EdgeKV: Decentralized, Scalable, and Consistent Storage for the Edge

by

Karim Mohamed Abdelazim Abouelmaati Sonbol

A Dissertation Submitted to the
Graduate School of Sciences and Engineering
in Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in

Computer Science and Engineering

March 9, 2020

EdgeKV: Decentralized, Scalable, and Consistent Storage for the Edge

Koç University

Graduate School of Sciences and Engineering

This is to certify that I have examined this copy of a master's thesis by

Karim Mohamed Abdelazim Abouelmaati Sonbol

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Committee Members:

_____

Prof. Öznur Özkasap (Advisor)

_____

Prof. Attila Gürsoy

_____

Asst. Prof. Pelin Angın

Date: _____

To my loving family who is always supporting me and believing in me.

ABSTRACT

Edge computing moves the computation closer to the data and the data closer to the user to overcome the high latency communication of cloud computing. Storage at the edge allows data access with high speeds that enable latency-sensitive applications in areas such as autonomous driving, machine vision, smart grid, and virtual reality. However, several distributed services are typically designed for the cloud. In addition, building an efficient edge-enabled storage system is challenging because of the distributed and heterogeneous nature of the edge and its limited resources.

In this thesis, we propose EdgeKV, a decentralized storage system designed for the network edge. EdgeKV does not require any central management and can scale with the edge infrastructure resources and the number of users. Besides, EdgeKV offers fast and reliable storage against failures, utilizing data replication while providing strong consistency guarantees. With a location-transparent and interface-based design, EdgeKV can scale with a heterogeneous system of edge nodes and varying resource capabilities. Combining Replicated State Machines (RSM) that provide consistent fault-tolerant storage with a Distributed Hash Table (DHT) that allows fair load balancing results in a geo-distributed system that can be easily configured to serve different use cases. Latency-sensitive applications typically have two types of data: local data that is relevant only to a small number of users close to each other and requires strict latency requirements, and global data that should be available to all users in the system and can tolerate higher latencies. EdgeKV considers this variation in latency requirements and provides low-latency access to local data, stored in local edge groups, and availability throughout the system to global data, fairly distributed over edge groups in different locations, with acceptable performance.

We implement a prototype of the EdgeKV modules in Golang and evaluate it in both the edge and cloud settings with an emulated setup on the Grid'5000 testbed. We utilize the Yahoo! Cloud Serving Benchmark (YCSB) to analyze the system's performance under realistic workloads. Our evaluation results show that EdgeKV outperforms the cloud storage setting with both local and global data access with an average write response time and throughput improvements of 26% and 19% respectively under the same settings. Our evaluations also show that EdgeKV can scale with the number of clients, without sacrificing performance. Finally, we discuss the energy efficiency improvement when utilizing edge resources with EdgeKV instead of a centralized cloud.

ÖZETÇE

Sınır bilişim, bulut bilişimin yüksek gecikmeli iletişim probleminin önüne geçebilmek için hesaplamayı veriye ve verileri kullanıcıya yakınlaştırır. Yerel veri depolama alanlarında depolama; otonom sürüş, makine görüşü, akıllı şebeke ve sanal gerçeklik gibi alanlarda gecikmeye duyarlı uygulamalar sağlayan yüksek hızlarla veri erişimine izin verir. Ancak, dağıtılmış hizmetler çoğunlukla bulut mimarisi için tasarlanmıştır. Ayrıca, etkin bir sınır özellikli depolama sistemi oluşturmak, kenarın dağıtılmış ve heterojen yapısı ve sınırlı kaynakları nedeniyle zorlayıcıdır.

Bu tezde, ağ kenarı için tasarlanmış merkezi olmayan bir depolama sistemi olan EdgeKV'yi öneriyoruz. EdgeKV herhangi bir merkezi yönetim gerektirmez, ayrıca lokal altyapı kaynakları ve kullanıcı sayısı ile ölçeklenebilir. Buna ek olarak, EdgeKV, güçlü tutarlılık garantileri sağlarken veri çoğaltmayı kullanarak arızalara karşı hızlı ve güvenilir depolama sunar. EdgeKV, konum içermeyen ve arayüz tabanlı tasarımı sayesinde, heterojen kenar düğümleri sistemi ve değişen kaynaklar ile ölçeklenebilir özelliktedir. Hataya dayanıklı depolama sağlayan RSM (Çoğaltılmış Durum Makineleri) ile adil yük dengelemesi yapan DHT (Dağıtılmış Komut Çizelgesini) yapısını birleştirerek farklı kullanım durumları için yapılandırılabilen coğrafi dağıtılmış bir sistem elde edilmesini sağlar. Gecikmeye duyarlı uygulamaların tipik olarak iki veri türü vardır: yalnızca birbirine yakın az sayıda kullanıcıyla ilgili olan ve sıkı gecikme gereksinimleri gerektiren yerel veriler, sistemdeki tüm kullanıcılara açık olması gereken ve gecikmelere daha yüksek tolerans gösterebilen küresel veriler. EdgeKV gecikme gereksinimlerindeki bu değişimi göz önünde bulundurur ve yerel sınır gruplarında depolanan yerel veriler için düşük gecikmeli erişim sağlar, ayrıca farklı konumlardaki sınır gruplarına adil bir şekilde dağıtılan küresel veriler için de sistem genelinde kabul edilebilir bir başarımla erişim sağlamaya çalışır.

Tez kapsamında, Golang ile EdgeKV modüllerinin sistem prototipi geliştirilmiş ve Grid'5000 dağıtık ortamında gerçekleştirilen deneylerle kapsamlı EdgeKV başarım analizi ve bulut depolama ile karşılaştırması yapılmıştır. Sistem performansını gerçekçi iş yükleri altında analiz etmek için YCSB (Yahoo! Cloud Serving Benchmark) kullanılmıştır. Analiz sonuçları EdgeKV'nin, aynı kurulum koşullarında hem yerel hem de küresel veri erişimiyle yapılan bulut depolama ayarından, ortalama yanıt süresinde %26, verimlilikte ise %19 daha iyi başarım göstermektedir. Ayrıca, EdgeKV'nin başarımdan ödün vermeden kullanıcı sayısıyla ölçeklenebileceğini göstermektedir. Son olarak, merkezi bir bulut yerine, EdgeKV ile yerel kaynakların kullanılması durumunda enerji verimliliğini iyileştirmesini tartışıyoruz.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor Prof. Öznur Özkasap for her continuous support, guidance, and motivation throughout this thesis journey. I have learned a lot from her during my study in Koç University not only about distributed systems, research and teaching experience, but also about organization, time and project management, and social skills. I am grateful to my thesis committee members Prof. Attila Gürsoy and Asst. Prof. Pelin Angın for their invaluable time and consideration on evaluating this thesis. The comments and suggestions given by Dr. Moayad Aloqaily and Dr. Ibrahim Al-Oqily were very helpful in improving this thesis. I am also indebted to all my professors for everything I have learned from them. I was lucky enough to work with Prof. Erdem Kabadayı in the Urban Occupations project for a short time and I am grateful for both the experience I gained and the friendships I made with the professor and other project mates.

I would like to thank my family for their continuous support and encouragement during my master's degree and for doing everything they can to help me be successful and happy. This thesis would not have been possible without the love, motivation, and encouragement of my family and friends from home and the ones I have made along the way. I owe a big thank you to Amr, Anas, Mowaffak, and Bayyar for being such great friends at all times. I am grateful to Osama and Desoky for all the good times we spent together. I am really grateful to have known all my Turkish language course friends. Thank you to Nada, Mai, Hoang, Miftah, Javid, Viviana, Enno, Haizayah, Faizan, Faris, Reccep, Heriberto, Solomon, Angelo, Uka, Khanda, Ammar, and Qumars. A special thank you to my teachers Yavuz, Umut, and Gözde for making life much easier for me in Turkey. I am delighted to know the beautiful couple Aseel and Ermanas and grateful for the good times we have spent together.

I was extremely lucky to be surrounded by such great people in the DISNET lab and in Koç University. I am grateful for my friendship with Beakal, and his lovely family, who has been like an older brother to me, and to Ipek for always being there for me. I am in debt for Seyhan, Yahya, and Sanaz for their guidance and kind friendship. Shahid, Waris, Umuralp, Faizan, Waleed, Ateeq, and Muanam, thank you for all the good times we have spent together. I am grateful for my friendship with Angy, Asmaa, Dzenaida, Seif, Meltem, and Rawana. I am also blessed to have made these great friendships in the recent months. I am grateful for Gökçe, Buse, Gizem, and Ecem for their support and encouragement and for the good times we spent together. I appreciate my friendship with Kerim, Çelen, Baturhan, and Bengi. I am grateful for my friendship with Maria and dr. Büşra and their support during my hard times.

I acknowledge that deployment and experiments in this study are carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER as well as other organizations (https://www.grid5000.fr).

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

1.1 Context

Distributed systems typically keep some state information regarding the application logic and the users of the system. Storing this state reliably is crucial to the correct behavior of the systems, but also fast and efficient access controls its performance and scalability. With more efficient parallel and distributed computing frameworks becoming available, communicating the state data between clients and servers, or between the different components of a system, lies on the critical path for many applications. Thus, there exists an increasingly-high demand for more efficient storage and communication architectures.

The cloud computing paradigm has been traditionally used for storing the state of several systems. The cloud means a small number of large-scale centralized data centers that offer reliable and cost-efficient storage and computation solutions. A data center hosts clusters of commodity servers that are virtualized for flexible resource provision and scalability [Bari et al., 2013]. These resources are all located in a single physical location and are managed under a single administrative domain, thus providing high security and reliability standards.

Building a data center has complex requirements for satisfying the required security standards, energy consumption, and physical safety against natural hazards. For example, the energy consumption of a typical data center is equal to the total consumption of 25,000 households [Dayarathna et al., 2016]. These high costs and special requirements are the reasons why each cloud service provider (e.g., Amazon,

Google, and Azure) only has a small number of data centers over the world [Amazon Web Services, , Google, , Microsoft Azure,]. This means that the average distance to the majority of users would be significant, and that a small portion of users would be very far from any data center. Such distances are important because they directly affect the latency of data transfer between clients and servers. [Choy et al., 2012] shows that the average cloud latency is as high as 80 ms, realized by users in different locations around the United States. Such latencies are even much higher outside Europe and North America, where fewer data centers are available.

The high latency of the cloud affects the performance of most distributed applications, and is especially more critical for latency-sensitive and latency-critical applications such as autonomous vehicles, Internet of Things (IoT) applications, and online multiplayer games. In addition to the latency issue, storing data on the cloud means sharing the data with a third-party which may be not be desired in privacy-oriented applications or with high-valued data.

To overcome the latency and privacy issues of the cloud, edge/fog computing paradigm [El-Sayed et al., 2018, Naha et al., 2018] came into existence. Edge computing aims to move the data and computation closer to the user. Therefore, instead of doing most of the computation and storage at the cloud, new computation and storage entities can be introduced between the client and data center in the client-to-cloud continuum to handle these tasks, at least partially. This not only reduces the average response time and saves network bandwidth but it also alleviates some of the workload from the cloud. In addition, privacy-oriented applications can depend totally on the edge, or use the cloud for storing less-sensitive or aggregate information only.

Designing an edge-optimized storage system is not a trivial task. Edge nodes generally have limited computation, storage, network, and/or power resources. They may have heterogeneous hardware and software architectures and are not typically placed in one central location, but instead distributed over a wide geographical area. Therefore, edge nodes in different locations may communicate over a Wide-Area Net-

work (WAN) or even use a wireless protocol (e.g., WiFi or 4G/5G) to communicate with each other or with their clients. Since most existing distributed applications and web services are built for the cloud, they may not be directly usable in the network edge. Instead, either a middleware layer can be introduced to make the existing software edge-aware or novel designs can be developed that are edge-optimized or that are generic enough to work efficiently with both cloud and edge architectures.

There exists a number of recent proposed edge-enabled storage systems. General-purpose systems include Eclipse Fog05 [Eclipse Foundation,] and FogStore [Gupta and Ramachandran, 2018]. Eclipse Fog05 is a compute, storage, and networking virtualization solution that allows running applications in decentralized heterogeneous fog and edge environments. FogStore is a geo-distributed key-value store that provides different latency and consistency guarantees in different contexts. Application-specific edge storage systems include [Ravindran and George, 2018] for machine vision applications, [Lebre et al., 2017] for edge infrastructure management, FBase [Hasenburg et al., 2019] for data-intensive fog applications, [Zhang et al., 2017] for virtual reality Massive Multiplayer Online Games (MMOG), CloudFlare's Workers-KV [CloudFlare,] for optimizing web applications, and DQlite [Canonical Ltd,] for SQL storage at the edge.

In this thesis, we propose EdgeKV, a novel general-purpose, decentralized, scalable, and distributed edge-enabled storage system that offers low-latency access, strong consistency guarantees, and high availability. EdgeKV is decentralized since, unlike the centralized cloud architecture, it does not require a central authority for storage management. It is scalable thanks to an efficient DHT overlay that extends independent RSMs in the geo-distributed edge groups to support fault-tolerance and a high number of users. EdgeKV defines two types of application data: local data that is stored in local edge groups close to the clients, providing very low-latencies and global data that is accessible by users in any edge group and uniformly distributed over all edge groups.

1.2 Contributions

In this thesis, the design and architecture details of the proposed EdgeKVsystem, its proof-of-concept implementation, and the comprehensive performance analysis using distributed Grid'5000 testbed and realistic YCSB workload traces are provided. Specifically, the thesis has the following original contributions.

- Present a review of the existing edge-enabled storage solutions with a summary of their architecture and contributions. We also explain the different use cases for edge storage solutions and their latency requirements.

- Propose EdgeKV, a novel storage system architecture for the edge, explain its modules and their interactions, and the algorithms used in each. In addition, we describe the implementation of our prototype in Golang, the tools used, and the evaluation platform setup.

- Provision of two levels of data locality and privacy to provide differential latency guarantees based on application requirements. The separation of local and global data allows deploying EdgeKV in different use cases.

- Provide Fault-tolerance and reliability through replication in the edge with strong consistency guarantees. In addition, we achieve fair load-balancing with a highly-scalable overlay that has minimal overhead.

- Combine Replicated State Machines (RSM) with a Distributed Hash Table (DHT) to scale EdgeKV to support the storage of large amounts of data, and to cover a large geographical area, hence serving thousands to millions of users. We also use a modular and flexible design that allows easy replacement of modules depending on the application use case.

- Present the performance analysis results of EdgeKV, showing its superior performance to the centralized cloud solution, especially with local data, and evaluate

the scalability of the system with the number of clients and requests. In addition, we provide a discussion about the energy efficiency aspect of the system. Moreover, we discuss possible optimizations for scalability, future research directions for EdgeKV, and useful insights for edge-enabled application designers.

1.3 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 provides a review of existing edge storage solutions organized by their use case. We discuss the following use cases: autonomous driving, smart grid, video streaming and vision applications, MMOG, augmented reality, and virtual reality applications. Chapter 3 explains the system design and architecture of EdgeKV, offering both a layered view and a modular view of the system. In chapter 4, we explain the system implementation tools, the platform setup, and the evaluation environment, followed, in chapter 5, by the performance analysis results, and a discussion on the system's energy efficiency. Finally, chapter 6 concludes with a summary of the contributions, future directions, and advice to edge-based application designers.

Chapter 2

REVIEW OF EDGE-ENABLED STORAGE SYSTEMS

A number of recent storage systems have been proposed that try to handle the challenges discussed in chapter 1. Since the edge infrastructure can be utilized to benefit applications in several different use cases, and since most existing edge storage solutions are optimized for a few number of use cases, we categorize the existing works by their motivating use cases. Specifically, the following use cases are discussed: autonomous driving, smart grid, video streaming and vision applications, Massive Multiplayer Online Games (MMOG), and Augmented Reality and Virtual Reality applications. We summarize the most relevant related works in table 2.1. FogStore [Gupta and Ramachandran, 2018] is designed for situation-awareness applications that use data annotated with context information such as location or timestamps. To use FogStore in any application, the context and mapping from context to consistency level needs to be defined. Similarly, Vision-Edge [Ravindran and George, 2018] is designed for video streaming applications that have two types of data, namely feature vectors, and key-frames. FBase [Hasenburg et al., 2019] provides a declarative way for programmers to choose replication paths and data flows for data-intensive fog applications, based on user-provided configuration data. FBase guarantees only eventual consistency for the application data and strong consistency for configuration data. Some works do not handle fault-tolerance such as Vision-Edge and EC+ [Zhang et al., 2017], while others provide fault-tolerance but with weak forms of consistency such as Fog05 and Workers-KV [CloudFlare,]. We note that each work has a specific use case or class of use cases for which they are designed. For example, Fog05 [Eclipse Foundation,] and OpenStack-Edge [Lebre et al., 2017] are Infrastructure-as-a-Service (IaaS) frameworks that are used for managing both cloud and edge resources. While

Table 2.1: Summary of existing edge-enabled storage solutions.

Project	Use Case	Data Content	Consistency	Fault-tolerance
FogStore [Gupta and Ramachandran, 2018]	situation-awareness, applications	contextual data	context-based	Yes
Vision-Edge [Ravindran and George, 2018]	Computer vision	feature vectors, key-frames	Data-type-based	No
FBase [Hasenburg et al., 2019]	Data-intensive fog applications	Application data	Eventual	Yes
		Configuration data	Strong	
EC+ [Zhang et al., 2017]	MMOG	Game events	Event-type-based	No
Workers-KV [CloudFlare,]	Web services	Web pages	Eventual	Yes
Fog05 [Eclipse Foundation,]	IaaS	Server states	Eventual	No
OpenStack-Edge [Lebre et al., 2017]	IaaS	Server states	Eventual	Yes
Dqlite [Canonical Ltd,]	Embedded devices	Sensor data	Strong	Yes
EdgeKV [Sonbol and Ozkasap, 2020]	General-purpose	Key-Value pairs	Strong	Yes

Dqlite [Canonical Ltd,] provides strong consistency and fault-tolerance, it can be used only with a small number of servers. Finally, the proposed EdgeKV system is a general-purpose system that provides fault-tolerance with strong consistency guarantees for storage of key-value pairs.

2.1 Autonomous Driving

Autonomous driving aims to make transportation safer, more organized, and energy-efficient. An autonomous vehicle uses its on-board sensing devices (e.g., cameras, radars, and proximity sensors) to understand its close surroundings. Additionally, it uses information from other vehicles and from the edge and cloud infrastructures to understand its out-of-sight environment and learn useful information such as high definition maps, traffic lights status, and traffic conditions. This allows for dynamic

path planning and making safer and more efficient trips. The decisions made by an autonomous vehicle are often complex and latency-critical and the cost of a mistake can be very high. Thus, the low latency data access provided by the edge is in many cases preferred to the high latency of long-haul communication with the cloud. The edge can be used for storage and for offloading critical computations.

FogStore [Gupta and Ramachandran, 2018], [Mayer et al., 2017] is a key-value store designed for situation-awareness applications where the *relevance* of data to a client depends on the user and data *contexts*. Specifically, to use FogStore, the application designer needs to define the *Context of Interest (CoI)*, which is the region of relevance to the user. For example, the CoI for autonomous vehicles could be all data items that are originated in a region with a distance less than X kilometers to the client, where X is customizable per application. FogSotre ensures data within the CoI of a client are retrieved with strong consistency. Otherwise, only eventual consistency is guaranteed. While the context definition for FogStore can be application-specific, both data and client requests must be annotated with such context information. To ensure both strong consistency for relevant data and fault-tolerance in a geo-distributed fog system, FogStore replicates data to two sets of replicas: InCoI replicas that lie within the CoI of the requester client ensure strong consistency for all requests and OutCoI replicas which lie outside the CoI only guarantee eventual consitency. OutCoI replicas are only eventually updated and are not included in the quorums for InCoI replicas. FogStore is able to achieve 24% lower update latencies, with 50% update operations, than eventually-consistent systems while providing context-dependent consistency with varying distribution of reads.

A two-level edge computing architecture for coordinated content delivery to automated driving services utilizing intelligence at both the edge and the autonomous vehicles is proposed in [Yuan et al., 2018]. The architecture implements dynamic caching policies at base stations and vehicles and cooperative content sharing schemes at the vehicles that consider the strict latency constraints of services. A vehicle control system that dynamically switches control between edge and cloud according to the

network condition, and shares internal state between the edge and cloud is proposed in [Sasaki et al., 2016]. The system assumes communication between autonomous vehicles and edge servers with a 5G network. The hybrid approach leads to a more stable driving trajectory.

2.2 Smart Grid

The smart grid (SG) has replaced traditional power grids in many places to achieve efficient energy distribution. SG allows real-time monitoring and prediction of consumer loads and the integration of renewable energy sources. By utilizing many sensors and learning models, SG allows energy suppliers to save energy by correctly predicting demand at peak hours without overestimating. SG also ensures a fair system by allowing suppliers to dynamically update the energy price in real-time to which consumer systems can adapt their loads [Samie et al., 2019]. Home Automation systems (HAS) can then help consumers reduce their electricity bills by scheduling the home appliances and shifting loads at peak hours based on the dynamic price.

Current SGs send all data to the cloud for storage and computation. However, the high-latency of the cloud may not be tolerated in some cases, and the large amount of data sent from an SG in real-time can overload the network, causing even higher latency and performance degradation. Additionally, since an SG system often involves sharing private information (e.g., readings from a smart meter in a smart home), sending such data to the cloud makes it more possible for it to be exposed to unwanted third parties.

Edge/fog computing solves these issues by storing data closer to its sources and doing some accumulation, pre-processing, or filtering on data before sending it to the cloud. This way, less data is sent through the network, better response times are achieved, and consumer privacy is protected by sending only accumulated or non-sensitive data to the cloud. Kumar et al. [Kumar et al., 2016] introduced a mobile edge computing solution for smart-grid data management utilizing vehicular delay-tolerant networks. Specifically, they use the VDTN store-and-carry forward mechanism for

data dissemination to the different devices in a smart grid environment.

2.3 Video Streaming and Vision Applications

The edge paradigm is used for latency-critical distributed machine vision applications (e.g., multi-camera real time vision) in [Ravindran and George, 2018]. It distinguishes two types of data coming from a surveillance camera that runs a machine vision application, namely feature vectors and key-frames. Feature-vectors are considered latency-critical and require consistency, while key-frames are considered latency-sensitive and can tolerate some mistakes. Therefore, latency control knobs are introduced to decide the level of required latency based on the type of data. FogStore [Gupta and Ramachandran, 2018] handles the same use case by defining a notion of data relevance based on the client's context. This relevance, or Context of Interest (CoI), is decided based on the distance (both physical and temporal) of the data items to the client's context. The CoI decides whether eventual consistency or strong consistency is used for an operation, hence the expected response time.

2.4 Massively Multiplayer Online Gaming (MMOG)

On-demand gaming is attractive for users since it offloads most of the computation and storage to remote servers, allowing the user to play games with complex graphics with minimal device requirements and no installations. Naturally, the game experience is greatly affected by the network performance, so games often have strict latency requirements (e.g., 80 ms for action games and 40 ms for more demanding games [Choy et al., 2012]). Many of these games are also not single-player but multiplayer or even massively multiplayer games supporting tens of thousands to millions of concurrent players. These games have even stricter latency requirements.

Cloud-based MMOG often suffer from lags caused by network delays for a considerable portion of the user base. [Choy et al., 2012] found that when utilizing a cloud provider such as Amazon EC2, more than 30% of the users are unable to have an acceptable gaming experience because of high network delays. While building more

data centers can increase user coverage with acceptable delays, a significant increase in the number of data centers is required [Choy et al., 2012], which comes with a high cost for building and maintenance. A more efficient approach is to utilize smaller edge clouds placed closer to the users.

Integration of virtual reality (VR) with MMOGs is both growing quickly and extremely challenging. EC+: an architecture for VR-MMOG that is augmented by edge computing is proposed in [Zhang et al., 2017] where VR-MMOGs have two main types of events: local *view change* events (affecting only the user's screen) and global *game events* that should be consistent for all users). While the former has a strict latency requirement of 20 ms, the latter has a more relaxed requirement of 100ms - 1 sec. Using this knowledge, EC+ utilizes the edge for view change updating and rendering, and leaves the game state updating task to the central cloud.

2.5 Augmented Reality (AR) & Virtual Reality (VR) Applications

AR and VR applications allow users to watch and interact with virtual objects in real-time in a reality that is partially or fully virtually created. Their applications are growing to include different fields such as navigation, education, healthcare, and entertainment.

Since most of the commercially available AR and VR devices have limited resources [Zhang et al., 2017], the Quality of Experience (QoE) for such applications is highly dependent on the network bandwidth and latency. Moreover, a high network delay can cause motion sickness or more severe physical harm since the user physically interacts with and navigates the virtual environment. For example, AR applications require latency smaller than 50 ms for acceptable subjective quality [Maheshwari et al., 2018], and VR-MMOGs require an ultra-low latency of 30 ms to avoid motion sickness and an even lower 20 ms for an acceptable gaming QoE [Zhang et al., 2017]. Since the average cloud latency can be as high as 80 ms [Choy et al., 2012], such applications have no option but to utilize the edge and fog resources to achieve acceptable response times. Design of a hybrid edge cloud framework for AR applications and a distributed

decision scheme for edge-cloud load distribution are described in [Maheshwari et al., 2018].

2.6 Discussion

The potential use cases for edge computing storage are not limited to the aforementioned ones. Here, we briefly mention a few other use cases and discuss how EdgeKV compares to the related systems. OpenStack Infrastructure as a Service (IaaS) framework was modified to enable managing edge resources [Lebre et al., 2017]. This is achieved by replacing OpenStack's centralized SQL database with a distributed Redis cluster resulting in faster responses with 80% of the API requests. FBase [Hasenburg et al., 2019] provides a replication service for data-intensive fog applications. It allows controlling data replication and flow across geo-distributed sites in a declarative way while hiding infrastructure heterogeneity. FBase achieves a small latency overhead of 10 ms on average for read operations and a larger 87 ms overhead for write operations which makes it suitable for non-realtime use cases. Eclipse fog05 [Eclipse Foundation,] is a virtualization solution for cloud, edge, and fog resources suitable for heterogeneous environments. It can integrate any key-value store and provide a location-transparent and unified view to it from anywhere in the network. CloudFlare's Workers-KV [CloudFlare,] utilizes CloudFlare's global edge network to build low-latency globally-available key-value storage. Workers-KV is mainly useful for building faster and customized web applications. Dqlite [Canonical Ltd,] provides a distributed, embedded, highly available, and lightweight SQLite implementation for fault-tolerant edge and IoT services. It utilizes the Raft consensus protocol for transactional consensus and fault-tolerance.

While most of the related works are specific to a small number of use cases, EdgeKV is a general purpose storage system with flexibility that enables using it in different use cases. FogStore is suitable for many use cases, but it depends on context embedding and requires a clear definition of the context before using it in any application. EdgeKV only requires the simple distinction of local/global data

and leaves their definitions to the application. Moreover, FogStore, and similarly FBase, do not include a storage module in their designs, unlike EdgeKV which offers the full architecture needed to run an edge-enabled storage system. The content of EdgeKV's key-value pairs will change based on the use case. For example, in a VR game scenario, the key-value pairs will represent local view events (for local data) and game events (for global data). Since EdgeKV has a modular design, if more complex data architecture or query types are needed, the key-value storage module can be replaced with relational or NoSQL databases.

Chapter 3

EDGEKV SYSTEM DESIGN AND ARCHITECTURE

EdgeKV offers a decentralized storage architecture for the edge with strong consistency guarantees through state machine replication. EdgeKV connects independent edge groups with a DHT overlay for high scalability. We explain the system design and architecture of EdgeKV in this chapter and present both a layered and modular views of the system in the following sections with detailed discussion on each module and the used algorithms. Then, we discuss some practical considerations for scalability. An initial version of this design was discussed in [Sonbol and Ozkasap, 2020]

3.1 Layered view

EdgeKV design adapts a hierarchical approach, building from a small number of local nodes to a large-scale system. Fig 3.1 shows the system architecture. There exists two main layers in the system, namely local and global.

In the local layer, a small number of edge nodes which are located in a close geographical proximity form a group. This group represents a replicated state machine (RSM). In other words, each node in this group would have a copy of the same state (i.e., key-value pairs) for fault-tolerance. A consensus protocol maintains strong consistency between these replica states. This means that all (concurrent) read and write requests will be applied to the state machine in the same order and no stale values would be returned to a user at any time.

The global layer is the upper layer in the hierarchy. In a typical deployment of EdgeKV, there would be many groups spread over a large geographical area (e.g., a city or a country). These groups together form an overlay or a global layer that

allows for scalability and the two layers together constitutes the system. In the global layer, different groups are connected through *gateway nodes*. A single gateway node is located close to at least one of the system groups and is responsible for forwarding data from that group to any remote group, and vice versa. It achieves this by first locating the remote group's gateway node in the overlay and then sending data to or asking for data from that group. Since different edge groups communicate only through the gateway nodes, this provides great flexibility and room for heterogeneity. Different edge groups can have different sizes, can use different internal replication mechanisms, and possibly build on different hardware and software architectures.

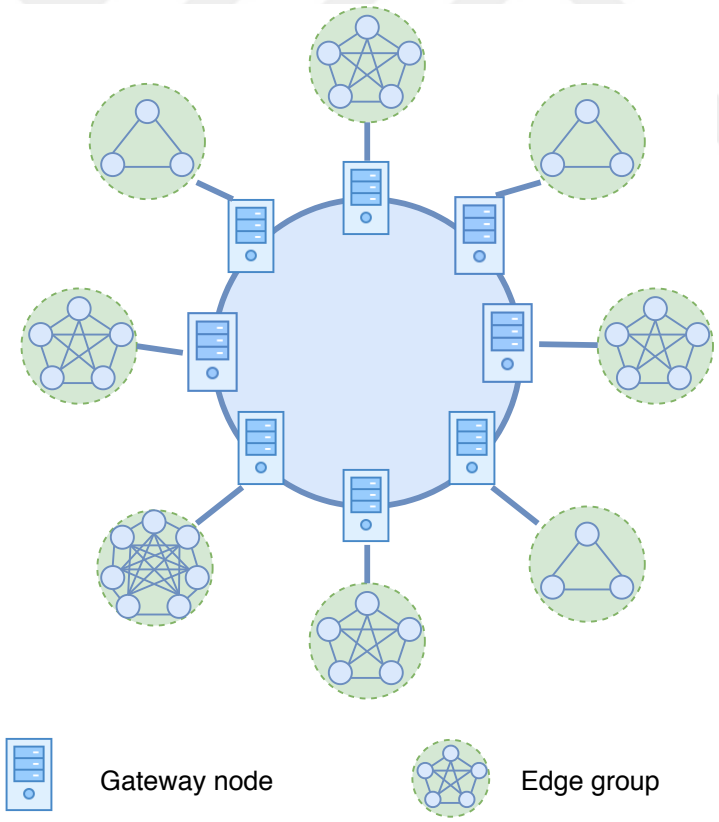

Figure 3.1: System architecture including both the local groups formed of nearby edge nodes and the DHT overlay of gateway nodes.

A distributed hash table (DHT) forms the global overlay of gateway nodes. A DHT ensures a fair distribution of keys from the key-space among the gateway nodes,

hence their corresponding groups. In a DHT, each node is given a unique hash based on its identifier (e.g., IP address) which specifies its location in the overlay. Utilizing a consistent hashing scheme ensures that such hashes are fairly distributed along the overlay and are collision-resistant. Similarly for key-value pairs, the hash of the key decides its location, hence its responsible node in the overlay.

In EdgeKV design, gateway nodes are used only for routing a key-value pair to its corresponding group. The key-value pairs are stored and replicated in the local edge groups. The gateway node itself stores only routing information needed for the DHT, known as *finger tables*. For n gateway nodes, DHT requires O(log(n)) storage complexity on each node to achieve O(log(n)) message complexity for locating any node in the overlay from any other node. Efficient routing and fair load distribution in a DHT allows for scalability.

We follow the popular Chord DHT [Stoica et al., 2001] design at the gateway nodes in EdgeKV for its simple design, proven correctness, and wide usage. The main function in Chord is the *findSuccessor(key)* which finds the successor node of key (i.e., the node responsible for storing the key). The pseudocode of *findSuccessor(key)* is shown in alg. 1.

Each DHT node stores a finger table of size O(log(m)) where m is the maximum possible number of identifiers (i.e., nodes) in the DHT. The *ith* entry in the finger table at node n contains the identifier of the first node, s, that succeeds n by at least 2^{i-1} on the DHT ring. In other words, $s = successor(n + 2^{i-1})$ where $1 \leq i \leq m$. The *findSuccessor(key)* method works by finding the closest node that precedes *key*, then returning the successor of that node, as shown in 1). To find the predecessor of a key, if a node is not the predecessor itself, it checks its finger table for the closest preceding node to *key*. Then, it forwards the request to that node with an RPC call to find the predecessor of *key*. Thus, *n.findPredecessor(id)* may make several RPC calls to other nodes in the ring. Since each request halves the distance to the key, the number of messages sent is O(log(M)) where M is the total number of nodes in the ring.

Algorithm 1 Circular DHT Lookup Protocol

```
1: function n.findSuccessor(id)
2:     pred ← findPredecessor(id)
3:     return pred.successor
4: end function

5: function n.findPredecessor(id)
6:     pred ← n
7:     while id ∉ (pred, pred.successor] do
8:         pred ← pred.closestPrecedingFinger(id)
9:     end while
10:    return pred
11: end function

12: function n.closestPrecedingFinger(id)
13:    for i ← m downto 1 do
14:        if finger[i].node ∈(n, id) then
15:            return finger[i].node
16:        end if
17:    end for
18:    return n
19: end function
```

3.2 Data Types in EdgeKV

The two-layered design allows EdgeKV to handle two types of data differently, namely local and global data. Each edge node in the local layer has a separate storage instance for each type of data.

1. **Local data** are stored only on the edge group closest to the client. When a client sends data to the edge node closest to it, specifying the data type as

local, the node follows the consensus protocol to replicate the data in the local edge group. The group-visible local data store at the edge node is independent of its global data store. Also, the DHT overlay is not used for accessing local data. This means that local data has the advantage of very low access latency compared to global data, which makes it suitable for the *latency-critical* data of an application. Since local data is not shared with other groups in the system, it is suitable for storing local events that are only relevant to a subset of users. Moreover, local data is suitable for storing sensitive data that should not be shared with other edge groups in the system. Nevertheless, local data is still replicated over all the edge nodes in the local group for fault-tolerance.

2. **Global data** are fairly distributed over the edge groups for load balancing and storage distribution. Global data is needed for applications to share their state with multiple edge nodes and users in different locations. In addition, since edge nodes typically have limited resources, global data helps reduce the storage load from a single edge group by fair distribution over the overlay groups. Since global data access requires routing through the overlay, it is more suitable for sharing *latency-sensitive* data.

3.3 Modular View

Our proposed EdgeKV system comprises the modules of RPC interface, placement protocol, resource finder, replication manager, and storage, and their interactions, as shown in Fig.3.2. The application flow is as follows: A client communicates with their closest edge node through the RPC interface, specifying the type of operation (e.g., get, put, and delete), the key (and possibly value) to perform the operation on, and the data type (i.e., local vs global). Then, the placement protocol on the edge node decides, based on the data type, whether to perform the operation in the current edge group or to forward it to to a remote group through the resource finder. The resource finder on the gateway node utilizes the DHT overlay to decide which edge group is

responsible for that key. Afterwards, it forwards the request to that group through its assigned gateway node. Finally, the assigned edge group performs the operation on all group members through the replication manager through a consensus quorum. Finally, the storage module handles the actual storage, retrieval, or deletion of data on the physical storage media. In the following sections, we describe each module in detail with its algorithm.

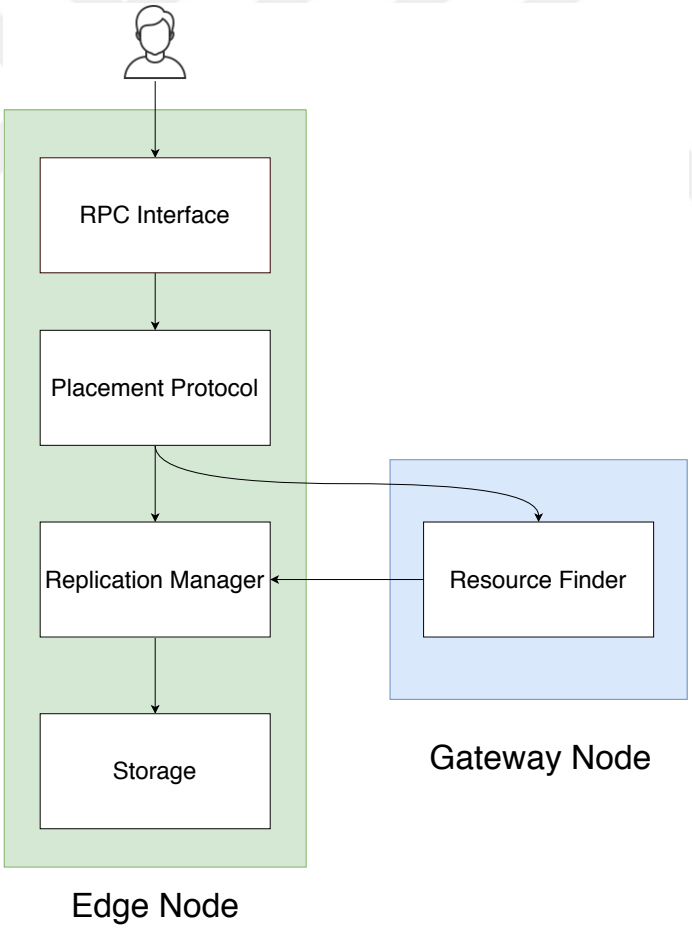

Figure 3.2: EdgeKV modules and their interactions, separated into edge node modules and gateway node module.

3.4 RPC interface

is provided to the end nodes to allow them to store or access the data. The interface supports GET, PUT, and DELETE operations for key-value pairs, where PUT is used for creating a new key-value pair or updating the value of an existing key. Utilizing an RPC interface allows the system to change its internal implementation without changing the users' code. Moreover, the RPC module provides a structured and efficient method of communication.

3.5 Placement Protocol

A placement protocol on the edge nodes decides where data should be placed based on its type: local or global. As depicted in Algorithm 2, local data are stored in the local storage group whereas global data are distributed over the system groups. The placement protocol forwards local data to the replication manager on the same edge node and global data to the resource finder on the local gateway node.

Algorithm 2 Placement Protocol

```
1: function placement(key, value, type)
2:     if type == local then
3:         if nodeType == Leader then
4:             replicate(key, value)
5:         else
6:             send(Leader, key, value, type)
7:         end if
8:     else
9:         send(gateway, key, value)
10:    end if
11: end function
```

3.6 Resource Finder

The resources finder runs on the gateway nodes and utilizes the DHT overlay to decide in which edge group a key-value pair should be stored and the location of that group in the overlay. When the resource finder receives a request from its local edge group for global data access or storage, it hashes the key and decides which edge group is responsible for that key. Then, it routes the request to the gateway node associated with that group using standard DHT routing. Once the remote gateway node receives the request, it forwards it to the replication manager in its assigned edge group. Pseudo-codes for the get and put functions of the resource finder are shown in Algorithm 3.

Algorithm 3 Resource Finder

1: **function** $put(key, value)$
2: $keyHash \leftarrow hash(key)$
3: $targetNode \leftarrow overlay.findSuccessor(keyHash)$
4: $response \leftarrow targetNode.put(key, value)$
5: **return** $response$
6: **end function**

7: **function** $get(key)$
8: $keyHash \leftarrow hash(key)$
9: $targetNode \leftarrow overlay.findSuccessor(keyHash)$
10: $value \leftarrow targetNode.get(key)$
11: **return** $value$
12: **end function**

3.7 Replication Manager

The replication manager runs on each edge node in an edge group and is responsible for replicating data on all the of group nodes. It uses a consensus protocol to ensure strong consistency among the edge nodes. The consensus protocol elects a leader from the group that handles the replication of all group data during its term. The leader may change dynamically if the current leader fails or becomes overloaded. The replication manager is responsible for the consistency of read and write operations to the storage module.

The Raft consensus protocol [Ongaro and Ousterhout, 2014] is used by the replication manager module. Raft is a leader-based consensus protocol providing strong consistency with a simple and understandable design. The Raft protocol is chosen since it is both efficient and suitable for practical systems. Its correctness has been proven and it has been used in practical systems such as etcd, which is used in Kubernetes, a popular container orchestration system, to store the critical data of the system. We explain below the two main processes of Raft, namely the leader election and append entries processes.

3.7.1 Leader Election

A leader is elected in Raft if it receives votes from a majority of the group members. The leader election process is shown in algorithms 4, 5, and 6.

Election and Heartbeat Timeouts

In alg. 4, the timeout callbacks are explained. Each node in a Raft group has two types of timeouts: a heartbeat timeout and an election timeout. The heartbeat timeout is used to detect a failed leader. If no heartbeats were received from the leader until the heartbeat timeout, the follower assumes the leader has failed, and starts an election process. This takes the following steps: the follower increments its term number, switches it states to *candidate*, and votes for itself (lines 2-6). Then, it sends a VoteRequest message to all other nodes in the consensus group (lines 7-10).

To make sure a leader gets eventually elected, and in a timely manner, the election timeout is used. If the election timeout passes before any candidate gets a majority of votes, a new election is started with a higher term number.

Algorithm 4 Leader election timeouts at server q

```
1:  function onHeartbeatTimeout( )
2:      currentTerm ← currentTerm + 1
3:      state ← CANDIDATE
4:      votes ← {q}
5:      votedFor ← q
6:      setElectionTimeout( )
7:      for p ∈ peerList do
8:          info ← (currentTerm, lastIndex, lastTerm)
9:          sendVoteRequest(p, info)
10:     end for
11: end function

12: function onElectionTimeout( )
13:     if Leader == NULL then
14:         onHeartbeatTimeout( )                    ▷ Start new election
15:     end if
16: end function
```

Vote Request Handler

We show how a follower handles a VoteRequest message in alg. 5. First, the follower checks the term number of the candidate. If it is smaller than the follower's term number, then the follower denies the vote and sends its term number to the candidate (lines 2-5). Otherwise, it checks if it has voted for any other candidate. To ensure only a single leader can get elected at any time, a candidate is required to collect a

majority of votes to be elected, and each member in the consensus group can only vote once in an election (with a specific term number). Thus, if the candidate has not voted, or has voted for the same server in the same term, it sends its vote to the candidate (lines 6-8). Otherwise, it denies the vote, specifying the successfully elected leader, if any (lines 9-15).

Algorithm 5 Vote request handler

```
1:  function onRecvVoteReq(req, p)
2:      if req.term < currentTerm then
3:          sendVote(p, False, currentTerm)                ▷ Deny vote
4:          return
5:      end if
6:      if votedFor ∈ {p, null} then
7:          votedFor ← p
8:          sendVote(p, True)                              ▷ Give vote
9:      else
10:         if currentLeader == NULL then
11:             sendVote(p, False)
12:         else
13:             sendVote(p, False, currentLeader)
14:         end if
15:     end if
16: end function
```

Vote Response Handler

The vote response message handler is shown in alg. 6. If the given vote is a positive one, the candidate adds it to the list of votes and checks the votes count. If at least a majority of votes is collected, the candidate switches it state to *leader* and declares its leadership to other group members (lines 2-9). This declaration is achieved by

creating an empty log entry, with the current term number, and replicating it to all other group members. For this, it uses the same appendEntries method used for log replication. Once that log entry is committed, the leader can start coordinating the normal operation by replicating log entries and responding to client requests.

On the other hand, if the received vote is negative, the leader checks if the follower had a higher term number. If so, it steps down to the *follower* state and updates its term number (lines 11-14). Moreover, if the follower has informed the candidate of another existing layer, it also steps down to the *follower* state and updates its leader information (lines 15-19).

3.7.2 Log Replication

After a leader is successfully elected, the normal operation of Raft continues, depending mainly on the AppendEntries RPC message. This is the message sent from the group leader to the followers in order to replicate log entries (i.e., key-value pairs) to all followers. However, the leader returns a response to the client once a majority of the nodes have successfully replicated the data. We briefly explain the AppendEntries function below, as described in algorithms 7, 8 for leader and follower algorithms respectively.

AppendEntries Request

When the leader receives a new request from the client, it creates a new log entry and replicates it to all group members in parallel (lines 2-5 in alg. 7). When it receives a response from a follower, there are three cases:

1. Follower has successfully replicated the entry, so the leader increments the number of votes for that entry, and marks it as committed if a majority is achieved (lines 8-14).

2. The follower is aware of a more recent leader, so the leader steps down to follower state (lines 16-19).

Algorithm 6 Vote response handler at candidate server

```
1: function onRecvVoteRes(p, givesVote, info)
2:     if givesVote == True then
3:         votes.add(p)
4:         if votes.length>numServers/2 then
5:             state ← LEADER
6:             entry ← (NO_OP, currentTerm)
7:             writeToLog(entry)
8:             appendEntriesRPC(entry)
9:         end if
10:    else
11:        if info.term > currentTerm then
12:            state ← FOLLOWER
13:            currentTerm ← info.term
14:        end if
15:        if info.leader ≠ NULL then
16:            state ← FOLLOWER
17:            leader ← info.leader
18:        end if
19:    end if
20: end function
```

3. The leader and follower logs are not synchronized, so the leader sends the previous log entries first (lines 21-24).

AppendEntries Response

Similarly, when the follower receives an appendEntries RPC request, it first checks if the sender is an outdated leader, in which case it informs it of the most recent leader it is aware of (lines 2-4 in alg. 8). Then, it checks the consistency of its log and the leader's log, with three possible cases:

1. If the entry index of the request is higher than its next entry index, it asks the leader for the older entries first (lines 5-7)

2. If the follower has uncommitted entries with a higher index, it removes the last entries until the last matched committed entry (lines 9-13). Then, it continues as in case 3.

3. If the request entry has the index the follower is expecting, then it adds it to its log, which is persisted on disk for resilience to failures (lines 14-16).

Finally, the follower updates its commit index from the commit index piggybacked in the AppendEntries request.

3.8 Storage Module

The storage module handles the actual storage of the key-value pairs on each edge node. Two separate key-value stores are available on each node, a local one for group-level data, and a global one for system-level data. An end node has access to the global storage and the local storage of its connected group only. In our design, we assume a persistent key-value store for simplicity. However, any storage implementation that implements the RPC interfaces can be used. For example, relational databases or NoSQL databases can easily be integrated into EdgeKV depending on the application

Algorithm 7 Sending appendEntries RPC from leader to followers

```
1:  function l.appendEntriesRPC(entryList)
2:      for p ∈ serverList do
3:          sendAppendEntriesTo(p, entryList, l.resHandler)        ▷ Set callback
4:          setTimeoutFor(p, entryList)
5:      end for
6:  end function

7:  function l.resHandler(p, res, info)        ▷ Handle response coming from node p
8:      if res == True then
9:          for ent ∈ res.entries do
10:             ent.votes.add(p)
11:             if ent.votes.length > numServers/2 then        ▷ Mark as committed
12:                 l.commitIdx ← max(leaderCommitIdx, ent.index)
13:             end if
14:         end for
15:     else
16:         if info.term > currentTerm then        ▷ More recent leader exists
17:             state ← FOLLOWER
18:             currentTerm ← info.term
19:             leader ← info.leader
20:         else                                   ▷ Follower log is outdated
21:             p.nextIndex ← p.nextIndex − 1
22:             entry ← log[p.nextIndex]
23:             sendAppendEntriesTo(p, entry)      ▷ Send previous entries first
24:         end if
25:     end if
26: end function
```

Algorithm 8 Replying to AppendEntries RPC from follower to leader

1: **function** $p.reqHandler$(l, req, info) ▷ Handle request coming from leader l
2: **if** $info.term < p.currentTerm$ **then**
3: **return** $False$, $p.myInfo$ ▷ Sender is an outdated leader
4: **end if**
5: $entry \leftarrow p.log[info.prevLogIndex]$
6: **if** $(entry == NULL)$ || $(entry.term \neq info.prevLogTerm)$ **then**
7: **return** $False$, $myInfo$ ▷ Older entries are required first
8: **end if**
9: $entry \leftarrow req.entries[0]$
10: $existingEntry \leftarrow p.log[entry.index]$
11: **if** $existingEntry$ & $existingEntry.term \neq entry.term$ **then**
12: $p.removeEntriesFrom(entry.index)$ ▷ Remove uncommitted entries
13: **end if**
14: **for** $entry \in req.entries$ **do**
15: $p.log[entry.index] \leftarrow entry$ ▷ Write entry to disk
16: **end for**
17: **if** $req.commitIdx > p.commitIdx$ **then**
18: $p.commitIdx \leftarrow min(req.commitIdx, p.lastIdx)$ ▷ update commit index
19: **end if**
20: **return** $True$, $p.myInfo$
21: **end function**

or use case. Also, in-memory storages can be used instead for quick prototyping or debugging.

3.9 Practical Requirements for Scalability

3.9.1 Virtual Nodes

To ensure a more uniform load-balancing between the edge groups for global data, the concept of *virtual nodes* could be utilized in the overlay. For each physical gateway node, multiple virtual nodes (e.g., log(N) virtual nodes where N is the number of physical gateway nodes) can be assigned on the overlay. This has been shown to significantly improve load balancing with the cost of increasing storage space for routing information. However, such storage is insignificant in practice. Virtual nodes can be especially useful when different edge groups have varying resources. More virtual nodes could be assigned to the gateway nodes associated with the more powerful edge groups. This means that such groups will store a bigger portion of the global data than other, less powerful, groups.

3.9.2 Caching for Global Data

With a large-scale deployment of EdgeKV, the average distance between a client and a random remote edge group will get larger, causing higher average latency for global data access. To improve the access latency, especially for frequently accessed remote data, caching can be utilized. Each edge node in the system can cache some of the global data that is stored in other edge groups. Choice of which data to cache depends on the application requirements but strategies such as choosing the most recent data or the most recently accessed data can be applied. The size of the cache can grow or shrink according to the available free storage on the edge node in a specific edge group. We note that to ensure strong consistency (i.e., linearizable reads), reading cached global data would still involve contacting a remote node to validate the cache is up-to-date. However, if reading stale values is tolerated (i.e., serializable reads), then the cached value would be directly returned without contacting the remote group.

Since finding a key's location on the DHT overlay has O(log(n)) time complexity, caching in the gateway node can also be useful. The locations of popular or recent

keys can be added to the gateway cache to avoid the key lookup overhead. The location information would contain the responsible gateway node's identifier in the overlay and its network address (e.g., IP address and port number).

3.9.3 Inter-group Fault Tolerance

A global key-value pair is replicated on each edge node in its assigned group, so it would still be accessible even if a minority of edge nodes fail. However, if a majority of the nodes fail, or if the entire edge group becomes inaccessible to other parts of the overlay (e.g., because of network partitions or link congestion), then the global data stored at that group becomes unavailable to the rest of the system. Because of the distributed nature of the edge setting, network partitions are more common in the edge than in the cloud. To solve these issues, we propose the idea of a *backup group*. For each edge group in the system, we assign another group as its backup group. This assignment would follow static rules so any node can identify the backup group given the original group identifier. A simple approach would be to specify the backup group as the first group directly following node in the overlay.

A backup group is kept up-to-date with the original group as follows: the backup group is included in the original group's members list as a non-voting member. This means that it receives all consensus requests as the other members and is notified of the committed entries. However, it is not counted in the consensus majority. A backup group does not receive data access requests from other groups until the original group becomes unreachable. Even then, the backup group is used for read operations only. This is important to ensure that the states of original and backup groups will not diverge. The backup group may have stale data for some time, but it will still be possible to correct the state once the original group becomes available again.

Chapter 4

EXPERIMENTAL PLATFORM AND SETUP

In this chapter, we discuss the tools used for the prototype implementation and explain our emulation setup and the evaluation framework we used for performing different performance benchmarks of EdgeKV. We also discuss the performance metrics used and their significance in the evaluation.

4.1 System Implementation Tools

We developed a prototype of the proposed EdgeKV edge storage system in Golang that we make available online [Karim Sonbol,]. Communication between the system entities is achieved through gRPC interfaces which allow a simple, structured, yet efficient means of communication.

The replication manager and storage modules are implemented using *etcd*, a popular key-value store used in many cloud deployments. etcd uses the Raft consensus protocol to ensure a strongly-consistent state between all replicas in an edge group. Raft is an efficient yet understandable leader-based consensus protocol widely used in RSMs. It starts by electing a leader from the group, then continues with data replication coordinated by the leader. Because edge groups do not directly communicate, different consensus protocols and different storage backends may be used in different groups, as long as they implement the same gRPC interface and provide the same performance and consistency guarantees. For simplicity, we use the etcd key-value store (for storage and replication management) in all groups in our prototype.

The resource finder is implemented with a DHT overlay. We chose to implement the Chord DHT, one of the most widely used DHTs, following the optimized algorithms in [Stoica et al., 2001]. It has a simple design and maintains the O(log(n))

communication and storage complexity. DHT nodes also communicate through well-defined gRPC interfaces. Using the DHT allows us to create a highly-scalable key-value storage system using small independent etcd clusters.

4.2 Evaluation Platform

To evaluate the performance of our EdgeKV prototype, we use the Grid'5000 testbed with the Distem Network emulator to create the system setup we need. We also use the Yahoo! Cloud Serving Benchmark (YCSB) [Cooper et al., 2010] tool for generating workloads and running a high number of operations against the system. Next, we explain each module in details.

4.2.1 Grid'5000

Grid'5000 is a popular large-scale testbed used by researchers for evaluating several kinds of systems with a focus on distributed and parallel systems, big data, AI, and HPC applications. It has 15,000 cores and 800 compute nodes organized in homogenous clusters which are distributed over 8 sites in France.

We chose Grid'5000 as our testbed for a number of reasons. First, it provides bare-metal access to the servers which allows for great flexibility in setting up the needed software stack. Second, it provides a handful of useful tools for node reservation and deployment, and for experiment monitoring and result collection. Moreover, providing both a REST API a Ruby client to the framework allowed automating most parts of the experiment process. Finally, an important advantage over other similar testbeds is the accuracy of node status data and the wide support available through the technical team and other users of the system.

4.2.2 Distem

While Grid'5000 provides flexible access to physical nodes with the desired compute, memory, network, and storage specifications, the Distem network emulator allows to build complex network topologies and setting up a large number of virtual nodes over

a limited number of physical nodes in a short time. It also allows emulating different scenarios (e.g., cloud vs edge) by creating virtual networks with customizable link specifications, regardless of the underlying network architecture. Similar to Grid'5000, Distem also provides a command-line client, a REST API, and a Ruby client allowing to script the experiments for easy reproductability. In addition, it allows controlling the whole experiment from a single node, known as the coordinator node.

4.2.3 YCSB

We use YCSB to generate realistic workloads and run them against our system. To integrate it with EdgeKV, we implemented a simple YCSB database interface layer that uses our EdgeKV client. YCSB generates realistic workloads, of which we choose the update-heavy workload "A" with 50% read operations and 50% write operations since our system performance gets affected by the percentage of write requests due to its disk access. YCSB also allows changing parameters such as the request distribution and data size. To simulate multiple concurrent connections, each client runs a 100 YCSB workers (i.e., threads) to send requests. In the next sections, we present different evaluations of different aspects of the system.

Experiments performed with YCSB have two phases. First, the *load phase* when key-value pairs are inserted into the EdgeKV storage. Specifically, 10,000 key-value pairs are inserted into the edge nodes storage. Second, the *run phase* when read and update operations are performed against the stored key-value pairs. To experiment with the concept of data types (i.e., local vs global), we made two changes to the EdgeKV database interface layer: When creating the key-value pairs, we store two copies of each pair, one in the local storage, and the other in the global one. Next, requests are randomly chosen to be run on the local or global storage with a probability that is defined by the *proportion of global data* parameter. This parameter is passed to YCSB at each benchmark run.

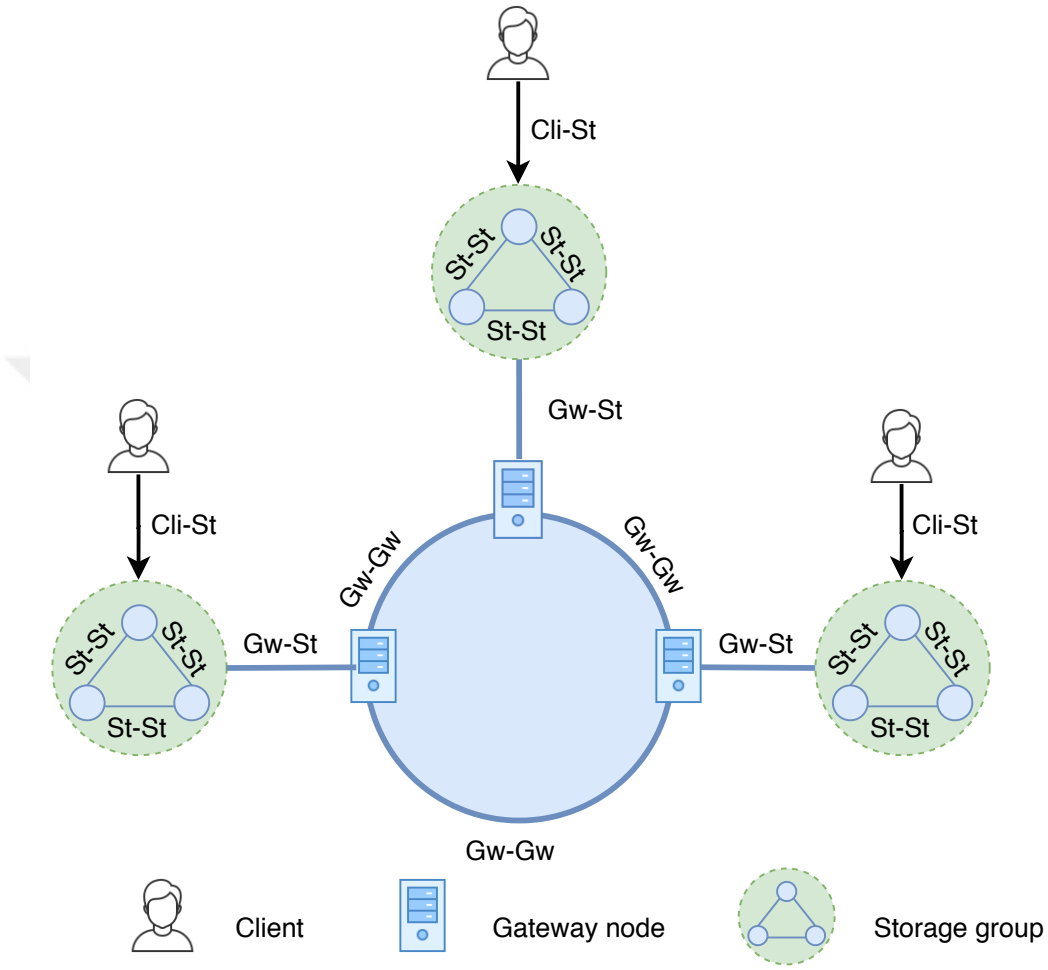

Figure 4.1: The setup used for evaluating EdgeKV: three edge groups and their assigned gateway nodes and clients. Link labels are also shown with abbreviations: Cli: client, St: storage node, and Gw: gateway node.

4.3 Experimental Setup

Our evaluation setup is as follows: we set up three edge groups, each consisting of three edge servers, and each is assigned one gateway node that is also a member in a DHT ring. We also initiate three clients, one for each group, and each running a 100 threads to simulate concurrent requests. The setup is shown in fig. 4.1.

Our setup has a total of 15 virtual nodes running on 15 physical machines in the 'grisou' cluster of the 'Nancy' site. Each node has 2 Intel Xeon E5-2630 v3 CPUs with 8 cores/CPU, 128 GiB RAM, and 600 GB HDD storage. The nodes are connected

with 10 Gbps Ethernet links but we modify their latencies and bandwidths for our experiments using Distem.

We use Grid'5000 to reserve the physical nodes and deploy the file system image to them. Then, we use Distem to create the virtual nodes, load Debian 9 file system images to them, and create network interfaces. We also create several virtual networks to isolate different edge groups and gateway nodes. Using Distem, we modify the links latency and bandwidth settings to simulate an edge setting and a cloud setting as shown in table 4.1.

1. **The cloud setting:** to simulate a centralized cloud setting, we assume a low-bandwidth (100 Mbps) and high-latency (50 ms) link between the client and storage nodes, and high-bandwidth (1000 Mbps), very-low latency (in nanoseconds) links between the other nodes. The high latency between the client and cloud simulates the typically large distance between clients and a data center. We carefully chose such parameters based on practical measurements of public cloud providers (e.g, Amazon EC2 and Microsoft Azure) average response times. Similar latency and bandwidth settings were used in [Zhang et al., 2017] but we assume a simpler network model and smaller distances between gateway nodes.

   We should note that cloud response time greatly depends on the client location. While clients closer to a data center will see lower response times, clients located farther away can have up to an order of magnitude higher response times. Also, clients outside North America and Europe generally get higher response times due to less efficient infrastructures and for the lack of enough data centers in other continents [Google, , Amazon Web Services, , Microsoft Azure,]. [Choy et al., 2012] reports that only 70% of users in different US locations get an average of 80 ms response time or less (40 ms link latency) from the Amazon EC2 cloud. We set a 50 ms link latency instead for a more general assumption. We also use the online WAN latency Estimator tool [WintelGuy.com,] for a lower-bound estimation of different link latencies since it only calculates propagation

	EdgeKV		Cloud	
Link	Latency (ms)	Bandwidth (Mbps)	Latency (ms)	Bandwidth (Mbps)
Client - Storage	5	100	50	100
Storage - Storage	2	1000	0.05	1000
Storage - Gateway	2	750	0.05	1000
Gateway - Gateway	10	500	0.05	1000

Table 4.1: Link specifications to simulate edge and cloud settings.

delay in fiber links.

2. **The edge setting:** simulates a typical edge deployment, with distances between hundreds of meters to a few kilometers. We simulate this with a latency of 2 ms between edge nodes, a latency of 5 ms between client and edge nodes in the same edge group, and higher bandwidth links in edge groups than farther-away gateway nodes. Unlike the cloud remote data centers, edge servers are assumed to be widespread to be within small distances from target clients and from neighboring edge servers (typically tens to hundreds of meters). Therefore, the links between such entities have low latencies. Based on the latencies required for optimal system performance, the edge nodes can be distributed to maximize the coverage while maintaining a manageable cost. However, solving such an optimization problem for node placement is outside the scope of this thesis.

Unless specified otherwise, all experiments run 10,000 operations on each client in parallel, and the average results over all clients and operations are reported. We perform experiments in both the edge and cloud settings.

4.4 Performance Metrics

We evaluate EdgeKV using multiple performance metrics to analyze the system's performance from different aspects. Below, we discuss each of the used metrics and their significance in such an edge-enabled system.

4.4.1 Average response time

The response time of an operation is the duration between sending a request to the storage node (i.e., edge node) and receiving the response at the client side. This may include latencies caused by the network, the consensus protocol, disk access, and overlay lookups. The data type specifies what the response time includes as follows:

For local data read and write operations, each operation requires communicating with at least a majority of storage nodes in a single edge group to achieve consensus, according to the Raft protocol. Therefore, the response time of a local data access operation includes the communication latency between group members, and the overhead of disk access to read or write the key-value pairs. If the stored data can fit into the memory of storage nodes, read operations can avoid disk access by directly reading from memory. However, for write operations, disk access is required to achieve fault-tolerance. Moreover, although EdgeKV uses linearizable reads, which require achieving a majority quorum, to always return consistent data, serializable reads can be used to return data from a single storage node avoiding the consensus overhead, with the cost of possibly returning stale data.

Global requests, on the other hand, may involve the overlay lookup overhead. When an edge node receives a data access request for a global key-value pair, it first checks if the key belongs to that edge group. If so, the key access is performed directly on that edge group. Otherwise, the DHT overlay is traversed according to the lookup protocol to find the edge group responsible for the key. The request is then forwarded to that edge group to perform the required operation. Thus, a global request may involve an additional overhead of overlay lookup and edge-to-gateway and gateway-to-gateway node communication. In addition, for both local and global

requests, the response time includes latency of communication between the client and storage nodes. Also, the reported response times are the averages of the individual response times for each request.

4.4.2 Throughput

In addition to the response times, we also measure the system throughput defined as the number of operations the system can successfully complete in a second, measured from the client side. In each experiment, the clients try to send as many requests per second as they can, unless otherwise specified, and the measured server throughput is reported. In experiments where multiple clients are used, the requests are distributed over the clients and the average response time and throughput values are reported.

Chapter 5

PERFORMANCE ANALYSIS AND RESULTS

In this chapter, we present the performance analysis results of EdgeKV using the emulation setup and performance metrics discussed in chapter 4. The performed experiments analyze the efficiency and scalability of the system under different workloads and different configurations. Also, a complexity analysis of EdgeKV is included, followed by a discussion on energy consumption.

5.1 Data-locality Effect on Performance

While a typical realistic edge use case would perform more local data access than global data access (e.g., VR and autonomous driving applications), we evaluate our system under loads with different global request percentages, as shown in figures 5.1, 5.2. We report the write latency and throughput results but the read operation results have an almost identical pattern. Both figures show that the edge setting outperforms the cloud one in both latency and throughput performance. It is interesting to notice that the change between 50% - 100% global data in throughput and latency is minimal. However, the performance decrease is high when the percentage of global requests is increased from 0% to 50%. Nevertheless, the edge still manages to keep its precedence over the cloud with 26% lower latency and 19% higher throughput at 50% global write requests. Even at 100% global requests, EdgeKV setting still outperforms the cloud, This is because in the cloud setting, the client-to-server latency dominates all other latencies for server-to-server communication and disk access latencies.

These results suggest with a low portion of the requests accessing global data, as in many practical applications, the system shows significantly better performance. This means the system is suitable for real-life scenarios but it also means careful care must

be taken when designing applications targeted for the edge to minimize the number of global requests needed as much as possible.

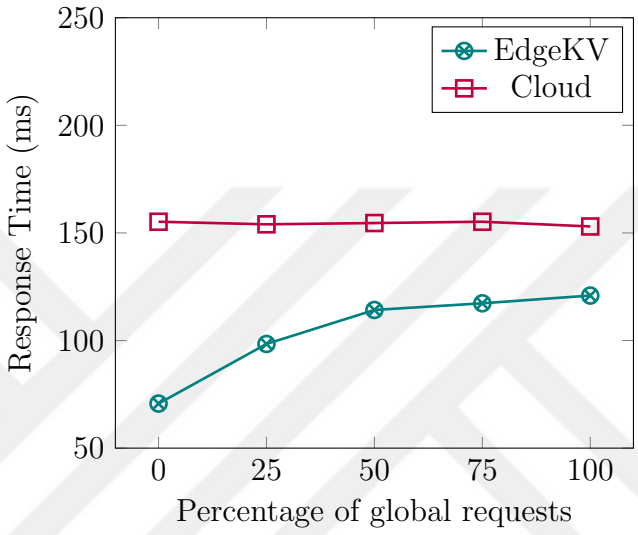

Figure 5.1: Average write response time change with the percentage of global data in the requests.

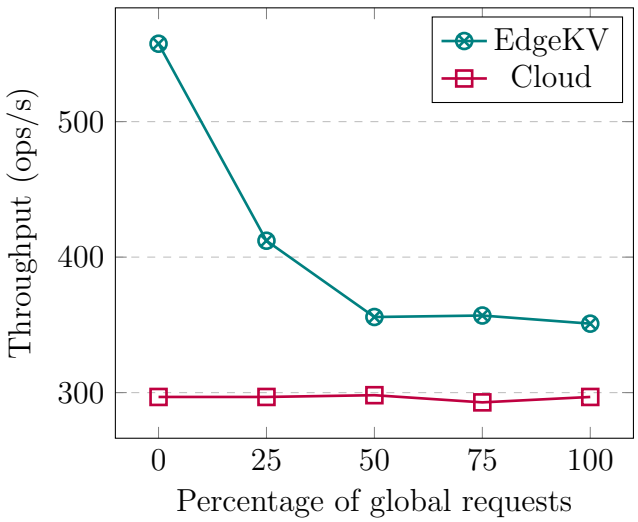

Figure 5.2: Average write throughput change with the percentage of global data in the requests.

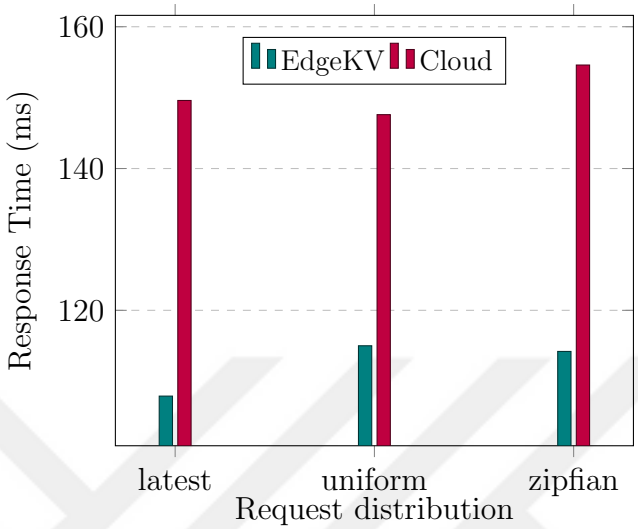

Figure 5.3: Average response time performance for update operations with 50% of the requests accessing global data.

5.2 Request Distribution Effect on Performance

Requests in practical applications often do not follow a uniform distribution (where keys have equal popularity in a key-value store). Instead, a small set of the keys are often the most popular, known as the "hotset" or "hotspot" which the majority of read operations target. For example, in a social media application, a small portion of user accounts or posts are the most popular. Another common real-life request distribution pattern is the "latest" pattern where recently-inserted keys are generally more popular than older ones. This is common in systems that use sensor-collected information such as autonomous driving and VR applications as the recent information have higher value than older ones.

Since the performance of a key-value store typically changes based on the request distribution pattern, we evaluate EdgeKV with the three discussed popular patterns, namely, uniform, zipfian, and latest patterns. For the zipfian request distribution, we set the hotset size to 20% of the total data and the percentage of operations that access the hot set to 80%. YCSB chooses the keys for the hotset randomly from the total data inserted.

In figures 5.3 and 5.4, we can see that EdgeKV in an edge setting outperforms the cloud setting with a large difference, with all three request distribution patterns. Specifically, fig. 5.3 shows that the write latency of the edge setting is 22% - 28% lower than the cloud setting with different request distributions. Similarly, in fig. 5.4, the edge setting outperforms the cloud one with 15% - 28% higher throughput. In both latency and throughput terms, EdgeKV achieves the best performance with the latest request distribution.

Figure 5.4: Throughput performance for update operations with 50% of the requests accessing global data.

5.3 Client Size Scalability with Local Requests

Here, we consider the performance of EdgeKV with local requests only by analyzing a single group of storage (etcd) nodes. We measure the response time and throughput of write operations from a client node in both the edge and cloud settings. Since the requests only access local data, the response time here is mainly the consensus latency (to reach a quorum among group members), including writing to disk, and communication with the client latency. In figures 5.5 and 5.6, we show the results of these experiments. Specifically, we show how the write operations latency and

throughput change with increasing the number of clients. The figures show that the edge setting achieves 1.5x - 2x higher throughput and 34 % - 60% lower latency than the cloud setting with different number of clients.

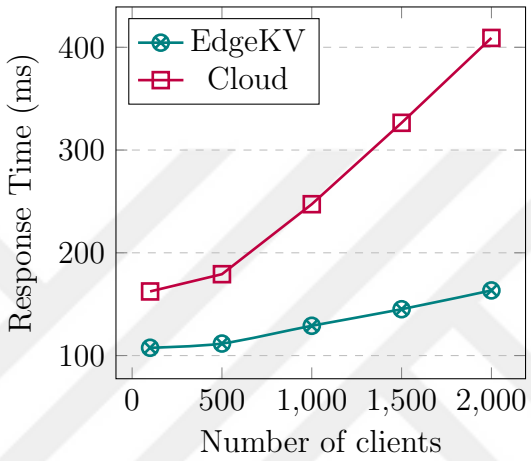

Figure 5.5: Write response time scalability with the number of clients, using local requests only.

The local requests performance results clearly show the advantage of utilizing EdgeKV in the edge setting. The close proximity of edge nodes to the clients allows for latency and throughput values not achievable by the remote cloud. Again, we see proof that defining the data locality in an application design can make a big difference in its performance.

5.4 Client Size Scalability with Global Requests

In this evaluation, we consider both local and global requests, by using workloads with 50% global requests under the same configurations for both the edge and cloud settings. Figures 5.8 and 5.7 show how the system scales almost linearly with the number of clients in both settings. While the edge throughput is slightly higher than that of the cloud (fig. 5.8), the difference in response time change is considerable (fig. 5.7). Note, especially, when the number of clients increases from 1,000 to 2,000 clients, how the cloud average response time increases by about 24% while the edge

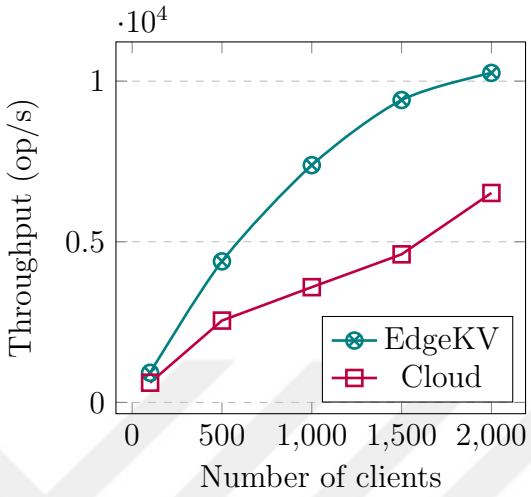

Figure 5.6: Write throughput scalability with the number of clients, using local requests only.

one is almost constant.

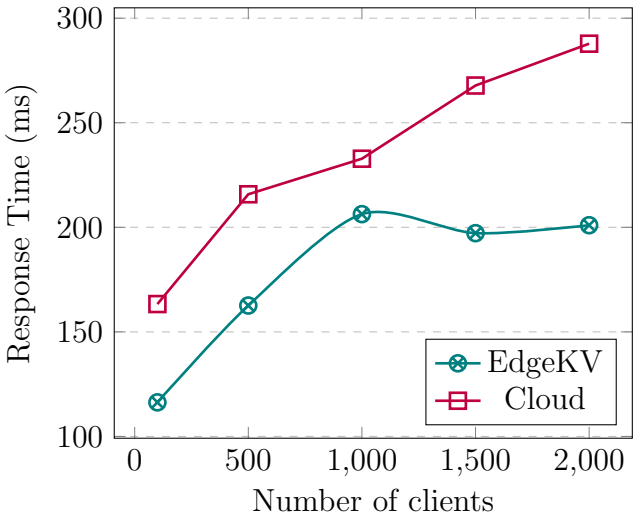

Figure 5.7: Write response time scalability with the number of clients, using 50% global requests

Comparing these results to the ones discussed in the previous section in figures 5.5 and 5.6, we see that with a high number of clients, the effects of global requests on performance is higher (up to a factor of 4x higher throughput with the edge).

Nevertheless, the edge setting keeps its advantage over the cloud in both cases.

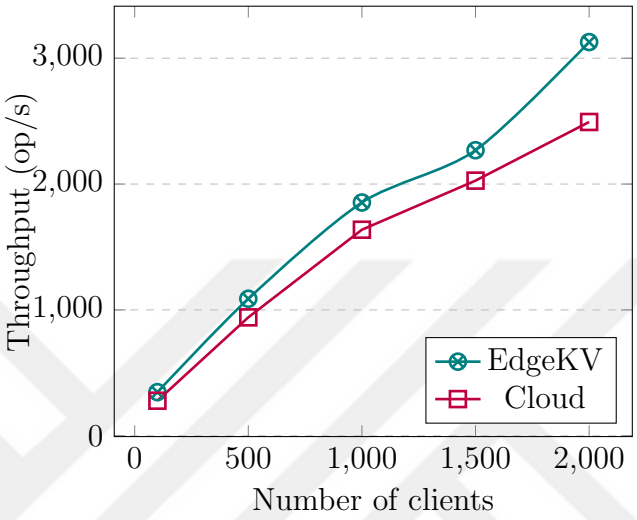

Figure 5.8: Write throughput scalability with the number of clients, using 50% global requests

5.5 Request Rate Scalability with Global Requests

In this experiment, we use different request rates to analyze how much the system performance degrades with increasing request rates. We compare the performance of the system with 50% global requests in both cloud and edge settings, using a 100 client threads, as shown in fig. 5.9. The results show that in both settings the write response time grows linearly with the request rate. Also, it is visible that there is a consistently large gap between the edge and cloud response times, where the edge has, on average, 42% lower response time than the cloud.

5.6 Complexity Analysis of EdgeKV

We provide a complexity analysis of EdgeKV's different operations in this section. Since the read operations discussed are linearizable reads (i.e., they require a consensus quorum to ensure data is up-to-date), the time complexity for both read and

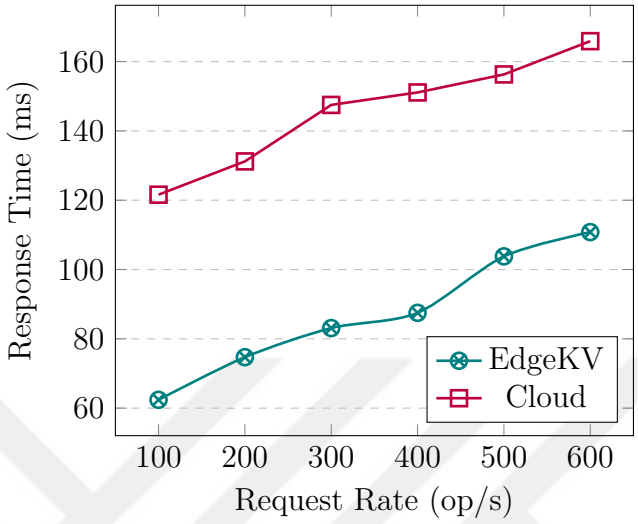

Figure 5.9: Response time change for write operations with increasing request rates when 50% of the requests are global requests

write operations are the same. However, the time complexity changes based on the data type:

1. **Local data access** does not involve communication with gateway nodes or usage of the DHT overlay. Instead, local data is directly accessed from the edge group they were requested. Still, for fault-tolerance, a consensus quorum among edge nodes in the group is required for both read and write operations to ensure strong consistency. Assuming an edge group with n nodes, a majority of at least n/2 nodes is required to achieve consensus.

   Time complexity of local data access: $O(n)$.

2. **Global data access**, on the other hand, may require lookup through the gateway nodes. If the key to access is in the responsibility of the edge group where the request is sent, the operation is performed directly in that group, achieving the same time complexity as local data. However, if another edge group is responsible for the key, then the responsible group needs to be decided first through a lookup operation in the DHT overlay. A DHT achieves lookup in

O(log(m)) with m gateway (i.e., DHT) nodes. After the responsible gateway node is found (hence its corresponding edge group), a consensus quorum needs to be achieved between the group nodes in O(n) time for the data access.

Time complexity of remote global data access: $O(n + log(m))$

Space complexity on each edge node is computed as follows: Each edge node has a copy of all the local data stored in the edge group, which is O(L*S) assuming L is the number of keys in the local group and S is the average size of a local key-value pair. Besides, since the global data are uniformly distributed over the edge groups, each group also gets global storage of O(G*T/m) where G is the total number of global keys in the system, T is the average size of a global key-value pair, and m is the number of edge groups (i.e., number of gateway nodes).

Space complexity of a single edge node: $O(L * S + G * T/m)$

On the other hand, gateway nodes do not persist any key-value pairs. They are only required to store the finger tables for lookup through the DHT overlay. So, for m gateway nodes:

Space complexity of a single gateway node: $O(log(m))$

5.7 Energy Considerations in the Edge

Considering the energy consumption in distributed P2P system design and developing mechanisms that improve energy efficiency are significant research areas [Brienza et al., 2016]. In a centralized cloud setting, each request needs to be sent to the remote cloud, through high latency links, which can cause the cloud servers and network links to the cloud to be overloaded, wasting both time and bandwidth. Besides, in the cloud, all application data is treated equally (i.e., giving the same performance guarantees) regardless of the location of the source and consumers of data and the number of consumers. On the other hand, EdgeKV considers the location context of users and

the fact that some data may need to be shared only with a small number of users in the same geographical area, namely local data. This allows some of the data to be stored in the proximity of its users, saving bandwidth and providing low-latency access while maintaining consistency. Therefore, the use of local data in EdgeKV achieves higher energy saving than the data-type-agnostic cloud, especially since in many applications such as autonomous driving and VR applications, local data is more frequent than global data [Zhang et al., 2017]. Moreover, since global data is distributed uniformly over multiple edge groups in different locations, the resource usage per each edge group becomes smaller as the system grows larger, making resources less likely to be overloaded.

Chapter 6

CONCLUSION

6.1 Remarks

While the cloud is suitable for latency-tolerant applications such as web services, latency-sensitive and latency-critical applications need an alternative. Edge and fog computing and storage resources provide the alternative to alleviate at least some of the workload from the cloud and provide faster and more efficient access to data. However, designing an edge-enabled application is a challenging task. In this thesis, we introduced EdgeKV, a distributed storage solution for the edge to help application developers worry less about the underlying infrastructure and focus more on the application design. EdgeKV abstracts away the details of the edge infrastructure with well-defined interfaces and a location-transparent data placement strategy. Moreover, EdgeKV provides fault-tolerance and strong consistency guarantees through data replication in the local edge groups. Nevertheless, high scalability is achievable with an efficient DHT-based overlay to connect edge groups in different locations. The modular design of EdgeKV allows the easy replacement of any of the modules to provide different storage types, varying consistency and latency guarantees, or to introduce application-specific requirements.

We have implemented a prototype of EdgeKV in Golang and presented our performance analysis results from different aspects. EdgeKV achieves 26% lower latency and 19% higher throughput than a centralized cloud solution with 50% global requests under the same testing conditions. We have also shown that EdgeKV scales better than the cloud with the number of requests with an average of 42% lower latency even with 50% of the requests accessing global data. Finally, we have demonstrated how the different request distribution patterns affect EdgeKV, and that EdgeKV performs

22% - 28% faster writes on average with 15% - 28% higher throughput.

Despite the challenging nature of latency-sensitive applications, most of them have multiple types of data or context that have different latency requirements. EdgeKV utilizes this knowledge to maximize an application's performance while maintaing an efficient load distribution and resource utilization. However, to fully benefit from EdgeKV, or other similar solutions for that matter, an application designer must pay great attention to the separation of different data types in their design. That said, EdgeKV does not require the integration of any application-specific logic to achieve its maximum performance.

6.2 Future Directions

EdgeKV is designed as a general-purpose strongly-consistent key-value store. However, due to its flexible and modular design, it can be used in different configurations to suit a multitude of use cases. As a future work, EdgeKV can be used with different replication techniques to provide weaker forms of consistency for lower response times. Similarly, different storage drivers can be used for storing different data types. For example, the key-value store can be replaced with a relational SQL database or with a NoSQL data store.

Additional evaluations can be performed for EdgeKV with different use cases. Latency-sensitive applications may be implemented to utilize EdgeKV as their distributed storage, and comparisons can be made between EdgeKV's performance and other related works in the same application. In the evaluations presented, all edge groups had the same number of clients and the same request rate. Instead, non-uniform load distributions can be used to evaluate the system's performance under skewed loads that reflect changes in population in different locations. Communication between different modules in EdgeKV is achieved through secure connections with optional use of certificates; however, only crash-failures are assumed. If an application assumes a malicious environment, a Byzantine-fault tolerant consensus protocol can be used in the replication manager module.

EdgeKV performance could be further improved by using low-latency persistent storage such as the Non-Volatile Memory (NVMe) and low-latency communication transport such as Remote Direct Memory Access (RDMA). RDMA avoids the overhead of the traditional TCP transport by performing zero-copy transfer and directly accessing memory or non-volatile memory. A few RDMA-enabled consensus protocols already exists [Sonbol and Ozkasap, 2019] and could be used in the replication manager module to perform faster replication.

BIBLIOGRAPHY

[Amazon Web Services,] Amazon Web Services. Global infrastructure. `https://aws.amazon.com/about-aws/global-infrastructure/`.

[Bari et al., 2013] Bari, M. F., Boutaba, R., Esteves, R., Granville, L. Z., Podlesny, M., Rabbani, M. G., Zhang, Q., and Zhani, M. F. (2013). Data center network virtualization: A survey. *IEEE Communications Surveys Tutorials*, 15(2):909–928.

[Brienza et al., 2016] Brienza, S., Cebeci, S. E., Masoumzadeh, S. S., Hlavacs, H., Ozkasap, O., and Anastasi, G. (2016). A survey on energy efficiency in p2p systems: File distribution, content streaming, and epidemics. *ACM Computing Surveys*, 48(3).

[Canonical Ltd,] Canonical Ltd. Dqlite – High-availability SQLite. `https://dqlite.io/`.

[Choy et al., 2012] Choy, S., Wong, B., Simon, G., and Rosenberg, C. (2012). The brewing storm in cloud gaming: A measurement study on cloud to end-user latency. In *2012 11th Annual Workshop on Network and Systems Support for Games (NetGames)*, pages 1–6.

[CloudFlare,] CloudFlare. Workers KV — Cloudflare's distributed database. `https://blog.cloudflare.com/workers-kv-is-ga/`.

[Cooper et al., 2010] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. (2010). Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA. Association for Computing Machinery.

[Dayarathna et al., 2016] Dayarathna, M., Wen, Y., and Fan, R. (2016). Data center energy consumption modeling: A survey. *IEEE Communications Surveys Tutorials*, 18(1):732–794.

[Eclipse Foundation,] Eclipse Foundation. Eclipse fog05 project. `https://fog05.io/`.

[El-Sayed et al., 2018] El-Sayed, H., Sankar, S., Prasad, M., Puthal, D., Gupta, A., Mohanty, M., and Lin, C. (2018). Edge of things: The big picture on the integration of edge, iot and the cloud in a distributed computing environment. *IEEE Access*, 6:1706–1717.

[Google,] Google. Data center locations. `https://www.google.com/about/datacenters/inside/locations/`.

[Gupta and Ramachandran, 2018] Gupta, H. and Ramachandran, U. (2018). Fogstore: A geo-distributed key-value store guaranteeing low latency for strongly consistent access. In *Proceedings of the 12th ACM International Conference on Distributed and Event-Based Systems*, DEBS '18, page 148–159, New York, NY, USA. Association for Computing Machinery.

[Hasenburg et al., 2019] Hasenburg, J., Grambow, M., and Bermbach, D. (2019). Fbase: A replication service for data-intensive fog applications.

[Karim Sonbol,] Karim Sonbol. Edgekv source code. `https://github.com/ksonbol/edgekv`.

[Kumar et al., 2016] Kumar, N., Zeadally, S., and Rodrigues, J. J. P. C. (2016). Vehicular delay-tolerant networks for smart grid data management using mobile edge computing. *IEEE Communications Magazine*, 54(10):60–66.

[Lebre et al., 2017] Lebre, A., Pastor, J., Simonet, A., and Desprez, F. (2017). Revising OpenStack to Operate Fog/Edge Computing Infrastructures. In *2017 IEEE International Conference on Cloud Engineering (IC2E)*, pages 138–148.

[Maheshwari et al., 2018] Maheshwari, S., Raychaudhuri, D., Seskar, I., and Bronzino, F. (2018). Scalability and performance evaluation of edge cloud systems for latency constrained applications. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 286–299.

[Mayer et al., 2017] Mayer, R., Gupta, H., Saurez, E., and Ramachandran, U. (2017). Fogstore: Toward a distributed data store for fog computing. In *2017 IEEE Fog World Congress (FWC)*, pages 1–6. IEEE.

[Microsoft Azure,] Microsoft Azure. Azure regions. `https://azure.microsoft.com/en-us/global-infrastructure/regions/`.

[Naha et al., 2018] Naha, R. K., Garg, S., Georgakopoulos, D., Jayaraman, P. P., Gao, L., Xiang, Y., and Ranjan, R. (2018). Fog computing: Survey of trends, architectures, requirements, and research directions. *IEEE Access*, 6:47980–48009.

[Ongaro and Ousterhout, 2014] Ongaro, D. and Ousterhout, J. (2014). In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA. USENIX Association.

[Ravindran and George, 2018] Ravindran, A. and George, A. (2018). An edge datastore architecture for latency-critical distributed machine vision applications. In *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*, Boston, MA. USENIX Association.

[Samie et al., 2019] Samie, F., Bauer, L., and Henkel, J. (2019). *Edge Computing for Smart Grid: An Overview on Architectures and Solutions*, pages 21–42. Springer International Publishing, Cham.

[Sasaki et al., 2016] Sasaki, K., Suzuki, N., Makido, S., and Nakao, A. (2016). Vehicle control system coordinated between cloud and mobile edge computing. In *2016 55th Annual Conference of the Society of Instrument and Control Engineers of Japan (SICE)*, pages 1122–1127.

[Sonbol and Ozkasap, 2019] Sonbol, K. and Ozkasap, O. (2019). Review of rdma-enabled consensus protocols. In *2019 International Symposium on Networks, Computers and Communications (ISNCC)*, pages 1–4.

[Sonbol and Ozkasap, 2020] Sonbol, K. and Ozkasap, O. (2020). Edgekv: Edge-optimized key-value storage. In *(submitted paper), The International Federation for Information Processing (IFIP) Networking 2020 Conference (NETWORKING 2020)*, pages 1–4.

[Stoica et al., 2001] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. (2001). Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31(4):149–160.

[WintelGuy.com,] WintelGuy.com. WAN Latency Estimator. `https://wintelguy.com/wanlat.html`.

[Yuan et al., 2018] Yuan, Q., Zhou, H., Li, J., Liu, Z., Yang, F., and Shen, X. S. (2018). Toward efficient content delivery for automated driving services: An edge computing solution. *IEEE Network*, 32(1):80–86.

[Zhang et al., 2017] Zhang, W., Chen, J., Zhang, Y., and Raychaudhuri, D. (2017). Towards efficient edge cloud augmentation for virtual reality mmogs. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC '17, New York, NY, USA. Association for Computing Machinery.