures the effect of platform interference as shown in Figure 5.24. Then, we have applied and implemented this performance model inside of an elasticity controller, namely **Hubbub-scale**. We evaluate the accuracy and effectiveness of this performance model by comparing with a performance model that does not consider the existence of platform interference.

5.4.1 Evaluation of Hubbub-scale

We implemented Hubbub-scale on top of a KVM virtualization platform and conducted extensive evaluation using Memcached and Redis for varying types of workload and varying degrees of interference.

Experiment Setup

All our experiments were conducted on the KTH private Cloud which is managed by Openstack [26]. Each host is an Intel Xeon 3.00 GHz CPU with 24 cores, 42GB memory and runs Ubuntu 12.04 on 3.2.0-63-generic kernel. It has a 12 MB L3 cache and uses KVM virtualization. The guest runs Ubuntu 12.04 with varying resource provisioning depending on the experiment. We co-locate memory intensive VMs with the storage system on the same socket for varying degrees of interference by adding and removing the number of instances. MBW [121], Stream [122] and SPEC CPU benchmarks [123] are run in different

combinations to generate interference. In all our experiments we disable DVFS(dynamic voltage scaling) from the host OS using the Linux CPU-freq subsystem.

Hubbub-scale performs fine-grained monitoring by frequently sampling the CPU utilization and the different performance counters for all the VMs on the host and repeatedly updates the interference index every 1 min. The time-frame chosen for monitoring the selected VMs after classification is 15 seconds and the counters are released for use by other processes for 45 seconds. The hosts running our experiments also run VMs from other users which introduces some amount of noise to our evaluation. However, our middleware also takes into account those VMs to quantify the amount of pressure exerted by them on the memory subsystem.

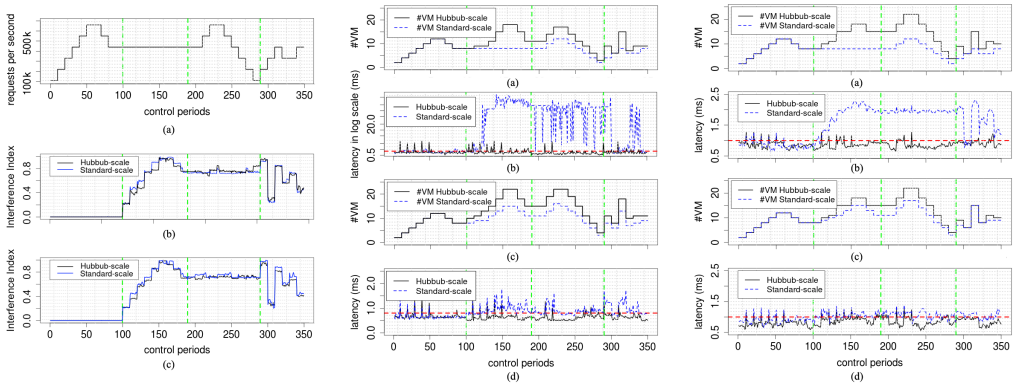
To focus on Hubbub-Scale rather than on the idiosyncrasies of our private Cloud environment, our experiments assume that the VM instances to be added are pre-created and stopped. These pre-created VMs are ready for immediate use and state management across the service is the responsibility of the running service, not Hubbub-Scale. Alternatively, interference generated from data migration can be accounted for by the middleware to redefine the SLO border to avoid excessive SLO violations from state transfer. In order to demonstrate the exact impact of varying interference on Hubbub-Scale, we generate equal amounts of interference on all physical hosts and decisions for scaling out are based on the model from any one of the hosts. The load is balanced in a round robin fashion to ensure all the instances receive an equal share of the workload. We note that none of this is a limitation of Hubbub-Scale and is performed only to accurately demonstrate the effectiveness of the system in adapting to varying levels of workload and interference with respect to the latency SLO.

The control model of Hubbub-scale is partially trained offline before putting it online. It identifies the operational region of the controlled system on a particular VM with various degrees of interference. However, the Hubbub-scale control model can never be fully trained offline, because inter-VM interferences are hard to artificially produce as a cloud tenant. So, this part of the model can only get trained in an online fashion. The control models used in our evaluations are well warmed up by training them with different workloads and interferences.

Results

Our experiments are designed to demonstrate the ability of Hubbub-Scale to dynamically adapt the number of instances to varying workload intensity and varying levels of interference, without compromising the latency SLO. The experiments are carried out in four phases, shown in figure 5.25a with each phase (separated by a vertical line) corresponding to a different combinations of workload and interference settings. We begin with a workload that increases and then drops with no interference in the system. The second phase corresponds to a constant workload with an increasing amount of interference and later drops. The third phase consists of a varying workload with a constant amount of interference and in the final phase, both workload and interference vary.

Figure 5.25b(b) and 5.25c(b) compares the latency of Memcached and Redis under the provision of two elasticity controllers, which are essentially different in the perfor-



(a) [Experimental setup] (b) [Memcached Results] (c) [Redis Results]

Figure 5.25 – (i) 5.25a shows the experimental setup. The workload and interference are divided into 4 phases of different combinations demarcated by vertical lines. 5.25a(b) is the interference index generated when running Memcached and 5.25a(c) is the interference index generated when running Redis. (ii) 5.25b shows the results of running Memcached across the different phases. 5.25b(a) and 5.25b(b) shows the number of VMs and latency of Memcached for a workload based model. 5.25b(c) and 5.25b(d) shows the number of VMs and latency of Memcached for a CPU based model. (iii) 5.25c shows the results of running Redis across the different phases. 5.25c(a) and 5.25c(b) shows the number of VMs and latency of Redis for a workload based model. 5.25c(c) and 5.25c(d) shows the number of VMs and latency of Redis for a CPU based model.

mance model. The elasticity controller, which is ignorant of performance interference, is referred as a standard approach while Hubbub-scale is based on the performance model presented in Figure 5.24. Both approaches provision Memcached and Redis for the four different phases. Without any interference (first phase), both approaches perform equally well. However, in the presence of interference, the SLO guarantees of the standard approaches begins to deteriorate significantly (figure 5.25b(b), plotted in log scale to show the scale of deterioration). Hubbub-scale performs well in the face of interference and upholds the SLO commitment. The occasional spikes are observed because the system reacts to the changes only after they are seen. Figure5.25a(b) plots the interference index captured by the hubbub-scale middleware during the run-time corresponding to the intensity of interference generated in the system. The index captures the pressure on the storage system for different intensities of interference. Certain phases of the interference index in the second phase do not overlap because of the interference from other users sharing the physical host (apart from generated interference). We found that during these periods services such as Zookeeper and Storm client were running alongside our experiments on the same Cloud platform increasing the effective interference generated in the system. Figure 5.25b(a) and 5.25c(a) plots the number of active VM instances and shows that Hubbub-Scale is aware of interference and spawns enough instances to satisfy the SLO.

5.4.2 Summary and Discussions of Hubbub-scale

We have conducted systematic experiments to understand the impact of performance interference when scaling a distributed storage system. Our observations show that input metrics for control models become unreliable and do not accurately reflect the measure of service quality in the face of performance interference. Discounting the number of VMs in a physical host and the amount of interference generated can lead to inefficient scaling decisions that result in under-provisioning or over-provisioning of resources. It becomes imperative to be aware of interference to facilitate accurate scaling decisions in a multi-tenant environment.

As a pioneer, we model and quantify performance interference as an index that can be used in the models of elasticity controllers. We demonstrate the usage of this index by building an elasticity controller, namely Hubbub-scale. We show that Hubbub-scale is able to make more reliable scaling decisions in the presence of interference. As a result, Hubbub-scale is able to elastically provision a distributed storage system with reduced SLO violations and improved resource utilization.

Chapter 6

Conclusions and Future Work

In this thesis, we have worked towards improving the performance of distributed storage systems in two directions. On one hand, we have investigated towards providing low latency storage solutions in a global scale. On the other hand, we have strived towards guaranteeing stable/predictable request latency of distributed storage systems under dynamic workloads.

Regarding the first direction, we have approached our goal by investigating the efficiency of node communications within storage systems. Then, we have tailored the communication protocols under the scenario of geo-distributed nodes. As a result, we are able to reduce request latency significantly. Three systems, GlobLease [12], MeteorShower, and Catenae, are implemented to demonstrate the benefits of our designs.

GlobLease employs lease mechanisms to cache and invalidate values of replicas that are deployed globally. As a result, it is able to reduce around 50% of high latency read requests while guaranteeing the same data consistency level.

MeteorShower leverages the caching idea. Replicas actively exchange their status/updates periodically instead of waiting for read queries. Based on the exchanged updates, even though a little out-dated because of message delays, the algorithm in MeteorShower is able to guarantee strong data consistency. As a result, MeteorShower significantly reduces read/write request latency.

Catenae applies similar idea as MeteorShower. Catenae uses the cached information of replicas to execute transactions against multiple data partitions, which are replicated in multiple sites/data centers. It employs and extends a transaction chain concurrency control algorithm to speculatively execute transactions in each data center with maximized execution concurrency and determinism of transaction ordering. As a result, Catenae is able to commit a transaction within half a RTT to a single RTT among DCs in most of the cases. Evaluation with TPC-C benchmark have shown that Catenae significantly outperforms Paxos Commit over 2-Phase Lock and Optimistic Concurrency Control. Catenae achieves more than twice of the throughput than both approaches with over 50% less commit latency.

Regarding the second direction, we have designed smart agents (elasticity controllers) to guarantee the performance of storage systems under dynamic workloads and environ-

ment. The major contributions that distinguish our work from the state-of-the-art elasticity controller designs are the consideration of data migration during elastic scaling of storage systems. Data migration is an unique dimension to consider when scaling a distributed storage system comparing to the scaling of stateless services. On one hand, data need to be properly migrated before a storage node can serve requests during the scaling process. We would like to accomplish this process as fast as possible. However, on the other hand, data migration hurts the performance, i.e. request latency. Thus, it needs to be throttled in a smart way. We have presented and discussed this issue while building three prototype elasticity controllers, i.e., BwMan [116], ProRenaTa [69], and Hubbub-scale [71].

BwMan arbitrates the bandwidth consumption between client requests and data migration workloads. Dynamic bandwidth quotas are allocated to both workloads based on empirical control models. We have shown that, with the help of BwMan, latency SLO violations of a distributed storage system can be reduced by a factor of two or more when the storage system has some data migration workload running in the background.

ProRenaTa systematically models the impact of data migration. The model helps ProRenaTa elasticity controllers to make smart decisions while scaling a distributed storage system. In essence, ProRenaTa balances the scaling speed (data migration speed) and the impact of data migration under scaling deadlines, which is given by a workload prediction module. As a result, ProRenaTa outperforms the state-of-the-art approaches in guaranteeing a higher level of latency SLO commitments while improving the overall resource utilization.

Hubbub-scale proposes an index that quantifies performance interference among virtual machines sharing the same host. We show that ignoring the interference among VMs leads to inaccurate scaling decisions that result in under-provisioning or over-provisioning of resources. We have built Hubbub-scale elasticity controller, which considers performance interference indicated by our index, for distributed storage systems. Evaluations have shown that Hubbub-scale is able to reliably make scaling decisions in a multi-tenant environment. As a result, it observes significantly less SLO violations and achieves higher overall resource utilization.

6.1 Future Works

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 workload, the performance interference from the underlying platform.

The research work described here have the opportunities to be improved in many ways. For the work in designing 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 have approached the research issue from the direction of using metadata and novel message propagation mechanisms to reduce replica communication overhead. However, the usage of periodic messages among data

centers consumes a considerable amount of network resources, which influences the tail latency of requests as shown in our evaluations. In other words, the network connections among data centers become the potential bottleneck when exploited extensively. We can foresee the improvements of these connections in the coming few years. Then, providing low latency services over all the world will be made possible by trading off the utilization of network resources.

Another direction is the design and application of various data consistency models. We believe that with the emergence of different Internet services and their usage scenarios, e.g., global deployment, strong data consistency model is not always required. Tailoring data consistency models for different applications or components will significantly alleviate the overhead of maintaining data and reduce the service latency. In general, larger system overhead is expected to achieve a stronger data consistency guarantee.

When designing elasticity controllers for distributed storage systems, there are more aspects to consider beyond the network bandwidth. Particularly, we have shown that performance interference among virtual machines sharing the same host also plays an essential role in affecting the quality of a storage service when deployed in the Cloud. Different efforts have been made to quantify this performance interference. However, none of the approaches can be applied easily. This is because that the accesses to host machines are not transparent to a Cloud user. Thus, the quantification of performance interference is not conducted directly on the hosts. Instead, it is usually estimated based on profiling and modeling from virtual machines. We believe that these approaches cannot quantify the interference accurately and impose a considerable amount of overhead to managed systems. Striving to provide transparent platform information and fair resource sharing mechanisms to Cloud users is another step to guarantee the QoS of Cloud-based services.

There is a gap between industry and academia regarding the design and application of elasticity controllers. Essentially, industrial approaches focus on simplicity and usability of elasticity controllers in practice. Most of these elasticity controllers are policy-based and rely on simple if-then threshold based triggers. As a result, they do not require pre-training or expertise to get it up and running. However, most of the elasticity controllers proposed by research community focus on improving the control accuracy but do not consider usability. As a result, these elasticity controllers are challenging or even impossible to be deployed and applied in the real world. Specifically, the first challenge is the selection of elasticity controllers. In fact, there is no way to consistently evaluate an elasticity controller proposed by the research community[124]. Even after an elasticity controller is chosen, it often requires expertise and thorough knowledge regarding the provisioned systems in order to instrument and retrieve the required metrics to make proper deployment of the controller. Additionally, the controllers from academia usually integrate complex components, which require complicated configurations, e.g., empirical training. Thus, we propose researches on elasticity controllers that minimize the gap between industrial and academic approaches. The future work is to propose and design elasticity controllers that achieve both usability and accuracy.

Bibliography

[1] Eric A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM.

[2] Eric Brewer. Cap twelve years later: How the" rules" have changed. *Computer*, 45(2):23–29, 2012.

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

[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] 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 *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[7] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Warp: Lightweight multi-key transactions for key-value stores. *CoRR*, abs/1509.07815, 2015.

[8] Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, March 2006.

[9] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, June 1981.

[10] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[11] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A. Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing*, 12(4):559–592, 2014.

[12] Y. Liu, X. Li, and V. Vlassov. Globlease: A globally consistent and elastic storage system using leases. In *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pages 701–709, Dec 2014.

[13] 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.

[14] Maria Kihl, Erik Elmroth, Johan Tordsson, Karl Erik Årzén, and Anders Robertsson. The challenge of cloud control. In *Presented as part of the 8th International Workshop on Feedback Computing*, Berkeley, CA, 2013. USENIX.

[15] Amazon cloudwatch. `http://aws.amazon.com/cloudwatch/`. accessed: June 2016.

[16] Right Scale. `http://www.rightscale.com/`.

[17] 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.

[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] 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.

[20] 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.

[21] 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.

[22] 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.

[23] Gurmeet Singh Manku, Mayank Bawa, and Prabhakar Raghavan. Symphony: Distributed hashing in a small world. In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, pages 10–10, Berkeley, CA, USA, 2003. USENIX Association.

[24] Jim Gray. The transaction concept: Virtues and limitations (invited paper). In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, VLDB '81, pages 144–154. VLDB Endowment, 1981.

[25] Yousef J. Al-Houmaily and George Samaras. *Three-Phase Commit*, pages 3091–3097. Springer US, Boston, MA, 2009.

[26] Openstack cloud software. `http://www.openstack.org/`. accessed: June 2016.

[27] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157, New York, NY, USA, 2011. ACM.

[28] 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*, FAST'12, pages 18–18, Berkeley, CA, USA, 2012. USENIX Association.

[29] 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.

[30] 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.

[31] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 513–527, Oakland, CA, May 2015. USENIX Association.

[32] Mongodb for giant ideas. `https://www.mongodb.org/`. accessed: June 2016.

[33] Rusty Klophaus. Riak core: Building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, CUFP '10, pages 14:1–14:1, New York, NY, USA, 2010. ACM.

[34] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

[35] Sérgio Almeida, João Leitão, and Luís Rodrigues. Chainreaction: A causal+ consistent datastore based on chain replication. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 85–98, New York, NY, USA, 2013. ACM.

[36] Jiaqing Du, Călin Iorgulescu, Amitabha Roy, and Willy Zwaenepoel. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 4:1–4:13, New York, NY, USA, 2014. ACM.

[37] Cosmin Arad, Tallat M. Shafaat, and Seif Haridi. Cats: A linearizable and self-organizing key-value store. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 37:1–37:2, New York, NY, USA, 2013. ACM.

[38] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

[39] Kfir Lev-Ari, Gregory Chockler, and Idit Keidar. *On Correctness of Data Structures under Reads-Write Concurrency*, pages 273–287. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[40] Hatem Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. Low-latency multi-datacenter databases using replicated commit. *Proc. VLDB Endow.*, 6(9):661–672, July 2013.

[41] 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*, EuroSys '13, pages 113–126, New York, NY, USA, 2013. ACM.

[42] Faisal Nawab, Vaibhav Arora, Divyakant Agrawal, and Amr El Abbadi. Minimizing commit latency of transactions in geo-replicated data stores. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1279–1294, New York, NY, USA, 2015. ACM.

[43] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware '01, pages 329–350, London, UK, UK, 2001. Springer-Verlag.

[44] Tallat M. Shafaat, Bilal Ahmad, and Seif Haridi. *ID-Replication for Structured Peer-to-Peer Systems*, pages 364–376. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[45] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas Anderson. Scalable consistency in scatter. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 15–28, New York, NY, USA, 2011. ACM.

[46] Ali Ghodsi, Luc Onana Alima, and Seif Haridi. Symmetric replication for structured peer-to-peer systems. In *Proceedings of the 2005/2006 International Conference on Databases, Information Systems, and Peer-to-peer Computing*, DBISP2P'05/06, pages 74–85, Berlin, Heidelberg, 2007. Springer-Verlag.

[47] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *SIGOPS Oper. Syst. Rev.*, 23(5):202–210, November 1989.

[48] Jian Yin, Lorenzo Alvisi, Michael Dahlin, and Calvin Lin. Volume leases for consistency in large-scale systems. *IEEE Trans. on Knowl. and Data Eng.*, 11(4):563–576, July 1999.

[49] Felix Hupfeld, Björn Kolbeck, Jan Stender, Mikael Högqvist, Toni Cortes, Jonathan Martí, and Jesús Malo. Fatlease: scalable fault-tolerant lease negotiation with paxos. *Cluster Computing*, 12(2):175–188, 2009.

[50] Jed Liu, Tom Magrino, Owen Arden, Michael D. George, and Andrew C. Myers. Warranties for faster strong consistency. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 503–517, Berkeley, CA, USA, 2014. USENIX Association.

[51] Nuno Carvalho, Paolo Romano, and Luís Rodrigues. *Asynchronous Lease-Based Replication of Software Transactional Memory*, pages 376–396. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[52] Danny Hendler, Alex Naiman, Sebastiano Peluso, Francesco Quaglia, Paolo Romano, and Adi Suissa. *Exploiting Locality in Lease-Based Replicated Transactional*

Memory via Task Migration, pages 121–133. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[53] Khuzaima Daudjee and Kenneth Salem. Lazy database replication with snapshot isolation. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 715–726. VLDB Endowment, 2006.

[54] Esther Pacitti, Pascale Minet, and Eric Simon. Fast algorithms for maintaining replica consistency in lazy master replicated databases. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 126–137, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

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

[56] NTP - The Network Time Protocol. `http://www.ntp.org/`. accessed: July 2016.

[57] Jiaqing Du, S. Elnikety, and W. Zwaenepoel. Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *Reliable Distributed Systems (SRDS), 2013 IEEE 32nd International Symposium on*, pages 173–184, Sept 2013.

[58] Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 11:1–11:14, New York, NY, USA, 2013. ACM.

[59] Gene T.J. Wuu and Arthur J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, PODC '84, pages 233–242, New York, NY, USA, 1984. ACM.

[60] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM.

[61] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. *Conflict-Free Replicated Data Types*, pages 386–400. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[62] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA, 2011. ACM.

[63] Jose M. Faleiro, Alexander Thomson, and Daniel J. Abadi. Lazy evaluation of transactions in database systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 15–26, New York, NY, USA, 2014. ACM.

[64] Yang Zhang, Russell Power, Siyuan Zhou, Yair Sovran, Marcos K. Aguilera, and Jinyang Li. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 276–291, New York, NY, USA, 2013. ACM.

[65] Soodeh Farokhi, Pooyan Jamshidi, Ewnetu Bayuh Lakew, Ivona Brandic, and Erik Elmroth. A hybrid cloud controller for vertical memory elasticity: A control-theoretic approach. *Future Generation Computer Systems*, 65:57 – 72, 2016. Special Issue on Big Data in the Cloud.

[66] Ewnetu Bayuh Lakew, Cristian Klein, Francisco Hernandez-Rodriguez, and Erik Elmroth. Towards faster response time models for vertical elasticity. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, UCC '14, pages 560–565, Washington, DC, USA, 2014. IEEE Computer Society.

[67] Soodeh Farokhi, Ewnetu Bayuh Lakew, Cristian Klein, Ivona Brandic, and Erik Elmroth. Coordinating cpu and memory elasticity controllers to meet service response time constraints. In *Proceedings of the 2015 International Conference on Cloud and Autonomic Computing*, ICCAC '15, pages 69–80, Washington, DC, USA, 2015. IEEE Computer Society.

[68] 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.

[69] Y. Liu, N. Rameshan, E. Monte, V. Vlassov, and L. Navarro. Prorenata: Proactive and reactive tuning to scale a distributed storage system. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pages 453–464, May 2015.

[70] 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.

[71] N. Rameshan, Y. Liu, L. Navarro, and V. Vlassov. Hubbub-scale: Towards reliable elastic scaling under multi-tenancy. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 233–244, May 2016.

[72] Amazon Elastic Compute Cloud. `http://aws.amazon.com/ec2/`.

[73] Google Compute Engine. `https://cloud.google.com/compute/docs/load-balancing-and-autoscaling`.

[74] 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.

[75] 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.

[76] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 69–82, San Jose, CA, 2013. USENIX.

[77] Leandro Navarro Vladimir Vlassov Navaneeth Rameshan, Ying Liu. Augmenting elasticity controllers for improved accuracy. Accepted for publication on 13rd IEEE International Conference on Autonomic Computing (ICAC), 2016.

[78] 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.

[79] D. Kreutz, F. M. V. Ramos, P. E. VerÃssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, Jan 2015.

[80] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[81] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Albert Greenberg, and Changhoon Kim. Eyeq: Practical network performance isolation at the edge. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 297–311, Lombard, IL, 2013. USENIX.

[82] Alan Shieh, Srikanth Kandula, Albert Greenberg, Changhoon Kim, and Bikas Saha. Sharing the data center network. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 309–322, Berkeley, CA, USA, 2011. USENIX Association.

[83] Lucian Popa, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. Faircloud: Sharing the network in cloud computing. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, HotNets-X, pages 22:1–22:6, New York, NY, USA, 2011. ACM.

[84] Vita Bortnikov, Gregory V. Chockler, Dmitri Perelman, Alexey Roytman, Shlomit Shachor, and Ilya Shnayderman. FRAPPE: fast replication platform for elastic services. *CoRR*, abs/1604.05959, 2016.

[85] Vita Bortnikov, Gregory V. Chockler, Dmitri Perelman, Alexey Roytman, Shlomit Shachor, and Ilya Shnayderman. Reconfigurable state machine replication from non-reconfigurable building blocks. *CoRR*, abs/1512.08943, 2015.

[86] Nedeljko Vasić, Dejan Novaković, Svetozar Miučin, Dejan Kostić, and Ricardo Bianchini. Dejavu: Accelerating resource allocation in virtualized environments. *SIGARCH Comput. Archit. News*, 40(1):423–436, March 2012.

[87] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 219–230, San Jose, CA, 2013. USENIX.

[88] Navaneeth Rameshan, Leandro Navarro, Enric Monte, and Vladimir Vlassov. Stay-away, protecting sensitive applications from performance interference. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pages 301–312, New York, NY, USA, 2014. ACM.

[89] Fei Guo, Yan Solihin, Li Zhao, and Ravishankar Iyer. A framework for providing quality of service in chip multi-processors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 343–355, Washington, DC, USA, 2007. IEEE Computer Society.

[90] Miquel Moreto, Francisco J. Cazorla, Alex Ramirez, Rizos Sakellariou, and Mateo Valero. Flexdcp: A qos framework for cmp architectures. *SIGOPS Oper. Syst. Rev.*, 43(2):86–96, April 2009.

[91] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. Qos policies and architecture for cache/memory in cmp platforms. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '07, pages 25–36, New York, NY, USA, 2007. ACM.

[92] Mihai Dobrescu, Katerina Argyraki, and Sylvia Ratnasamy. Toward predictable performance in software packet-processing platforms. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 141–154, San Jose, CA, 2012. USENIX.

[93] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. *SIGPLAN Not.*, 45(3):129–142, March 2010.

[94] Jacob Machina and Angela Sodan. Predicting cache needs and cache sensitivity for applications in cloud computing on cmp servers with configurable caches. *Parallel and Distributed Processing Symposium, International*, 0:1–8, 2009.

[95] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 248–259, New York, NY, USA, 2011. ACM.

[96] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. *SIGPLAN Not.*, 48(4):77–88, March 2013.

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

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

[99] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst.*, 1(3):222–238, August 1983.

[100] 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.

[101] Clinton Gormley and Zachary Tong. *Elasticsearch: The Definitive Guide*. " O'Reilly Media, Inc.", 2015.

[102] Hagit Attiya and Roy Friedman. A correctness condition for high-performance multiprocessors (extended abstract). In *Proceedings of the Twenty-fourth Annual ACM Symposium on Theory of Computing*, STOC '92, pages 679–690, New York, NY, USA, 1992. ACM.

[103] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 12(2):91–122, May 1994.

[104] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

[105] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, 1st edition, 2004.

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

[107] Cloudping. `http://www.cloudping.info/`. accessed: June 2016.

[108] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, June 2005.

[109] Daniel J. Rosenkrantz, Richard E. Stearns, and Philip M. Lewis, II. System level concurrency control for distributed database systems. *ACM Trans. Database Syst.*, 3(2):178–198, June 1978.

[110] Tpc-c, the order-entry benchmark. `http://www.tpc.org/tpcc/`. accessed: June 2015.

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

[112] Kevin Jackson. *OpenStack Cloud Computing Cookbook*. Packt Publishing, 2012.

[113] Openstack swift's documentation. `http://docs.openstack.org/developer/swift/`. accessed: June 2013.

[114] Scaling media storage at wikimedia with swift. `http://blog.wikimedia.org/2012/02/09/scaling-media-storage-at-wikimedia-with-swift/`. accessed: June 2013.

[115] Wikipedia traffic statistics v2. `http://aws.amazon.com/datasets/4182`. accessed: June 2015.

[116] 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.

[117] George Edward Pelham Box and Gwilym Jenkins. *Time Series Analysis, Forecasting and Control*. Holden-Day, Incorporated, 1990.

[118] Timothy Masters. *Neural, Novel and Hybrid Algorithms for Time Series Prediction*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1995.

[119] Thomas Kailath, Ali H Sayed, and Babak Hassibi. *Linear estimation*, volume 1. Prentice Hall Upper Saddle River, NJ, 2000.

[120] 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.

[121] MBW. http://manpages.ubuntu.com/manpages/utopic/man1/mbw.1.html. accessed: April 2015.

[122] Stream Benchmark. http://www.cs.virginia.edu/stream/. accessed: February 2015.

[123] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[124] Alessandro Vittorio Papadopoulos, Ahmed Ali-Eldin, Karl-Erik Årzén, Johan Tordsson, and Erik Elmroth. Peas: A performance evaluation framework for auto-scaling strategies in cloud applications. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 1(4):15:1–15:31, August 2016.