© 2018 Shadi Abdollahian Noghabi

BUILDING INTERACTIVE DISTRIBUTED PROCESSING APPLICATIONS AT A GLOBAL SCALE

BY

SHADI ABDOLLAHIAN NOGHABI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Doctoral Committee:

Professor Roy H. Campbell, Chair
Professor Indranil Gupta, Chair
Professor Klara Nahrstedt
Dr. Victor Bahl, Microsoft Research

ABSTRACT

Along with the continuous engagement with technology, many latency-sensitive *interactive applications* have emerged, e.g., global content sharing in social networks, adaptive lights/temperatures in smart buildings, and online multi-user games. These applications typically process a massive amount of data at a global scale. In this cases, distributing storage and processing is key to handling the large scale. Distribution necessitates handling two main aspects: a) the *placement* of data/processing and b) the *data motion* across the distributed locations. However, handling the distribution while meeting latency guarantees at large scale comes with many challenges around hiding heterogeneity and diversity of devices and workload, handling dynamism in the environment, providing continuous availability despite failures, and supporting persistent large state.

In this thesis, we show how *latency-driven designs for placement and data-motion* can be used to build *production infrastructures for interactive applications at a global scale*, while also being able to address myriad challenges on heterogeneity, dynamism, state, and availability. We demonstrate a latency-driven approach is general and applicable *at all layers* of the stack: from storage, to processing, down to networking.

We designed and built four distinct systems across the spectrum. We have developed Ambry (collaboration with LinkedIn), a geo-distributed storage system for interactive data sharing across the globe. Ambry is LinkedIn's mainstream production system for all its media content running across 4 datacenters and over 500 million users. Ambry minimizes user perceived latency via smart data placement and propagation. Second, we have built two processing systems, a traditional model, Samza, and the avant-garde model, Steel. Samza (collaboration with LinkedIn) is a production stream processing framework used at 15 companies (including LinkedIn, Uber, Netflix, and TripAdvisor), powering >200 pipelines at LinkedIn alone. Samza minimizes the impact of data motion on the end-to-end latency,

thus, enabling large persistent state (100s of TB) along with processing. Steel (collaboration with Microsoft) extends processing to the emerging edge. Integrated with Azure, Steel dynamically optimizes placement and data-motion across the entire edge-cloud environment. Finally, we have designed FreeFlow, a high performance networking mechanisms for containers. Using the container placement, FreeFlow opportunistically bypasses networking layers, minimizing data motion and reducing latency (up to 3 orders of magnitude).

To my husband and parents for their eternal love, support, and belief in me.

ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest gratitude to my advisors, Prof. Indranil Gupta (Indy) and Prof. Roy Campbell, for their invaluable guidance and continuous support throughout my Ph.D. studies. Indy always pushed me to shoot for the stars and be the best version of myself, and Roy's deep belief in me and the freedom he gave me, shaped me into a confident and independent person. I cannot believe the tremendous transformation I have gone through in my Ph.D. journey, and simply put, I could not have done it without the guidance and help of these two amazing advisors.

I would like to sincerely thank the rest of my committee members, Dr. Victor Bahl and Prof. Klara Nahrstedt. Their invaluable feedback has played a big part in taking my work from good to great. However, they were not just committee members for me. Klara has always been a big inspiration for me, encouraging me that I can also become a successful female researcher. Meeting Victor has been a big turning point in my life. Having a good mentor is a treasured gift, and I have been extremely lucky to not only have good but, an outstanding mentor.

I would like to thank all the collaborators throughout my journey. I was fortunate to work with amazing people at LinkedIn for a big part of my Ph.D. Many thanks to my managers Sriram Subramanian and Kartik Paramasivam who have also been amazing mentors for me and thought me to think deeply about my life goals. Also thanks to the many other collaborators, Yi Pan, Navina Ramesh, Jon Bringhurst, Wei Song, Xinyu Liu, Jagadish Venkatraman, Jacob Maes, Chris Riccomini, Priyesh Narayanan, Sivabalan Narayanan, Gopalakrishna Holla, Mammad Zadeh, Tianwei Li, Tofig Suleymanov, Arjun Shenoy and Dmitry Nikiforov and Jay Wylie for their invaluable contributions and feedback towards my projects. Many thanks also to outstanding collaborators during my time at Microsoft Research, especially, John Kolb, Peter Bodik, Eduardo Cuervo, Tianlong Yu, Shachar Raindel, Hongqiang Liu, Jitu

Padhye, and Vyas Sekar, for their tremendous help and input.

I would also like to express my sincere gratitude to my colleagues in the Computer Science Department. I was fortunate to be part of both the System Research Group (SRG) and Distributed Protocols Research Group (DPRG). Many thanks to the members of both these groups including Sayed Hadi Hashemi, Faraz Faghri, Read Sprabery, Imani Palmer, Chris Cai, Mohammad Ahmad, Shayan Saeed, Faria Kalim, Le Xu, Shegufta Bakht Ahsan, Mainak Ghosh, Mayank Bhatt, Luke Leslie, Jayasi Mehar, Mayank Pundir, Muntasir Raihan Rahman, Shiv Verma, and Srujun Gupta. I am forever indebted to them for their help, guidance, and support. Also, thanks to my many friends who made the small city of Champaign seem like the most fun place on earth!

Many thanks to the staff of the Computer Science department, including but not limited to: Kathy Runck, Kara MacGregor, Mary Beth Kelley, Viveka Perera Kudaligama, and Maggie Metzger Chappell for their prompt help and support at multiple junctures of my Ph.D. studies.

Finally, I could not have completed this difficult journey without the unconditional love and support of my family. My husband, Hassan Eslami, has not only been a partner, but also a friend and an anchor. He has always been there for me, putting me above all, and a tower of strength helping me overcome any small or large challenge I faced. My family had to endure the pain of being separated from me for several years. My dad is the reason I am where I am and my mom is the reason I am who I am. I am forever grateful for the inspiration my dad has been–as he is why I pursued a Ph.D.–and the endless effort my mom has put to shape me as a fearless young woman that can reach anything she wishes. My sister, Shana, and brother, Ali, are the most wonderful siblings one can ask for. I am extremely proud of them, and even though I was not there for them as the older sister, they have turned into outstanding individuals.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1
1.1 Thesis Contributions 3
1.2 Related Work 7
1.3 Thesis Organization 9

CHAPTER 2 AMBRY: LINKEDIN'S SCALABLE GEO-DISTRIBUTED OBJECT STORE 10
2.1 Introduction 10
2.2 System Overview 14
2.3 Load Balancing 19
2.4 Components in Detail 22
2.5 Replication 31
2.6 Experimental Results 33
2.7 Related Work 50
2.8 Conclusion 52

CHAPTER 3 SAMZA: STATEFUL STREAM PROCESSING AT SCALE 54
3.1 Introduction 54
3.2 Motivation 58
3.3 System Overview 63
3.4 System Design 70
3.5 Checkpointing VS. ChangeLog 79
3.6 Evaluation 81
3.7 Related Work 93
3.8 Conclusion 95

CHAPTER 4 STEEL: UNIFIED AND OPTIMIZED EDGE-CLOUD ENVIRONMENT 96
4.1 Introduction 96
4.2 Motivation and Background 100
4.3 Design of Steel 105
4.4 Experimental Evaluation 123
4.5 Related Work 129
4.6 Conclusion 130

CHAPTER 5 FREEFLOW: HIGH PERFORMANCE CONTAINER NETWORKING 132
5.1 Introduction 132
5.2 Background 136
5.3 Overview 137
5.4 Design 142

5.5 Preliminary Implementation 145
5.6 Related Work 145
5.7 Conclusion and Discussion 147

CHAPTER 6 CONCLUSION AND FUTURE WORK 149
6.1 Summary 149
6.2 Future Work 150

REFERENCES 153

CHAPTER 1: INTRODUCTION

In this era of increased engagement with technology, many latency-sensitive applications, or so called *interactive* applications have emerged. For example, we expect social networks to show our uploaded photos and videos immediately to all our friends (within seconds); ad campaigns to orient and update ads based on the current (at most the past few minutes of) user activity; Internet of Things (IoT) environments, like smart cities, to process and react to sensor data within seconds; and multi-user online games to support delays of a couple milliseconds. The sensitivity to latency is diverse across the applications, from milliseconds to even minutes. However, in all cases, being reactive with low latency is a governing factor with impacts on safety, engagement, or revenue [1–6].

In addition, a major portion of these interactive applications, are operating at a massive scale from a large region (e.g., smart city) to even the entire globe (e.g., social networks). When building an interactive application at a global scale, distributing the computation and data becomes key. Distribution originates two main questions to be solved. First, *where to place the distributed processing or data (placement)?* Second, *how to move and propagate data across the distributed locations (data-motion)?*

At large scale, providing low latency and interactivity becomes increasingly challenging with many complexities in placement and data-motion. Along with scale comes diversity and heterogeneity across the environment including in: the data objects, the processing operations, the physical machines/devices, and the network links. This diversity significantly complicates optimizations [7–10]. Similarly, the environment becomes increasingly more dynamic, with a combination of spiky, long-term, and periodic changes in the workloads, necessitating adaptive approaches to hide this dynamism. To support certain latencies, proximity to data/processing becomes extremely crucial, requiring geo-distribution across multiple locations around the globe (or an area). Additionally, a majority of the computations generates state (persistent data) alongside computation. When operating at a

Table 1.1: Systems developed/analyzed in the thesis and the placement and data-motion notion in each.

System	System Type	Placement	Data Motion
Ambry	storage	load balanced data placement	geo-distributed data propagation
Samza	processing	task placement	stateful stream processing
Steel	processing	task placement on edge	edge cloud communication
FreeFlow	networking	abstract container placement	container communication

large scale, the state also becomes large, and a main limiting factor for both placement and movement.

In this thesis we have focused on the placement and data motion in a *variety of production* systems used across *multiple companies*. The work spans a wide range of systems, from a storage system, two processing systems–the traditional stream processing and the avant-garde edge processing–and a networking mechanism. We worked on transparently mitigating the challenges and requirements of a global (or sub-global) environment, including the heterogeneity, dynamism, stateful processing and geo-distribution. This leads to our thesis:

We show how *latency-driven designs for placement and data-motion* can be used to build *production infrastructures for interactive applications at a global scale*, while addressing myriad challenges on heterogeneity, dynamism, state, and availability.

A *latency-driven design* is an approach where achieving latency has the highest priority–as latency is the main factor for interactivity. As suggested by the CAP theorem and its PACELC extension [11,12], there is a fundamental trade-off between consistency, availability and partition-tolerance, and in case of no failures, between latency, availability and consistency. In a latency-driven design, when faced with such trade-offs, the priority is given to latency. The decision of what to compromise instead, is use-case dependent, e.g., a photo sharing network would probably care for having availability of service over strong consistency.

This latency-driven approach is applicable across all layers of storage, processing and

networking. Table 1.1, outlines the various systems we have developed and analyzed in this thesis, which layer they belong to, and the definition of placement and data-motion in that context. Our works spans across all layers, including:

- **Storage:** we have developed *Ambry*, a geo-distributed storage system where data placement is designed with the goal of achieving load balance, and data is constantly moved across multiple datacenters.

- **Processing:** we have built two processing systems, a traditional model, *Samza*, and the avant garde model, *Steel*. Samza is a stream processing system, where we place processing and support state at large scale, with the goal of minimizing data movement. Steel extends processing to the emerging edge environment with smart placement and data-motion across the entire edge-cloud environment.

- **Networking:** we have designed *FreeFlow*, a high performance networking mechanisms for containerized environments, abstracting container placements and minimizing data movement in container communications.

Figure 1.1 shows the big picture and how these various layers build the overall global system.

In the remaining of this chapter, we briefly discuss the projects, the broad impact of each system, and the intellectual merit of each.

1.1 THESIS CONTRIBUTIONS

Solving real-world problems faced by costumers has been a first class citizen in my work. Most of my work is deployed at large-scale and in mainstream production systems (e.g., Ambry and Samza) with hundreds of millions of users. In this section, we highlight the main contributions and impact of our proposed solutions.

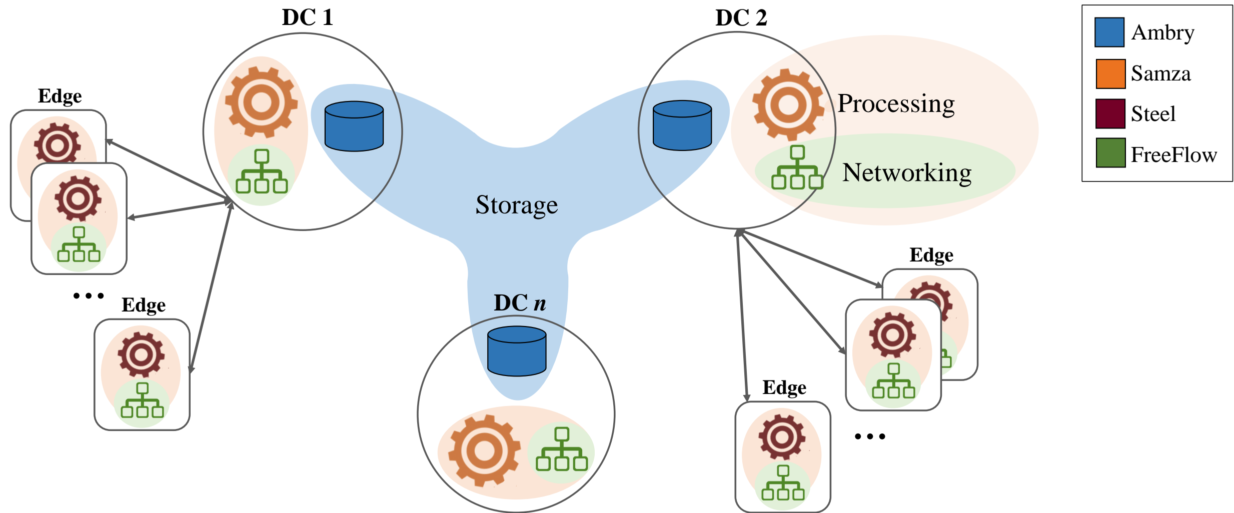


Figure 1.1: Systems studied in the thesis and their interaction.

1.1.1 Ambry: Geo-distributed Low Latency Object Storage

In today's high-tech connected world, an immense amount of data is generated every second from all around the world. Social networks alone generated billions of variable-sized media objects per day. This excessive amount of diverse data needs to be served with low latency from all across the globe, while transparently scaling to match the ever-growing amount of data. In collaboration with LinkedIn, we developed Ambry, a geo-distributed, scalable, and low-latency object store for a broad range of objects (a few KBs to a few GBs) [13]. For over 36 months, Ambry has been the **mainstream** storage for all LinkedIn's media objects, across all of its four datacenters, serving more than 500 million users.

1.1.2 Samza: Stateful Stream Processing at Scale

When it comes to large state, relying on an external storage is not a viable solution (orders of magnitude degradation in latency and throughput). Therefore, we developed a state handling mechanism leveraging the local storage of the computing nodes [14]. Samza utilizes a partitioned local state along with a low-overhead background changelog mechanism, allowing it to scale to massive state sizes (hundreds of TB) per application. Samza is

currently in use at LinkedIn by hundreds of production applications with more than 10,000 containers while processing **trillions of events per day**. Samza is an open-source Apache project adopted by more than 15 companies (Uber, Netflix, TripAdvisor, etc.)

1.1.3 Steel: Optimizing and Simplifying Edge-Cloud Applications

Edge Computing is becoming a leading technology, opening the venue for a wide range of interesting applications ranging from self-driving cars, smart cities/homes, to wireless Virtual and Augmented Reality. Edges are heterogeneous, fast (near the data source), and with limited resources and bandwidth. On the other hand, the cloud is a homogeneous high(er)-latency resource. The cloud already provides a wide diversity of cloud service, e.g., streaming as a service. In this work we developed Steel that integrates the cloud and the edge into one unified geo-distributed environment [15, 16] . We support end-to-end development, deployment, and monitoring, extending cloud services to the edge. In addition, we provide optimizations for placement and communication in order to adapt to this more dynamic and heterogeneous environment. Steel is developed on top of the Azure and Windows IoT stack, and is in progress to be integrated with the production Azure code base.

1.1.4 FreeFlow: High Performance Container Networking

Containerization has piqued a strong interest in Big Data Analytics particularly because of its high portability (easily moving around), and low overhead (as they are a process on a common guest OS)[17]. However, the current cross-container networking mechanisms do not take into account the hardware capabilities (e.g. RDMA-enabled) or the locations of containers, thus, resulting in poor performance. Manual efforts to optimize networking impacts the portability to new environments. To fill this gap, we designed and developed FreeFlow, a high performance and portable container networking solution [18]. FreeFlow acts as a middle layer between the applications and underlying network, transparently choosing the optimal network option based on location (e.g., shared memory on the same host),

and hardware capabilities (e.g., use RDAM if available). FreeFlow is a general container networking technology and can be applied to any of the containerization frameworks such as Docker, Kubernetes, Mesos, YARN, CoreOS, etc. [19–23]. The team I worked with at Microsoft are analyzing adding this contribution to their production container technology.

1.1.5 Overview of Techniques

In this thesis we have focused on placement and data-motion across a wide range of system. Each system was developed with its own set of requirements fit for that system and unique set of challenges around heterogeneity and dynamism. Towards reaching interactivity while mitigating the challenges, we have employed a number of latency-driven techniques including:

1. Leveraging *locality* by preferring local data, local storage, sticky processing and location awareness. The term locality is used in the broad sense where it incorporates resource locality (e.g., stickiness to a machine or using local storage), data locality (e.g., re-access of data), and location awareness.
2. *Background processing* by pushing computation/data propagation to the background out of the critical path of processing, along with optimization such as batching, compression and compaction.
3. *Prioritizing latency* over consistency and availability
4. *Tiered data access* along with caching, indexing, and bloom filters
5. *Opportunistic processing*
6. *Partitioning and parallelism*
7. *Load balancing* and *scaling*

Table 1.2 summarizes the main requirements and challenges in each system, and governing placement and data-motion technique used to reach the goal of the system. In the following chapters we describe each system and techniques used in more detail.

1.2 RELATED WORK

This thesis argues for a latency-driven approach when building frameworks for interactive applications. We have shown this is a practical principal, employing it across a wide range of systems. As PACELC theorem proves [12], latency comes in a trade-off with availability and consistency. Others have used a similar latency-driven approach across many frameworks. The whole trend of NoSQL systems, such as Cassandra, Amazon's DynamoDB, Pileus and others [24–27], prefer latency (in a static or dynamic manner) over consistency.

Despite the suitable fit of a latency-driven approach for interactivity, in many cases latency is not the only or main requirement. Sometimes, guaranteeing consistency comes at higher priority, e.g., a banking system, user profile settings, and an inventory system. Many distributed storage systems, such as BigTable, Spanner, Yahoo's PNUTS, and LinkedIn's Espresso [28–31] support strong consistency. These consistency-driven designs typically come at cost on latency [27, 32] with techniques such as synchronous replications in multiple locations (vs. lazy background), master-slave and multi-phase approaches creating extra hops in processing a request, or bottleneck-prone locking schemas with high idle times waiting for locks. Similar trends are existent in processing systems. Many stream processing frameworks, including Flink, Millwheel, Spark Streaming, and Trident [33–37], aim for "exactly-once" processing (i.e., each incoming message is processed exactly once) using complex tracking mechanism and by relying on external consistent storages.

Another approach is a resource/throughput oriented design. In cases with limited resources or excessive amount of data, reaching high resources utilization and throughput becomes the main priority. Many batch systems (e.g., Hadoop, Spark, Tez, Pig and Hive

Table 1.2: Requirements, challenges, and solutions used in systems of this thesis with focus on placement and data-motion.

System	Requirement	Challenge: Heterogeneity	Challenge: Dynamism	Latency-driven Techniques
Ambry	**- geo-distributed:** across multiple datacenters **- state:** persistent and reliable data store **- latency:** interactive sharing experience	- object sizes and types - machines across DCs - object access pattern - links between DCs and users	- workload (diurnal, spikes) - cluster expansion - dropping popularity over time	- locality (local data & location-aware) - background processing (w/ batching & compaction) - prioritizing latency over consistency - tiered storage (w/ caching, indexing, & bloom filters) - partitioning and parallelism - load balancing
Samza	**- state:** stateful stream processing **- latency:** near real-time processing at scale	- input source (batch/stream/db) - storage option (mem, disk, remote) - operation type	- input stream load changes - failures and restarts	- locality (local storage & sticky processing) - background processing (w/ batching & compaction) - prioritizing latency over availability - tiered storage (w/ caching) - partitioning and parallelism - scaling
Steel	**- geo-distributed:** across many locations close to users **- latency:** interactive processing experience	- edge devices - edge-cloud network - cloud services - cost/perf model	- permanent load changes - failures and migration - short spikes	- locality (local data & location-aware) - background processing (w/ batching & compression) - prioritizing latency over availability - partitioning and parallelism
FreeFlow	**- latency:** Fast high performance container networking	- networking hardware's capabilities - networking technologies	- container failures or move to new location	- locality (location-aware) - prioritizing latency over isolation - opportunistic bypassing

[38–42]), throughput oriented stream processing systems (e.g., Spark Streaming [36]), and high throughput storage systems (e.g., HDFS, NFS, and GFS [43–45]) have been built with this promise. Typically, throughput and resource utilization come in trade-off with latency, for example when employing approaches such as large batching, compaction, compression, and the use of slow but cheaper storage.

1.3 THESIS ORGANIZATION

The rest of the thesis is structured as follows. In the subsequent chapter we dive into each system in more detail, along with experimental evaluations for each. Chapter 2 describes the details of Ambry, a geo-distributed storage system. Chapters 3 and 4 respectively describe Samza and Steel, a stateful stream processing system and extending processing to the edge. Chapter 5 outlines FreeFlow, a high performance container technology. Finally, we conclude by presenting our future directions Chapter 6.

CHAPTER 2: AMBRY: LINKEDIN'S SCALABLE GEO-DISTRIBUTED OBJECT STORE

In this chapter we present Ambry, a geo-distributed storage system designed for managing large objects with low latency, high throughput, and in a balanced manner. The infrastructure beneath a worldwide social network has to continually serve billions of variable-sized media objects such as photos, videos, and audio clips. These objects must be stored and served with low latency and high throughput by a system that is geo-distributed, highly scalable, and load-balanced. Existing file systems and object stores face several challenges when serving such large objects. We present Ambry, a production-quality system for storing large immutable data (called blobs). Ambry is designed in a decentralized way and leverages techniques such as logical object grouping abstractions, asynchronous replication, rebalancing mechanisms, zero-cost failure detection, and OS caching, towards smart placement and data-motion. Ambry has been running in LinkedIn's production environment for the past 3 years, serving up to 10K requests per second across more than 500 million users. Our experimental evaluation reveals that Ambry offers high efficiency (utilizing up to 88% of the network bandwidth), low latency (less than 50 ms latency for a 1 MB object), and load balancing (improving imbalance of request rate among disks by 8x-10x).

2.1 INTRODUCTION

During the past decade, social networks have become popular communication channels worldwide. Hundreds of millions of users continually upload and view billions of diverse massive media objects, from photos and videos to documents. These large media objects, called *blobs*, are uploaded once, frequently accessed from all around the world, never modified, and rarely deleted. LinkedIn, as a global large-scale social network company, has faced the need for a geographically distributed system that stores and retrieves these read-heavy blobs in an efficient and scalable manner.

Handling blobs poses a number of unique challenges. First, due to diversity in media types, blob sizes vary significantly from tens of KBs (e.g., profile pictures) to a few GBs (e.g., videos). The system needs to store both massive blobs and a large number of small blobs efficiently. Second, there is an ever-growing number of blobs that need to be stored and served. Currently, LinkedIn serves more than 800 million put and get operations per day (over 120 TB in size). In the past 12 months, the request rate has almost doubled, from 5k requests/s to 9.5k requests/s. This rapid growth in requests magnifies the necessity for a linearly scalable system (with low overhead). Third, the variability in workload and cluster expansions can create unbalanced load, degrading the latency and throughput of the system. This creates a need for load-balancing. Finally, users expect the uploading process to be fast, durable, and highly available. When a user uploads a blob, all his/her friends from all around the globe should be able to see the blob with very low latency, even if parts of the internal infrastructure fail. To provide these properties, data has to be reliably replicated across the globe in multiple datacenters, while maintaining low latency for each request.

LinkedIn had its own home-grown solution called Media Server, built using network attached storage filers (for file storage), Oracle database (for metadata), and Solaris boxes. Media Server had multiple drawbacks. It faced CPU and IO spikes caused by numerous metadata operations for small objects, was not horizontally scalable, and was very expensive. Given that LinkedIn was scaling rapidly and the future web content will be largely dominated by media, it needed to find a replacement.

Several systems have been designed for handling a large amount of data, but none of them satisfactorily meet the requirements and scale LinkedIn needs. There has been extensive research into distributed file systems [43–47]. These systems have a number of limitations when used for storing blobs, as pointed out by [48, 49]. For instance, the hierarchical directory structure and rich metadata are an overkill for a blob store and impose unnecessary additional overhead.

Many key value stores [24, 25, 28, 50] have also been designed for storing a large number

of objects. Although these systems can handle many small objects, they are not optimized for storing large objects (tens of MBs to GBs). Further, they impose extra overhead for providing consistency guarantees while these are typically not needed for immutable data. Some examples of these overheads include using vector clocks, conflict resolution mechanism, logging, and central coordinators.

A few systems have been designed specifically for large immutable objects including Facebook's Haystack [48] along with f4 [51] and Twitter's Blob Store [52]. However, these systems do not resolve load imbalance, especially when cluster expansions occur.

In this work we present Ambry[1], a production-quality system designed specifically for diverse large and small immutable data with read-heavy traffic, where data is written once, and read many times (>95% read traffic). Ambry is designed with four main goals in mind:

1. **Low Latency and High Throughput:** The system needs to serve a large number of requests per second in a timely fashion, while working on cheap commodity hardware (e.g., HDDs). In order to reach this goal, Ambry utilizes a number of techniques including exploiting the OS cache, using zero copy when reading data from disk to network, chunking data along with retrieving/storing chunks in parallel from multiple nodes, providing configurable polices for the number of replicas to write and read, and zero-cost failure detection mechanisms (Sections 2.2.3, 2.4.2, and 2.4.4).

2. **Geo-Distributed Operation:** Blobs have to be replicated in other geographically distributed datacenters for high durability and availability, even in the presence of failures. To achieve low latency and high throughput in this geo-distributed setting, Ambry is designed as a decentralized multi-master system where data can be written to or read from any of the replicas. Additionally, it uses asynchronous writes that write data to the closest datacenter and asynchronously replicate to other datacenter(s). Also, for higher availability, it uses proxy requests that forward requests to other

[1] Ambry is open-source and can be found at `https://github.com/linkedin/ambry`

datacenters when the data is not replicated in the current datacenter yet (Sections 2.2.3 and 2.4.2).

3. **Scalability:** With the ever-growing amount of data, the system has to scale out efficiently with low overhead. To achieve this goal, Ambry makes three main design choices. First, Ambry separates the logical placement of blobs from their physical placement, allowing it to change the physical placement transparently from the logical placement. Second, Ambry is designed as a completely decentralized system, with no manager/master. Third, Ambry uses on-disk segmented indexing along with Bloom filters and an in-memory cache of the latest segment, allowing for scalable and efficient indexing of blobs. (Section 2.4.4).

4. **Load Balancing:** The system has to stay balanced in spite of growth. Ambry uses chunking of large blobs along with a random selection approach to remain balanced in a static cluster, and a re-balancing mechanism to return to a balanced state whenever cluster expansion occurs (Section 2.3).

Ambry has successfully been in production for the last 36 months, across four datacenters, serving more than 500 million users. Our experimental results show that Ambry reaches high throughput (reaching up to 88% of the network bandwidth) and low latency (serving 1 MB blobs in less than 50 ms), works efficiently across multiple geo-distributed datacenters, and improves the imbalance among disks by a factor of 8x-10x while moving minimal data.

This chapter discusses the design and implementation of Ambry. The main contributions of this work are:

- Design and implementation of a scalable load-balanced blob Store working across datacenters, while maintaining low latency and high throughput.

- Evaluating various aspects of the system including latency, throughput, load-imbalance and scalability in real-world clusters.

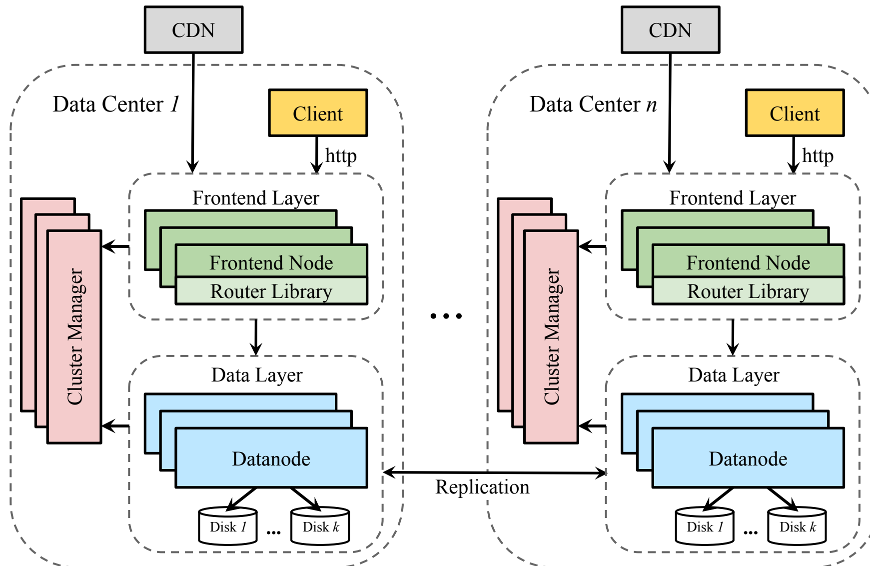


Figure 2.1: Architecture of Ambry.

- Analyzing the system over long period of time and varying workload using a simulator based on real-world data.

The rest of the chapter is organized as follows. In Section 2.2, we discuss the design and architecture of Ambry. Section 2.6 evaluates the system with a number of experimental results. Finally, we analyze the state-of-art related work in this are in Section 2.7 and conclude in Section 2.8.

2.2 SYSTEM OVERVIEW

In this section we discuss the overall design of Ambry including the high-level architecture of the system (Section 2.2.1), the notion of partition (Section 2.2.2), and supported operations (Section 2.2.3).

2.2.1 Architecture

Ambry is designed as a completely decentralized multi-tenant system across geographically distributed data centers. The overall architecture of Ambry is shown in Figure 2.1. The

system is composed of three main components: *Frontends* that receive and route requests, *Datanodes* that store the actual data, and *Cluster Managers* that maintain the state of the cluster. Each datacenter owns and runs its own set of these components in a decentralized fashion. The Frontends and Datanodes are completely independent of one another, and the Cluster Managers are synchronized using Zookeeper [53]. We provide an overview of each component below (details in Section 2.4):

- **Cluster Manager:** Ambry organizes its data in virtual units called *partitions* (Section 2.2.2). A partition is a logical grouping of a number of blobs, implemented as a large replicated file. On creation, partitions are read-write, i.e., immutable blobs are read and new blobs can be added. When a logical partition reaches its capacity, it turns read-only. The Cluster Manager keeps track of the state (read-write/read-only) and location of each partition replica, along with the physical layout of the cluster (nodes and disk placement).

- **Frontend:** The Frontends are in charge of receiving and routing requests in a multi-tenant environment. The system serves three request types: put, get, and delete. Popular data is handled by a Content Delivery Network (CDN) layer above Ambry. Frontends receive requests directly from clients or through the CDN (if the data is cached). The Frontends forward a request to the corresponding Datanode(s) and return the response to the client/CDN originating the request.

- **Datanode:** Datanodes store and retrieve the actual data. Each Datanode manages a number of disks. Datanodes receive operations from Frontends and apply the operations on the disks they are in charge of. For better performance, Datanodes maintain a number of additional data structures including: indexing of blobs, journals and Bloom filters (Section 2.4.4).

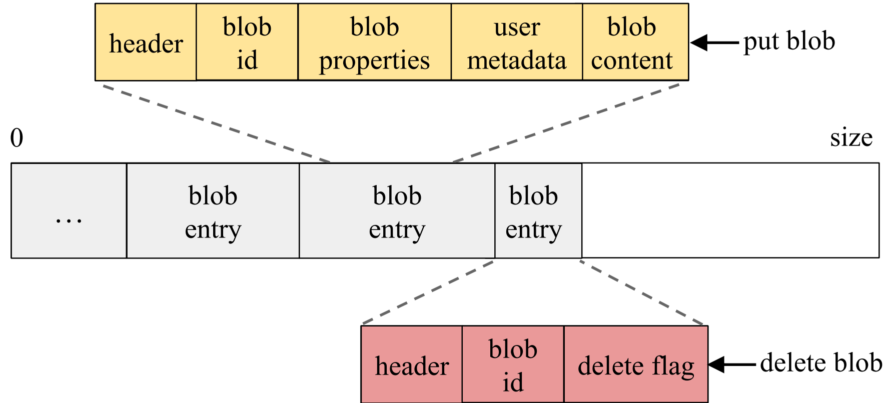


Figure 2.2: Partition and Blob layout.

2.2.2 Partition

Instead of directly mapping blobs to physical machines, e.g., Chord [54] and CRUSH [55], Ambry randomly groups blobs together into virtual units called *partitions*. The physical placement of partitions on machines is done in a separate procedure. This decoupling of the logical and physical placement enables transparent data movement (necessary for re-balancing) and avoids immediate rehashing of data during cluster expansion.

A partition is implemented as an append-only log in a pre-allocated large file. Currently, partitions are fixed-size during the life-time of the system [2]. The partition size should be large enough that the overhead of partitions, i.e., the additional data structures maintained per partition such as indexing, journals, and Bloom filters (Section 2.4.4), are negligible. On the other hand, the failure recovery and rebuild time should be small. We use 100 GB partitions in our clusters. Since rebuilding is done in parallel from multiple replicas, we found that even 100 GB partitions can be rebuilt in a few minutes.

Blobs are sequentially written to partitions as put and delete entries (Figure 2.2). Both entries contain a header (storing the offsets of fields in the entry) and a *blob id*. The blob id is a unique identifier, generated by the Frontend during a put operation, and used during get/delete operations for locating the blob. This id consists of the partition id in which

[2] As part of future work we plan to investigate potential improvements by using variable-size partitions.

the blob is placed (8 Bytes), followed by a 32 Byte universally unique id (UUID) for the blob. Collisions in blob ids are possible, but very unlikely (the probability is $< 2^{-320}$). For a collision to occur, two put operations have to generate equal UUIDs and chose similar partitions for the blob. Collisions are handled at the Datanodes by failing the late put request.

Put entries also include predefined properties including: blob size, time-to-live, creation time, and content type. Also, there is an optional map of user defined properties followed by the blob. Delete entries include a delete flag as an indicator of a delete entry.

In order to offer high availability and fault-tolerance, each partition is replicated on multiple Datanodes. For replica placement, Ambry uses a greedy approach based on disk spaces. This algorithm chooses the disk with the most unallocated space while ensuring constraints such as: 1) not having more than one replica per Datanode and 2) having replicas in multiple data centers. Currently, the number of replicas per partition is configurable by the system administrator. As part of future work, we plan to adaptively change the number of replicas based on the popularity of the partition, and use erasure coding for cold data to even further reduce the replication factor.

On creation, partitions are read-write, serving all operations (put, get and delete). When the partition hits its upper threshold on size (capacity threshold) it becomes read-only, thereafter serving only get and delete operations.

The capacity threshold should be slightly less than the max capacity (80-90%) of the partition for two reasons. First, after becoming read-only, replicas might not be completely in-sync and need free space to catch-up later (because of asynchronous writes). Second, delete requests still append delete entries.

Deletes are similar to put operations, but on an existing blob. By default, deletes result in appending a delete entry (with the delete flag set) for the blob (soft delete). However, Ambry also supports hard deletes where data is overwritten with random values. Deleted blobs are periodically cleaned up using an in-place compaction mechanism. After compaction,

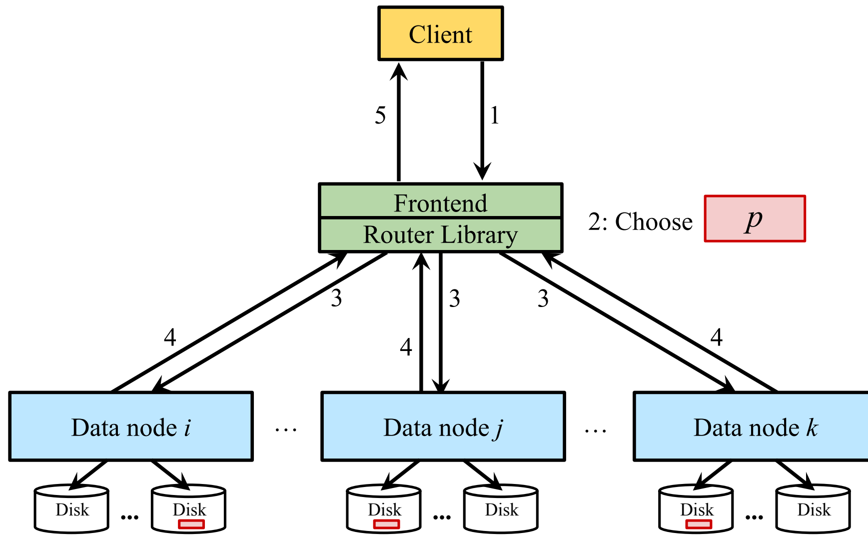


Figure 2.3: Steps in processing an operation.

read-only partitions can become read-write if enough space is freed-up. In the rest of the chapter we mainly focus on puts, due to the similarity of delete and put operations.

2.2.3 Operations

Ambry has a lightweight API supporting only 3 operations: put, get, and delete. The request handling procedure is shown in Figure 2.3. On receiving a request, the Frontend optionally conducts some security checks on the request. Then, using the Router Library (that contains the core logic of operation handling) it chooses a partition, communicates with the Datanode(s) in charge, and serves the request. In the put operation, the partition is chosen randomly (for data balancing purposes), and in the get/delete operation the partition is extracted from the blob id.

Operations are handled in a multi-master design where operations can be served by any of the replicas. The decision of how many replicas to contact is based on user-defined policies. These policies are similar to consistency levels in Cassandra [24], where they control how many (one, k, majority, all) replicas to involve in an operation. For puts (or deletes), the request is forwarded to all replicas, and policies define the number of acknowledgments

needed for a success (trade-off between durability and latency). For gets, policies determine how many randomly selected replicas to contact for the operation (trade-off between resources usage and latency). In practice, we found that for all operations the $k = 2$ replica policy gives us the balance we desire. Stricter polices (involving more replicas) can be used to provide stronger consistency guarantees.

Additionally, performing write operations to all replicas placed in multiple geo-distributed datacenters in a synchronous fashion can affect the latency and throughput. In order to alleviate this issue, Ambry uses asynchronous writes where puts are performed synchronously only in the local datacenter, i.e., the datacenter in which the Frontend receiving the request is located. The request is counted as successfully finished at this point. Later on, the blob is replicated to other datacenters using a lightweight replication algorithm (Section 2.5).

In order to provide read-after-write consistency in a datacenter which a blob has not been replicated yet (e.g., writing to one datacenter and reading from another), Ambry uses proxy requests. If the Frontend cannot retrieve a blob from its local datacenter, it proxies the request to another datacenter and returns the result from there. Although a proxy request is expensive, in practice we found that proxy requests happen infrequently (less than 0.001 % of the time).

2.3 LOAD BALANCING

Skewed workloads, massively large blobs, and cluster expansions create load imbalance and impact the throughput and latency of the system. Ambrytries to achieve balance in terms of two factor a) disk usage and b) request rates. We study load balancing in two cases: a *static cluster* where nodes and disks are not added or deleted and a *dynamic cluster* with continuous expansion of the cluster.

2.3.1 Load Balancing a Static Cluster

In a static cluster Ambry uses a combination of three simple techniques: random placement, chunking, and caching. By routing put operations to random partitions, it is expected to have a balanced number of puts per partition. By splitting large blobs into multiple small chunks, it is expected to have an balance size across blobs, thus, overall reaching a balance in terms of the overall data stored at each partition and the disk usage. However, this does not mean balanced request rates across disks. Ambry relies on caching, in a CDN layer above it, to serve the extremely hot and outlier blobs. Since partitions are fairly large, with millions of blobs per partition, partitions have a similar distribution of data, especially since the hot outliers are served outside of Ambry. Using these techniques the load imbalance of request rates and partition sizes in production gets to as low as 5% amongst Datanodes.

2.3.2 Load Balancing a Dynamic Cluster

In practice, read-write partitions receive all the write traffic and also the majority of the read traffic (due to popularity). Since partitions grow in a semi-balanced manner, the number of read-write partitions becomes the main factor of load imbalance. After cluster expansion, new Datanodes contain only read-write partitions, while older Datanodes contain mostly read-only partitions. This skewed distribution of read-write partitions creates a large imbalance in the system. In our initial version, the average request rates of new Datanodes were up to 100x higher than old Datanodes and 10x higher than the average-aged ones.

To alleviate this issue, Ambry employs a rebalancing mechanism that returns the cluster to a semi-balanced state (in terms of disk usage and request rate) with minimal data movement. The rebalancing approach reduces request rate and disk usage imbalance by 6-10x and 9-10x respectively.

Ambry defines the ideal (load balanced) state as a triplet (idealRW, idealRO, idealUsed) representing the ideal number of read-write partitions, ideal number of read-only partitions

and ideal disk usage each disk should have. This ideal state (idealRW, idealRO, idealUsed) is computed by dividing the total number of read-write/read-only partitions and total used disk space by the number of disks in the cluster, respectively. A disk is considered above (or below) ideal if it has more (or less) read-write/read-only partitions or disk usage than the ideal state.

The rebalancing algorithm attempts to reach this ideal state. This is done by moving partitions from disks above ideal to disks below ideal using a two-phase approach, as shown in the pseudo-code below.

Algorithm 2.1 Rebalancing Algorithm

```
1: // Compute ideal state.
2: idealRW=totalNumRW / numDisks
3: idealRO=totalNumRO / numDisks
4: idealUsed=totalUsed / numDisks

5: // Phase1: move extra partitions into a partition pool.
6: partitionPool = {}
7: for each disk d do
8:     // Move extra read-write partitions.
9:     while d.NumRW > idealRW do
10:        partitionPool += chooseMinimumUsedRW(d)
11:    // Move extra read-only partitions.
12:    while d.NumRO > idealRO & d.used > idealUsed do
13:        partitionPool += chooseRandomRO(d)

14: // Phase2: Move partitions to disks needing partitions.
15: placePartitions(read-write)
16: placePartitions(read-only)

17: function placePartitions(Type t)
18:    while partitionPool contains partitions type t do
19:        D=shuffleDisksBelowIdeal()
20:        for disk d in D and partition p in pool do
21:            d.addPartition(p)
22:            partitionPool.remove(p)
```

Phase1 - Move to Partition Pool: In this phase, Ambry moves partitions from disks above ideal into a pool, called *partitionPool* (Lines 6-13). At the end of this phase no disk

should remain above ideal, unless removing any partition would cause it to fall below ideal.

Ambry starts from read-write partitions (which are the main factor), and moves extra ones solely based on idealRW threshold. The same process is repeated for read-only partitions, but with considering both idealRO and idealUsed when moving partitions. The strategy of choosing which partition to move is based on minimizing data movement. For read-write partitions, the one with the minimum used capacity is chosen, while for read-only partitions, a random one is chosen since all such partitions are full.

Phase2 - Place Partitions on Disks: In this phase, Ambry places partitions from the partition pool on disks below ideal (Lines 14-16), starting from read-write partitions and then read-only ones. Partitions are placed using a random round-robin approach (Line 17-22). Ambry finds all disks below ideal, shuffles them, and assigns partitions to them in a round-robin fashion. This procedure is repeated until the pool becomes empty.

After finding the the new placement, replicas are seamlessly moved by: 1) creating a new replica in the destination, 2) syncing the new replica with old ones using the replication protocol while serving new writes in all replicas, and 3) deleting the old replica after syncing. Although the rebalancing algorithm has low complexity, moving data around can impact the performance of the system. Thus, we are working on performing rebalancing in the background when the load of the system is low, e.g., overnight.

2.4 COMPONENTS IN DETAIL

In this section we further discuss the main components of Ambry. We describe the detailed state stored by the Cluster Manager (Section 2.4.1), extra responsibilities of Frontends including chunking and failure detection (Section 2.4.2), and additional structures maintained by the Datanodes (Section 2.4.4).

Table 2.1: Hardware layout in Cluster Manager.

Datacenter	Datanode	Disk	Size	Status
DC 1	Datanode 1	disk 1	4 TB	UP
		...	...	...
		disk k	4 TB	UP
DC 1	Datanode 2	disk 1	4 TB	DOWN
		...	...	...
		disk k'	4 TB	UP
	...	...	...	...
DC n	Datanode j	disk 1	1 TB	DOWN
		...	...	...
		disk k''	1 TB	UP

2.4.1 Cluster Manager

The Cluster Manager is in charge of maintaining the state of the cluster. Each datacenter has its local Cluster Manager instance(s) kept in-sync with others using Zookeeper. The state stored by the Cluster Manager is very small (less than a few MBs in total), consisting of a hardware and logical layout.

Hardware Layout

The hardware layout includes information about the physical structure of the cluster, i.e., the arrangement of datacenters, Datanodes, and disks. It also maintains the raw capacity and status, i.e., healthy (UP) or failed (DOWN), for each disk. An example hardware layout is shown in Table 2.1. As shown, Ambry works in a heterogeneous environment with different hardware and configuration used inside and across different datacenters.

Logical Layout

Partitions act as logical placeholders for blobs such that blobs are mapped to partitions completely unaware of the physical placement. Each partition has number of replicas, where

Table 2.2: Logical Layout in Cluster Manager.

Partition id	**State**	**Placement**
partition 1	read-write	DC 1: Datanode 1: disk 1 DC 1: Datanode 4: disk 5 ... DC 3: Datanode 7: disk 2
partition 2	read-only	DC 1: node 1: disk 1 ... DC 3: node 5: disk 1
...	...	...
partition p	read-only	DC 1: Datanode 1: disk 1 ... DC 4: Datanode 5: disk 2

each replica is placed on a disk. The logical layout maintains the physical location of partition replicas, and the state (read-only/read-write) of each partition. In order to find the state of a partition, the Cluster Manager periodically contacts the Datanodes, and requests the state of their partitions. This layout is used for choosing a partition to write a new blob to (put operation), and locating the Datanode in charge of a given replica (all operations). An example of this layout is shown in Table 2.2. As shown, replicas of a partition can be placed on multiple Datanodes in one datacenter, and/or in different datacenters. Additionally, one disk (e.g., DC 1: Datanode 1: disk 1) can contain replicas of distinct partitions, where some are read-only and some are read-write. Partitions are added by updating the logical layout stored in the Cluster Manager instances[3].

2.4.2 Frontend Layer

The Frontend is the entry point to Ambry for external requests. Each datacenter has its own set of Frontends. Frontends are decentralized involving no master or coordination,

[3]Currently, the system administrator manually adds partitions in order to prevent unwanted and rapid cluster growths. However, this can easily be automated.

identically performing the same task, and stateless with all state stored in the Cluster Manager (which is periodically pulled). This design enhances scalability (new Frontends can be added without much performance penalty), fault-tolerance (requests can be forwarded to any Frontend), and failure recovery (failed Frontends can quickly be replaced) for Frontends. Frontends have three main responsibilities:

1. Request Handling: This involves receiving requests, routing them to the corresponding Datanode(s) using the Router Library and sending back the response.
2. Security Checks: Optionally performing security checks, such as virus scanning and authentication on requests.
3. Capturing Operations: Pushing events to a change capture system out of Ambry for further offline analysis, such as finding request patterns of the system. We use Kafka [56] as our change-capture system due to the high durability, high throughput, and low overhead it provides.

At the core of the Frontend is the Router Library which actually performs the routing. Upon receiving a request, the Frontend performs security checks, logs the operation, uses the Router Library to route the request, and returns the response from the Router Library back to the user. The following section discusses the details of the Router Library, and the functionalities it supports.

2.4.3 Router Library

The Router Library contains all the core logic of handling requests and communicating with Datanodes. Frontends simply embed and use this library. Clients can bypass Frontends by embedding this library and directly serving requests. This library includes four main procedures: A) policy-based routing, B) chunking large blobs, C) failure detection, and D) proxy requests.

# Chunks	ChunkId1		...	ChunkIdk	
	partitionId	UUID		partitionId	UUID

Figure 2.4: Content of the metadata blob used for chunked blobs.

A. Policy Based Routing: On receiving a request, the library decides which partition to involve (randomly chosen for puts and extracted from blob id for gets/deletes). Then, based on the policy used it communicates with the corresponding replica(s) until the request is served/failed. It supports a variety of policies including {one, k, majority, all}, as discussed in Section 2.2.3.

B. Chunking: Extremely large blobs (e.g., videos) create load imbalance, block smaller blobs, and inherently have high latency. To mitigate these issues, Ambry splits large blobs into smaller equal-size units called chunks. A large chunk size does not fully resolve the large blob challenges and a small chunk size adds too much overhead. Based on our current large blob size distribution, we found the sweet spot for the chunk size to be in the range of 4 to 8 MB[4].

During a put operation, a blob b is split into k chunks $\{c_1, c_2, ..., c_k\}$, each treated as an independent blob. Each chunk goes through the same steps as a normal put blob operation (Section 2.2.3), most likely being placed on a different partition. It is also assigned a unique chunk id with the same format as a blob id. In order to be able to retrieve b Ambry creates a metadata blob $b_{metadata}$ for b. $b_{metadata}$ stores the number of chunks and chunk ids in order, as shown in Figure 2.4. This metadata blob is then put into Ambry as a normal blob and the blob id of $b_{metadata}$ is returned to the user as the blob id of b. If the put fails before writing all chunks, the system will issue deletes for written chunks and the operation has to

[4] Chunk size is not fixed and can be adapted to follow the growth in blob sizes, improvements in network, etc.

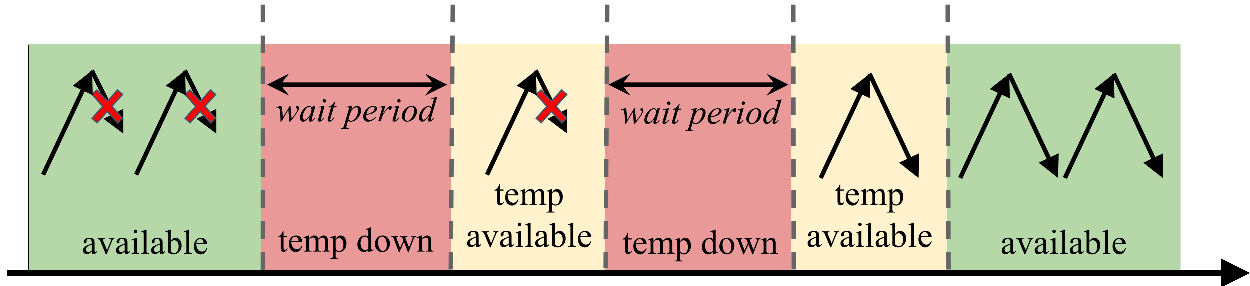


Figure 2.5: Failure detection algorithm with maximum tolerance of 2 consecutive failed responses.

be redone.

During a get, the metadata blob is retrieved and chunk ids are extracted from it. Then, Ambry uses a sliding buffer of size s to retrieve the blob. Ambry queries the first s chunks of the blob independently and in parallel (since they are most likely written on unique partitions placed on separate Datanodes). When the first chunk in the buffer is retrieved, Ambry slides the buffer to the next chunk, and so on. The whole blob starts being returned to the user the moment the first chunk of the blob is retrieved.

Although an extra put/get is needed in this chunking mechanism (for the metadata blob), overall, our approach improves latency since multiple chunks are written and retrieved in parallel.

C. Zero-cost Failure Detection: Failures happen frequently in a large system. They range from unresponsiveness and connection timeouts, to disk I/O problems. Thus, Ambry needs a failure detection mechanism to discover unavailable Datanodes/disks and avoid forwarding requests to them.

Ambry employs a zero-cost failure detection mechanism involving no extra messages, such as heartbeats and pings, by leveraging request messages. In practice, we found our failure detection mechanism is effective, simple, and consumes very little bandwidth. This mechanism is shown in Figure 2.5. In this approach, Ambry keeps track of the number of

consecutive failed requests for a particular Datanode (or disk) in the last *check_period* of time. If this number exceeds a MAX_FAIL threshold (in our example set to 2) the Datanode is marked as *temporarily_down* for a *wait_period* of time. In this state all queued requests for this Datanode will eventually time out and need to be reattempted by the user. After the *wait_period* has passed, the Datanode becomes *temporarily_available*. When a Datanode is in the *temporarily_available* phase, if the next request sent to that Datanode fails, it will move to the *temporarily_down* phase again. Otherwise, it will be marked as *available*, working as normal again.

D. Proxy Requests: As described in Section 2.2.3, Ambry uses proxy requests to reach higher availability and read-after-write consistency in remote datacenters. When a blob has not been replicated in the local datacenter yet, requests for that blob are forwarded to other datacenters and served there (proxy requests). However, datacenter partitions can cause unavailability of unreplicated data until the partition is healed and replicas converge.

Proxy requests are handled by the Router Library, transparently from the user issuing the request. In practice, we found proxy requests occur less than 0.001% of the time, thus minimally affecting the user experience.

2.4.4 Datanode Layer

Datanodes are in charge of maintaining the actual data. Each Datanode manages a number of disks, and responds to requests for partition replicas placed on its disks. Puts are handled by writing to the end of the partition file. Gets can be more time-consuming, especially since the location of the blob in the partition is not known. To minimize both read and write latencies, Datanodes employ a few techniques:

- *Indexing blobs:* Ambry stores an index of blob offsets per partition replica to prevent sequential sweeps for finding blobs.

- *Exploiting OS cache:* Ambry utilizes OS caching to serve most reads from the RAM, by limiting the RAM usage of other components.

- *Batched writes, with a single disk seek:* Ambry batches writes for a particular partition together and periodically flushes the writes to disk. Thus, it incurs at most one disk seek for a batch of sequential writes. The flush period is configurable and trades off latency for durability. Although, batching can introduce overheads of flushing, dirty buffers, and tuning, the benefits outweigh these overheads.

- *Keeping all file handles open:* Since partitions are typically very large (100 GB in our setting), the number of partition replicas placed on a Datanode is small (a few hundred). Thus, Ambry keeps all file handles open at all times.

- *Zero copy gets:* When reading a blob, Ambry utilizes a zero copy [57] mechanism, i.e., the kernel directly copies data from disk to the network buffer without going through the application. This is feasible since the Datanodes do not perform any computation on the data at get operations. Note that data read from disk is usually old and less popular. A significant advantage of zero copy is that it also helps toward better utilizing the OS cache by not copying less popular data into it.

These few simple techniques have significantly improved the latency and throughput in Ambry. As we show in 2.6.1, we are able to reach 75%-88% of the maximum network bandwidth by using these techniques. Below, we discuss two of these techniques (indexing and OS Caches) in more detail.

Indexing

To find the location of a blob in a partition replica with low latency, the Datanode maintains a light-weight in-memory indexing per replica, as shown in Figure 2.6. The indexing is sorted by blob id, mapping the blob id to the start offset of the blob entry. The indexing is updated in an online fashion whenever blobs are put (e.g., blob 60) or deleted (e.g., blob 20).

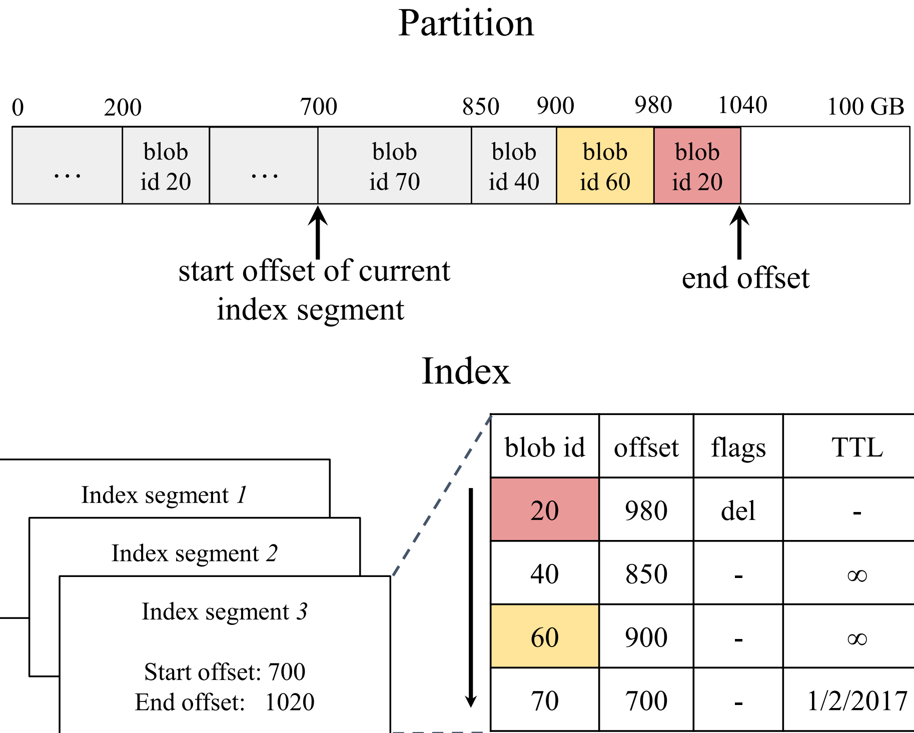


Figure 2.6: Indexing of blob offsets in a partition replica. When blobs are put (blob 60) or deleted (blob 20) the indexing stucture is updated.

Similar to SSTables [28], Ambry limits the size of the index by splitting it into segments, storing old segments on disk, and maintaining a Bloom filter for each on-disk segment.

The indexing also stores a flag indicating if a blob has been deleted and optionally a time-to-live (TTL). During get operations, if the blob has expired or been deleted, an error will be returned before reading the actual data.

Note that the indexing does not contain any additional information affecting the correctness of the system, and just improves performance. If a Datanode fails, the whole indexing can be reconstructed from the partition.

Exploiting The OS Cache

Recently written data, which is the popular data as well, is automatically cached without any extra overhead (by the operating system). By exploiting this feature, many reads can be served from memory, which significantly improves performance. Thus, Ambry limits

the memory usage of other data structures in the Datanode. Ambry bounds the indexing by splitting it into segments, with only the latest segment remaining in-memory (Figure 2.6). New entries are added to the in-memory segment of the indexing. When the in-memory segment exceeds a maximum size it is flushed to disk as a read-only segment. This design also helps toward failure recovery since only the in-memory segment needs to be reconstructed. Looking up blob offsets is done in reverse chronological order, starting with the latest segment (in-memory segment). Thus, a delete entry will be found before a put entry when reading a blob. This ensures deleted blobs are not retrievable.

Bloom Filters: To reduce lookup latency for on-disk segments, Ambry maintains an in-memory Bloom filter for each segment, containing the blob ids in that index segment. Using Bloom filters, Ambry quickly filters out which on-disk segment to load. Thus, with high probability, it incurs only one disk seek. However, due to our skewed workload a majority of reads just hit the in-memory segment, without any disk seeks.

2.5 REPLICATION

Replicas belonging to the same partition can become out of sync due to either failures, or asynchronous writes that write to only one datacenter. In order to fix inconsistent replicas, Ambry uses an asynchronous replication algorithm that periodically synchronizes replicas. This algorithm is completely decentralized. In this procedure each replica individually acts as a master and syncs-up with other replicas, in an all-to-all fashion. Synchronization is done using an asynchronous two-phase replication protocol as follows. This protocol is a pull-based approach where each replica independently requests for missing blobs from other replicas.

- **First Phase:** This phase finds missing blobs since the last synchronization point. This is done by requesting blob ids of all blobs written since the latest syncing offset and then filtering the ones missing locally.

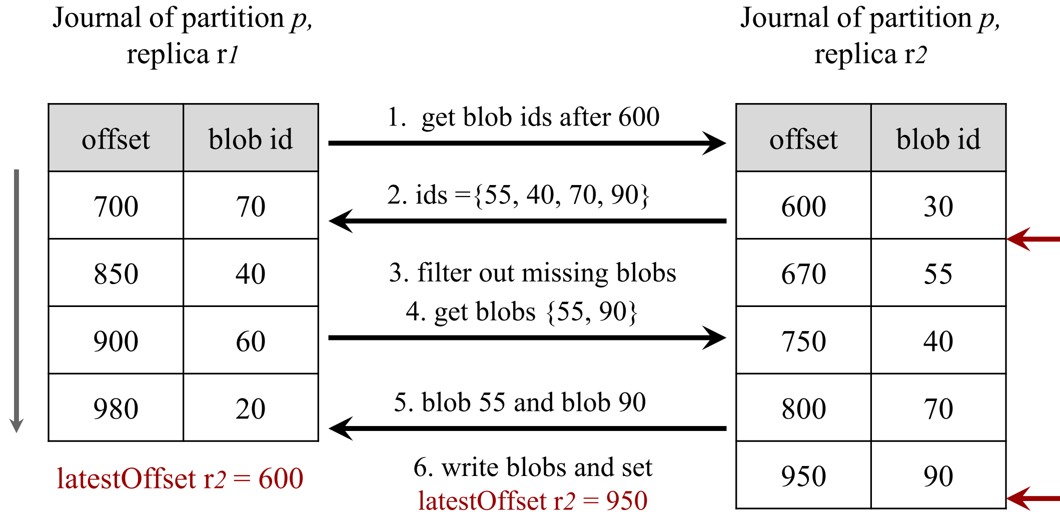


Figure 2.7: Journals for two replicas of the same partition and an example of the replication algorithm.

- **Second Phase:** This phase replicates missing blobs. A request for only the missing blobs is sent. Then, the missing blobs are transferred and appended to the replica.

In order to find the recently written blobs quickly, the replication algorithm maintains an additional data structure per replica, called a *journal*. The journal is an in-memory cache of recent blobs ordered by their offset. The journal is the inverse of the Indexing table. It can promptly return which new blobs have been added since a given offset.

Figure 2.7 shows example journals of two replicas (r_1 and r_2) and the two phase replication procedure for r_1 syncing with r_2 from latestOffset 600. In the first phase, r_1 requests all recently added blobs in r_2 after latestOffset; using the journal r_2 returns a list B ={55, 40, 70, 90} of blob ids; and r_1 filters out the missing blobs (blob 55 and 90). In the second phase, r_1 receives the missing blobs, appends the blobs to the end of the replica, and updates the journal, indexing and latestOffset.

This replication protocol is periodically used in an all-to-all fashion between each pair of replicas. The overall replication procedure is shown in the algorithm below. Since it is desirable to minimize cross-datacenter traffic, replication algorithm is used separately for in intra- and inter-datacenter with different periods. Inter-datacenter has a higher period

so that, with high probability, only one replica receives data from another datacenter and replicates the data inside the datacenter, instead of multiple cross datacenter replications.

Algorithm 2.2 Replication Algorithm

```
1: for each partition p replica r_1 do
2:     for each remote replica r_2 of p do
3:         BlobIds = get_blob_IDs (latestOffset[r_2])
4:         MissingIds = find_missing_blobs(BlobIds)
5:         Blobs=get_blobs(MissingIds)
6:         writeBlobs(Blobs)
7:         update_indexing()
8:         update_journal()
9:         update_last_offset(r_2)
```

To provide improved efficiency and scalability of the system, the replication algorithm employs a number of further optimizations:

- Using separate thread pools for inter- and intra-datacenter replication with different periods.
- Batching requests between common partition replicas of two Datanodes, and batching blobs transferred across datacenters.
- Batching blobs transfered across datacenters
- Prioritizing lagging replicas to catch up at a faster rate (by using dedicated threads for lagging replicas).

2.6 EXPERIMENTAL RESULTS

We perform three kinds of experiments: small cluster (Section 2.6.1), production cluster (Section 2.6.2), and simulations (Section 2.6.3). We divide our evaluation into three parts, each focusing on one of the main goals in Ambry. First we evaluate the latency and throughput of the system. Second, we show the effectiveness of our multi-colo optimizations and last we analyze the load-balancing mechanism over time.

2.6.1 Throughput and Latency

In this section we measure the latency and throughput of the system using a micro-benchmark that stress-tests the system, under read-only, write-only and read-write workloads.

Micro-Benchmark: We first measure the peak throughput. We designed a micro-benchmark that linearly adds load to the system (by adding more clients), until the saturation point where no more requests can be handled. Each client sends requests one at a time with the next request sent right after a response.

This benchmark has three modes: Write, Read, and ReadWrite. In Write mode, random byte array blobs are being put by varying number of clients. In Read mode, first blobs are written at saturation rate for a *write-period* of time. Then, randomly chosen blobs are read from the written blobs[5]. In most experiments we set the write-period long enough that most read requests ($> 80\%$) are served by disk, rather than RAM. Read-Write is a similar to Read, but serving 50% reads and 50% writes after the write-period. Since latency and throughput are highly correlated with blob size, we use fixed-size blobs in each run, but vary the blob size.

Experimental Setup: We deployed Ambry with a single Datanode. The Datanode was running on a 24 core CPU with 64 GB of RAM, 14 1TB HDD disks, and a full-duplex 1 Gb/s Ethernet network. 4 GB of the RAM was set aside for the Datanode's internal use and the rest was left to be used as Linux Cache. We created 8 single-replica 100 GB partitions on each disk, with a total of 122 partitions. Using 14 disks with a 1 Gb/s network might look like an overkill. However, disk seeks are the dominant latency factor for small blobs. Since a large portion of blobs are small (< 50 KB), we need the parallelism created by using multiple disks. Note that Ambry is designed as a cost-effective system using cheap HDD disks.

[5] The random strategy gives a lower bound on the system's performance since real-world access patterns are skewed toward recent data.

Clients send requests from a few machines located in the same datacenter as the Datanode. These clients, that are acting as Frontends, directly send requests to the Datanode. The micro-benchmark discussed above was used with varying blob sizes {25 KB, 50 KB, 100 KB, 250 KB, 500 KB, 1 MB, 5 MB}. We did not go above 5 MB since blobs are chunked beyond that point.

Effect of Number of Clients

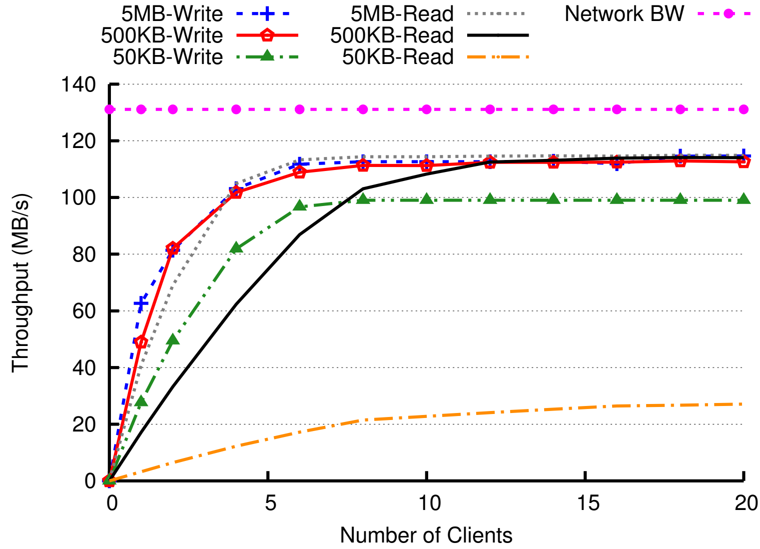
We ran the micro-benchmark with varying blob sizes, while linearly increasing the number of clients. For Read mode, the write-period was set such that 6 times the RAM size, was first written. Figure 2.8a shows the throughput in terms of MB/s served by the system. Adding clients proportionally increases the throughput until the saturation point. Saturation occurs at 75%-88% of the network bandwidth. The only exception is reading small blobs due to frequent disk seeks (more details in following sections). Saturation is reached quickly (usually ≤ 6 clients) since clients send requests as fast as possible in the benchmark.

Figure 2.8b shows the latency normalized by blob size (i.e., average latency divided by blob size). Latency stays almost constant before reaching saturation point, and then increases linearly beyond the throughput saturation point. The linear increase in latency after saturation indicates the system does not add additional overhead beyond request serving.

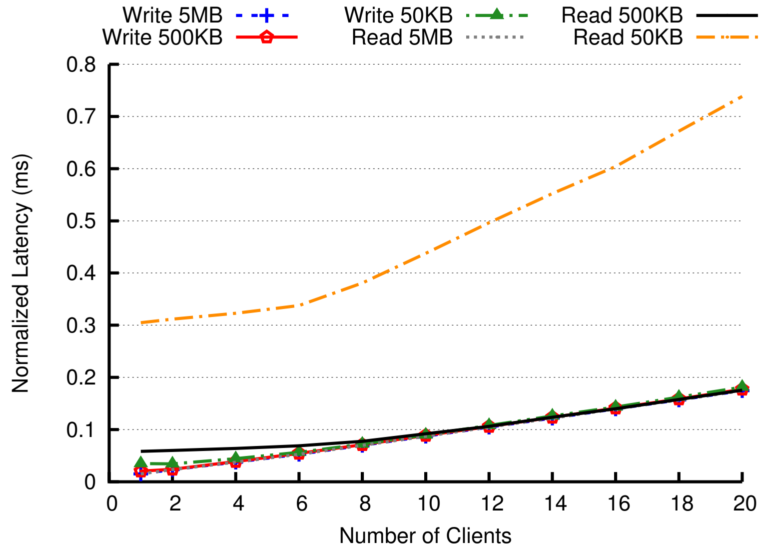
Effect of Blob Size

In Figures 2.9a and 2.9b we analyzed the maximum throughput (with 20 clients) under different blob sizes and workloads. For large objects (> 200 KB), the maximum throughput (in MB/s) stays constant and close to maximum network bandwidth across all blob sizes. Similarly, throughput in terms of requests/s scales proportionally.

However, for Read and Read-Write, the read throughput in terms of MB/s drops linearly for smaller sizes. This drop is because our micro-benchmark reads blobs randomly, incurring frequent disk seeks. The effect of disk seeks is amplified for smaller blobs. By further

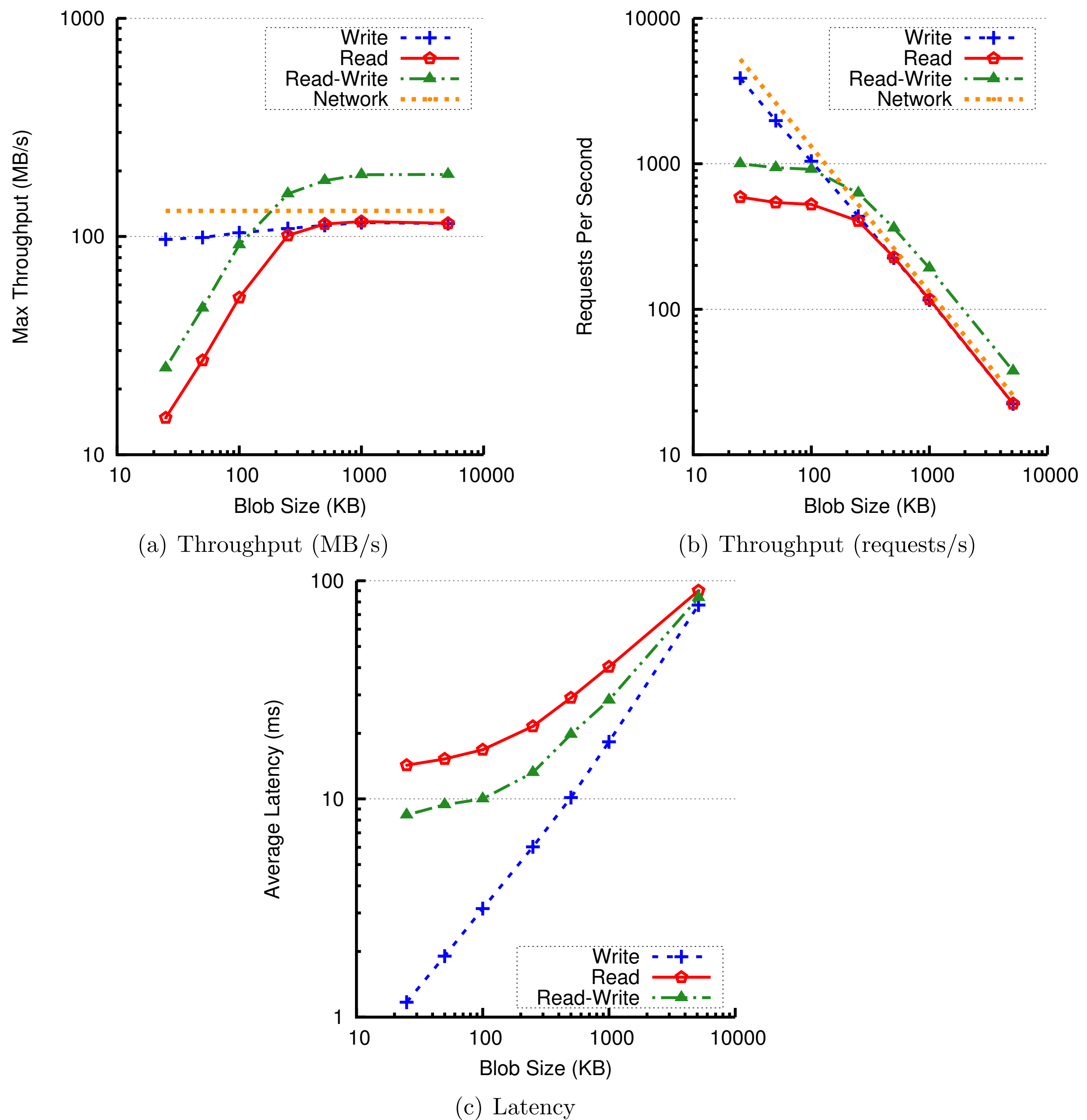


(a) Throughput



(b) Latency normalized by blob size

Figure 2.8: Throughput and latency of read and write requests with varying number of clients on different blob sizes. These results were gathered on a single Datanode deployment and across various blob sizes {50 KB, 500 KB, 5 MB}



(a) Throughput (MB/s)

(b) Throughput (requests/s)

(c) Latency

Figure 2.9: Effect of blob size on maximum throughput, both in terms of MB/s and requests/s, and latency. Results were gathered on a write-only, read-only, and mixed (50%-50%) workload. Reads for small blob sizes (<200 KB) are slowed down by frequent disk seeks, while other requests saturate the network link.

profiling the disk using Bonnie++ [58] (an IO benchmark for measuring disk performance), we confirmed that disk seeks are the dominant source of latency for small blobs. For example, when reading a 50 KB blob, more than 94% of latency is due to disk seek (6.49 ms for disk seek, and 0.4 ms for reading the data).

Read and Write workload mostly utilize only the outbound and inbound link on the Datanode, respectively. However, Read-Write workload has a better mix and evenly utilizes both links reaching higher throughput. Therefore, in our full-duplex network infrastructure the Read-Write mode is able to saturate both links reaching almost double the network bandwidth ($\simeq$ 1.7 Gb/s in total out of the 2 Gb/s available). For smaller size objects it reaches twice the Read throughput, since Read-Write is a 50-50 workload with reads being the limiting factor.

Figure 2.9c demonstrates the trend in latency under various blob sizes. These results are before the saturation point with 2 clients. Similar to throughput, latency grows linearly except for reading small blobs. The higher latency in Read is because most read requests incur a disk seek, while write requests are batched and written together. The Read-Write latency falls halfway between Read and Write latency because of its mixed workload.

Variance in Latency

The tail and variance in request latencies are important. Figure 2.10 shows the CDFs of Write, Read, and Read-Write mode experiments with 2 clients. The CDF of Read and Write mode is very close to a vertical line, with a short tail and a majority of values close to the median. The jump around 0.15 in Read mode (for 50 KB blob size) is because a small fraction of requests are served using the Linux cache which is orders of magnitudes faster than disk (discussed more in following Section). The Read-Write mode is a mixture of the Read and Write CDF with a change around 0.5, following the 50% read - 50% write workload pattern.

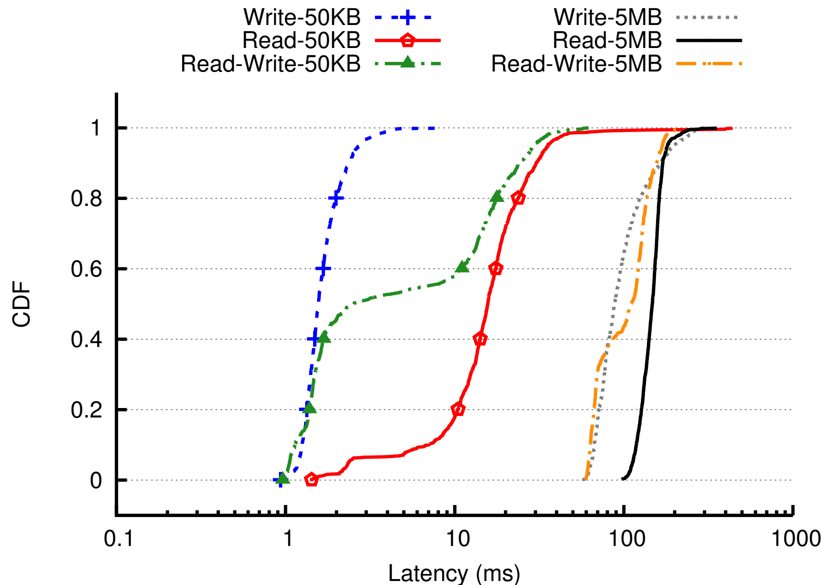


Figure 2.10: CDF of read and write latency with 2 clients for various blob sizes. Read-Write falls in the middle with change around 0.5 due to the 50-50 mixed workload.

Table 2.3: Comparision of get request latency when most of requests (83%) are served by disk (Disk Reads) and when all requests are served by linux cache (Cached Reads) for 50 KB blobs.

	Average	**Max**	**Min**	**StdDev**
Disk Reads	17 ms	67 ms	1.6 ms	9 ms
Cached Reads	3 ms	5 ms	1.6 ms	0.4 ms
Improvement	**5.5x**	13x	0	23x

Effect of Linux Cache

We ran the micro-benchmark on 50 KB blobs and 2 clients in two configurations: 1) writing 6 times more than the RAM size before reading, so that most requests (83 %) are served by disk (Disk Read), and 2) writing data equal to the RAM size to keep all data in RAM (Cached Read). Table 2.3 compares these two modes.

The Cached Read experiment performed more than 2100 requests/s (104 MB/s reaching 79% network bandwidth) matching the maximum write throughput, compared to 540 requests/s for Disk Reads. We also measured the average, maximum, minimum, and standard deviation of latency, shown in Table 2.3. In both cases, the minimum is equal to reading from

the Linux Cache. However, the Cached Read case improves the average and max latency by 5.5x and 13x, respectively. This shows the significance of exploiting the Linux Cache (Section 2.4.4).

2.6.2 Geo-distributed Optimizations

Ambry is designed to work across multiple datacenters geographically distributed. We analyzed our replication algorithm among 3 different datacenters at LinkedIn {DC1, DC2, DC3}, located all across the US. All experiments in this section are from production workloads. Since Ambry treats each datacenter independently, except for the background replication that is cross-datacenter. This section focuses on the replication, where we evaluate multiple aspects including the lag among replicas, the time to replicate, and bandwidth used for replication.

Replication Lag

We define *replication lag* between a pair of replicas (r_1, r_2) as the difference of r_2's highest used offset and r_1's latest synced offset with r_2. Note that not all data in the replication lag needs to be transfered to r_1, since it could receive the missing blobs from other replicas as well.

We measured the replication lag among all replicas of a given Datanode and the rest of the cluster, and found that more than 85% of the values were 0. Figure 2.11 shows the CDF of non-zero values grouped by datacenter. The 95th percentile is less than 1 KB for 100 GB partitions (in all datacenters), with slightly worse lag in datacenter 3 since it is placed farther apart from others.

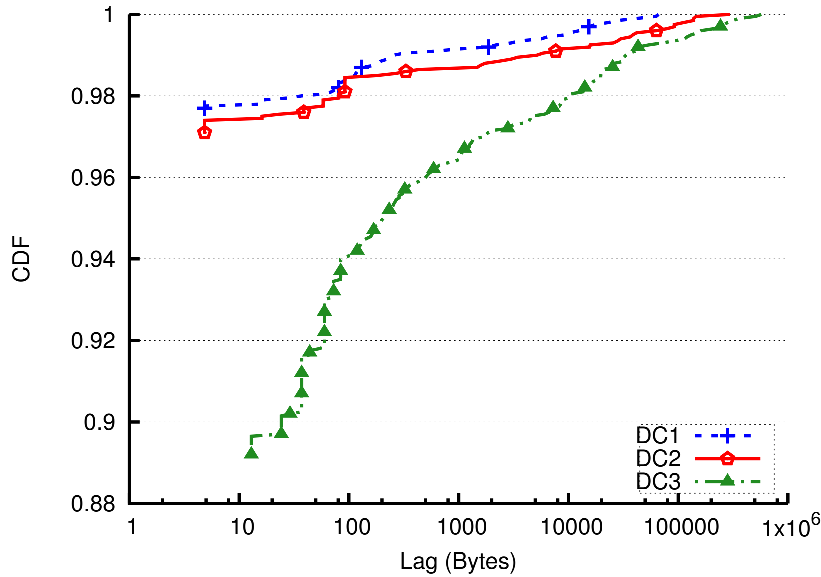


Figure 2.11: CDF of Replication Lag among replica pairs of a given Datanode and the rest of the cluster. Most values were zero, and are omitted from the graph.

Replication Bandwidth

Ambry relies on background replication to write data to other datacenters. We measured the aggregate network bandwidth used for inter-datacenter replication during a 24 hour period, shown in Figure 2.12. The aggregate bandwidth is small (< 10 MB/s), similar across all datacenters, and correlated to request rates with a diurnal pattern. This value is small because we batch replication between common replicas and due to the read-heavy nature of the workload.

Figure 2.13 demonstrates the CDF of average replication bandwidth per Datanode, for both intra- and inter-datacenter replication. Intra-datacenter bandwidth is minimal (< 200 B/s at 95th percentile), especially compared to inter-datacenter with 150-200 KB/s at 95th percentile (1000x larger). The higher value for inter-datacenter is because of asynchronous writes. However, the inter-datacenter bandwidth is still negligible ($\sim$ 0.2% of a 1 Gb/s link). The small difference among the 3 datacenters is due to the different request rates they receive.

Figure 2.14 shows a zoomed in graph of only inter-datacenter bandwidth. Inter-datacenter

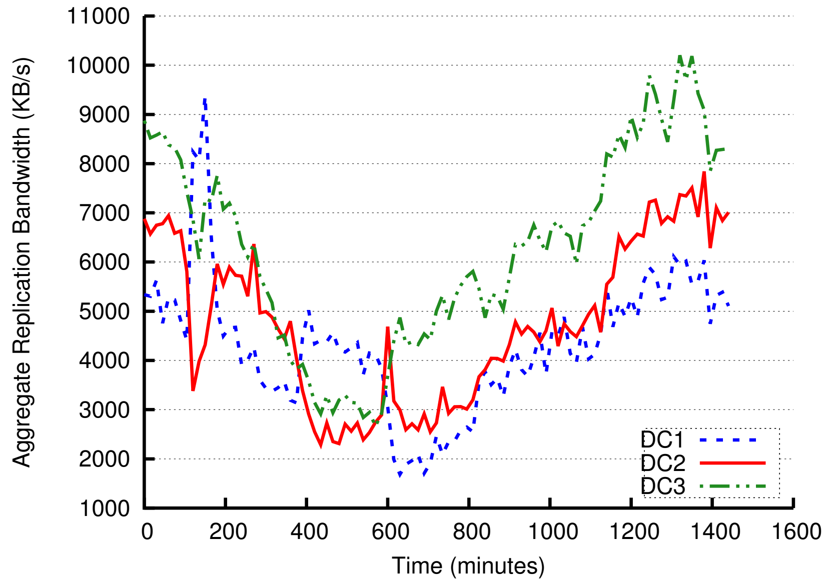


Figure 2.12: Aggregate network bandwidth used for inter-datacenter replication during a 24 hour period in production.

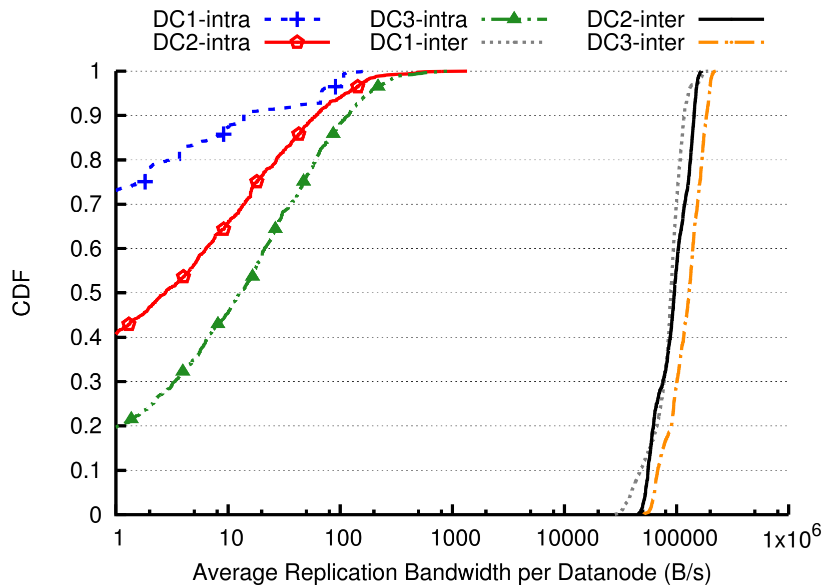


Figure 2.13: CDF of average network bandwidth used per Datanode for intra- and inter-datacenter replication over a 24 hour period in production.

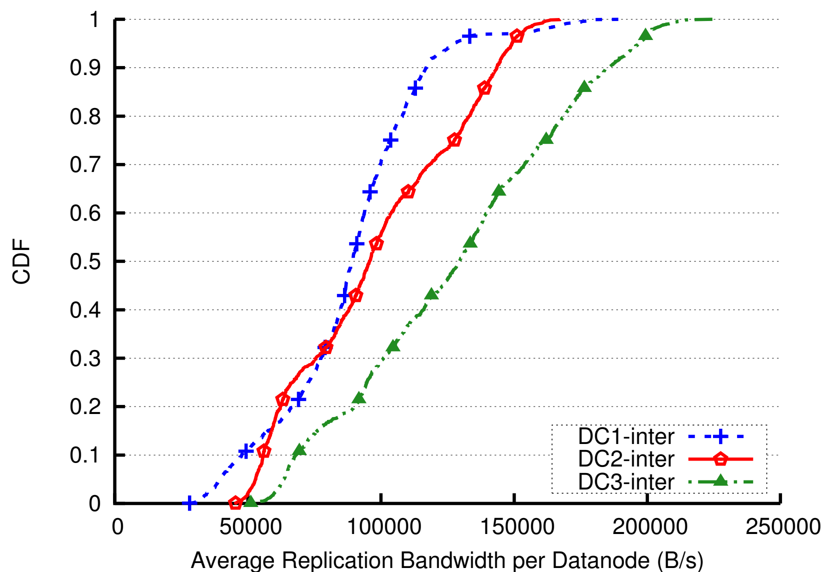


Figure 2.14: CDF of average inter-datacenter network bandwidth used per Datanode over a 24 hour period in production.

replication has a short tail with the 95th to 5th percentile ratio of about 3x. This short tail is because of the load-balanced design in Ambry. Intra-datacenter replication has a longer tail, as well as many zero values (omitted from the graph). Table 2.4 shows the exact values of the 95th and 5th percentiles and their absolute difference and ratio. Replication either uses almost zero bandwidth (intra-datacenter) or almost balanced bandwidth (inter-datacenter).

Table 2.4: Comparirsion of the 95th and 5th percentile of average replication bandwidth used for both intra- and inter-datacenter, in three datacenters

	95th	**5th**	**95th-5th**	**95th ÷ 5th**
Intra-DC1	68	0	68	∞
Intra-DC2	113	0	113	∞
Intra-DC3	178	0.007	178	27K
Inter-DC1	128K	40K	88K	3.2
Inter-DC2	150K	53K	97K	2.8
Inter-DC3	197K	63K	134K	3.1

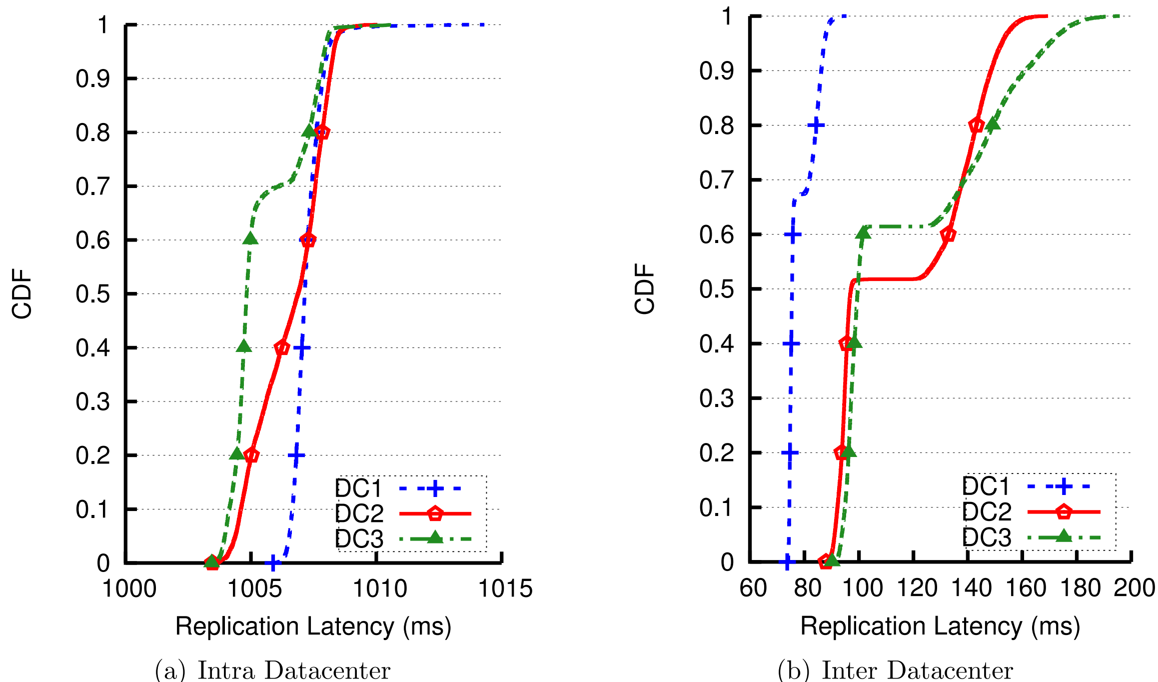


Figure 2.15: CDF of average replication latency (i.e., time spent to recieve missing blobs) for intra- and inter-datacenter in production environment.

Replication Latency

We define *replication latency* as the time spent in one iteration of the replication protocol, i.e., $T_{missing_blobs_received}$ minus $T_{replication_initiated}$. Figure 2.15 demonstrates the CDF of average replication latency for intra- and inter-datacenter, in our production workload.

Inter-datacenter replication latency is low with a median of less than 150 ms, and a very short tail. Although this latency might appear to be high, the number of proxy requests remain near-zero ($< 0.001\%$). This is because users usually read data from the same local datacenter to which they have recently written. Therefore, replication has a minimal effect on user experience.

Surprisingly, intra-datacenter replication latency is relatively high (6x more than inter-datacenter) and with little variance. This pattern is because of a pre-existing and pre-fixed artificial added delay of 1 second, intended to prevent incorrect blob collision detections. If a blob is replicated in a datacenter faster than the Datanode receives the initial put

request (which is possible with less strict policies), the Datanode might consider the put as a blob collision and fail the request. The artificial delay is used to prevent this issue in intra-datacenter replication. This relatively small delay does not have a significant impact on durability or availability of data since intra-replication is only used to fix failed/slow Datanodes, which rarely occurs.

2.6.3 Load Balancing

Since cluster growth occurs infrequently (every few months at most), we implemented a simulator to show the behavior of Ambry over a long period of time (several years), and at large scale (hundreds of Datanodes). We tried a workload that is based on production workloads. All results in this section are gathered using the simulator.

Simulator Design

The simulator's design resembles Ambry's architecture with all its components such as the Frontends, Data Nodes, Partitions and etc. Requests are also served using the same path. However, there are no real physical disks. For handling requests, only the metadata (blob id and size) is stored/retrieved, and the effect of requests are reflected (e.g., disk space increase).

Workload: We use a synthetic workload closely resembling the real-world traffic at LinkedIn. This workload preserves the rate of each type of request (read, write, and delete), the blob size distribution, and the access pattern of blobs (based on age).

Cluster Expansion: The simulator starts with an initial set of Datanodes, disks, and partitions in one datacenter. Over time, when partitions reach the capacity threshold, a new batch of partitions are added using the replica placement strategy from Section 2.2.2. If partitions cannot be added (e.g., if there is not enough unallocated space on disks), a batch of new Datanodes are added.

Experiment Setup

The simulation is run in a single datacenter, with 10 Frontend nodes. The experiment starts with 15 Datanodes, each with 10 4TB disks, and 1200 100GB partitions with 3 replicas. At each partition and Datanode addition point, a batch of 600 partitions and 15 Datanodes are added, respectively. The simulation is run over 400 weeks (almost 8 years) and up to 240 Datanodes. The simulation is run with and without rebalancing with the exact same configuration, while measuring request rates, disk usage, and data movement.

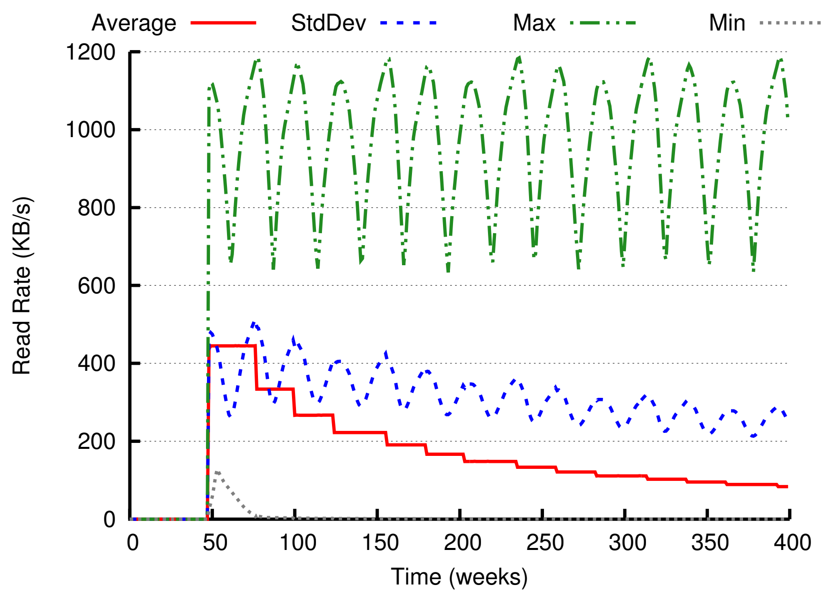
Request Rate

We measured the read rate (KB/s) for each disk at any point of time. Figure 2.16 demonstrates the average, standard deviation, maximum and minimum among these values, for the system with and without rebalancing. The results for write rates were similar and removed due to lack of space.

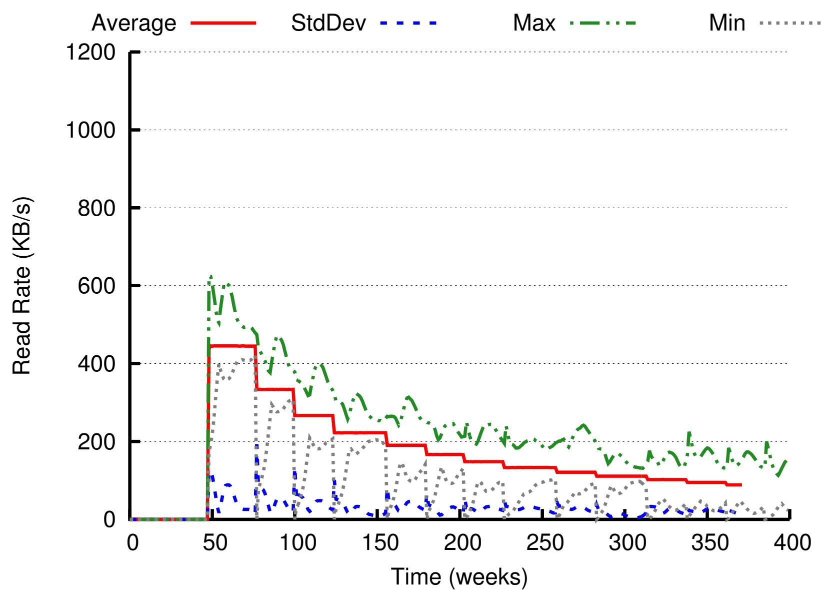
The average, which is also the ideal, is a dropping step function. The drops are points where new Datanodes were added to the cluster. In case of no rebalancing, a majority of the disks are old read-only disks with almost zero traffic, while a few disks receive most of the request. Thus, the minimum is close to zero. Also, the maximum and standard deviation are significant (3x-7x and 1x-2x larger than the average, respectively). When rebalancing is added, the minimum and maximum move close to the average, and the standard deviation drops close to zero. We conclude that Ambry's load balancing is effective.

Disk Usage

We analyzed the disk usage ratio, i.e., used space divided by total space among disks, with and without rebalancing. As seen in Figure 2.17, without rebalancing, the maximum stays at the capacity threshold since some disks become and remain full, while the minimum drops to zero whenever new Datanodes are added. With rebalancing, the maximum and



(a) Without rebalancing



(b) With rebalancing

Figure 2.16: Average, standard deviation, maximum and minimum of average read rate (KB/s) among disks over a 400 week interval. The system is bootstrapping in the first few weeks, and the results are omitted.

Table 2.5: Improvement of range (max-min) and standard deviation of request rates and disk usage over a 400 week interval. Results are from the system with rebalancing (w/ RB) and without rebalancing (w/o RB).

Integral over 400 weeks	**Write Avg**	**Read Avg**	**Disk Usage**
Range w/o RB	63,000	340,000	200
Range w/ RB	8,500	52,000	22
Improvement	7.5x	6x	9x
StdDev w/o RB	21,00	112,000	67
StdDev w/ RB	2500	11,000	6.7
Improvement	8x	10x	10x

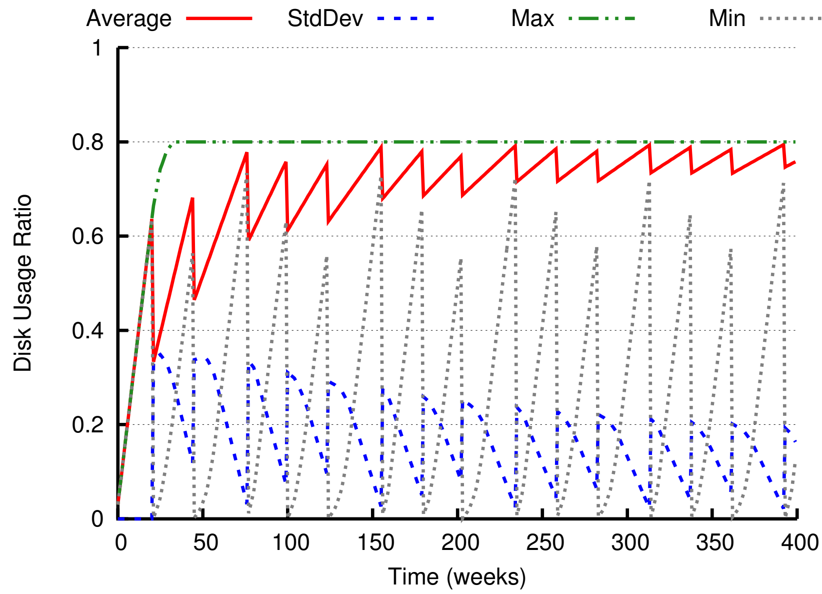
minimum move closer to the average with temporary drops in the minimum until rebalancing is completed. Additionally, the standard deviation drops significantly, becoming almost zero with temporary spikes on Datanode addition points.

Evaluation Over Time

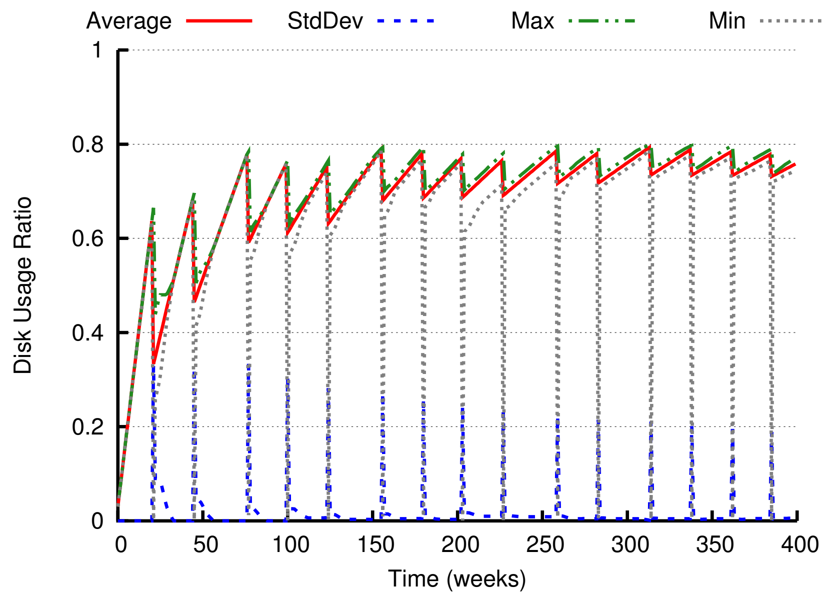
We evaluated the improvement over time by measuring the integral of range (max-min) and standard deviation for request rate and disk usage over the 400 week interval. As shown in Table 2.5, rebalancing has a prominent effect improving the range and standard deviation by 6x-9x and 8x-10x, respectively.

Data Movement

At points where rebalancing is triggered, we measure the minimum data movement needed to reach an ideal state and the data movement caused by rebalancing. We calculate the minimum data movement by adding the difference between ideal and current disk usage among all disks above ideal disk usage. This value is a lower bound on the minimum data movement since data is moved in granularity of partitions, and reaching this ideal case might not be feasible. As shown in Figure 2.18, the data movement of rebalancing is very close and always below the minimum. This is because the rebalancing algorithms trades off



(a) Without rebalancing



(b) With rebalancing

Figure 2.17: Average, standard deviation, maximum and minimum of disk usage ratio (i.e., used space divided by total space) over a 400 week interval.

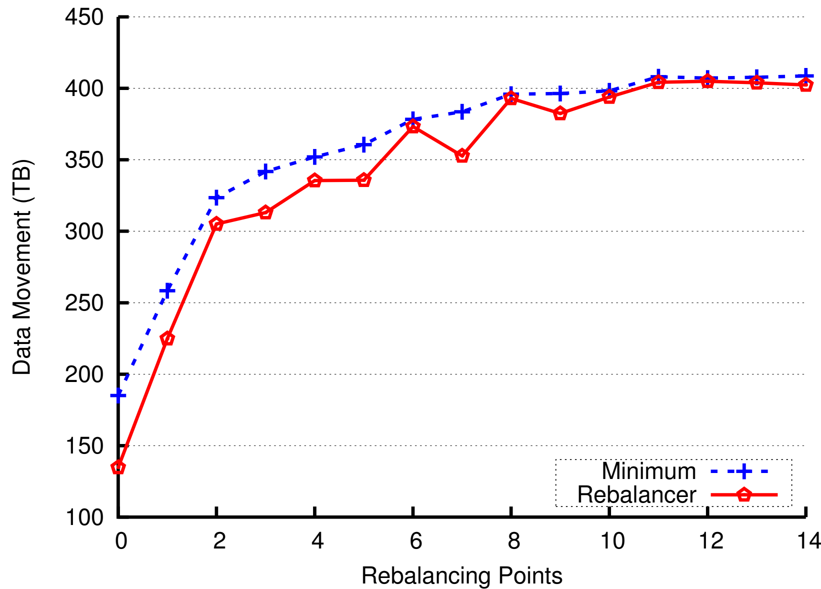


Figure 2.18: Data movement of the rebalancing algorithm at each rebalancing point (i.e., whenever new Datanodes are added) over a 400 week interval.

perfect balance (ideal state) for less data movement. Specifically, the rebalancing algorithm usually does not remove (or add) a partition from a disk if it would go below (or above) the ideal state, even if this were to cause slight imbalance.

2.7 RELATED WORK

2.7.1 File Systems

The design of Ambry is inspired by log-structure file systems (LFS) [59, 60]. These file systems are optimized for write throughput by sequentially writing in log-like data structures and relying on the OS cache for reads. Although these single machine file systems suffer from fragmentation issues and cleaning overhead, the core ideas are very relevant, especially since blobs are immutable. The main differences are the skewed data access pattern in our workload and additional optimization such as segmented indexing and Bloom filters used in Ambry.

There has been work on handling metadata and small files more efficiently. Some of

these techniques include reducing disk seeks [61], using a combination of log-structured file systems (for metadata and small data) and fast file systems (for large data) [62], and storing the initial segment of data in the index block [63]. Our system resolves this issue by using in-memory segmented indexing plus Bloom filters and batching techniques.

2.7.2 Distributed File Systems

Due to the extremely large amount of data and data sharing needs, many distributed file systems such as NFS [44] and AFS [46], and even more reliable ones handling failures, such as GFS, HDFS, and Ceph [43, 45, 47] have emerged. However, all these systems suffer from the high metadata overhead and additional capabilities (e.g., nested directories, permissions, etc.) unnecessary for a simple blob store. In many of these systems (e.g., HDFS, GFS, and NFS) the metadata overhead is magnified by having a separate single metadata server. This server adds an extra hub in each request, becomes a single point of failure, and limits scalability beyond a point. Recent research has addressed this problem by either distributing the metadata [47] or caching it [64]. Although these systems alleviate accessing metadata, each small object still has a large metadata (usually stored on disk), decreasing the effective throughput of the system.

2.7.3 Distributed Data Stores

Many key-value stores, such as [24,25,28,50], have been designed to handle a large number of requests per second. However, these systems cannot handle massively large objects (tens of MBs to GBs) efficiently, and add unnecessary overhead to provide consistency. Also, some systems [24, 25, 50] hash data to machines, creating large data movement whenever nodes are added/deleted.

PNUTS [30] and Spanner [29] are scalable geographically distributed systems, where PNUTS maintains load balance as well. However, both systems provide more features and stronger guarantees than needed in a simple immutable blob store.

Table 2.6: Placement and data-motion optimization in Ambry

Requirements	**Placement**	**Data Motion**
- geo-distributed across multiple datacenters - low-latency upload and view - persistent data - continuous scaling	- partition abstraction for placement - random placement of objects - mix of many objects in one partition - adaptive background load balancing	- CDN cache and OS cache - indexing & bloom filters - chunking - async replication with journaling - batching writes - zero copy reads - active-active (master free) design - zero-cost failure detection

2.7.4 Blob Stores

A similar concept to partitions in Ambry has been used in other systems. Haystack uses logical volumes [48], Twitter's blob store uses virtual buckets [52], and Petal file system introduces virtual disks [65]. Ambry is amenable to some optimizations in these systems such as the additional internal caching in Haystack. However, neither Haystack nor Twitter's blob store tackle the problem of load-imbalance. Additionally, Haystack uses synchronous writes across all replicas impacting efficiency in a geo-distributed setting.

Facebook has also designed f4 [51], a blob store using erasure coding to reduce replication factor of old data (that has become cold). Despite the novel ideas in this system, which potentially can be included in Ambry, our main focus is on both new and old data. Oracle's Database [66] and Windows Azure Storage (WAS) [67] also store mutable blobs, and WAS is even optimized for a geo-distributed environment. However, they both provide additional functionalities such as support for many data types other than blobs, strong consistency guarantees, and modification to blobs, that are not needed in our use case.

2.8 CONCLUSION

This chapter described Ambry, a distributed storage system designed specifically for storing large immutable media objects, called blobs. We designed Ambry to serve requests in a geographically distributed environment of multiple datacenters while maintaining low la-

tency and high throughput. Using a decentralized design, rebalancing mechanism, chunking, and logical blob grouping, we provide load balancing and horizontal scalability to meet the rapid growth at LinkedIn. Table 2.6 summarizes the requirements and techniques used in Ambry, and how they are all focused around placement and data-motion.

As part of future work we plan to adaptively change the replication factor of data based on the popularity, and use erasure coding mechanisms for cold data. We also plan to investigate using compression mechanisms and its costs and benefits. Additionally, we are working on reducing the replication footprint using location aware mechanism.

CHAPTER 3: SAMZA: STATEFUL STREAM PROCESSING AT SCALE

In this chapter, we present Apache Samza, a distributed system for stateful and fault-tolerant stream processing. Distributed stream processing systems need to support *stateful processing*, *recover quickly from failures* to resume such processing, and *reprocess* an entire data stream quickly. Samza utilizes a partitioned local state along with a low-overhead background changelog mechanism to minimize data-motion, allowing it to scale to massive state sizes (hundreds of TB) per application. Recovery from failures is sped up by re-scheduling the placement based on Host Affinity. In addition to processing infinite streams of events, Samza supports processing a finite dataset as a stream, from either a streaming source (e.g., Kafka), a database snapshot (e.g., Databus), or a file system (e.g. HDFS), without having to change the application code (unlike the popular Lambda-based architectures which necessitate maintenance of separate code bases for batch and stream path processing).

Samza is currently in use at LinkedIn by hundreds of production applications with more than $10,000$ containers. Samza is an open-source Apache project adopted by many top-tier companies (e.g., LinkedIn, Uber, Netflix, TripAdvisor, etc.). Our experiments show that Samza: a) handles state efficiently, improving latency and throughput by more than $100\times$ compared to using a remote storage; b) provides recovery time independent of state size; c) scales performance linearly with number of containers; and d) supports reprocessing of the data stream quickly and with minimal interference on real-time traffic.

3.1 INTRODUCTION

Many modern applications require processing large amount of data in a real-time fashion. We expect our websites and mobile apps to be deeply interactive and show us content based on users' most recent activities. We expect social networks to show us current global and local hashtag trends within seconds, ad campaigns to orient ads based on current user activity, and data from IoT (Internet of Things) to be processed within minutes.

Processing these streams of data in a real-time fashion poses some unique challenges. First, at LinkedIn, as a global social network company, trillions of events are fed to our production messaging system (Apache Kafka) and change capture system (Databus) per day. To process this massive amount of data, we need to be able to use resources efficiently and at scale, and to handle failures gracefully. Second, it is common for applications to access and store additional *stateful* data while processing each received event. At LinkedIn, examples of state include (depending on the application): user profiles, email digests, aggregate counts, etc. State computations include aggregations/counts over a window, joining a stream with a database, etc. Thus, we need mechanisms to: i) handle such state efficiently while maintaining performance (high throughput and low latency), and ii) recover quickly after a failure in spite of large state [68].

Third, it is common to require a whole database or the full received stream to be *reprocessed* completely. Such reprocessing is triggered by reasons ranging from software bugs to changes in business logic. This is one of the primary reasons why many companies employ the Lambda architecture. In a Lambda architecture [69], a streaming framework is used to process real-time events, and in a parallel "fork", a batch framework (e.g., Hadoop/Spark [38,39,70]) is deployed to process the entire dataset (perhaps periodically). Results from the parallel pipelines are then merged. However, implementing and maintaining two separate frameworks is hard and error-prone. The logic in each fork evolves over time, and keeping them in sync involves duplicated and complex manual effort, often with different languages.

Today, there are many popular distributed stream processing systems including Storm, MillWheel, Heron, Flink [33,35,71,72], etc. These systems either do not support reliable state (Storm, Heron, S4 [71–73]), or they rely on remote storage (e.g., Millwheel, Trident, Dataflow [35, 37, 74]) to store state. Using external (remote) storage increases latency, consumes resources, and can overwhelm the remote storage. A few systems (Flink, Spark [33,36,75]) try to overcome this issue by using partitioned local stores, along with periodically checkpointing the full application state (snapshot) for fault tolerance. However, full-state checkpointing is

known to be prohibitively expensive, and users in many domains disable it as a result [76]. Some systems like Borealis [77] run multiple copies of the same job, but this requires the luxury of extra available resources [75].

In this work we present Samza, a distributed stream processing system that supports stateful processing, and adopts a *unified* (Lambda-less) design for processing both real-time as well as batch data using the same dataflow structure. Samza interacts with a change capture system (e.g., Databus) and a replayable messaging system (e.g., Apache Kafka, AWS Kinesis, Azure EventHub) [56, 78–80]. Samza incorporates support for fast failure recovery particularly when stateful operators fail.

The Lambda-less approach is used by Spark and Flink [33, 36]. However, Flink still requires the programmer to access two APIs for streaming and batch processing. We present experimental comparisons against Spark. Samza's unique features are:

- **Efficient Support for State:** Many applications need to store/access state along with their processing. For example, to compute the click-through rate of ads, the application has to keep the number of clicks and views for each ad. Samza splits a job into parallel tasks and offers high throughput and low latency by maintaining a local (in-memory or on-disk) state partitioned among tasks, as opposed to using a remote data store. If the task's memory is insufficient to store all its state, Samza stores state on the disk. We couple this with caching mechanisms to provide similar latency to, and better failure recovery than, a memory-only approach. Finally Samza maintains a changelog capturing changes to the state, which can be replayed after a failure. We argue that having a changelog (saving the incremental changes in state) is far more efficient than full state checkpointing, especially when the state is non-trivial in size.

- **Fast Failure Recovery and Job Restart:** When a failure occurs or when the job needs to be explicitly stopped and resumed, Samza is able to restart multiple tasks in parallel. This keeps recovery time low, and makes it independent of the number of

affected tasks. To reduce overhead of rebuilding state at a restarted task, Samza uses a mechanism called Host Affinity. This helps us reduce the restart time to a constant value, rather than linearly growing with the state size.

- **Reprocessing and Lambda-less Architecture:** It is very common to reprocess an entire stream or database. Common scenarios are rerunning using a different processing logic or after a bug discovery. Due to the large scale of reprocessing, many companies use a parallel batch [38] framework, creating duplicate and separate implementations of the same job (high maintenance overhead). Ideally, reprocessing should be done within one system (Lambda-less). Further, the reprocessing often needs to be done alongside processing of streaming data while not interfering with the stream job and without creating conflicting data. Samza provides a common stream-based API that allows the same logic to be used for both stream processing and batch reprocessing (if data is treated as a finite stream). Our architecture reprocesses data without affecting the processing of real-time events, by: a) temporarily scaling the job, b) throttling reprocessing, and c) resolving conflicts and stale data from reprocessing.

- **Scalability:** To handle large data volumes and large numbers of input sources, the system has to scale horizontally. To achieve this goal, Samza: i) splits the input source(s) using consistent hashing into partitions, and ii) maps each partition to a single task. Tasks are identical and independent of each other, with a lightweight coordinator per job. This enables near-linear scaling with number of containers.

Samza has successfully been in production at LinkedIn for the last 4 years, running across multiple datacenters with 100s of TB total data. This deployment spans more than 200 applications on over $10,000$ containers processing Trillions of events per day. Samza is open-source and over 15 companies, including Uber, Netflix and TripAdvisor, rely on it today [81].

Our experimental results show that Samza handles state efficiently (improving latency

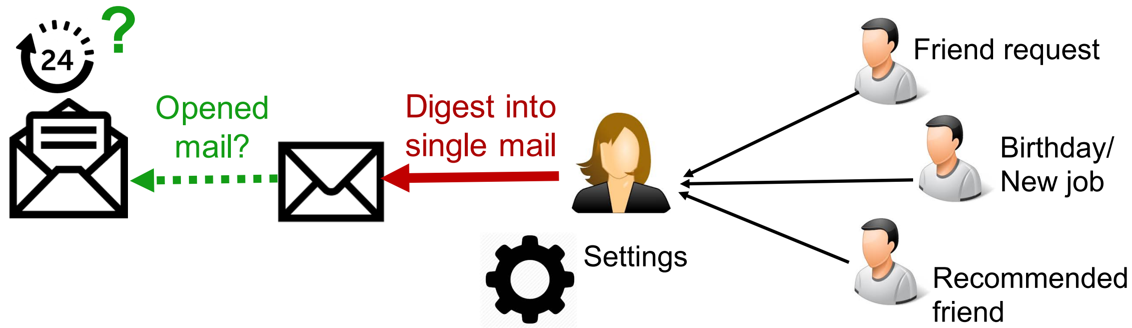


Figure 3.1: Email Digestion System (EDS).

and throughput by more than 100× compared to using remote storage), provides parallel recovery with almost constant time (regardless of the size of the state), scales linearly with adding more containers, and supports reprocessing data with minimal effect on real-time traffic, while outperforming batch systems. We experimentally compare against both variants of our own system (some of which capture other existing systems), and against Spark and Hadoop in both production and test clusters.

3.2 MOTIVATION

3.2.1 Stateful Processing

Most event processing applications need to access state beyond the mere contents of the events. In this chapter, we refer to state as any persistent data-structure defined by the application or used internally in the system. Such state may arise from cumulative or compact aggregates (computed from the stream), or static settings (e.g., user parameters), or stream joins. To illustrate, we describe *Email Digestion System (EDS)*, a production application running at LinkedIn using Samza. EDS controls email delivery to users by digesting all updates into a single email (Figure 3.1). EDS stores and accesses a large amount of state across multiple users. For each user, EDS accumulates and aggregates updates over a large window of time. To know the window size, EDS looks up the user digestion settings (e.g., every 4 hours) stored in a remote database. Finally, to find the effectiveness of the

digested email, it computes if the email was opened in the last couple days, by joining two streams of sent emails and opened emails over a multi-day window.

State is typically categorized into two types:

- **Read-Only state:** Applications look up "adjunct" read-only data, perhaps for each event, to get the necessary information to process it. Examples of such static state include user digestion settings in EDS or the user profile on each ad view event (accessed to find the user's field of expertise).

- **Read-Write state:** Some state is maintained and updated as the stream events continue to be processed. Examples of this type of state include: state required for joins of streams/tables over a windows, aggregations, buffers, and machine learning models. Some applications of this state include rates/counter over a window of time (used for monitoring ads or detecting Denial of Service attacks) and guaranteeing exactly-once semantics by storing all processed message ids to verify uniqueness of incoming message ids. In all these cases the state needs to be fault-tolerant while providing non-trivial consistency guarantees such as "read your writes".

3.2.2 Data Reprocessing

As described earlier, it is common to reprocess a stream or database, either in part or entirety. For example, at LinkedIn, we use a critical production job to standardize user profile information in order to offer relevant recommendations and advertisements. This job uses a machine learning model (derived offline) to standardize incoming profile updates in real-time. However, the model continually gets updated (even multiple times per week). Upon each update, all existing user profiles ($>$ 450 millions) have to be reprocessed while still processing incoming updates in real-time and without creating conflicts.

In other scenarios, only a few hours worth of data has to be reprocessed (instead of a whole database). For example, during an application upgrade, a software bug may come up.

With proper monitoring, the bug will most likely be detected within minutes or hours. The need after that is to revert the application (or fix the bug), rewind the input stream, and reprocess the data since the upgrade.

3.2.3 Application Summary

We summarize 9 major and diverse stream applications built using Samza that are currently running in LinkedIn's production environments across multiple datacenters, in Table 3.1. The location of the application is determined by the data source, varying from a single cluster to all clusters.

They are shown in Table 3.1. These are gathered from 10 YARN clusters with houndreds of nodes and 110 TB of memory, spread across multiple geo-distributed datacenters [1].

There is a wide variety amongst applications with a diverse combination of joins, aggregations, filters to even complex machine learning models. The applications studied are in order:

- *EDS*: Digesting updates into one email. State is used in multiple phases: updates are aggregated, user preference for digestion is looked up, and the stream of sent emails and opened emails are joined.

- *Call graph*: Generating the graph of the route a service call traverses. For each service call, a graph is incrementally generated (as the state) by aggregated the individual routes the call has traversed.

- *Inception*: Extracting exception information from error logs. This is a stateless filtering of the incoming data.

- *Exception Tracing*: Enriching exceptions with the exact source (machine) of the exception. This is done by joining the input stream with the table of source informations.

[1] Not all of the cluster capacity was used by the studied applications and other applications were also running.

Table 3.1: Applications running in LinkedIn's production across multiple datacenters. State size ranges from to 10s of GB to 100s of TBs (actual sizes not shown due to confidentiality). Max values shown in bold.

Name	Definition	Containers	Tasks	Inputs	Throughput msg/s	State type
EDS	Digesting updates into one email (aggregation, look-up, and join).	350	2500	14	40 K	on-disk
Call graph	Generating the graph of the route a service call traverses (aggregation).	150	**9500**	620	**1 Million**	in-mem
Inception	Extracting exception information from error logs (stateless filter).	300	300	**880**	700 K	stateless
Exception Tracing	Enriching exceptions with the source (machine) of the exception (join).	150	450	5	150 K	in-mem
Data Popularity	Calculating the top k most relevant categories of data items (join and machine learning).	70	420	9	3.5 K	on-disk
Data Enriching	Enriching the stream of data items with more detailed information (join).	350	700	2	100 K	on-disk
Site Speed	Computing site speed metrics (e.g., average or percentiles) from the stream of monitoring events over a 5-minute window (aggregation).	350	600	2	60 K	in-mem
A/B testing	Measuring the impact of a new feature. This application first categorizes input data (by their tag) into new and old versions and then computes various metrics for each category (split and aggregate).	450	900	2	100 K	in-mem
Standardization (>15 jobs)	Standardizing profile updates using machine learning models. This application includes > 15 jobs, each processing a distinct features such as title, gender, and company (join, look-up, machine learning).	**550**	5500	3	60 K	in-mem remote on-disk

- *Data Popularity*: Calculating the top k most relevant categories of data items. This involves enriching the data by doing joins and then using machine learning algorithms to find the top k.

- *Data Enriching*: Enriching the stream of data items with more detailed information by joining the stream with other tables.

- *Site Speed*: Computing site speed metrics (such as average and percentiles) from the stream of monitoring events over a 5-minute window. This involves multiple aggregations for the various metrics gathered.

- *A/B testing*: Measuring the impact of a new feature. This application first categorizes input data (by their tag) into new and old versions by splitting the input stream. Then, for each category various metrics are computed using aggregations.

- *Standardization*: Standardizing profile updates using machine learning models. This application includes > 15 jobs, each focusing on distinct features, such as title, gender, and company. Each job has a unique machine learning model, and in some jobs joins and data enriching is also used.

These applications exhibit a wide diversity along several angles: 1) *Scale*: throughput (input messages processed per second) and the number of containers, tasks, and inputs; 2) *State handled*: size and type of state; and 3) *Lifetime*: how long the job has been running.

Scale: The scale of applications varies widely based on the computational need of the application, from 70 containers to more than 500 containers. The higher scale is either due to higher throughput requirements (e.g., Inception) or computation load per event (e.g., EDS and Standardization). Samza supports various input source types (Kafka, Databus, Kinesis, etc.) as well as many input streams. Our applications range from 2 inputs to roughly 900 input streams with $> 27,000$ total partitions. For example, Inception processes 880 input streams (capturing exceptions) from multiple applications.

The number of tasks per container also varies significantly from 1 task per container (Inception) to ≈65 tasks per container (Call graph), with an average value of 10 tasks. A higher task per container ratio provides more flexibility when scaling out/in which is a positive factor for stateful jobs.

State: Applications range widely from stateless jobs (e.g., filtering done by Inception) to ones using a variety of different stores, ranging from fast in-memory stores to on-disk local stores with higher capacity (100s of TB vs. a few TBs) and remote stores with faster failure recovery. The type/size of the store is determined based on application requirements on performance, capacity, and failure recovery.

Lifetime: At LinkedIn, the number of production applications built using Samza has been growing rapidly, with a tenfold growth in the past 2 years (from 20 to 200). While we have focused on only the most mature applications here, newer applications continue to emerge.

3.3 SYSTEM OVERVIEW

In this section we present our end to end processing pipeline, Samza's high-level architecture, and how jobs are handled.

3.3.1 Processing Pipeline

Our stream processing pipeline works as a feedback loop (Figure 3.2). It receives events and updates from the service tier, processes them, and updates the service tier in return. This is a common pattern adopted by many companies.

The service tier (top of Figure 3.2, e.g., the website and mobile app, where clients interact) generates two main types of data that need to be processed. First, more than trillions of *events* are generated per day. Use cases vary widely from capturing interactions with the service tier (e.g., viewed ads, and shared articles) to background monitoring and logging

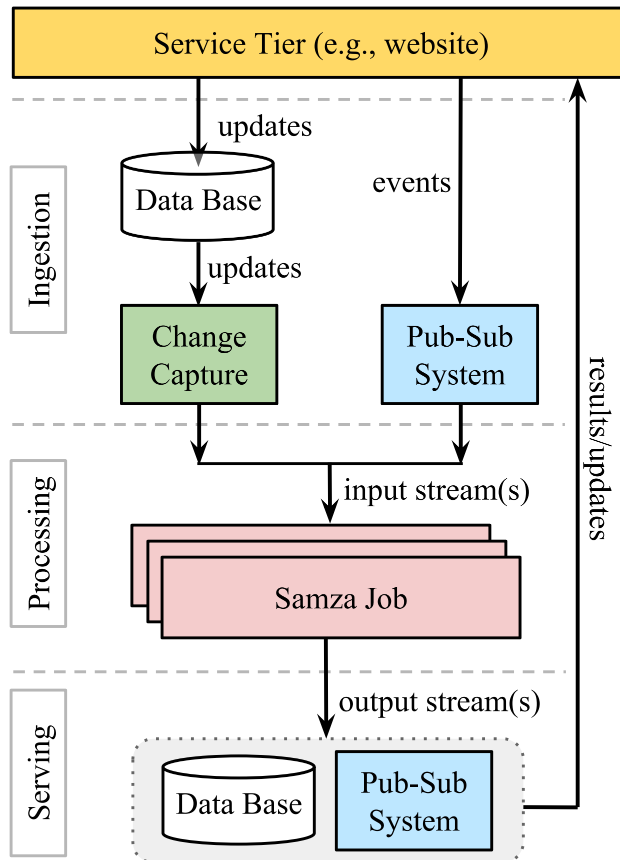


Figure 3.2: Stream processing pipeline at LinkedIn.

(e.g., site latency, exceptions and call tracing). Additionally, an often-overlooked source of data are *updates* occurring on databases (both SQL and NoSQL). Our databases have a transaction log capturing the stream of updates. For instance, every time a user changes their profile, several downstream applications need to know this and react to it.

At the first phase in Figure 3.2, called the *Ingestion* layer, these stream of events and updates are ingested into fault-tolerant and replayable messaging systems. We use Apache Kafka [56], a large-scale publish-subscribe system (widely adopted by >80 other companies [82]), and Databus [80], a database change capture system, as our messaging system for events and updates, respectively. Both these systems have the ability to replay streams to multiple subscribers (or applications), from any offset per subscriber. Databus also supports streaming a whole database from a snapshot.

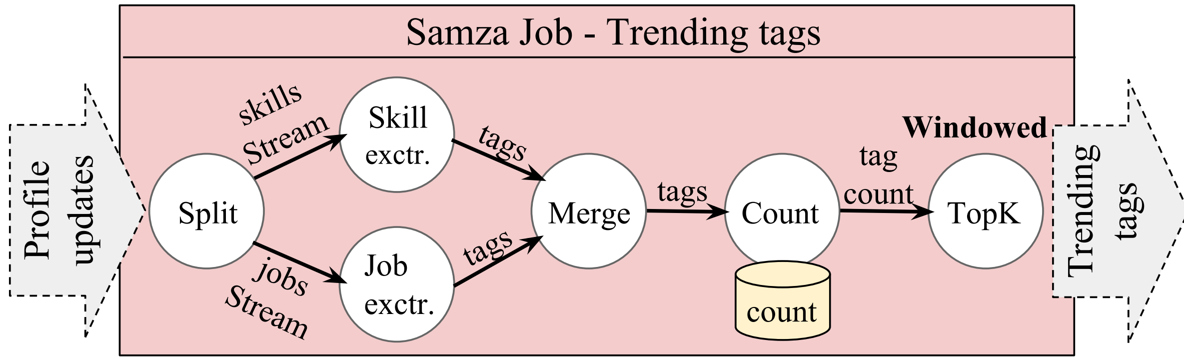


Figure 3.3: Example Samza job to find trending tags.

In the next phase, called the *Processing* layer, these multiple streams of events and updates are fed to one or many Samza jobs. Samza acts as our core processing framework, processing input and generating output in real-time. Finally, in the *Serving* layer, results of the processing layer (e.g., connection recommendations and suggested feeds), are persisted to an external database or a pub-sub system (e.g., Kafka). Results are also returned to the service tier to update the services accordingly.

3.3.2 Processing Layer Design

A Samza job is an intact stage of computation: one or many input streams are fed to the job; various processing–from simple operations (e.g., filter, join, and aggregation) to complex machine learning algorithms–are performed on the input; and one or many new output streams are generated.

Logical Representation

Samza represents jobs as a directed graph of *operators* (vertices) connected by *streams* of data (edges). Figure 3.3 shows an example Samza job consuming a stream of user profile updates, splitting the stream into skill and job updates, extracting tags, and computing the top k trending tags (we use this as a running example in our discussion).

A stream is an infinite sequence of messages, each in the form of a $(key, value)$ pair,

Table 3.2: Operators supported in Samza.

Type	Options	Definition
1:1	`map`	applying a defined function on each message.
	`filter`	filtering messages based on a function.
	`window`	splitting a stream into windows and aggregating elements in the window.
	`partition`	repartitioning a stream on a different key.
m:1	`join`	joining ≥ 2 streams into one stream based on a given function
	`merge`	merging ≥ 2 two streams into one stream.
1:m	user-defined	user-defined split or replication of a stream into ≥ 2 streams. This is achieved by allowing multiple operators consume the same stream.

flowing through the system. A stream is internally divided into multiple partitions, based on a given entry. There are three types of streams: 1) input streams that enter the job, without a source operator (e.g., Profile updates); 2) output streams that exit the job, without a destination operator (e.g., Trending tags); and 3) intermediate streams that connect and carry messages between two operators (e.g., skills, jobs, tags and counts).

An operator is a transformation of one or many streams to another stream(s). Based on the number of input and output streams, Samza supports three types of operators: a) *1:1 operators* transforming one stream to another (e.g., Count), b) *m:1 operators* transforming many streams to one (e.g., Merge), and c) *1:m operators* transforming one stream to many (e.g., Split), as shown in Table 3.2.

System API: The API of Samza is based on Java 8 Stream package [83] because of its ease of programming and functional programming capabilities. Listing 3.1 demonstrates the sample code for the Trending Tags job (Figure 3.3).

The basic 1:1 operators are: a) `map`: applying a user-defined function on each message (e.g., `SkillTagExtractor` extracting tags using a machine learning model or `MyCounter` updating a local store); b) `filter`: comparing each message against a filter condition (e.g.,

Listing 3.1: Sample API – Trending Tags Job.

```
public void create(StreamGraph graph, Config conf) {
   //initialize the graph
   graph = StreamGraph.fromConfig(conf);
   MsgStream<> updates = graph.createInStream();
   OutputStream<> topTags = graph.createOutStream();

   //create and connect operators
   MsgStream skillTags = updates.filter(SkillFilter f_s)
                              .map(SkillTagExtractor e_s);
   MsgStream JobTags = updates.filter(JobFilter f_j)
                              .map(JobTagExtractor e_j);
   skillTags.merge(jobTags).map(MyCounter)
         .window(10, TopKFinder).sendto(topTags);
         //10 sec window
}

class MyCounter implements Map<In, Out>{
//state definition
 Store<String, int> counts = new  Store();
 public Out apply (In msg){
    int cur = counts.get(msg.id) + 1;
    counts.put(msg.id, cur);
    return new Out(msg.id, cur)
 }
}
```

`SkillFilter`); c) `window`: partitioning a stream into windows and applying a user-defined function on the window (e.g., `TopKFinder` over 10 s windows); and d) `partition`: repartitioning and shuffling a stream on a different key. The main m:1 operators are: e) `join`: joining two streams on a user-defined condition, and f) `merge`: merging two streams into one (e.g., merging `skillTags` and `jobTags`). Finally, the 1:m operators are defined by feeding the same stream into different operators (e.g., feeding `update` stream into two different filters).

The combination of diverse operator types and support for arbitrary user-defined functions enables handling a wide range of applications. For example, to perform aggregation

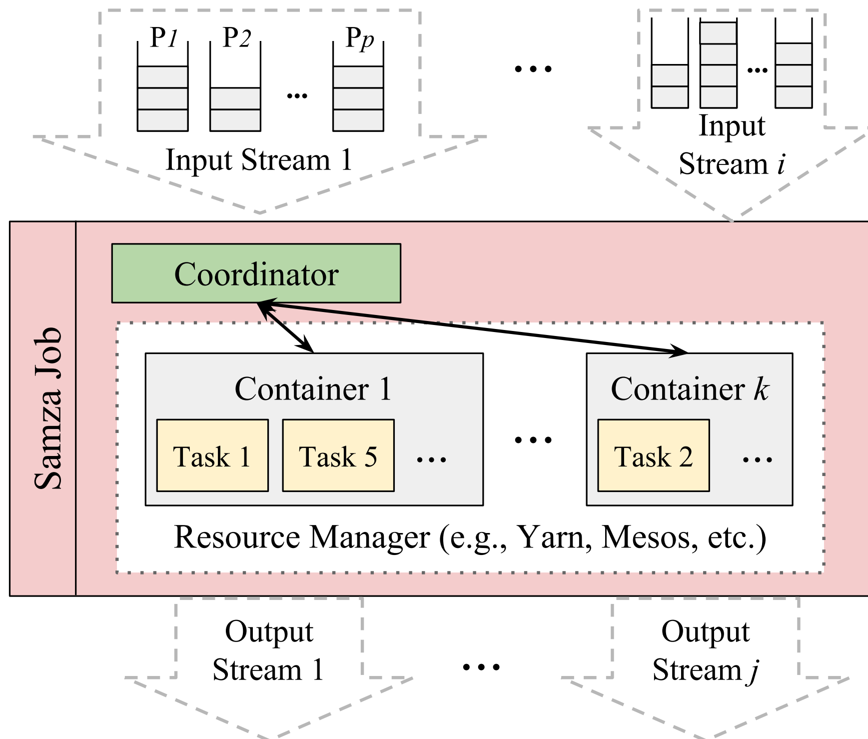


Figure 3.4: The internal architecture of a job.

(1:1 operator), depending on whether to be done over an entire stream or a window of data, a single aggregation logic (e.g., count) can be used in a `map` or `window` operator, respectively.

Physical Deployment

Internally, as depicted in Figure 3.4, a job is divided into multiple parallel, independent, and identical tasks, and an input stream is divided into partitions (e.g., $\{P_1, ..., P_p\}$). Each task executes the identical logic, but on its own input partition (a data parallelism approach). Each task runs the entire graph of operators. For each incoming message, the task flows the message through the graph (executing operators on the message), until an operator with no output or the final output stream is reached.

Most intermediate stream edges stay local to the task, i.e., they do not cross the task boundary. This keeps most communications local and minimizes network I/O. The only exception is the `partition` operator, where messages are redistributed across all tasks based

on the partitioning logic. For non-local streams and the job's input and output streams, Samza utilizes a fault-tolerant (no message loss) and replayable (with large buffering capabilities) communication mechanism. At LinkedIn, we mainly use Kafka, although other communications mechanism supporting partitioning, e.g., Kinesis or Azure EventHub [78,79] can be used instead.

By employing replayable communication with large buffering capabilities, Samza can temporarily overcome congestion. Lagging messages are buffered without impacting upstream jobs, and replayed with the pace of the slow job. This is particularly important at non-local streams with high potential of creating congestion. Although this approach does not fully resolve the issue, it gives enough time for a temporary spike to pass, or to scale-out a slow job.

We leverage the partitioning already performed by the input streams in order to split jobs into tasks. The number of partitions of the input streams (configured by the application developer) indicates the number of tasks. For a single stream, each partition is mapped to a single task. However, partitions of different streams (e.g., partition 1 stream A and partition 1 stream B) can be mapped to the same task (used for joining two streams). A higher number of tasks provides more parallelism and finer granularity when scaling. However, too many tasks can create excessive overhead.

Resource Allocation: Tasks are grouped together into containers using a round-robin, random, or user-defined strategy. The number of threads is configurable, ranging from one per container up to one per task[2]. The spectrum of choices defined by these extremes also defines a trade-off between ease of programming (with no race conditions in a single-threaded model) and performance (with potentially higher throughput by exploiting more parallelism).

The application developer configures the number and capacity of containers, which de-

[2]Within a tasks users can implement multi-threaded logic.

fines the amount of resources assigned to a job[3]. Samza offloads container allocation and placement to a separate Resource Manager layer. This layer manages the available resources in cluster by handling: resource allocation, monitoring, failure detection, and failure recovery (by restarting or reallocation). This layered and modular design provides pluggability and flexibility. Currently, we use Apache YARN [22], one of the most popular resource managers, in our pipeline. Samza is also available as *standalone Samza*, an embeddable client library allowing applications to be hosted in any environment.

Coordinator: Each job has a lightweight *Coordinator* managing and tracking the job. The Coordinator maintains several pieces of metadata pertinent to the job including: i) job configuration (such as the number of containers and input sources); ii) placements (mapping of containers to machines, tasks to containers, and input partitions to tasks). When using YARN, the coordinator is part of YARN's Application Master, and when using standalone Samza, the coordinator uses Zookeeper to elect a singleton leader.

3.4 SYSTEM DESIGN

In this section we discuss Samza's goal (Section 1), existing ways to address it, and key design techniques in Samza.

3.4.1 Efficient Access to State

Several applications (Section 3.2.1) access/store large amou/nts of state along with processing incoming events. Some streaming engines have tackled this problem by using a reliable external *remote store* [35, 37], e.g., MillWheel persists data in Bigtable [28]. This remote store has to independently handle fault-tolerance (by replicating data) while still providing some notion of consistency (the weakest requirement is usually read-your-writes consistency per task).

[3] Configuring the optimal number of containers is a challenging problem, especially in the presence of unpredictable workload changes [84]. As future work, we are working on dynamically and adaptively scaling the number of containers (based on the job's load and requirements).

While storing state in an external file system outsources the responsibility of fault-tolerance, this approach is not efficient. It consumes network and CPU resources, increases average and tail latency, and limits throughput. It may also overwhelm the remote store (e.g., in presence of spikes), negatively impacting other applications using the shared store. When a single input message generates multiple remote requests, this is further amplified. For example, Millwheel and Trident provide exactly-once semantics by storing processed message keys (one write per message) along with verifying that incoming messages are unique (one read per message).

Another approach is to keep data local and, for fault-tolerance, use periodic *checkpointing*, i.e., a snapshot of the entire state is periodically stored in a persistent storage [33, 36, 75]. However, full state checkpointing in general slows down the application. It is particularly expensive when state is large, such as 100s of TB (Section 3.5); users tend to disable full state checkpointing for even smaller state sizes [76].

State in Samza

Samza moves the state from a remote store to instead being local to tasks – the task's *local store* (memory and disk) is used to store that task's state. This is particularly feasible in Samza with independent tasks (Figure 3.5).

Samza supports both in-memory and on-disk stores as options to trade off performance, failure recovery, and capacity. The in-memory approach is the fastest, especially for applications with random access (poor data locality). The on-disk store can handle state that is orders of magnitude larger while reducing failure recovery time. For our on-disk store we use RocksDB, a high-performance and low-latency single machine storage, widely used [85]. Other embeddable stores, e.g., LevelDB and LMDB, can be used as well.

Samza further improves on-disk stores by leveraging memory as a 3-layer cache. At the deepest layer, each RocksDB instance caches the most popular items using a least recently used (LRU) mechanism. To mitigate the deserialization cost of RocksDB, Samza provides

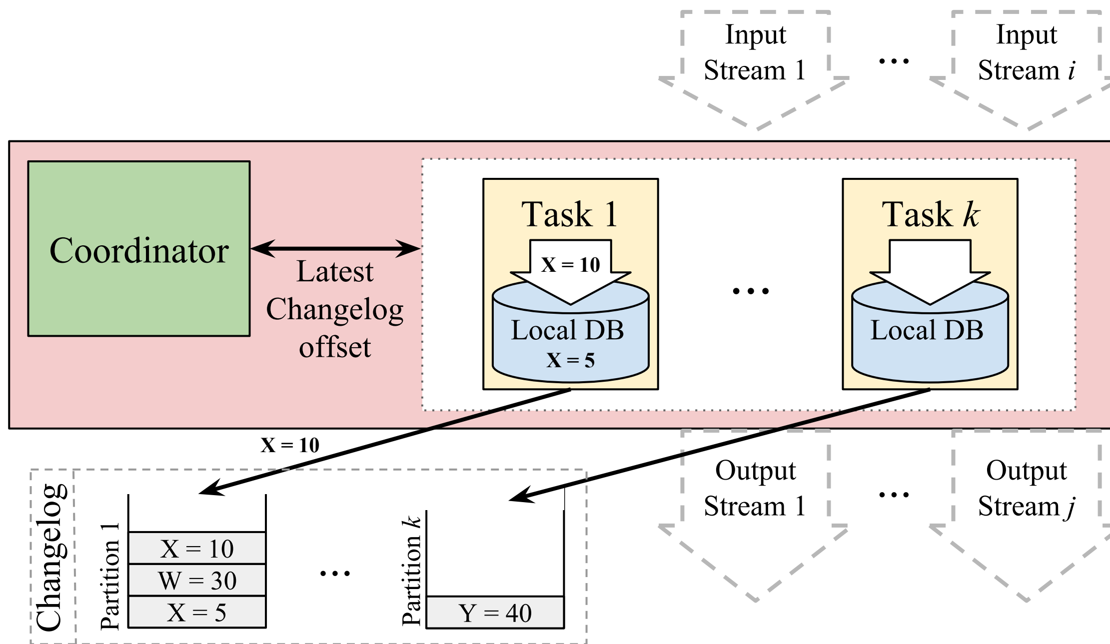


Figure 3.5: Layout of local state in Samza, and how fault-tolerance is provided.

a caching layer of deserialized data in front of RocksDB. Finally, we rely on OS caches to keep the frequently accessed pages around (similar to [13]). Our experiments show that for applications with good-locality workloads, caching mechanisms ensure on-disk stores perform close to in-memory stores. For random access workloads, on-disk still achieves acceptable performance.

In most cases, state is partitioned across tasks using the same partitioning function and key as used for the input stream. Hash or range partitioning can be used. For instance, in a word-count job, a task is assigned to process words in a specified range (e.g., words starting with $[a-g]$) and stores state for the same range (e.g., counts of words starting with $[a-g]$). Joins, aggregations, metric computation (count/rates), are all supported in this manner.

For applications that absolutely need to use a remote store, Samza supports *asynchronous* processing of remote requests for efficiency and concurrency. When using asynchronous processing, Samza handles out of order event processing while ensuring at-least-once guarantees (even in the event of failures). This may be needed if the input partitioning is different from the state partitioning. In a job enriching place-of-birth updates with country information,

the input is a stream of profile updates (key = *userid*) while the store is keyed by country (key = *countryid*). In such cases, if the state is small (tens of GB), Samza can broadcast the state to all tasks and store it locally, otherwise, Samza uses a remote store along with caching (for performance). Another use case is when tasks need to share state, or state needs to be queried from outside the job, where a remote store satisfying the consistency requirements of the job is used.

Although local state provides better performance, it comes at the cost of lower flexibility and losing independence of state from computation. This brings up major challenges including: *what if a failure occurs?*, *how to recover from failures with low overhead?*, and *How to scale jobs out/in?*. These challenges are addressed in the following sections.

Fault-Tolerance

Using local state, requires solving a main challenge that arises out of it: *how to provide efficient fault-tolerance?* Samza equips each task with a changelog that captures all updates done on the local store (Figure 3.5). A key feature of the changelog is that it is only capturing incremental changes rather than the entire state. The changelog is an append-only log maintained outside of the job architecture. Samza stores the changelog in Kafka, enabling fast and easy replays in case of a failure, although, any other durable, replayable and ordered messaging system can be used.

For efficiency, the changelog is kept out of the hot path of computation. Updates are batched and periodically sent to Kafka in the background using the spare network bandwidth. After successfully writing a batch of updates, the *latest offset*–indicating the latest successfully processed input message–is persisted in the Coordinator (Figure 3.5). After a failure, state is rebuilt by replaying the changelog, then, all messages after the latest offset are reprocessed.

Moreover, to reduce changelog overheads and prevent an indefinitely growing changelog, Samza utilizes Kafka's compaction features. Compaction retains the latest value for each

key by removing duplicate updates. Compaction is performed in the background and outside the hot path. Compaction is used in two cases: 1) compacting the batch of updates sent to the changelog (reducing the network overhead); 2) compacting the changelog itself (reducing storage overhead). Right after compaction, the changelog is no larger than a snapshot of the task's most critical state.

It is well-known that providing exactly-once semantics is either slow or overloads the remote file system [35]. Samza guarantees at-least-once processing, preferring performance over consistency. In practice, we observe that at-least-once is sufficient for our applications requirements. For a few cases requiring exactly-once, it is implemented by the application (with low overhead) by leveraging local state.

The changelog approach in Samza provides a *read-your-writes* consistency level on a per task basis. Without failures, data is stored locally on single replica, straightforwardly providing read-your-writes consistency. In presence of a failure, processing and state are rolled back to the point of time where consistency is conserved, i.e., the latest persisted offset, wherein all updates from processed messages up to the latest offset are reliably reflected in the state.

The changelog in Samza adds less than 5% performance overhead. An append-only log has been measured to be far more efficient (2 million op/s with 3 machines in Kafka [86]) compared to accessing a remote store (at most 100K op/s with 3 machines [87, 88]).

Fast state recovery

After a failure, a job or a portion of it needs to be restarted. Failures may include node, container, or disk failures. Restart may also be warranted upon preventive maintenance (either stop-the-world or one container at a time), and configuration updates (due to misconfiguration or workload changes).

Replaying the changelog (even compacted) can still introduce excessive overhead and long pauses, e.g., with 100s of TBs of state. This will be especially pronounced when the changelog

is accessed remotely. To mitigate this, Samza uses a fast state recovery mechanism called *Host Affinity (HAff)*. The key idea in HAff is to leverage the state already stored on disk (in RocksDB) by preferring to place a restarting task on the same physical machine where it was running prior to the failure (stored in the Coordinator). This is a best-effort mechanism, and will continually try to optimize placement, even in presence of repeated failures. However, HAff is not effective in case of permanent machine failures, where replaying the changelog is used instead.

To make HAff feasible, Samza stores state in a known directory (in the native file system) outside of the container namespace. This allows state to live independent of the application lifecycle. A garbage collection agent runs in the background, removing state of permanently deleted applications. Since the underlying system cannot distinguish between stopped and deleted applications, we rely on the application developer to manually mark applications as deleted.

In production, we found that HAff is effective in over 85% of restart cases. By using HAff in our large stateful applications ($\approx$ 100 of TBs of state), we were able to reduce recovery time by 60$\times$ (from 30 minutes to 30 seconds).

3.4.2 Lambda-less

Inevitable software bugs and changes along with inaccuracies (late or out-of-order arrivals) can require parts (or even a whole) stream to be reprocessed. To mitigate this issue, many companies [89] utilize a Lambda architecture, wherein data is dispatched in a parallel "fork" to both an online stream and offline batch path (e.g., Hadoop or Spark), as shown in Figure 3.6). The stream path processes incoming data in real-time (latency is first-class) while the batch path acts as source-of-truth, periodically generating batch views of accurate results (accuracy is first-class). Final results are computed by merging stream and refined batch views [69]. To reprocess data it is sent via the batch path. This separation allows data to be reprocessed completely via the batch path (if needed).

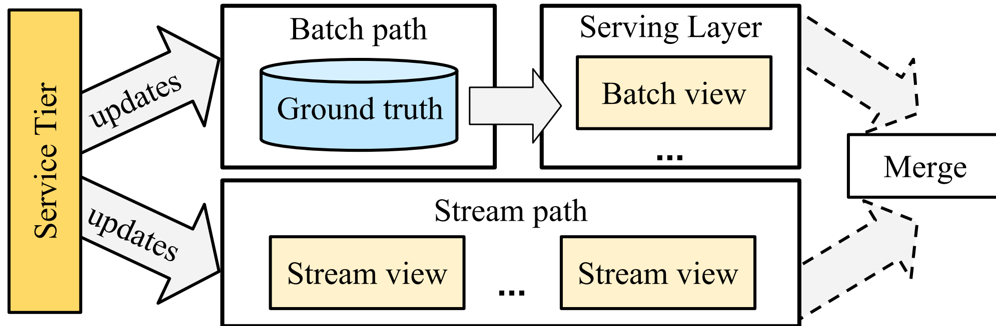


Figure 3.6: Lambda architecture.

However, the Lambda architecture comes at a high management cost, requiring duplicate development of stream and batch logic and code for the same application, and keeping these logics in sync as the application evolves over time. The Lambda approach also consumes double resources (two systems need to be running for both stream and batch processing). In the batch path inaccuracies could still occur–there can be late arrivals at the beginning, and missing data at the end of the batch interval.

Samza instead adopts a unified model supporting both stream and batch. The main challenges are: 1) to process *late events*, and 2) to *reprocess* a stream or database without impacting incoming messages or pressuring the database/service. 3) to support an easy-to-use API (Section 3.3.2) readily available in batch systems [36, 41, 42].

Unified Model: Similar to [74,90], Samza treats batch data as a finite stream of data, with a special token indicating the end of the stream. Application logic is developed and maintained in one place using a unified API. A single application can switch between real-time traffic, batch data from HDFS (integrated with Samza), or a database snapshot. Depending on the input source fed to the application, stream of incoming messages or old batched data, the job will become stream or batch job.

Processing Late Events: Samza employs a *reactive approach*, i.e., processing and fixing previous results when late or out-of-order results arrive (this bears similarities to Millwheel [35]). To avoid reprocessing the entire stream, the input is split into windows. Upon pro-

cessing a late message, the impacted windows are found, rolled back, and recomputed [74]. State management is a key element in late event handling. Generally, the whole window of messages should be stored (e.g., a join operation). For some operations, storage can be optimized where a compact final "result" is available (e.g., for a counter or aggregations).

Currently, the application is in charge of implementing the late arrival logic [4]. However, the windowing functionality along with the efficient state handling make Samza an perfect fit for this I/O intensive feature.

Reprocessing: To reprocess an entire stream or database (Section 3.2.2), Samza leverages: a) Kafka's replaying capability to reprocess a stream, and b) Databus' *bootstrapping* capability to reprocess a database. During bootstrapping, Databus generates a stream from a database snapshot (without impacting the database service) followed by the stream of new updates after the snapshot.

To perform reprocessing, Samza simply switches between different inputs: real-time traffic, replayed stream, or bootstrap stream, in single intact application. Reprocessing can be done in two modes: 1) *blocking* where real-time computation blocks until all reprocessing is complete, or 2) *non-blocking* where reprocessing is done in parallel with real-time processing. Typically, blocking reprocessing is used with small datasets, e.g., rolling-back latest upgrade due to a bug, while non-blocking processing is used with massive datasets, e.g., business logic change requiring reprocessing of whole database. In non-blocking reprocessing, Samza minimizes the impact on the real-time processing via: i) throttling reprocessing, ii) temporary job scale out.

Late events may create conflicts. A merge job is used to resolve conflicts (between the reprocessing and real-time stream) and prioritize the real-time results. This is developer specified logic. For instance, in the Standardization job, the user may change the profile, p_{new}, while the reprocessing will also process the user's old profile, p_{old}. If reprocessing of p_{old} occurs after processing p_{new}, it can override the results of the new profile. Thus, a merge

[4] As future work, we are adding late event handling as a built-in support in Samza.

job is needed to merge both updates and prioritize the results of p_{new}.

Although there is no technical size limit to what you can reprocess in a streaming system like Samza, there comes a point where you might need thousands of machines to do the reprocessing in a reasonable amount of time, e.g. a dataset which is hundreds of terabytes or more. At LinkedIn, such datasets are reprocessed only in our Hadoop grids. This separation of clusters avoids network saturation and other DOS triggers in our online clusters.

3.4.3 Scalable Design

Samza provides scalability via a decentralized design, maximizing independence among its components. These design principals have helped Samzato scale in multiple fronts:

1. *Scaling resources:* As discussed in Section 3.3.2, a job is split into independent and identical tasks (with input/state partitioning). Then, independently tasks are allocated on containers. This decoupling allows tasks to be flexibly scheduled and migrated if necessary. The system easily scales by spreading tasks across a larger number of containers. Based on our result, we scale almost linearly with adding more containers.

2. *Scaling state:* Samza scales to a massive amount of state, by leveraging independent partitioned local stores. State can easily scale by spreading the local stores across more machines (or containers). Also, state recovery is done in parallel across tasks and is not impacted by the number of failed containers.

3. *Scaling input sources:* Samza treats each input stream autonomously from other inputs. This enables scaling to many inputs, e.g., the Inception application (Table 3.1) processes exceptions from more than 850 different streams sources. Due to its modular design, Samza works with a variety of systems including: Databus, DynamoDB Streams, Kinesis, ZeroMQ and Mongo DB [56,79,80,91–93], and this set is continuously growing.

Table 3.3: Main parameters of data generation in each approach, and the range of values studied.

Approach	Parameter	Definition	Range
Checkpoint	*interval*	time between two consecutive checkpoints.	10 min - 90 min
	state_size	total size of state in Bytes	100 GB - 100 TB
Changelog	*change_rate*	rate of entry changes in the state (msg/s).	10 K - 10 M
	entry_size	size of each entry of the state in Bytes	10 B - 1 KB

4. *Scaling the number of jobs:* Samza utilizes a decentralized design with no system-wide master. Instead, each job has a lightweight Coordinator managing it. Also, jobs maintain their independence from each other, and each job is placed on its own set of containers. This enables Samza to scale to large numbers of jobs. We have seen a 10× growth in the number of applications over the past 2 years.

3.5 CHECKPOINTING VS. CHANGELOG

To provide fault-tolerance, Samza uses a changelog capturing *changes* to the state in the background. Another popular approach is full state checkpointing, where periodically a snapshot of the *entire state* is taken and stored in an external storage [33, 36, 75, 94]. Checkpointing can be either synchronous (pause, checkpoint, and resume) or asynchronous (in the background)–a more performant but also more complex approach. In both cases, the overhead of checkpointing can be prohibitive especially for large state sizes.

In this section, we quantitatively compare full-state checkpointing vs. Samza's changelog approach, taking into account characteristics of real applications from production.

The average amount of additional data generated (Bytes/s) is the main source of overhead in both checkpointing and changelog. Table 3.3 summarizes the parameters that affect it, and thus, the focus of this study.

For checkpointing, data generation depends on checkpointing interval (*interval*) and

size of each checkpoint (*state_size*). The *interval* trades off checkpointing overhead (less for larger intervals) and the amount of work needed to be redone in the case of a failure (more for larger intervals). On the other hand, changelog depends on the rate of changes (*change_rate*) and the size of each change (*entry_size*). Thus, the average rate of data generation for these approaches are:

$$Data_{checkpoint} = \frac{state_size}{interval}$$
$$Data_{changelog} = change_rate \times entry_size$$

We define the break-even point, *bp*, as where $Data_{checkpoint}$ equals to $Data_{changelog}$. For any *change_rate* value below *bp*, changelog is the preferred approach, and for any value above, checkpointing. For various checkpointing configurations (*interval* and *state_size*) and *entry_size* values, we measure *change_rate* at break-even point. This is depicted as the lines in Figure 3.7. For example, for a *state_size* of 100 TB, an *interval* of 20 minutes, and *entry_size* of 10 B, *bp* is ≈ 10 Trillion changes/s. For any *change_rate* below 10 Trillion/s, changelog would be a better option.

Based on our production application configurations (Section 3.2.3) a *change_rate* of Trillions of changes/s is not realistic. As a pessimistic estimate of the *change_rate* (accounting for our application growth over the next few years), we use the throughput achieved in our production applications (Table 3.1) as a proxy for the *change_rate* and multiply it by 10. This range, is shown by the shaded area in Figure 3.7.

We observe that for large *state_size* values (100 TB), changelog is clearly a better choice (the shaded area is below the 100 TB lines). A small *state_size* with a large *entry_size* (100 GB - 1 KB, the lowest line in the plot) is also uncommon in production-scale applications, since the state should compose of only a few entries. For a small *state_size* and a small *entry_size* (100 GB–10 B, second lowest line in plot), at a *change_rate* of around 10 M

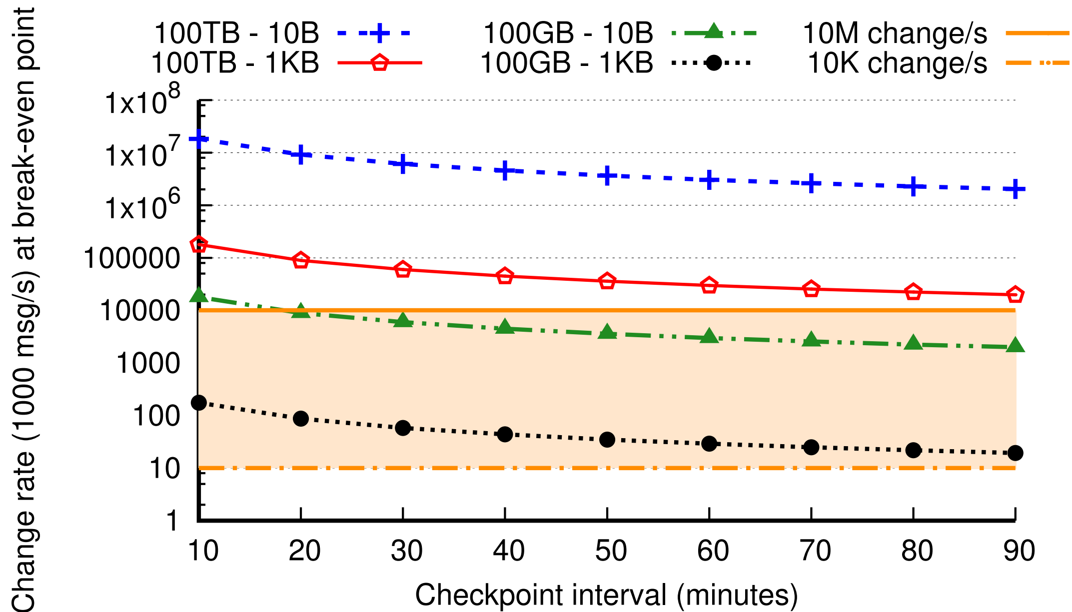


Figure 3.7: Comparison of checkpointing under various *state_size* (100 TB and 100 GB) and *interval* values with changelog under various *entry_size* (10 B and 1 KB) and *change_rate* values. Shaded region shows 10× typical values from applications (Table 3.1)

change/s, changelog performs worse than checkpointing. To mitigate this issue, Samza utilizes batching along with a compaction mechanism (removing redundant keys) to reduce the effective *change_rate*. By batching data for a couple of seconds, even with a *change_rate* of 10 M change/s (given the total state has ≈ 10 Million entries in the 100GB–10B case), the effective *change_rate* is reduced significantly, keeping changelog efficient and the more preferable technique.

3.6 EVALUATION

Our evaluation addresses the following questions:

1. How effective is local state, versus alternative options?
2. What is the effect of failures, and how fast is recovery? How much does Host Affinity help in failure recovery?
3. How fast is reprocessing, especially compared to existing systems?

4. How does Samza scale?

In doing so, we compare Samza with existing systems including Spark and Hadoop, as well as against other alternative Samza-like designs.

3.6.1 Experimental Setup

We evaluated the system using both production jobs and microbenchmarks, subjecting the system to much higher stress than production workloads. Our experiments were performed on both small (6 nodes) and large (500 nodes) production clusters at LinkedIn.

Microbenchmarks were performed on a test YARN and Kafka cluster. We used a 6 node YARN cluster, with 4 Resource Managers (RMs) and 2 Node Managers (NMs). Each NM was a high-end machine with 64GB RAM, 24 core CPUs, a 1.6 TB SSD, 2 1TB HDDs, and a full-duplex 1 Gbps Ethernet network. We also used an 8 node Kafka cluster of similar machines. We tested the system using two applications: a *ReadWrite* and *ReadOnly* job.

The ReadWrite job contains a map of ids to counters. For each input message, an embedded *id* is extracted, current count for *id* is read, the counter is incremented, and then written back. This job mimics the trend in real-world aggregation and metrics collecting jobs, e.g., EDS, Call Graph, Site Speed, and A/B Testing in Table 3.1.

The ReadOnly job consists of a join between a database and an input stream. For each message, an embedded *id* is extracted, value *val* for id is read from a database, *val* is joined with (a fraction of) the input message, and outputted as a new message. This follows the pattern used in many real-world enriching jobs, e.g., Data Enriching (enriching a stream of data with additional details) and Exception Tracing (enriching exceptions with source information).

We use a single input stream with infinite tuples (*id*, padding). *id* is a randomly generated number in the range $[1, 10^k]$ and padding is a randomly generated string of size m. We use k and m as tuning knobs of the workload. k trades off state size for locality–a larger k creates

more entries (larger state) while decreasing the chance of reading the same data twice. m is used to tune CPU/network usage. Since the serialization/deserialization overhead and header overhead per message is almost constant, m tunes the ratio of overhead to Bytes/s processed. We chose m such that the system is under stress (CPU and network utilization $\geq 60\%$). We found 100 and 130 Bytes padding to be the appropriate values for ReadWrite and ReadOnly, respectively.

Before submitting a job, we pre-populate the input stream, so that no time is spent on waiting for new data (inter-arrival between messages is 0). Additionally, in ReadOnly case, we pre-populate the store with random values for all keys.

3.6.2 Effectiveness of Local State

In order to evaluate our design of local state, we compare our choice against other alternative designs. The stores we consider are as following:

- *in-mem* and *on-disk*: A partitioned in-memory store (our homegrown key-value store) or on-disk store (Rocks/DB), *without* any fault-tolerance mechanism. Stateless systems, such as Storm and Heron, use these type of stores (typically in-mem). Additionally, without considering the checkpoint overhead which depends on the interval and state size (Section 3.5), systems using checkpointing, e.g., Flink and Spark [33, 36, 75], also fall here.

- *in-mem + Clog* and *on-disk + Clog*: Samza's in-mem or on-disk store along with changelog for fault-tolerance. This a new store developed by Samza.

- *on-disk no cache*: On-disk with no in-memory caching. This mimics the behavior of applications with large state and poor data locality (high cache misses).

- *remote store*: An external remote storage, used in many systems including Millwheel, Trident, Dataflow [35, 37, 74].

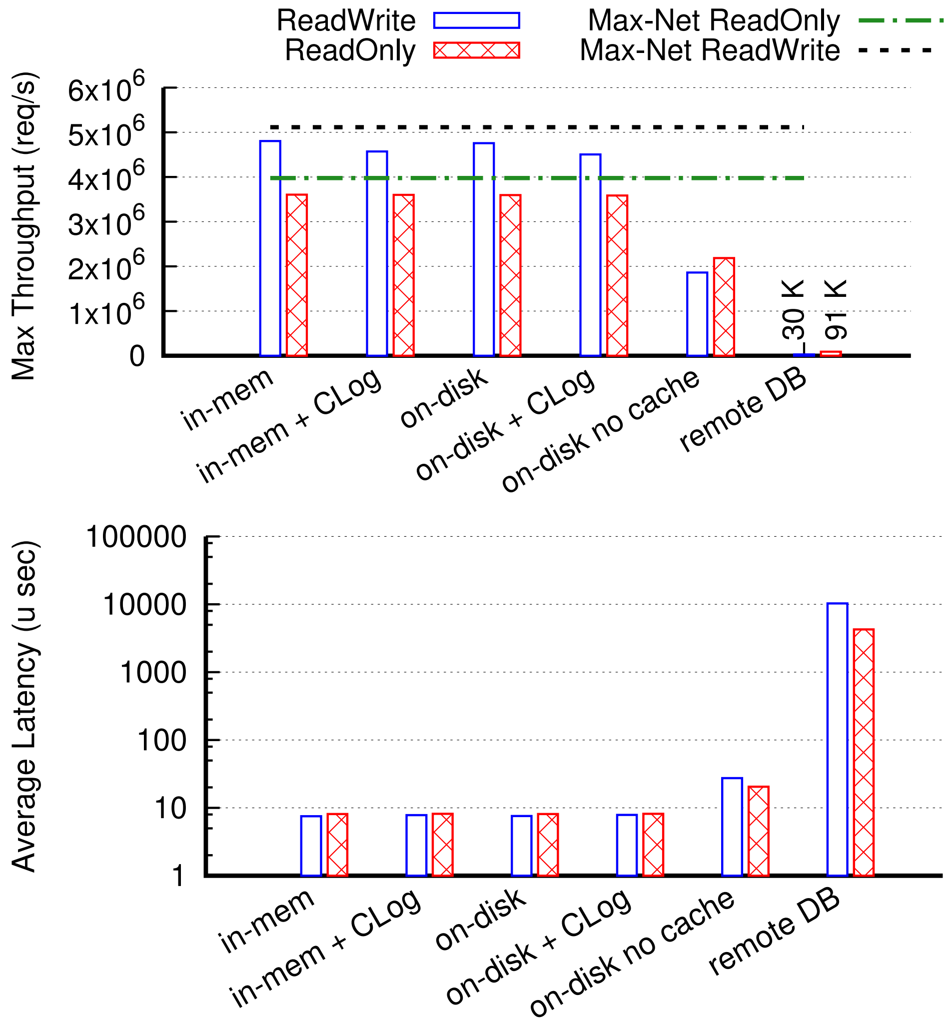


Figure 3.8: Comparision of storing state using an in-memory structure, local on-disk storage, or a remote database, with and without a changelog (CLog). Theses graph show throughput and latency in a read only and a 50-50 read write workload.

Although our Samza implementation supports all these variants, the default is on-disk + Clog. This variant performs the best, has large state support (hundreds of TBs), and offers low cost failure recovery (close to stateless).

To evaluate state, we used ReadWrite and ReadOnly micro-benchmarks. In each test we continuously added containers until throughput is saturated. Figure 3.8 shows maximum achieved throughput and average latency for different stores. We computed the theoretical maximum throughput achievable by the network (Max-Net), i.e., network bandwidth divided

by the message size. Since ReadOnly messages are larger than ReadWrite, both maximum and achieved network throughput are smaller.

In-memory vs. On-disk

As shown in Figure 3.8, in-mem and on-disk stores perform similarly, and both approach the network maximum (Max-Net). The in-mem and on-disk stores do not handle fault-tolerance. However, even when we add fault-tolerance using a changelog, the overhead is negligible.

To measure the effect of caches, we also plot numbers from disabling all internal caches, including the caching layer provided by Samza and Rocks DB (on-disk no cache). This reduced throughput by only 50-60%, indicating the caching is not solely responsible for our performance gains.

We conclude that on-disk state coupled with a caching strategy can achieve the same performance as using in-mem store, but it also achieves better fault-tolerance and supports larger state than in-mem (TBs vs tens of GBs).

Local vs. Remote state

We compared using local state (in-mem or on-disk) to remote state. As our remote state we used Espresso [31], a scalable key-value store widely used in LinkedIn's production (e.g., storing user profiles). We used an additional 5 node cluster (4 data nodes and a router) with nodes similar to the Kafka cluster. As shown in Figure 3.8, even with additional resources used for the remote store, latency increases by 3 orders of magnitude (a few μs to a few ms). This is due to traversing multiple hubs (router, data nodes, replication, and back to the user) which each takes hundreds of μs.

Throughput is impacted less than latency, and drops by two orders of magnitude (100-150 $\times$), since requests are issued in parallel. ReadOnly achieves 3$\times$ better throughput than

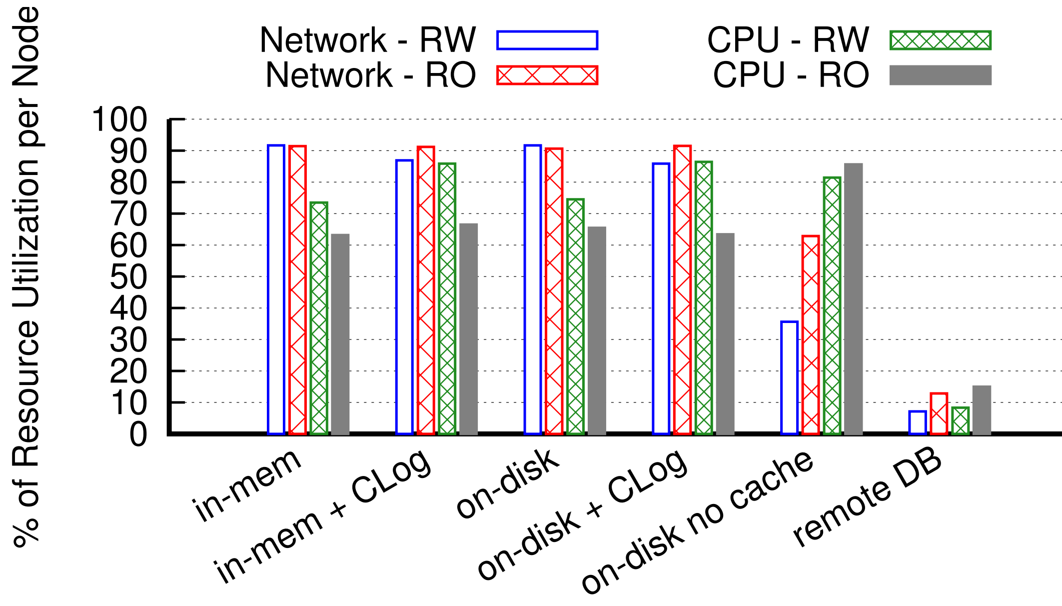


Figure 3.9: Utilization of network (in bound link) and CPU when using in-mem, on-disk and remote state, with and without a changelog (CLog).

ReadWrite because a) the former issues fewer requests per message (one vs. two), and b) reads have lower overhead vs. writes (no replication needed).

We expect this large difference between local and remote state to hold beyond Espresso. Other studies [87,88] show that most popular stores, such as Cassandra, HBase, Voldemort, MySQL, Couchbase, and Redis [24, 50, 95–98], can only reach tens of 1000s of requests/s using of 4 nodes. When using local state we perform millions of requests/s.

Resource Utilization

Figure 3.9 measures the resource utilization (CPU, disk, network) for each test. We elide disk utilization (being $< 5\%$ for all except for no-cache case) and outbound network (following the same pattern as inbound link) due to space.

When using in-mem store or on-disk with caching (with or without changelog), we saturate the network (utilizing $\geq 85\%$). Note that our benchmarks are configured to stress-test the system using $\geq$60% of CPU resources, while in production this value is typically below 20%.

Adding changelog has a small impact ($\approx 15\%$) on CPU utilization (additional serialization and deserialization overhead), and less than 2% effect on network. Similarly, removing the internal caches (on-disk no cache) causes a spike in CPU usage, though it is processing fewer messages–this is because of RocksDB's serialization/deserialization overhead.

The remote DB has a low utilization ($< 20\%$) in all resources, since the job is mostly idle–waiting for a response from the database. The resources are used inefficiently as well. For example, using remote store, the amount data transfered over the network for processing a single message is 5-10$\times$ higher than local store.

Latency Tail and Variance

The tail and variance of latency are important factors in many applications. We define latency as the total time spent in processing a message (event), including time spent in fetching the message from the input source. Figure 3.10 shows the Cumulative Distribution Function (CDF) of latencies in all cases. Since Samza fetches messages in batches (50 K messages in our test), a few messages incur very long latencies causing long tails in the CDF. However, for the rest, the variance is low and a majority of values are close to the median (CDF is very close to a vertical line).

3.6.3 Failure Recovery

The main advantage remote store is fast seamless state recovery (with no overhead on the stream job), a luxury diminished when using local storage: state has be restored locally before continuing processing. To measure failure recovery overhead when using local state, we randomly killed a percentage of containers (6% to 50%) in a stateful job. We measured the recovery time–the time spent between the first failure until all containers are up and running–both with Host Affinity disabled and enabled (w/ HAff). In Host Affinity, we used a success rate of 100%, i.e., ratio of containers placed on the same machine as before. Although this might seem extreme, it is not far from our production success rate (85-90%).

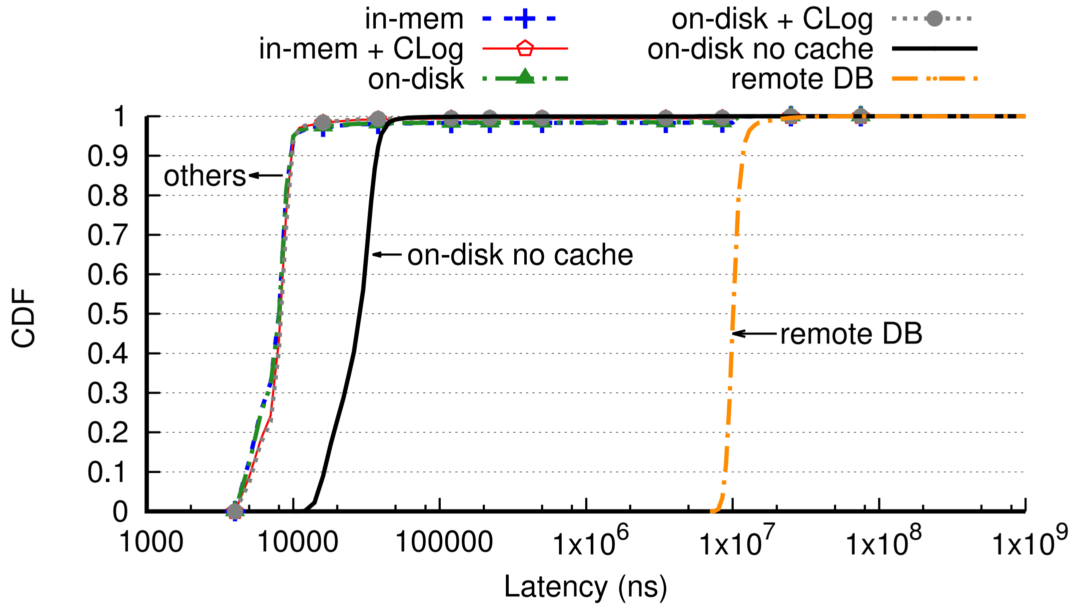


Figure 3.10: CDF of latency when using local and remote state, with and without a changelog (CLog).

In production, the main reason for misses are permanent node failures, and being a shared cluster with other jobs filling the free capacity.

In this experiment we used the ReadWrite workload, 16 8GB containers, on-disk + Clog store, and an input stream containing all keys in the range $[1 - 10^{12}]$ in order. For each input message processed a new entry was stored locally and added to the changelog.

As Figure 3.11 shows, without Host Affinity, recovery time increases proportionally with state size. With Host Affinity recovery time becomes near constant independent of the state size. In our production jobs, recovery time reduced from 30 minutes to less than 30 seconds using Host Affinity.

Furthermore, failure recovery time was nearly independent of the percentage of containers failing. This is because tasks are recovered in parallel. This shows the significance of having a scalable design that partitions jobs into tasks with little interdependency.

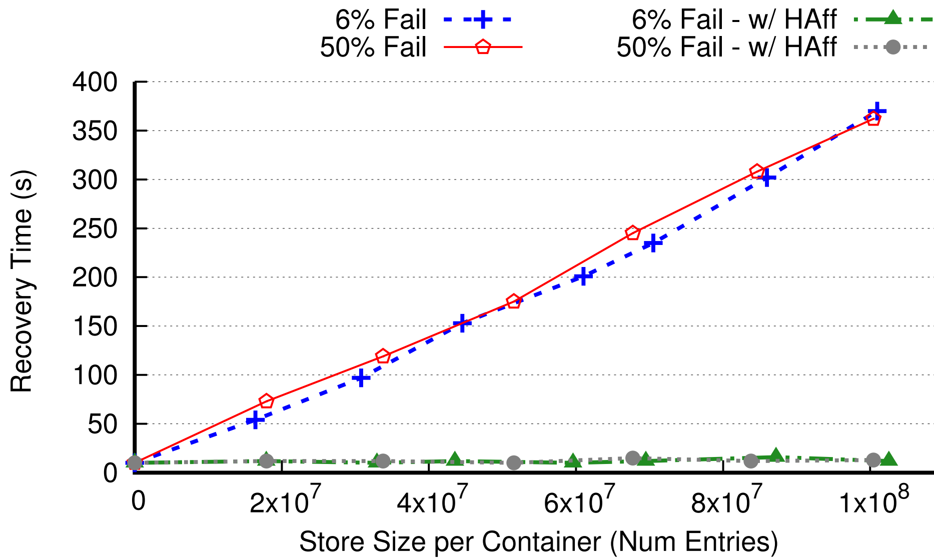


Figure 3.11: Failure recovery time using different store sizes with and without Host Affinity (HAff). The results are with a ReadWrite workload when 6% and 50% of the containers fail.

3.6.4 Reprocessing

We analyzed the impact of reprocessing in our production jobs. We evaluated the Standardization job, our most frequently reprocessed job, over a 24 hour period. Standardization consumes profile updates and using a machine learning model, transforms the update to a standardized text. After 2 hours, we started reprocessing the entire database of user profiles ($>$ 450 Million entries). Simultaneously, we scaled-out the job from 8 containers to 24.

Figure 3.12 shows the reprocessing throughput. Reprocessing peaks and remains at 10,000 messages per second (due to our throttling mechanism). After all the data is processed ($\approx$16 hours), reprocessing throughput drops and starts catching up with the real-time data. At this point, we stop reprocessing and scale-in the job. Reprocessing time can be reduced, similar to a batch job, by simply allocating more resources. The combination of scale-out and throttling mechanisms ensure that reprocessing does not affect real-time processing performance.

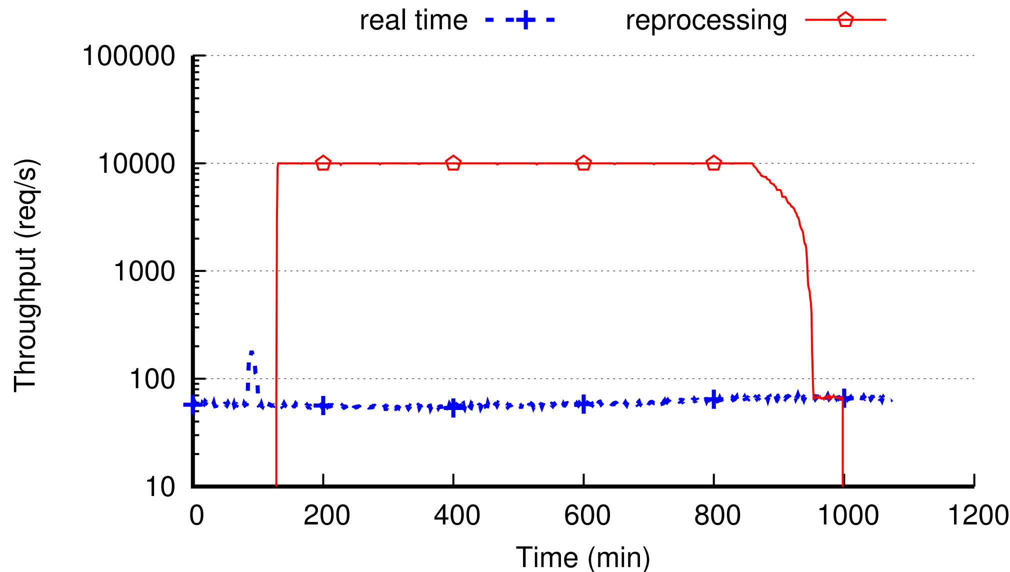


Figure 3.12: Throughput of Standardization job while performing a reprocessing (starting after 2 hours).

Batch Processing using Samza

We compared Samza's reprocessing/batch solution with other mainstream batch processing solutions including Spark and Hadoop [38, 39]. Spark offers high similarity of code for batch and stream processing, thus making it a near-Lambda-less architecture. Hadoop is a system that might be used modularly inside the Apache Beam architecture.

In this experiment we used Members Per Country (MPC), a real-world batch job running at LinkedIn, and reimplemented the job in Samza (with HDFS consumer). MPC reads a snapshot of all user profiles, groups them by country (Map), and then counts the members in each country (Reduce). We used 450 million profile records stored across 500 files (250 GB of data) in a production YARN cluster ($\approx$ 500 nodes), and single core containers with 4GB RAM.

Figure 3.13 shows Samza has better throughput than Spark and Hadoop[5]. This is because it streams data to downstream operators as soon as it becomes available, while Hadoop and Spark (in batch mode) are limited by the barrier between Map and Reduce phases [99] [6].

[5]Latencies are higher in Hadoop due to the barrier and Spark due to micro-batching; these are not plotted.
[6]Samza is also able to exploit more parallelism than the other frameworks, better utilizing CPU cores.

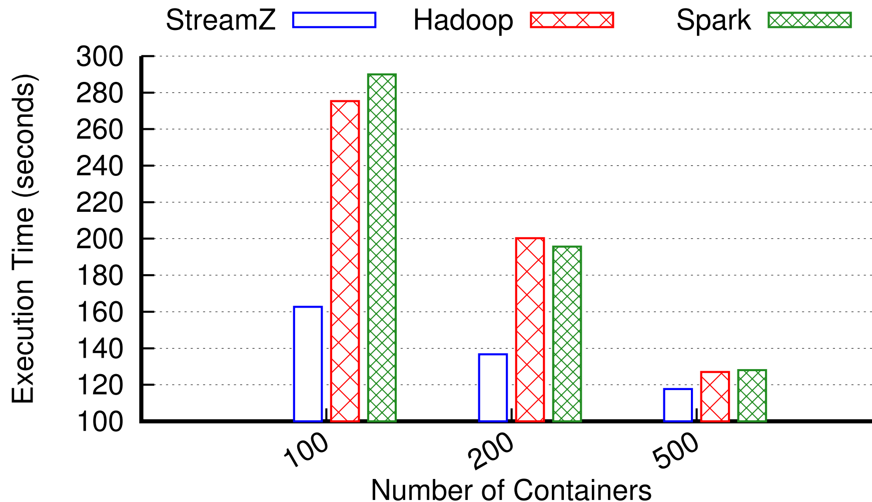
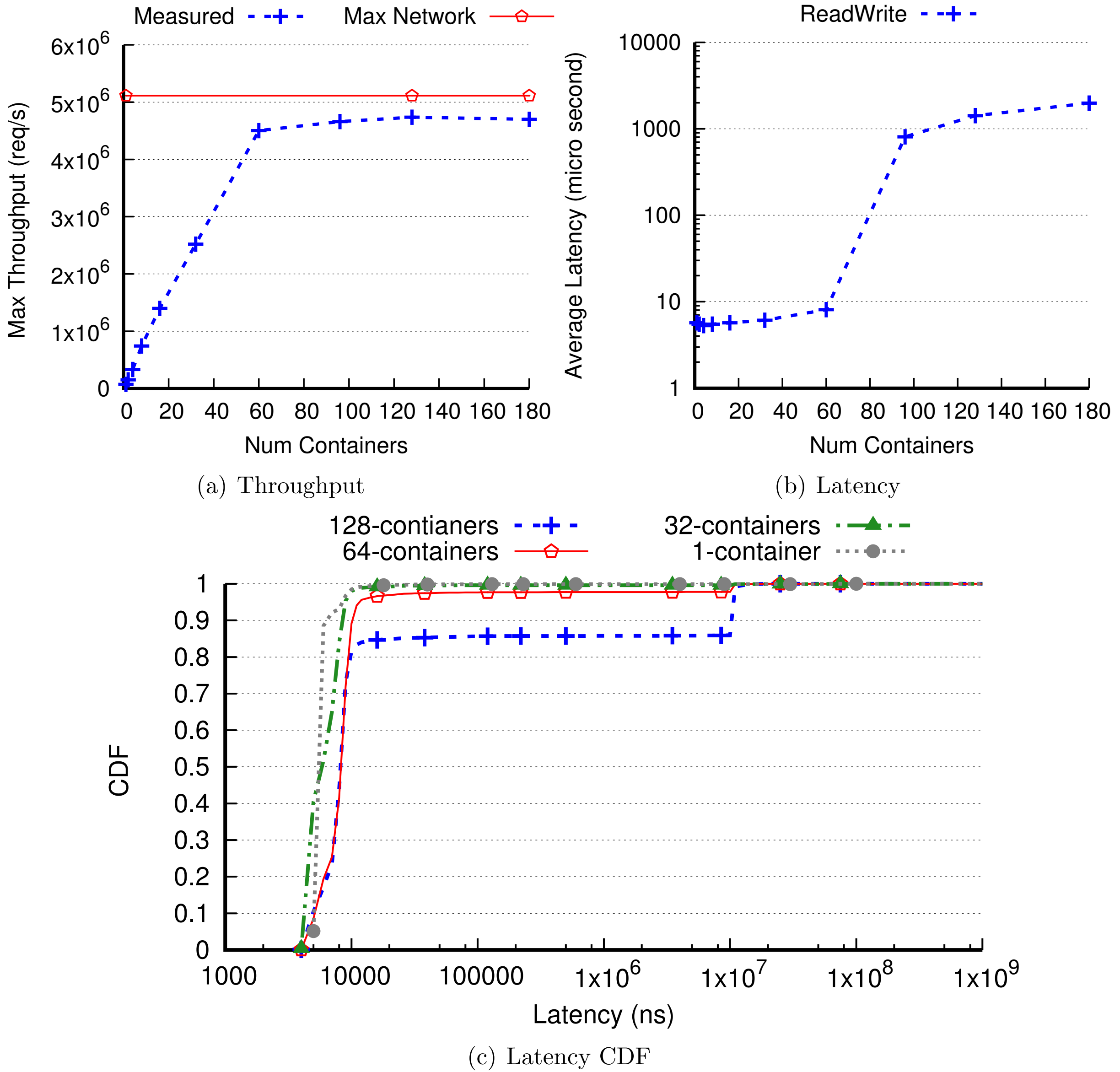


Figure 3.13: Comparison of Samza with other batch processing frameworks in reprocessing data.

3.6.5 Scalability

Figure 3.14 shows maximum throughput and average latency in Samza as the number of containers increases (in the ReadWrite workload). Throughput increases linearly, saturating just beyond 60 containers. The saturation point is very close to the optimum throughput possible in the network. Latency stays low at first and increases thereafter. This knee of increase in latency coincides with the throughput saturation, and thus, can be used as an indicator of when to stop scaling. For maximizing throughput, there is low marginal utility in scaling beyond the saturation point.

Figure 3.14(c) shows the CDF of the latency. Even with twice more containers than needed, a majority of messages ($> 80\%$) are processed within a few microseconds, and with small variance. The tail prolongs when containers are beyond the throughput saturation point, primarily because more time is spent waiting for the next events than processing them. We also observe that latencies are higher with more containers (e.g., 128 vs. 32). This is because the latency is calculated from message fetch time to processing completion. With more containers, more outlier messages need to be fetched remotely, and this drives up the average.



(a) Throughput

(b) Latency

(c) Latency CDF

Figure 3.14: Throughput, average latency and CDF of latency in scalability study adding containers, under a 50-50 read write workload using local state. The saturation point of the system is 64 containers.

3.7 RELATED WORK

State management: State management varies significantly among stream processing solutions. Many industrial scale systems, such as Apache Storm, Heron and S4 [71–73] are built with no support for state. Trident and MillWheel [35, 37] manage state by using a combination of "hard state" persisted in an external store along with "soft state" stored in memory as a non-fault-tolerant cache. Thus, they either incur high overhead by relying on a remote storage or accept the chance of losing data.

There has been some work on partitioning state similar to the idea of local state [33, 36]. StreamCloud [100] discusses elastic and load-balanced state partitioning. However, partitioning is only supported for specific operators (join and aggregation) and it does not address fault-tolerance. S-Store [101] proposes transactional state management for stream data that is a potential add-on to Samza.

Fault-tolerance in local state: Upstream backup recovery [71, 77] successfully restore processing, but not the state. One approach to add fault-tolerance is by using replication [77] (as studied in [102]). However, this requires the luxury of extra available resources [75], and approaches like Sweeping checkpointing [103] do not ameliorate this problem.

Another popular approach is continuous full-state checkpointing of state along with input replay in presence of failures. Fernandez et al. [75] discuss scale-out state management for all operators by partitioning state and using checkpoint. Many others [33, 36, 104–106] also employ checkpointing mechanism to ensure fault-tolerance. The SDG approach [94] enables asynchronous checkpointing by locking the state, keeping a dirty buffer for incoming changes during checkpointing, and then applying the dirty buffer on the state. [105] generates a global snapshot by using a blocking variation of Chandy-Lamport snapshot [107] where it blocks on on-the-fly messages before generating the snapshot. Instead of blocking, IBM System S [106] persists checkpoints in the external DBMS (which is slow), and [75] captures pending asynchronous operations as part of the state (which is complex). The excessive overhead of

full-state checkpointing, especially with large state sizes, make these approaches prohibitive. Sebepou et al. [108] partition state into smaller chunks, with incremental updates. However, it was only evaluated for aggregation operators, and it is unclear how effective it will be on user-defined logic.

Unified stream and batch: MapReduce Online [99] has explored processing batch jobs in an online barrier-free manner, but they do not fully support stream-processing. Liquid [109] also has a unified integration stack, but still maintains two separate subsystems.

Apache Beam, Dataflow, and Flink [33, 74, 90] have moved toward integrating batch into stream as a unified environment. Dataflow and Borealis [74, 77] have investigated how to handle inaccuracies caused by out-of-order messages occurring in stream frameworks. However, Dataflow relies on a remote store (not handling large state efficiently), and Flink is not fully unified (separate APIs for batch and stream). Samza can be used modularly inside Beam which acts as a wrapper API. Besides, Dataflow and Beam incur extra overhead by not leveraging the inherent partitioning capabilities of systems like Kafka, Kinesis, or EventHub. Spark Streaming [36] also has a unified environment, however, it processes data in micro-batches incurring higher processing latency. Also, Flink and Spark Steaming are not available as a standalone version and lose the deployment flexibility.

Scalability: Scaling to large state necessitates going beyond relying on memory, e.g., by using disk spilling [110]. This is orthogonal to our approach and could be used as an extra optimization in Samza. For better scalability, operators need to work with maximum independence. Thus, many systems have opted to use reliable, replayable communication mechanisms to handle data buffering between operators, e.g., Streamscope and Heron [72, 104]. IBM System S [106, 111] utilizes fault-tolerant replayable communication and distributes operations into a set of independent component-local operators. These systems deploy a similar approach to the scalable design in Samza. However, none of them target large state or reprocessing.

Table 3.4: Placement and data-motion optimizations in Samza

Requirements	Placement	Data Motion
- large state handling - low-latency processing - fast failure recovery - batch as a stream (lambda-less) - continuous scaling	- abstracting operations and placement - locality-aware task placement - sticky placement on restart (Host Affinity) - scaling at spikes (batch as a stream)	- locality aware (local) storage - lazy background replication - batching updates - background compaction - abstracting inputs and data flows - caching in memory - throttled processing at spikes (batch as a stream)

3.8 CONCLUSION

This chapter described Samza, a distributed system that supports stateful processing of real-time streams, along with reprocessing of entire data streams. Samza recovers quickly from failures, with recovery time independent of application scale (number of containers). It can support very large scales of state in spite of limited memory, by combining local on-disk storage, an efficient changelog, and caching. Table 3.4 summarizes the techniques used for data placement and data motion in Samza and the requirements they satisfy.

Our experiments showed Samza has higher throughput than existing systems like Spark and Hadoop. Samza runs both batch and stream processing in a unified way while minimizing interference between them. We also described several applications that rely on Samza.

Samza's approach opens up many interesting future directions including: dynamic rebalancing and task re-splitting (changing number of tasks), automatic configuring and scaling of resources (containers), investigating stragglers (not a major issue so far), and handling hot vs. cold partitions.

CHAPTER 4: STEEL: UNIFIED AND OPTIMIZED EDGE-CLOUD ENVIRONMENT

In this chapter we discuss Steel, an unified and optimized framework for building edge-cloud applications. Internet of Things (IoT) applications have seen a phenomenal growth with estimates of growing to a 25 Billion dollar industry by 2020. With the scale of IoT application growing and stricter requirements on latency, edge computing has piqued the interest for such environments. However, the industry is still in its infancy, with no proper support for applications running the entire edge-cloud environment, and an array of manual tedious per-application optimizations. In this work, we propose Steel, a unified framework for developing, deploying, and monitoring applications in the edge-cloud. Steel supports dynamically adapting and easily moving services back and forth between the edge and cloud. Steel is extensible where common optimizations (but crucial for the edge) can be built as pluggable and configurable modules. We have added two very common optimizations: placement and adaptive communication, to cope with both short and long term changes in the workload and environment.

Steel is integrated with Azure Cloud services. Based on our evaluation, we reduce the initial development effort (1.7x–3.5x reduction in lines of config) and support dynamic moves with minimal changes ($\sim$2 lines of config per move, reducing 95% of the overhead). Our placement optimizer (added with only 500 lines of code) reduces cost (by up to 40%) and improves edge resource utilization (by up to 75%), and our communication optimizer reacts quickly to network or workload changes (within less than a second).

4.1 INTRODUCTION

Internet of Things (IoT) solutions are growing more popular and disruptively in scale. Gartner Inc. estimates that the number of deployed IoT devices will grow from 5 Billion in 2015 to 25 Billion in 2020 creating a multi-billion dollar market [112, 113]. This growth

has led to many industrial services offering to simplify development and management of IoT applications, e.g., C3 IoT and IoT frameworks from all major cloud providers [114–117]. Initially, AWS, Azure and Google offered managed solutions based completely in the cloud. An IoT management service (such as IoT Hub from Azure) securely manages the remote devices, their configuration, and receives the sensor data. The rest of the processing pipeline is built using regular cloud services for computation (Azure Function, or Stream Analytics), communication (Event Hub), and storage (CosmosDB, Blobs).

As the IoT solutions scale up in the number of devices and messages sent, there is greater need to process data closer to the IoT sensors using the edge. Processing on the edge has several advantages including: reducing end-to-end latency especially when the application is actually controlling the devices (e.g., shutting down operations in case of failures), providing continuation of service despite low connectivity to the cloud, reducing the bandwidth usage, and reducing monetary cost of using cloud services.

However, the IoT industry is still in its infancy. Building an IoT solution comes with a lot of challenges and complexities in deployment, monitoring, and optimizing the end-to-end solution. Even a simple remote monitoring application will consist of several components on the edge to receive, preprocess, and send the data to the cloud along with services in the cloud for additional processing, machine learning, publish-subscribe systems, and storage. Deploying such an application requires a lot of error-prone and complicated scripting. Since each cloud service has grown organically and independently, there is a great diversity among them in terms of their cost models, monitoring, API, etc, and this diversity has created many compatibility constraints across services.

Moreover, naive implementations of IoT applications can often be very inefficient both in terms of cost and performance. For example, processing all in the cloud a) requires a lot of bandwidth to send sensor measurements (typically encoded as JSON strings), and b) becomes expensive for large datasets. Therefore, developers manually optimize the processing pipeline by batching data for upload, compressing data, and determining component

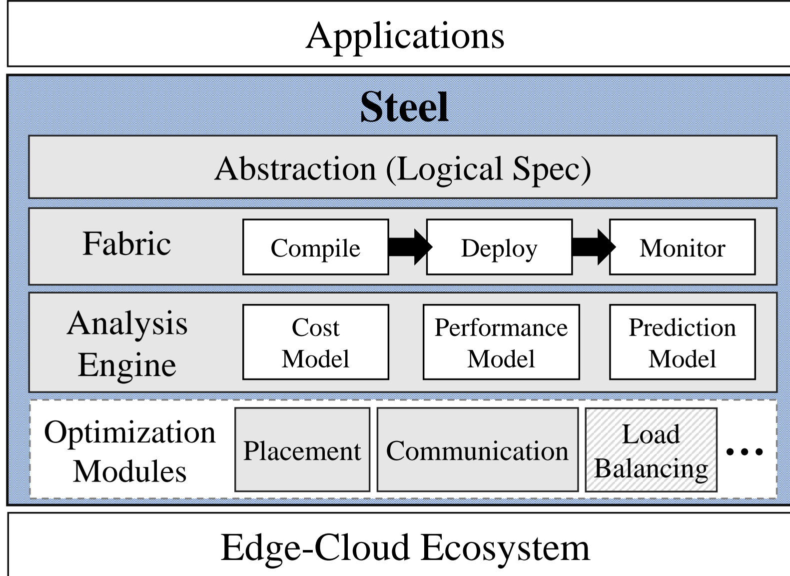


Figure 4.1: High level architecture of Steel, a middle layer between the applications and edge-cloud environment.

placements. Making these decisions is non-trivial because the best configuration depends on the available resources, the workload, and accurate estimations of a new configuration. Also, companies deploy a wide range of heterogeneous edge devices [118] (from Raspberry Pi to large servers) and use different connectivity options (optical, WiFi, cellular, or satellite) further complicating this decision.

We present a system, Steel, that allows users to declaratively describe the IoT application, while managing its deployment and monitoring and automatically optimizing it for performance and cost. As shown in Figure 4.1, Steel acts as a middle layer between the application and the edge-cloud environment. Steel consists of four main components: 1) a high level *abstraction* to describe location agnostic applications, 2) a *fabric* materializing and deploying the application, 3) an *analysis engine* building analysis and prediction models of the application, and 4) a set of *control modules* that optimize the performance and cost of the application in response to changes in workload, resource demands, or failures.

Instead of directly deploying individual modules, in Steel, developers describe their application in a logical abstraction. The abstraction defines the source data for the application

(e.g., all temperature sensors in building 43), the processing modules (e.g., computing average temperature on each floor), and how the modules are connected into a directed acyclic graph (DAG). The IoT Fabric takes the abstraction as an input, adds other necessary components (such as data compression and decompression) and deploys the application automatically. The IoT Fabric also monitors all executing components and reports performance and resource usage metrics.

On top of the fabric, the analysis engine consumes monitoring metrics and builds cost and performance models across the operations in a application. These models along with a predicting model are used to answer *"what-if"* questions.

Then, using the analysis engine, Steel runs several control modules. These modules monitor the running IoT application and adjust its configuration at runtime in response to changes in workload and environment. In this work, we focus on placement and communication optimizations. However, Steel provides a pluggable and extensible framework for adding other control modules such as a load-balancer.

The main contributions of this work are:

- Steel, an integrated framework for automated compilation, deployment, and monitoring of declaratively specified complex IoT applications, across entire edge-cloud environment

- global optimization of component placement across edge and cloud environments in response to changes in workload and resource availability based on *shadow profiling* and *cloud cost prediction*,

- adaptive data compression policy that periodically *adapts data batching, encoding, and compression* based on available CPU and bandwidth resources and uses *efficient JSON parsing.*

We implemented Steel on top of an IoT edge/cloud stack from one of the leading cloud providers (Azure IoT stack) and demonstrate that we can deploy realistic IoT applications

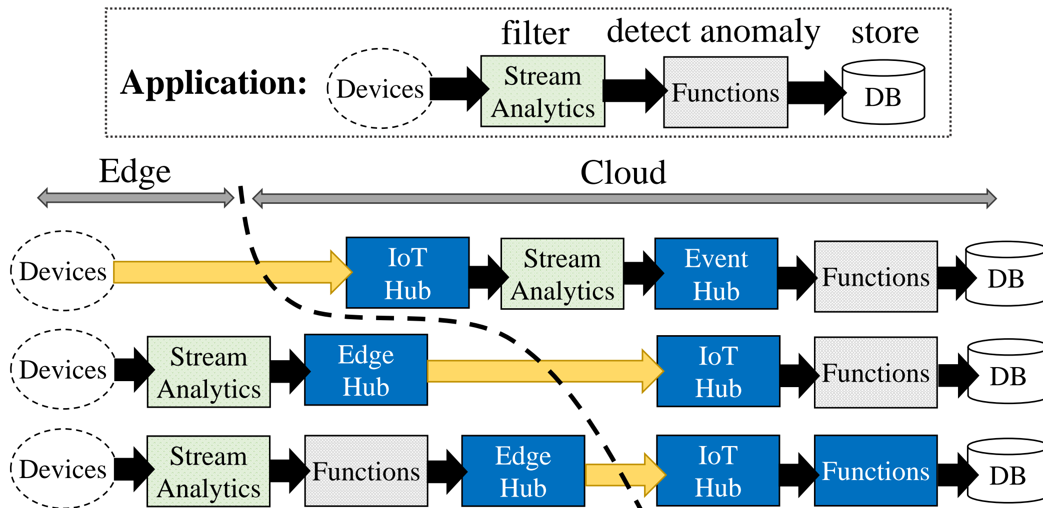


Figure 4.2: Sample anomaly detector IoT application.

from a simple job spec. We evaluate our system with set of 6 diverse real-world production applications. Based on our results, we reduce the number of lines of config (*loc*) for the initial application development by 1.7x–4.8x, and in presence of a change or move, we reduce the changes to ∼2 *loc* per move, a >95% reduction. Our placement optimizer (added with only 500 lines of code) reduces cost (by up to 40%) and improves edge resource utilization (by up to 75%), and our communication optimizer effectively adapts to dramatic changes to network within less than 1 second.

4.2 MOTIVATION AND BACKGROUND

Steel's target are large scale enterprise IoT scenarios using the cloud with several sensors (millions) and edge devices (hundreds). The sensors usually produces data in the form of time series, i.e., a stream of (time stamp, data point) tuples. These applications have a common pipeline: measure and send data, process, store and visualize. This pipeline is typically constructed of multiple cloud services, connected to each other in form of a directed acyclic graph (DAG). The processing is commonly from a small set of services, for example, in Azure the typical services are Stream Analytics, Functions, Machine Learning, and Storage [15, 114, 119] (equivalents used in Google Cloud Engine [120]). Figure 4.2 shows

a simple anomaly detector application built using Azure services. This application reads sensor data, filters values (Stream Analytics), detects anomalies (Functions), and writes the results to a database (Document DB).

There data enters the cloud through an entry point service, e.g., IoT Hub in Azure or AWS Greengrass. This entry point reliably routes the data to the appropriate cloud service(s). In case of using an edge, data first is sent to communication service on the edge, e.g, Edge Hub in Azure, which in turn communicates with the cloud. At the edge, for portability (being able to quickly move services) and isolation across services, it is common to run each service in a separate container [89, 121–124].

For the rest of this chapter, we focus on Azure, one of the leading cloud service providers in the IoT space. We use Azure IoT hub in the cloud and Azure IoT Edge at the edge. However, the problems and challenges outlines are a fundamental problem applicable to other platforms as well.

4.2.1 Challenges

Using the cloud for these enterprise IoT applications is a trending and well-established option given the many scalable, reliable, and flexible cloud services available. However, porting these applications to work across this emerging edge-cloud environment brings up a number of new challenges. Bellow we mention a few of the major ones.

Dynamic Environment at the Edge

Edges also have limited resources (for economic and practical reasons) while workloads are dynamic. Thus, the only cost-effective, peak-tolerant option is to adaptively balance and move services. Also, edges are faulty, with more sources of failure (e.g., a power outage), and longer recovery procedures, necessitating dynamic movement and replication for continued availability. Furthermore, the connection from the edge is poor, costly, and intermittent, demanding dynamic batching and compression [125–127]. To enable these optimizations, it is

essential to be able to *readily move* services back and forth between various edge deployments and the cloud and *dynamically adapt* to the environment.

Configuration, Deployment and Monitoring

Manually developing, deploying, and monitoring these multi-service applications is hard, time-consuming and error-prone. First, most cloud services have each developed organically and independently, therefore, they exhibit wide diversity and suffer from inter-service compatibility constraints. These compatibility issues can be resolved by adding intermediate *glue services*. However, the glue services depend on the location of services (edge or cloud). For example, in the anomaly detector (Figure 4.2), depending on the location of the different services, the glue services required (shown in blue) change substantially. Manually configuring these glue services to support migration is intractable because it is error-prone and cannot be done in real time.

Second, the deployment process is complex. Deploying the anomaly detector application in Figure 4.2 (all in cloud) through the web portal takes roughly 25 minutes for an expert, and building a deployment template requires >250 lines of config. Although this overhead can be acceptable as a one-time effort, the conventional case for cloud-only applications, it becomes a major barrier to dynamic movement of services between the cloud and the edge. Listing 4.1 shows a sample Azure template required just for deploying (without fully connecting) a Document DB service. The template requires a verbose specification of the service's properties, such as its detailed location and unique name, on lines 1–16. Lines 17–27 specify the output values that should be passed back to the user after the service is deployed and cannot be generated beforehand, e.g., the unique ID and URI that must be known to applications and services that will connect to the database.

Furthermore, after deployment, each service within an application is monitored independently. There is no global view of the entire application and end-to-end monitoring across all services, especially for the parts running in the edge.

Listing 4.1: "Sample Azure Config for deploying a simple Document DB service"

```
1 {  "$schema": ``http://schema.management.azure.com/...'',
2    ``contentVersion'': ``1.0.0.0'',
3    ``resources'': [{
4       ``apiVersion'': ``2015-04-08'',
5       ``kind'': ``GlobalDocumentDB'',
6       ``type'': ``Microsoft.DocumentDb/databaseAccounts'',
7       ``name'': ``test-name'',
8       ``location'': ``westus'',
9       ``properties'': {
10         ``databaseAccountOfferType'': ``Standard'',
11         ``locations'': [
12         {``id'': ``test-id'',
13            ``failoverPriority'': 0,
14            ``locationName'': ``West Us''}]},
15      ``tags'': {``defaultExperience'': ``DocumentDB''}
16   }],
17   ``outputs'': {
18      ``EndpointUri'': {
19         ``value'': ``[reference('...').documentEndpoint]'',
20         ``type'': ``string''},
21      ``PrimaryKey'': {``value'': ``[listKeys(...)']',
22         ``type'': ``string''},
23      ``resourceId'': {
24         "value": ``[concat('subscriptions/', subscription().subscriptionId,↩
             ...')]'',
25         ``type'': ``string''}
26   }
27 }
```

Heterogeneity and Diversity

There are many sources of diversity and heterogeneity in both the cloud and edge [7–10]. Cloud services have very diverse models and structure. As a result, they exhibit a wide diversity in their *pricing models* with varieties in: granularity, metrics to charge (bytes, CPU, etc.) with some combined opaque metrics (Streaming Units for Azure Stream Analytics), time scales, and provisioning model. Table 4.1 shows a few of these aspects across three different Azure services. This diversity complicates cost prediction, especially in case of a move where hidden glue components are added and removed.

Table 4.1: Pricing models of a select Azure services.

System	Scaling Model	Primary Metrics	Granularity	Time Scale
IoT Hub	manual provisioning	incoming messages	400K messages	daily
Azure Functions	dynamic scaling	exec. time, memory footprint	100 ms, 128 MB	function invocation
Stream Analytics	manual provisioning	Streaming Units	1 Unit	hourly

The edges also exhibit great diversity in the hardware, network links, and locations (proximity). Edge hardware ranges from raspberry pi (or even smaller) to large servers. This diversity complicates predicting the resource demand and performance of a move, e.g., whether a service running in the cloud can be moved to a specific edge. The limited resources on the edge make running and profiling a target move an expensive and intrusive operation. Since the network is commodity networks, the bandwidth is variable and the connectivity is intermittent, both across edges and time.

Manual optimizations

Naive implementations of IoT applications can be inefficient and unable to cope with dynamism of the edge. When using the edge, many optimizations, that are usually optional in the cloud, become crucial. Examples of such optimizations include placement, load balancing, adaptive communication, and fault-tolerance. Individual edge deployments and their networks are heterogeneous and diverse in terms of cost, performance, availability, etc., generating a large search space of potential placements. The network is also limited and variable. Naively sending measured data as small JSON messages, as typically done today, wastes the communication link. Currently, application developers have to manually incorporate these optimizations, which is a significant amount of work, without the ability to reuse them. However, ideally, these should be modular pluggable features.

4.3 DESIGN OF STEEL

Steel's goal is to simplify a) developing multi-service applications, b) deploying them across the entire environment, c) dynamically adapting and moving parts of them, and d) applying common optimizations to them. We found the placement and communication optimizations as the main optimizations for IoT, and therefore the main focus of this work. Steel acts as middle layer between application developers and the edge-cloud, hiding the underlying complexities. As shown in Figure 4.1, Steel consists of four main layers: abstraction, fabric, analysis engine, and optimization modules.

4.3.1 Abstraction

The abstraction includes: a) the *logical DAG* of the application, i.e., the main services and their connections excluding the glue services and b) the *location* of each service. We argue this information is sufficient to support adaptable and movable services across the entire environment.

Listing 4.2 shows the anomaly detector (Figure 4.2) in this abstraction. As shown, it provides full flexibility and extensibility of the internals of each service. Each service can be defined similar to using the typical templates, e.g., using a SQL query as a streaming job or setting service specific configs like "Streaming Units". This abstraction gives flexibility and transparency for placement. To move a service, simply, the location mapping needs to be updated.

4.3.2 Fabric

The Fabric materializes the abstraction of an application into an actual physical deployment. The Fabric *compiles* the abstraction, *deploys* it across the edge-cloud environment, and *monitors* it end-to-end. Figure 4.3 shows this pipeline and the transformation steps.

Listing 4.2: "Sample Application in Steel's Abstraction"

```
1{  "Name": "test job",
2"Services": [
3{"Name": "filter",
4"Type": "AsaJob",
5"Query": "SELECT * FROM input0 WHERE tmp > 60"},
6"Streaming Units": 10
7...
8],
9"Connections": [
10{"From": "filter", "To": "anomaly detector"},
11{"From": "anomaly detector", "To": "anomaly db"} ],
12"Locations": {
13"filter": {"Type": "edge","Id": "dev1"},
14"anomaly detector": {"Type": "cloud", "Id": "westus"},
15"anomaly db": {"Type": "cloud","Id": "westus"}
16}  }
```

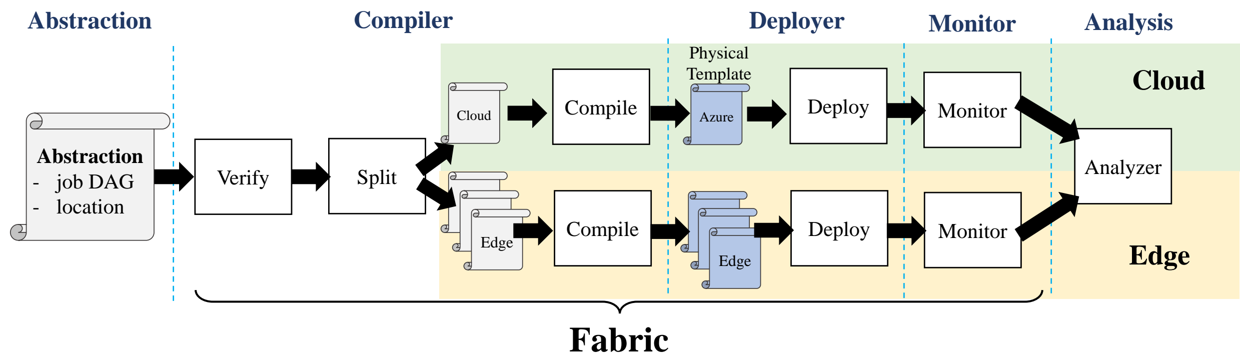


Figure 4.3: Compilation and deployment pipeline in the fabric

The fabric pipeline is split to a cloud and edge path. Since the cloud and the edges exhibit very different behaviors, they are treated and managed separately. Clouds have well-established resource mangers and deployment templates for instantiation of services (e.g., Listing 4.1 is a sample for Document DB service in the cloud) along with already defined APIs for metric collection. At the edge there is a number of emerging libraries and frameworks with their unique (different from the Cloud) APIs for starting services and metric collection. Steel uses the Azure IoT Edge [122] at the edge because of its better compatibility. However it can be easily and modularly extended to other options, e.g, AWS Greengrass [128]. We describe each step in more detail below.

Compiler

The compiler converts the abstraction to detailed deployment templates, based on the location of each service. First, it verifies the location settings are valid, both preventing misconfiguration and also imposing user-specified constraints. Some services cannot move to the edge or cloud, e.g, a permanent storage on the edge, and there can be user-specified constrains, e.g., service A and B should be placed together. Second, the overall job abstraction and DAG is split into sub-jobs per location i.e., the sub-DAG running at that location, and treated in separate edge and cloud paths.

Finally, the job is compiled into physical templates. In the compilation, glue services are added to fix the compatibility constraints and routes are set-up. Figure 4.2 shows a simple 3-service application, and how the glue components (shown in blue) change based on the location settings. The routes and connections are configured to connect services, including glue services, and the actual deployable templates are built. These template differ based on whether they are targeted for the edge (edge version wrapped as containers) or the cloud (Azure templates). The compiler masks these complexities.

Deployer

The deployer deploys the templates across the entire environment. Using the application's DAG of services, it automatically detects dependencies among services to ensure that a service is deployed only after its dependencies, and deploys these services stage-by-stage, with multiple parallel deployments within each stage. For example, a Functions instance connected to a Document DB needs to know the endpoint URI of the DB service (which is only available once the DB has been deployed), and therefore, must be instantiated in a stage after the DB. The deployment is distributed and done independently at each location, thus, resulting in parallel deployments across locations.

Monitoring

Similar to cloud-based tools [129–133], the monitoring gathers real-time metrics for services, but across the entire edge-cloud environment. It digests and unifies the metrics in an end-to-end application level, rather than per service. We collect 4 main categories of metrics: a. resource usage metrics (CPU, memory, network, etc.), b. message flow metrics (input rate, output rate, process rate), c. latency metrics: processing latency, idle latency, etc., and d. cost metrics. We use a combination of already provided metrics by services and our own instrumentations to collect these metrics.

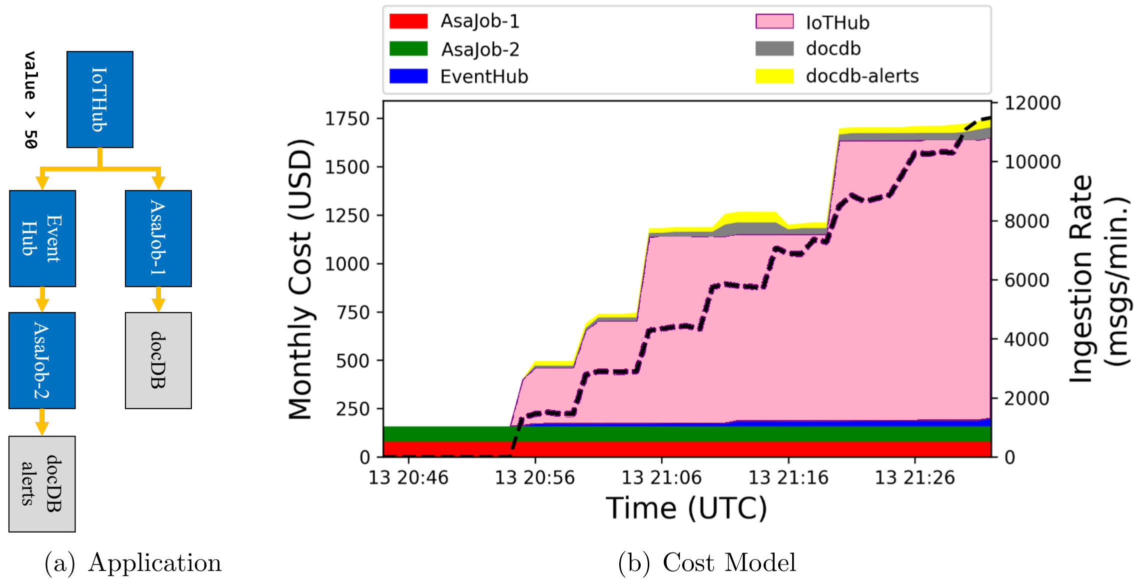
The monitoring hides the diversity of cloud services (each with their own metrics and models), and heterogeneity of the edge devices. For example, in terms of cost, it hides the differences of pricing structures across services: the different metrics (bytes, CPU, IO), granularities, reserve costs, etc. Monitoring is crucial for checking the health of the program, and also a major piece for further analysis such as for bottleneck detection, allocation and placement optimizations, load balancing, adaptive communication, etc.

4.3.3 Analysis Engine

The edge-cloud environment exhibits wide diversity, both across the edge devices and the cloud services. The analysis engine aims at hiding these complexities, and providing a holistic view of the application for further analysis. In particular, it builds models for cost and performance of the different components in an application, along with a prediction model to predict effect of changes (e.g., moving a components) and answer "what-if" analysis question.

Cost Model

As mentioned in Section 4.2.1, services have diverse pricing structures with different metrics and granularities across services. In addition, there is the cost of both a) the network



(a) Application (b) Cost Model

Figure 4.4: Cost model over time of a simple alerting application

links between edges and the cloud, and b) the hidden cost of glue services (depending on the location). This component builds cost models for the various components of an application. At runtime, depending on the placement and using the metrics gathered by the fabric, it computes the monetary cost of running an application. This component can also be used at development time with predefined metrics to estimate cost of running an application, before actually deploying it.

Figure 4.4 shows the output cost model of a simple alerting application. This application filters incoming data (detecting alerts) using a Stream Analytics job, and stores the raw data and alerts in two different stores, accordingly. We linearly increase the incoming rate of messages (dotted line), and the cost models adjust accordingly. As seen the various services demonstrate different behaviors: a) some services are more sensitive to load, b) the cost growth is a step function rather than a smooth linear one, and c) the step function follows the cost model of the service demonstrating irregular steps, typically due to the lower amortized cost at higher scale.

Performance Model

When using the edge, the environment is heterogeneous at its extreme. A typical enterprise deployment has hundreds or thousands of edges, with many different types, architectures, and computation powers, ranging from a small Raspberry Pi to a large multi core machine. The goal of the performance model component is to predict performance of running a component on a (new) edge, i.e., how much resources are required to run that component. It builds performance models across the entire application, also detecting bottlenecks within the application DAG. Performance models are built using a combination of *modeling* and *profiling*.

Modeling: Similar to [134,135], these models try to normalize the resources at the edges using resource vectors, e.g., $\vec{Edge}_1$ = (1.5 cpu, 2GB mem, 1 TB disk, ...). Then, using the performance characteristic of a component on one edge, they predict the performance on another edge (based on normalized resources).

While normalizing and modeling work well for most resources, it performs poor in normalizing the CPU with more sources of unpredictability [9, 10, 136]. For example, if an application uses 1 core in one machine, it would not simply use 0.5 core on another machine that has a CPU specification that is 2 times faster. The performance depends on various factors such as the architecture, the cache sizes, the type of operations, etc., which are not captured in a resource vector model. Therefore, along with modeling, Steel uses profiling of actual runs to build comprehensive performance models.

Profiling: Steel employs a *shadow profiling* approach. In order to build a performance model of a component C on a given edge $edge_i$, it starts a shadow instance of the component C_{shadow} on the edge $edge_i$. A copy of the input traffic of C is redirected to C_{shadow}. Using the performance metrics gathered by the fabric a performance model on C on $edge_i$ is built.

Edges have limited resources and are not scalable. When using the cloud, an enterprise or company can continuously scale by launching more instances on-demand, however, at the edge the enterprise is limited to whatever resources they already own. These resources should

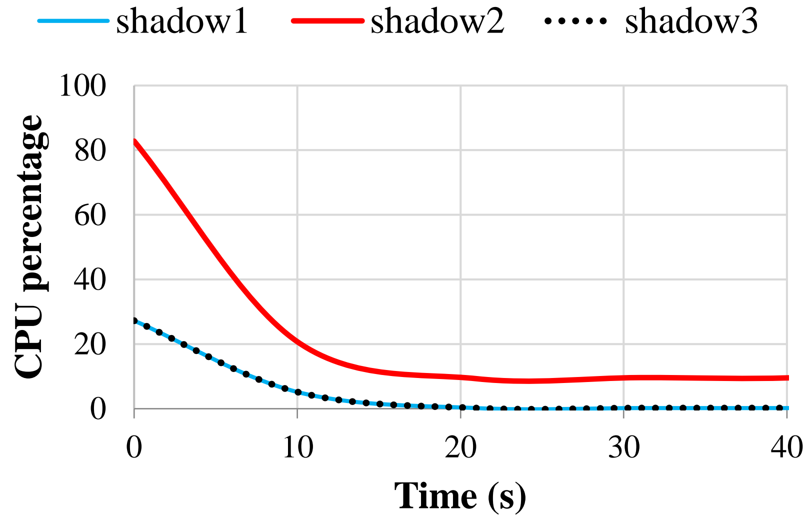


Figure 4.5: Performance monitoring over time (CPU usage) for 3 shadow components.

be shared between the actual computation running on the edge and for profiling. Profiling is also noisy and variable requiring a run over a larger window until the computation stabilizes. Figure 4.5 shows metrics collected during a shadow profiling, where it takes a couple minutes for the shadow profiling to stabilize. Therefore, Steel runs a sampled profiling. Profiling is assigned a constant and limited share of the edge resources. The input data is randomly sampled and sent to the shadow component. The sampling ratio is gradually increased until the fixed profiling cap is reached, where the performance model is extrapolated from there (for a 100% sampling rate).

In addition, to reduce the overhead of profiling attempts, edges with similar hardware specifications are grouped into categories, e.g, Raspberry Pi as one category. Profiling is done per category, instead of per edge. Results from similar historic runs are used to build models for current runs[1].

[1]Steel currently uses the average of 3 historic runs. As part of future work, we plan to adaptively choose the number of required historic runs based on the variance in the runs.

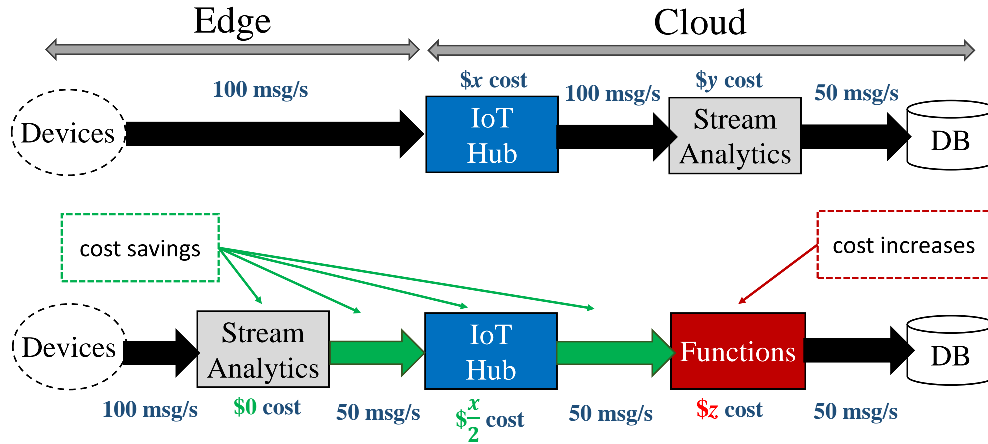


Figure 4.6: Profile prediction example of a component move

Profile Prediction

The analysis engine enables "what-if" analysis, i.e., to predict impact of changes without actually applying a change. Examples of such changes are moving a component, enabling compression, changing batch sizes, or scaling a job. The profile prediction uses the cost and performance models to enable this.

The profile prediction relies on the fabric (specifically, the compiler) to predict the new physical layout of a job given a change, e.g., the new glue components added. It populates profiles of current layout to the new layout for unchanged components; propagates profiles and data-flow metrics accordingly (rates of data across the edges); predicts profile for missing components and edges (particularly, new glue services); and uses cost models to predict the new cost. The abstraction and transparent compiler provided by the fabric, along with its end-to-end view of the application are an essential part to building this analysis engine, as they remove the manual configuration and compilation otherwise required. Figure 4.6 shows a sample application and its profile, and the output of the prediction engine from analyzing the move of a component. The prediction engine takes into account changes into many factors including the new physical layout, glue components added and deleted, the network savings of sending less data, and the entry point (IoT Hub) saving of processing fewer messages.

4.3.4 Optimization Modules

Inherently, the edge comes with many complexities such as increased chances/causes of failures, higher recovery time, poor and variable network links, extreme heterogeneity among resources, and unbalanced distribution of workloads over space and time. These complexities give rise to the need for additional, but common, optimizations. Steel's design enables the modular, pluggable, and extensible implementation of these optimization. Below, we list a few of the common optimizations, and how they benefit from Steel's Fabric, abstraction, and design.

- **Communication:** Edges are often characterized by limited bandwidth or intermittent connectivity to the larger Internet, such as at a remote oil refinery, making the network a critical factor for performance. Also, both the network and the entry point at the cloud have a high cost, e.g., the Azure IoT hub (entry point to Azure) charges per message or cellular networks charge Byte. To this end, it is common to optimize the communication by using smart batching, compression, and compaction. Steel's connection abstraction enables these optimizations to be done behind the scenes, in a pluggable and configurable manner, and its unified monitoring enables intelligent optimizations for both cost and performance.

- **Placement:** Applications running across the edge-cloud need a placement plan: Which parts of the application should be placed on the cloud and which on the edge? Which edge to choose? In an edge deployment, the search space of potential placements grows significantly. Edges are heterogeneous (from a Raspberry Pi to multi-GPU machines), large in number ($\sim$ 1000s of edges in a typical factory), and with diverse network bandwidths. An application is distributed across the edge and cloud. The optimal placement depends on the workload, available capacity, cost of bandwidth, etc. Manually deciding the placement is tedious, and in presence of a change, e.g., a new application, the procedure has to be repeated. Thus, a common optimization is

to automatically decide the placement. The decoupling of the application and its location, the end-to-end monitoring provided by fabric, and the what-if analysis provided by the analysis engine, are essential to make placement automated and smart.

- **Load balancing:** Edges decentralize computation across a spatial area, where ideally each edge processes workloads geographically closest to it. However, in practice, workloads and edges are not uniformly distributed across the area. For example, there could be a large cluster of sensors all close to one edge, or, the load could see sudden spikes or long-term changes over time. Since edges do not have the luxury of gathering and distributing data in a centralized location, as in the cloud, they need to dynamically balance load by moving services to other edge locations or the cloud while also brining services back upon drop of load. The ability to transparently and easily move services provided by Steel, is an important feature for load balancing.

- **Fault-tolerance:** Compared to the cloud, the edge is a more unreliable environment with many conventionally uncommon failure cases, e.g., hours of power outage (with no backup power), or falling and breaking [137, 138]. Fixing and/or replacing failed resources also takes longer, either because the edges are owned by non-experts (rather than cloud providers), or the inherent physical challenges. Thus, having a proper fault-tolerance mechanism–that continues the service despite the failures–is essential at the edge. In case of failures, it is common to move services either back to the cloud or a nearby edge. If high availability is a priority, using multiple instances of the service or a hot standby may also be needed. Steel's abstraction provides the flexibility to move services or create multiple instances of them, without impacting the application's logic.

Steel includes a 2-tier optimization schema of a *global* and *local* optimizer, handling both short-term and long-term spikes in the workload and changes in the resources available in the environment (e.g., expansions or failures). The goal of Steel's optimizers are to minimize the monetary cost of running applications (therefore, increasing the edge utilization),

without hurting the performance of applications (i.e., reaching similar or better performance compared to the all-cloud model). In other words, the optimizers try to find the cheapest setting that satisfies the performance requirement.

Global Optimizer: This tier has a global view of the application and all the different components in the application DAG. The global optimizer is in charge of handling long-term changes in the environment or workload. The global optimizer runs in the cloud with a relatively high reaction time–reacting to changes in a order of minutes. Placement and load balancing optimization would run in this tier.

Local Optimizer: This tier has local views of particular devices and specific components of applications. The local optimizer is running on each edge device, fine-tuning the optimization. This tier handles short spikes[2], running at a low reaction time–reacting to changes in a order of seconds. Communication optimization (e.g., batching and compression to increase network capacity) and fine-tuning the allocation fall in this tier.

In this work we focus on the placement and communication optimization, as we found these two optimizations the most essential optimizations based on the need of current Azure IoT customers.

4.3.5 Placement Optimizer

One of the common and complex optimizations for IoT applications is the placement of different components. In a typical industrial setting with hundreds of thousands of sensors and hundreds of edges, manually optimizing the placement, if even possible, is sub-optimal, error-prone and not adaptive. We developed a placement optimizer that takes the cost and performance models of various components and automatically optimizes the placement (edge or cloud, and which edge).

The optimizer, automatically and iteratively, decides the placement of operators across all the applications running in a deployment, e.g., all application in a factory. The optimizer

[2] The spikes could be due to the workload or changes in resources/network.

starts with an initial all-cloud deployment. Then, in a greedy manner moves the most expensive operators (normalized by resource usage) to the edge, until the edge capacities are full. Algorithm 4.1 shows the pseudo code of the detail steps.

The optimizer deploys all applications in-cloud, monitoring them and building cost models for each operator in each application. Then, in a greedy manner based on cost, in finds a candidate list of operators to move, up to a maximum count of MAX_CANDIDATE (lines 18-32). Candidates are operators on the boundary of the edge and the cloud (so called the frontier). For each candidate operator, a couple locations are suggested (up to a fixed MAX_LOC count). These locations are either from the location of upstream operators before this candidate (if an upstream exists) or randomly chosen locations from locations close to the data source. Then, the performance profile is built for these candidates via the analysis engine, with shadow profiling if needed. Based on the cost and performance profile of the candidates, the most expensive operators (normalized by resources) are moved. Moves are monitored and verified that they do not created performance bottlenecks, and if so, the move is reverted and performance profile updated. This optimization loop continues until there are no more feasible moves, i.e., moves that do not create bottlenecks performance).

The optimizer works in a multi-tenant environment, optimizing across multiple applications. The current version treats all applications equally, and prioritizes among them based on cost and resources. As part of future work, we are investigating other priority schemas including user defined priorities and service level agreement (SLA) driven policies (e.g., prioritizing the more latency-sensitive parts).

4.3.6 Communication Optimization

One of the critical factors in the performance and cost of an IoT application is the network traffic between its cloud and edge components. Edges have commodity network links to the broader Internet, with variability in the bandwidth, latency and connectivity,

Algorithm 4.1 Placement Optimization Algorithm

```
1: List<App> apps ={app1, app2, ...}
2: Map< op, loc > locMap; // init all-in-cloud for all apps
3: deploy(apps, locMap)
4: while ! End of Optimization do
5:     //Build cost model
6:     cost=analysisEngine.buildCostModel(apps, locMap)
7:     candidateList< op, loc > = chooseCandidates(cost);
8:     perf = analysisEngine.buildPerModel(candidateList)
9:     moveList< op, loc >=chooseMoves(candidateList)
10:    if !moveList.isEmpty() then
11:        deploy(moveList, locMap) //updates locMap
12:        verifyMove() //reverts if needed
13:    else
14:        End of Optimization = true

15: function buildPerfModel(List< op, loc > list)
16:    for candidate c< op, loc > in list do
17:        if !analysisEngine.contains(op, deviceType(loc)) then
18:            shadow (op, loc)

19: function chooseCandidates(CostProfile cost, apps)
20:    List< op > frontiers, candidates < op, loc >
21:    MAX_LOC, MAX_CANDIDATE
22:    for application app_i in apps do
23:        //traverse application DAG
24:        //add ops on boundry of edge & cloud to frontiers
25:        traverseAndAddFrontier(app_i, frontiers)
26:    sort(frontiers) //sort based on cost
27:    int count = 0
28:    for operator op in frontiers do
29:        for i in 1 to MAX_LOC do
30:            if count < MAX_CANDIDATE then
31:                //loc either where op's upstream operator placed or random location close to the
    data source
32:                candidates.put(op, chooseLoc(op))
33:                count++

34: function chooseMoves(candidateList< op, loc >)
35:    Map< op, loc > moves
36:    sortedList=sort(candidateList) // based on cost/perf
37:    for candidate c< op, loc > in sortedList do
38:        if !moves.contains.(c.op) then
39:            moves.put (c.op, c.loc)
40:    return moves
```

e.g., when remotely located devices transmit data over cellular networks[3]. Furthermore, network communications incur a financial cost in two major forms: fees assessed by the ISP for bandwidth consumption and charges assessed by the cloud provider for ingesting and processing the incoming messages.

In this work, we specifically consider the case of streams of timeseries data that are emitted by sensors, arguably the dominant form of traffic that flows from the edge to the cloud in the vast majority of IoT deployments today. In many applications, including the two real-world deployments that form the basis of our evaluation, sensor messages are naively sent to the cloud as JSON documents. This approach is admittedly convenient, as the ubiquity of JSON lends itself to easy interoperability between components, but it induces unnecessary traffic from the edge to the cloud. JSON is an inefficient compression for numerical data, and JSON documents commonly feature redundant fields that are static for the lifetime of the timeseries stream and need not be included in every message.

We therefore introduce an intelligent batching and compression control module that gives IoT applications the ability to cope with variability or outages in network service and to dynamically adapt to constantly changing network conditions in order to minimize cost. The implementation of this control module addresses several challenges. Sensor messages are batched as necessary to deal with network outages and to facilitate transformations like compression. The communication control module must also periodically identify an appropriate strategy for transforming and compression timeseries data batches on the fly, as network conditions change, from a large search space of potential choices.

These techniques are not new, yet they have historically been integrated directly into IoT applications in the form of one-off implementations. Not only does this situation force developers to re-implement logic and optimizations that can be shared across a wide array of IoT applications, it also means that these applications become unnecessarily complicated

[3]With the deployment of 5G in the future, other interesting optimizations can be performed which is out of the scope of this project.

with communication logic. Our control module enables applications to optimize their communications without significant changes to the code, relieving developers of the burden of implementing their own optimization and adaption strategies and simplifying their production code.

Data compression Strategies

We have implemented 12 different techniques for transforming message batches within a timeseries stream. Each of these techniques is a combination of one or more of the following compression strategies:

- *Omitting Metadata from JSON*: Static information that is redundantly included in each element within a timeseries stream, such as a sensor ID or the units of measurement, is exchanged just once between the sender and receiver, when initializing the channel. Redundant metadata is ommitted from all subsequent document batches, which are sent as JSON.

- *Binary Packing*: Metadata is omitted from all batches, and each timeseries element is represented as an 8-byte integer for the timestamp and an 8-byte float for the corresponding value.

- *Timestamp Delta*: Metadata is omitted from all batches, and time series elements are represented in binary. However, only the first timeseries element in each batch contains a full 8-byte timestamp. All subsequent elements contain a 2-byte timestamp representing the delta from the previous element's timestamp. Values remain 8-byte floats.

- *Fixed Precision*: Metadata is omitted from all batches, and timestamps are represented as 8-byte integers. Because users often only care about measurements up to a certain precision, values are not represented as 8-byte floats. Instead, the channel is

initialized with a specific lower bound, upper bound, and precision, and the value in each timeseries element expresses a point within this range. For example, a channel for temperature data might have numerical bounds of 50 and 100 and a desired precision of 0.5. With only 100 distinct possibilities, timeseries values can be represented in 1 byte instead of 8.

- *Gzip*: Gzip compression is applied to each batch of timeseries values (however they happen to be encoded) before they are sent. Specifically, this strategy opts for the best possible compression at the potential cost of increased computation.

- *Fast Gzip*: Gzip compression is applied to each timeseries batch, but with settings that offer greater speed (i.e. less computational cost) at the potential expense of the effectiveness of the compression.

Adaptive Communication Optimizer

The communication optimizer dynamically choses the best a) compression strategy, and b) batch size.

Compression Strategy: The choice of compression strategy must make judicious use of both the available network bandwidth and available CPU resources at the edge. For example, an aggressive compression algorithm might dramatically reduce the volume of data transmitted from cloud to edge, but at the same time incur a prohibitively high load on the edge node. Each technique presents a different balance between computational intensity and the compression effectiveness.

Figure 4.7 illustrates the trade-off space for the 12 compression techniques in a real-world smart building data set (from Microsoft's smart building deployments in the Redmond Campus). Each compression scheme was used to process a fixed-size batch of messages, and we measured the size of the uncompressed batch relative to the compressed batch as well as the CPU cost incurred by the compression. In the figure, schemes further to the right on the

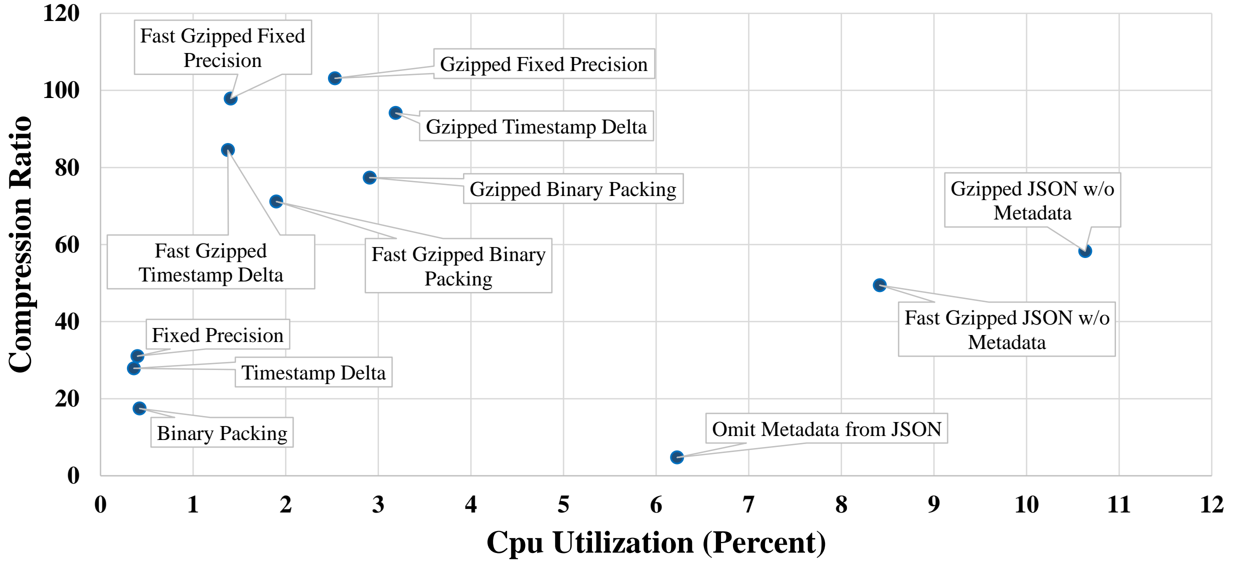


Figure 4.7: The tradeoff between compression and computation for various data compression schemes

x axis exhibited higher CPU consumption, while schemes higher on the y axis achieved a better compression ratio. The best schemes are therefore in the top left region of the figure.

Batch Size: Additionally, there is a trade-off between data staleness and the effectiveness of compression applied at the edge. A larger batch size enables better compression ratios for some compression strategies, but increases the lag time between the generation of a sensor message and its arrival at the cloud. The benefits of larger batch sizes also exhibit diminishing returns. Figure 4.8 shows the effect of batch size for different strategies and the diminishing return point for each (on the smart building data-set). For some compression strategies, such as the fixed precision scheme, batch size does not affect compression ratio because each message in the batch is compressed individually. For Gzip-based schemes, however, larger batches yield better compression ratios, up to a size of about 250 items, because messages are compressed collectively.

Optimization Algorithm: The communication optimizer presents the abstraction of a set of *channels*, each representing a stream of messages to be sent from the edge to the cloud. Periodically (once per second in the current implementation), the communication

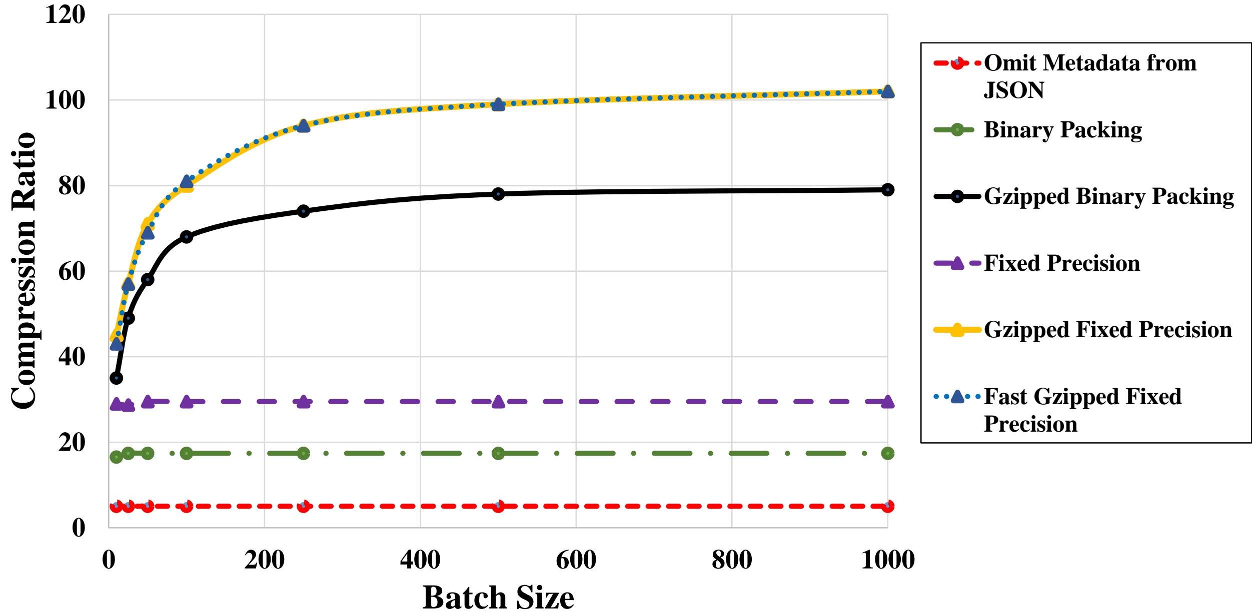


Figure 4.8: Diminishing returns with larger batch sizes

optimizer compresses the messages that have accumulated for each channel and sends them to the cloud, transmitting as many messages as possible while respecting two constraints:

1. A cap on the amount of CPU time that can be devoted to compression, specified by the user

2. A cap on the total size of all transmitted message batches (bandwidth consumption), specified by the user or derived from periodic observations of network conditions

To choose the best compression scheme for each channel while respecting these constraints, the communication optimizer relies on one table containing the CPU compression cost and another containing the compressed size of batches of timeseries messages as batch size and compression scheme vary. For such a table T, $T[i][j]$ gives the relevant measurement when applying scheme i to compress a batch of j messages. These tables can be easily precomputed by running a controlled set of experiments on the relevant hardware before the communication optimizer is used.

We restrict our attention only to compression schemes along the Pareto curve of Figure

4.7 and order these schemes from those that are most computationally efficient but yield the worst compression to those that are most computationally expensive but yield the best compression, i.e., scheme $i+1$ has a higher CPU cost but also higher compression ratio than scheme i.

Each time it is invoked, the compression optimizer first predicts the CPU and bandwidth required to send all messages accumulated across all channels while using the least computationally intensive compression scheme. The optimizer iteratively improves the choice of compression schemes for the channels until both constraints are satisfied. Pseudocode for the optimization is presented in Algorithm 4.2.

4.4 EXPERIMENTAL EVALUATION

Our Evaluation addresses the following questions

1. How easy is it to develop and configure an application spanning the edge and cloud?
2. How easy is it to move parts of an application between edge and cloud?
3. Can Steel efficiently automate the placement of applications?
4. Can Steel effectively cope with network and workload changes?

We evaluate Steel across a diverse set of real-world IoT Applications, as well as emulated results.

4.4.1 Development Effort

The big promise of Steel is ease of development, and adaption of services. To compare our abstraction (*Steel*) to conventional templates/scripts (*current*), we implement real world applications in both systems on top of Azure. We measure the development effort in terms

Algorithm 4.2 Communication Optimization Algorithm

```
1: C = Precomputed table where C[i][j] = CPU cost of applying scheme i to batch of j messages
2: B = Precomputed table where B[i][j] = Bandwidth cost of applying scheme i to batch of j
   messages
3: function OptimizeConfig(channels, cpuCap, bwCap)
4:     for c ∈ channels do
5:         c.scheme = 0
6:     i = 0
7:     while i < MAX_ITERS do
8:         if channels = ∅ then return Failure
9:         totalCPU = ∑{C[c.scheme][c.size] | c ∈ channels}
10:        totalBw = ∑{B[c.scheme][c.size] | c ∈ channels}
11:        if totalCPU < cpuCap and totalBw < bwCap then return {c.scheme | c ∈ channels}
12:        else if totalCPU > cpuCap and totalBw > bwCap then
13:            // We have failed to find a feasible solution, remove a channel
14:            chan = argmax_c {c.size | c ∈ channels}
15:            channels = channels − {chan}
16:        else if totalCPU > cpuCap and totalBw ≤ bwCap then
17:            // We've gone from meeting CPU cap to exceeding it without ever satisfying BW
   cap
18:            // Remove a channel
19:            chan = argmax_c {c.size | c ∈ channels}
20:            channels = channels − {chan}
21:        else
22:            // totalCPU ≤ cpuCap and totalBw > bwCap
23:            // Adjust channel compression scheme to yield greatest ratio of BW reduction to
   CPU increase
24:            if {c | c ∈ channels, c.scheme ≠ Max Scheme} = ∅ then
25:                // We've tried as much compression as possible; remove a channel
26:                chan = argmax_c {c.size | c ∈ channels}
27:                channels = channels − {chan}
28:            else
29:                chan = argmax_c (B[c.scheme][c.size] − B[c.scheme + 1][c.size]) / (C[c.scheme + 1][c.size] − C[c.scheme][c.size]),
                          c ∈ channels, c.scheme ≠ Max Scheme
30:                chan.scheme = chan.scheme + 1
31:        i = i + 1
```

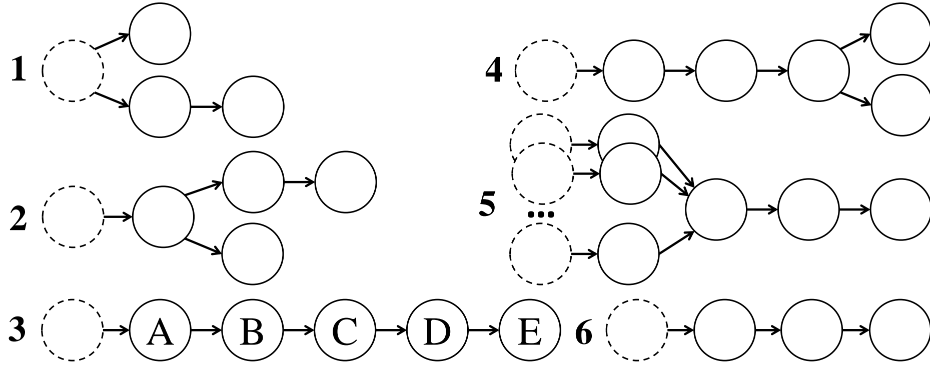


Figure 4.9: DAG of studied applications. Dotted and full circles show the data source and services, respectively.

of lines of config (*loc*), i.e., the number of lines of changes required in the configuration[4]. We evaluate both the initial development and the changes required for moving services.

We chose 6 diverse applications from a combination of common and pre-configured applications [139]. Figure 4.9 shows the logical DAG (without glue services) of the applications: 1. a simple data persisting application (two stores, raw and filtered), 2. a predictive maintenance application using machine learning, 3. Bluetooth sensor connector and analyzer (convert format, add meta-data, filter, average, and store), 4. a factory remote monitoring and alert generator, 5. a campus-wide statistics generator (aggregate locally, join globally, build statistics, and store), and 6. an anomaly detector (Figure 4.2)

4.4.2 Initial Development

We compare the initial development effort in terms of *loc* in both Steel and the current cloud environments. Figure 4.10 shows the results for the 6 chosen applications. Our abstraction reduces the *loc* between 1.7x to 4.8x across the different applications compared to the current system. This improvement is due to two main factors: 1. the hidden glue services added automatically by the system, and 2. the substantially reduced configurations ($\approx$ 1–2 *loc*) required for connecting services. Our abstraction works best when there are multiple

[4]Steel only modifies deployment configurations and code binaries remain untouched.

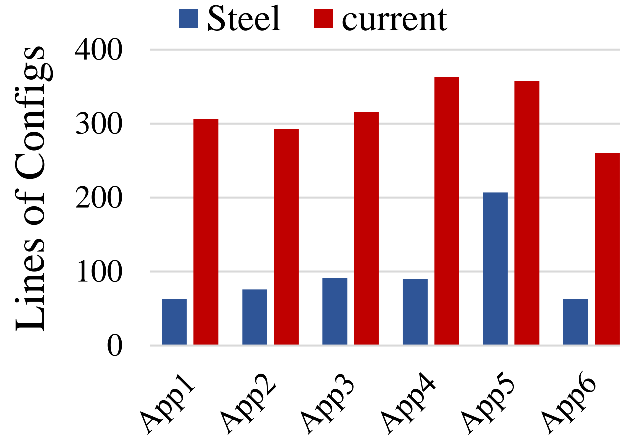


Figure 4.10: Comparison of lines of config required for developing applications in Steel and current systems.

services per location. For examples, Application 5 is spread across many edges, each edge performing one service and the cloud joining the results (followed by additional computation). Since the edge part is simple (single stage), there is little improvement there, and the improvement mostly comes from the cloud part, resulting in an overall smaller improvement (1.7x).

4.4.3 Modifications

Being able to move services dynamically is a crucial feature when using the edge. We evaluate the changes required when making a move. We start with an all-cloud deployment of Application 3 (the longest chain), and move its services {A,B,C,D} one by one to the edge[5]. We measure the *loc* changes (add, modify, or delete) required at each step, compared to the all-cloud deployment. As shown in Figure 4.11, Steel provides a constant and small over head of around 2 *loc* per change of a service location, as only the location map needs to be updated (in Figure 4.3.1). However, current systems need 100s of *loc* changes, between 260–360 *loc* for Application 3, both to add glue components and to reconnect the services. Steel reduces over 95% of this overhead. Note that the *loc* is an almost constant value

[5]Steel modifies configuration during runtime. There is a short period of pause (≈ 10s of seconds) during reconfiguration.

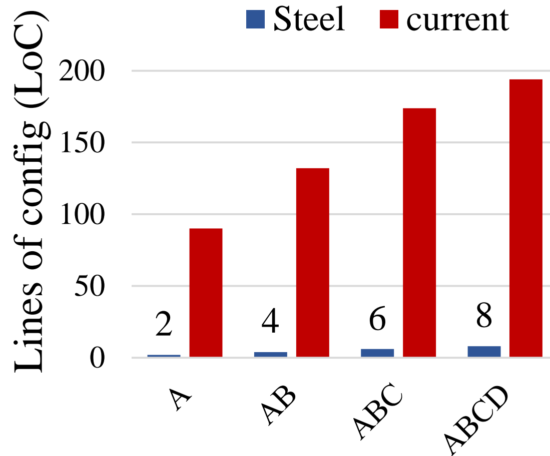


Figure 4.11: Comparison of lines of config required for moving services within an application in Steel and current systems. This experiment moves services {A,B,C,D} in Application 3 (Figure 4.9)

for *one* component move, thus, when a change involves multiple components (e.g., in the placement optimizer) the total *loc* would be the constant overhead times the number of moving components.

4.4.4 Placement Optimization

We measured the effectiveness of our Placement optimizer using a set of multi-stage applications. We start with an all-cloud configuration, then, in an iterative manner, we shadow components, find the best ones to move and push them to the edge. The goal of this process is to reduce the cost of running an application (by pushing processing to the edge). This process is continued until the edge resources are (almost) fully utilized. Figure 4.12 shows a sample run of a multi-stage application. At each iteration, the overall cost is reduced by $\approx$ 40% as components move from the cloud to the edge. Along with that, the resource utilization of the edge increases from 10% to 75%.

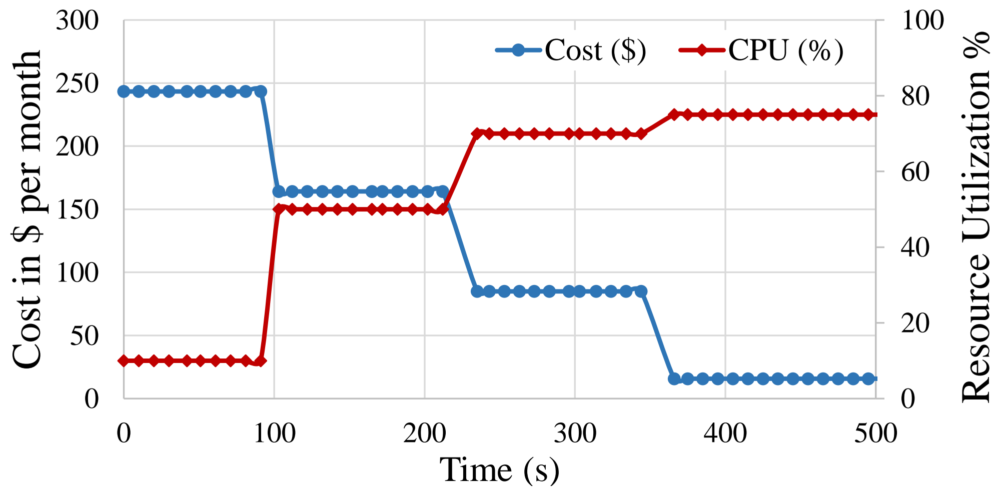


Figure 4.12: Reduction of price and improvement of edge utilization with the placement optimizer over time.

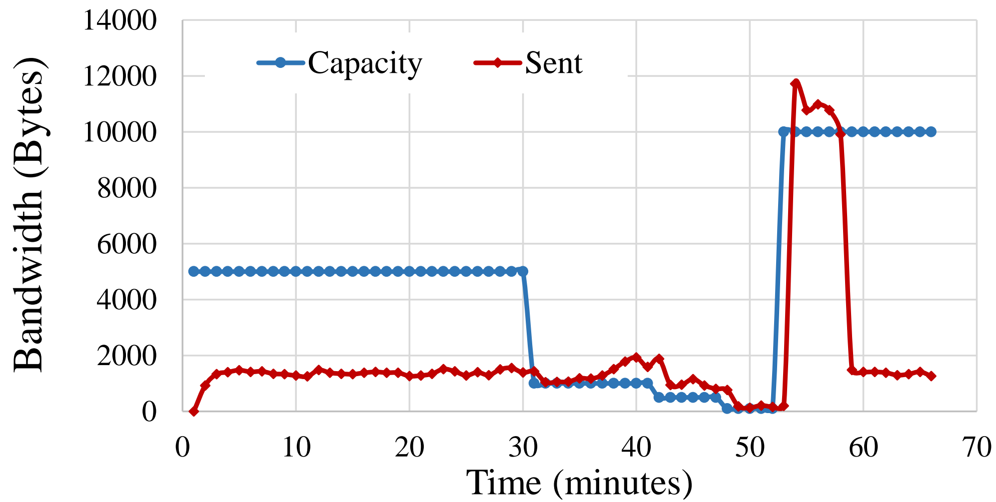


Figure 4.13: Bandwidth consumed by the communications optimizer as network constraints are changed

4.4.5 Communication Optimizer

We also evaluated performance of our communication optimizer module. Figure 4.13 shows the compression control module in action with replayed messages drawn from a real-world smart factory deployment. This trace begins with a generous bandwidth cap (shown in blue) that is later lowered, mimicking temporary congestion. The optimizer adapts the encoding scheme and batch size to limit its bandwidth consumption (shown in red). Although the consumption does not stay strictly under the specified cap, it does stay reasonably close (up to 2.3x at peak). When cap is finally increased, the optimizer drains its buffer of backlogged data, explaining the brief peak in bandwidth consumption before it levels off to a steady rate at the end of the trace.

4.5 RELATED WORK

Abstractions: Mobile fog [140], Beam [90], and [141] propose abstractions to hide the complexities and standardize the heterogeneity of the edge. General purpose abstractions, such as Orleans [142], have also emerged for building large-scale, elastic, and reliable applications. However, none address of integrating with existing cloud services for multi-service applications.

Edge-Cloud Frameworks: To leverage the edge, many end-to-end edge-cloud frameworks have emerged, e.g., FarmBeats for agricultural IoT, Race, and GigaSight [137, 143, 144]. Although these systems handle abstraction, deployment, and optimizations, they are each tailored for a specific use case (e.g, remote agriculture fields), and do not generalize to custom data-processing applications.

Edge Deployment: Dynamic deployment across heterogeneous edges is essential for the usability of the edge. Micro-cloud, Cloudlets, Openstack++, and others [141, 145–150] have proposed infrastructures to provide portability and dynamism using either containers or virtual machines. These infrastructures provide interoperability between edges and the cloud,

Table 4.2: Placement and data-motion optimizations in Steel

Requirements	Placement	Data Motion
- geo-distributed and transparent processing across many locations - interactive processing experience - reduced network usage w/ tolerable latency impact	- abstracting and unifying environment and placement - network, device, and locality aware placement - placement aware models for performance and cost with background profiling and a prediction engine	- accessing local data and storage (instead of remote cloud) - data motion reduction with adaptive batching and compression - throttled-based batching at spikes with prioritization

a crucial feature for the edge. However, this is orthogonal to the need of simplifying the development of edge-cloud applications.

Migration and Placement: Dynamically placing and moving computation is not a new concept. Maui and CloneCloud [151, 152] have explored this with mobiles as the edge. MigCEP [126], VM Handoff [125], and [127] optimize the dynamic migration between edges, based on different constraints, e.g., latency. These works do not support current cloud services, while works like [153] emphasize the importance of doing so. However, their approaches can be added as optimizations modules to Steel.

4.6 CONCLUSION

This chapter describes Steel, a high-level framework designed specifically for building complex data processing applications in the emerging edge-cloud environment. We design Steel to hide the complexities of developing, deploying, and monitoring data processing applications using many cloud services, and to support dynamically adapting and easily moving services back and forth between the edge and the cloud. Steel is an extensible framework where common but crucial optimizations for the edge can be built as pluggable and configurable modules. We have added two very common optimizations: placement and adaptive communication, to cope with both short and long term changes in the workload and environment. Table 4.2 summarizes the techniques used for data placement and data

motion in Steel and the requirements they satisfy.

As part of future work, we plan to further investigate other main optimizations required in an edge-cloud environment. We plan to develop efficient edge-oriented solutions for resource allocation, fault-tolerance, and load balancing and add them to Steel as pluggable modules. In addition, we are working on extending Steel's abstraction to include service level agreements (SLAs) for latency, throughput, and availability, and incorporate these SLAs into the optimization modules.

CHAPTER 5: FREEFLOW: HIGH PERFORMANCE CONTAINER NETWORKING

In this chapter, we present FreeFlow, a solution for reaching high performance and portable networking between containers. As the popularity of container technology grows, many applications are being developed, deployed and managed as groups of containers that communicate among each other to deliver the desired service to users. However, current container networking solutions have either poor performance or poor portability, which undermines the advantages of containerization. We propose FreeFlow, a container networking solution which achieves both high performance and good portability. FreeFlow leverages two insights: first, in most container deployments a central entity (i.e. the orchestrator) exists that is fully aware of the location of each container (placement-awareness). Second, strict isolation is unnecessary among containers belonging to the same application. Leveraging these two observations allows FreeFlow to use a variety of technologies such as shared memory and RDMA to improve network performance (reducing data-motion), while maintaining full portability – and do all this in a manner that is completely transparent to application developers.

5.1 INTRODUCTION

> *The history of all hitherto computer science is (often) the history of a struggle between isolation, portability and performance.*
> *(With apologies to Karl Marx.)*

At the dawn of computing, applications had access to (and had to manage) raw hardware. Applications were not portable, and isolation between applications was non-existent. Operating systems emerged and offered a modicum of isolation and portability. As users demanded more portability and better isolation across applications, OSes became more sophisticated, and deep layering became the norm; the modern TCP/IP stack is a classic example.

Layering improves the application's portability across systems and types of networks, but incurs well-known performance issues [154, 155]. Soon enough, solutions like DPDK [156] and RDMA [157] emerged that traded off some portability and isolation to provide better performance. The trend continued with virtualization, which offers even more isolation, and additional portability (e.g., you can pack up and move VMs at will, even live-migrate them). In return, performance – especially the network performance is further reduced [158]. Numerous technologies have been proposed to remedy the situation (e.g., [155, 156, 158, 159]) – again at the cost of isolation and portability.

The latest step in this trend is *containerization* [19,160,161]. By wrapping a process into a complete filesystem and namespace cell, a container has everything needed to run the process, including executables, libraries and system tools. A container has no external dependencies (it only has a set of dependencies on a standard API), which makes it highly portable. The namespace of the container is isolated from other containers, eliminating worries about naming and version conflicts. Such portability and independence significantly simplifies the life cycle of a containerized application, from testing to high availability maintenance.

Unfortunately, containers too suffer the curse of having to sacrifice one or more of performance, isolation, and portability. To understand these potential performance bottlenecks, we conducted a simple experiment. We set up two Docker containers on a single server[1]. We consider three ways for the containers to communicate with each other: (1) *Shared Memory:* This requires special setup[2] to bypass the namespace isolation, and offers the least isolation, and the least portability; (2) *Host mode:* in which a container binds an interface and a port on the host and uses the host's IP to communicate, like an ordinary process. Hence, containers are not truly isolated as they must share the port space; and (3) *Overlay mode:* in which the host runs a software router which connects all containers on the host via a bridge network. The software routers enable overlay routing across multiple hosts to provide

[1]See Section 5.5 for HW and SW details

[2]We setup the shared memory data transfer through `shared memory object` in a shared IPC namespace and measure the time to pass the pointer and make one copy of the data.

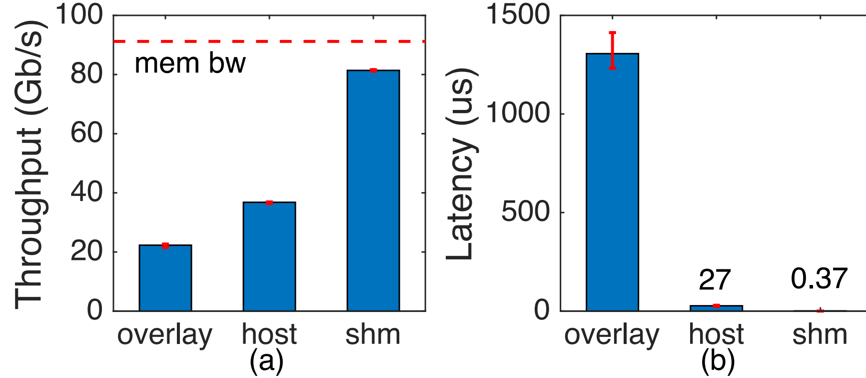


Figure 5.1: Performance comparison of two modes of container networking and shared memory IPC.

maximum portability as each container can even have public IPs assigned.

Figure 5.1 is a demonstration of the fundamental tussle between portability, isolation, and performance. We make two observations from this figure. First, the throughput and latency of host and overlay modes of inter-container communication are significantly worse than the throughput and latency of shared memory based communication. The reason is obvious: both host and overlay modes require a "hairpin" path through the full TCP/IP stack.

Second, the performance of overlay networking is worse than host mode. The reason, again, is simple: in case of overlay networking, hairpinning happens twice, since the packets must traverse through the software router as well. This figure thus clearly illustrates the performance cost of isolation and portability.

As the popularity of container networking grows, this inefficiency must be addressed. On one hand, the low throughput and high latency directly impacts the overall performance of large scale distributed systems, such as big data analytics [162–165], key-value stores [166], machine learning frameworks [167], etc. On the other hand, it forces the applications to reserve substantial CPU resources to merely perform traffic processing, which significantly raises the cost of running the applications.

One may argue that there is nothing new here: virtualization suffers from similar in-

efficiencies and we know how to address them using techniques like SR-IOV [159] and NetVM [158]. Unfortunately, these ideas cannot be directly applied to the container world. SR-IOV typically scales to tens of VMs per server. In typical deployments, there are hundreds of containers per server. The cost of supporting so many containers in the NIC hardware will be prohibitive. NetVM [158] cannot be used directly with containers, since it destroys portability – it only works when two VMs are on the same server. Developers prize containers especially for their portability: indeed, one of the main selling points of containerization is that a containerized application that runs on developer's desktop will run in the cloud without any changes! [168].

In this work we outline a solution, called FreeFlow to address this issue. Our vision is to develop a container networking solution that provides high throughput, low latency and negligible CPU overhead and fully preserves container portability in a manner that is completely transparent to application developers.

To achieve these seemingly conflicting goals, we observe an opportunity to leverage two key aspects of typical container deployments: (1) they are typically managed by a central orchestrator (E.g., Mesos, YARN and Kubernetes [21, 22, 160]) and (2) they are typically deployed over managed network fabrics (e.g., a public cloud provider). Taking advantage of these easily available additional bits of information, we sketch a roadmap of an overlay-based solution that obtains the relevant deployment-specific information from the aforementioned container orchestrator and fabric manager and use this in conjunction with the "right" I/O mechanism (e.g., shared memory when containers are co-located, vs. RDMA when they are not).

While this sounds conceptually simple, there are several architectural and system design challenges in realizing this vision in practice. In the rest of the chapter, we discuss these challenges and sketch a preliminary design. We will also present results from an early prototype.

5.2 BACKGROUND

Container technology has gained tremendous popularity [17, 169, 170] since it enables a fast and easy way to package, distribute and deploy applications and services.

Containers are essentially processes, but they use mechanisms as `chroot` [171] to provide namespace isolation. The dependencies of containerized applications are bundled together with the application, making them highly portable: they can be easily distributed and deployed [19]. Compared to VMs, containers are much more lightweight and can be initialized much faster [172, 173]. Containers can be deployed both within a VM or on bare metal. Containers are supported by both Linux and Windows operating system. Docker [19] is perhaps the most popular container management system, although there are many others as well [160, 161].

Most containerized applications are usually composed of multiple containers. For example, each mapper and reducer node in Hadoop [38] is an individual container. A modern web service with multiple layers (load balancers, web servers, in-memory caches and backend databases) is deployed with multiple containers in each layer. These containers are usually deployed into a multi-host server cluster, and the deployment is often controlled by a cluster orchestrator, e.g. Mesos [21] or Kubernetes [160]. Such an architecture makes it easier to upgrade the nodes or mitigate failures, since a stopped container can be quickly replaced by a new one on the same or a different host. Working as a single application, containers need to exchange data, and the network performance has a significant impact on the overall application performance [162–165]

Depending on whether containers run on bare-metal hosts or VM hosts, there are four cases any container networking solution must handle. These cases are illustrated in Figure 5.2. For maximum portability, containers today often use overlay networks. A number of software solutions are available to provide overlay fabrics, such as Weave [174] and Calico [175]. In these solutions, the host (i.e., the server or the VM) runs the software router

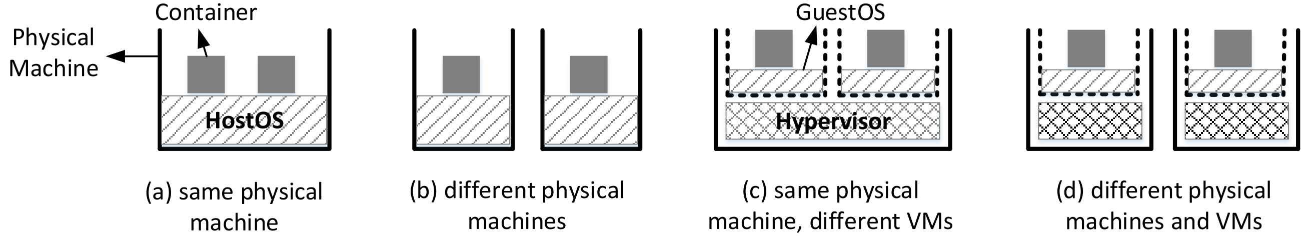


Figure 5.2: Representative running environments of containers.

which connects to the NIC of the host and to the virtual interfaces of the containers on the host via a software bridge. The router also performs appropriate tunneling (encapsulation and decapsulation) to move traffic between the physical network and the overlay fabric. The router uses standard networking protocols like BGP to connect with software routers on other hosts. Containers send and receive IP packets via this overlay, and hence are agnostic to location of other containers they are communicating with.

The CPU overhead of processing packets in the software bridge, as well as in the software router (for off-host communication) is the primary performance bottleneck in overlay networks, as illustrated in Figure 5.3. This figure shows throughput and CPU utilization of two containers, running iPerf3. For Figure 5.3(a), the containers were on the same server, communicating via host mode. For Figure 5.3(b), the containers were on different servers, connected via overlay routing using Weave [174]. We vary the iPerf traffic generation rate, and plot the achieved throughput and corresponding CPU utilization. We see that in host mode, the throughput tops off at 30Gbps, while in overlay mode it is just 20Gbps. In both cases, the CPU is the bottleneck – the sender and the receiver CPUs are fully utilized.

5.3 OVERVIEW

In this section, we discuss the overall architecture of FreeFlow, and discuss the key insights that enable FreeFlow to achieve a high network performance without sacrificing portability of containers.

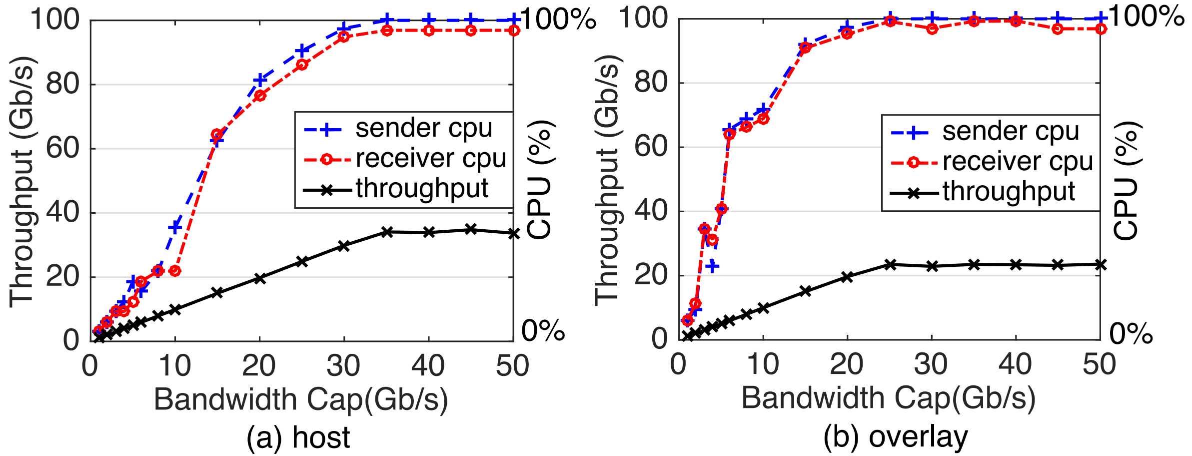
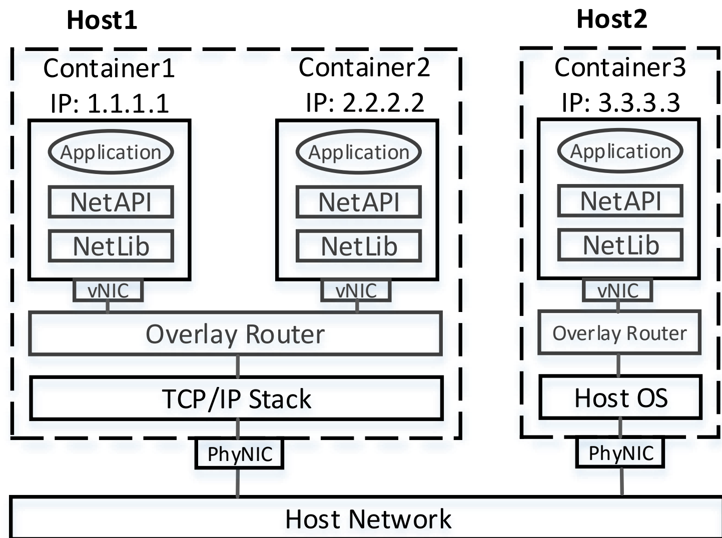


Figure 5.3: CPU limits overlay networking throughput.

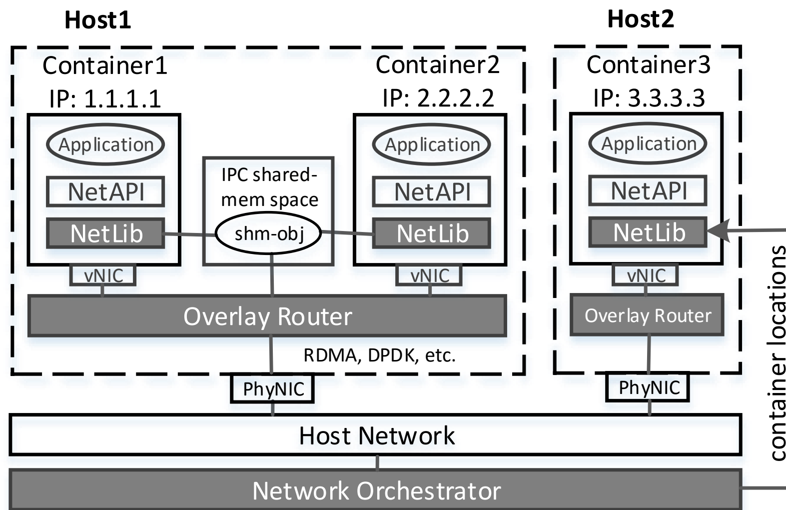
5.3.1 High network performance without sacrificing portability

Container deployments opt for overlay-based networking since it is most portable: a container does not have to worry about where the other endpoint is. For example, in Figure 5.4(a), Container 1 and Container 3 cannot distinguish whether Container 2 is on Host 1 or Host 2, as long as Container 2 keeps its overlay IP address (2.2.2.2) and the overlay routers know how to route packets to this IP.

Existing overlay-based container networks sacrifice performance for good portability, because traffic needs to go through a deep software stack, as shown in Figure 5.4(a). The key to achieve high performance and low overhead overlay network for containers is to avoid, in the data-plane, any performance bottlenecks such as bridges, software routers and host OS kernel. Given that containers are essentially processes, the communication channels provided by host-based IPC and hardware offloaded network transports (RDMA) give us numerous options to build a better container network. For instance, containers within a single host, like Container 1 and Container 2 in Figure 5.4(a), can communicate via a shared-memory channel, and overlay routers in different hosts can talk via RDMA (or DPDK) to bypass the performance bottlenecks. Note that communication paradigms like shared-memory and RDMA partially sacrifice the isolation of containers. However, since in most cases containers that communicate with each other are part of the same larger, distributed application



(a) Legacy overlay container networks



(b) FreeFlow

Figure 5.4: The overall system architecture of existing overlay network and FreeFlow. Gray boxes are building blocks of FreeFlow.

deployed by a single tenant. Therefore, we believe that trading off a little isolation for a large boost in performance is acceptable. We will discuss security implications of our design in more detail later in the chapter.

Generally, one container should decide how to communicate with another according to the latter's location, using the optimal transport for high networking performance. There are two issues to realize this key idea: (1) How to discover the real-time locations of containers; (2) How to enable containers to use different mechanisms to communicate with different peers.

One way is to solve these two issues is to depend on containerized applications themselves: the applications can discover and exchange location information and agree on a communication mechanism to use. This method requires applications to do extra work (and the code can become quite complicated, as the programmer deals with different communication methods), and hence is undesirable.

Instead, we take an alternative approach: using a (conceptually) centralized orchestrator to decide how containers should communicate, and keeping the container locations and the actual communication mechanisms transparent to containerized applications. Our key insight is that since currently most of the container clusters are managed by a centralized cluster orchestrator (e.g., Mesos, Kubernetes, and Docker Swarm)[3], the information about the location of the other endpoints can be easily obtained by querying the orchestrator. By leveraging this information, we can choose the right communication paradigm for the specific scenario. Furthermore, all of the complexity of communication mechanism selection and execution can be hidden from the application by bundling it into a customized network library supporting standard network APIs. Next, we sketch the architecture of our solution.

[3]They can easily be deployed on private bare-metal clusters or cloud VM clusters without any special supports from cloud providers.

5.3.2 The architecture of FreeFlow

Figure 5.4(a) shows the architecture of existing overlay networking solutions for containers. Each container has a virtual NIC that is attached to the overlay router of the host via software bridge. Different overlay routers exchange routing information and build routing tables via standard routing protocols, such as BGP. The fabric built by virtual NICs, bridges, overlay routers, physical NICs and the host network is the actual data-plane for packets traversing the overlay from container to another one. Inside each container, applications use standard network APIs to access the network. The API calls are implemented in network libraries, such as `glibc` for Socket API, and `libibverbs` for RDMA Verbs API.

FreeFlow reuses many control-plane features like IP allocation and routing implemented by existing solutions such as *Weave*. However, FreeFlow modifies multiple existing modules in the networking stack to achieve a smarter and more efficient data-plane.

Figure 5.4(b) shows the overall architecture of FreeFlow. The gray boxes in Figure 5.4(b) represent the three key building blocks of FreeFlow: customized network library, customized overlay router and customized orchestrator.

FreeFlow's network library is the core component which decides which communication paradigm to use. It supports standard network programming APIs, e.g. Socket for TCP/IP, MPI and Verbs for RDMA, etc. It queries the network orchestrator for the location of the container it wishes to communicate with. Whenever possible, it uses shared memory to communicate with the other container, bypassing the overlay router. FreeFlow's overlay routers are based on existing overlay routers. We add two new features: (1) the traffic between routers and its local containers goes through shared-memory instead of software bridge; and (2) the traffic between different routers is delivered via kernel bypassing techniques, e.g. RDMA or DPDK, if the hardware on the hosts is capable. The network orchestrator keeps track of the realtime locations of each container in the cluster. Our solution extends existing network orchestration solutions, and allows FreeFlow's network library to query for the physical deployment location of each container.

The architecture enables the possibility to make traffic among containers flow through an efficient data-plane: shared-memory for intra-host cases and shared-memory plus kernel bypassing networking for inter-host cases.

However, we have two challenges to achieve FreeFlow's simple vision. First, the network library of FreeFlow should naturally support multiple standard network APIs for transparency and backward compatibility. Second, for inter-host cases, overlay routers should connect the shared-memory channel with local containers and the kernel bypassing channel between physical NICs to avoid overhead caused by memory copying. In next section, we discuss how FreeFlow addresses these challenges.

5.4 DESIGN

This section presents the designs of FreeFlow's major components. More details of each component is discussed below.

5.4.1 The network library of FreeFlow

The network library of FreeFlow is the key component which makes the actual communication paradigm transparent to applications in the containers. It has two goals: (1) supporting most common network APIs, such as Socket (TCP/IP), Verbs (RDMA), MPI (parallel computing) and so on; and (2) selecting the most efficient communication paradigm no matter which network API is used.

One straightforward way to build the network library is to develop several independent libraries each of which deals with a specific network API. For instance, we extend the socket implementation of `glibc` for supporting Socket API and design a new `libibverbs` for RDMA Verbs API. However, writing different libraries is clearly suboptimal. Instead, as shown in Figure 5.5, we merely develop a new library for RDMA API, and use existing "translation" libraries such as [176–179] to support socket and MPI APIs atop the RDMA API. Note that

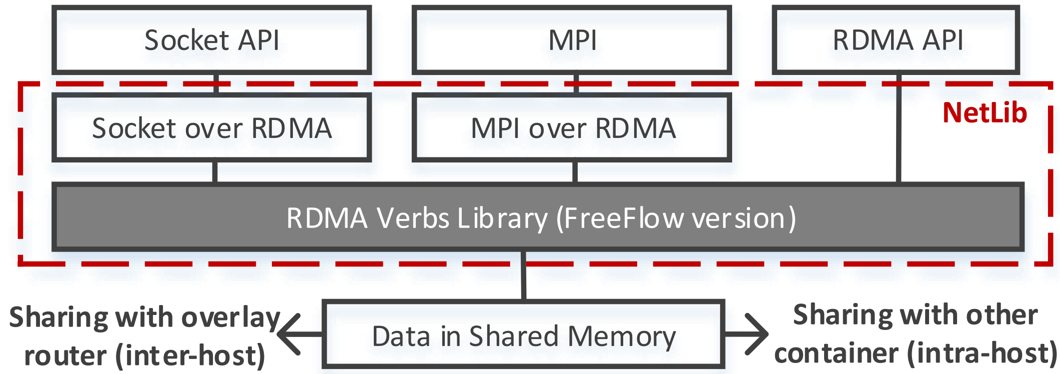


Figure 5.5: The internal structure of FreeFlow's network library. The gray box is the built by FreeFlow.

we could have made the choice the other way as well: e.g. support socket API natively and use translation libraries to support other APIs atop it. We chose RDMA API as our primary interface, since we believe that the message-oriented interface it supports maps naturally to communication patterns of many containerized applications. The shared memory IPC API also maps naturally and easily to the RDMA API.

After the network library receives calls to send data to another container, it first checks (from the network orchestrator) the receiver container's location. If the container is on the same host, it will put the data into a shared memory block and send the pointer of the memory block to the receiver's network library module. The latter will use correct API semantics to notify the application on the receiver container that the data is ready to read. Otherwise, if the receiver container is on a different host, the network library will share the data with the overlay router on the same host, and tell the overlay router send the data to the receiver container's IP. Then it relies on the overlay routers to deliver the data to the destination.

5.4.2 The overlay router of FreeFlow

Overlay router has functionalities in both control-plane and data-plane. In control-plane, it allocates IP addresses for new containers according to default or manually configured

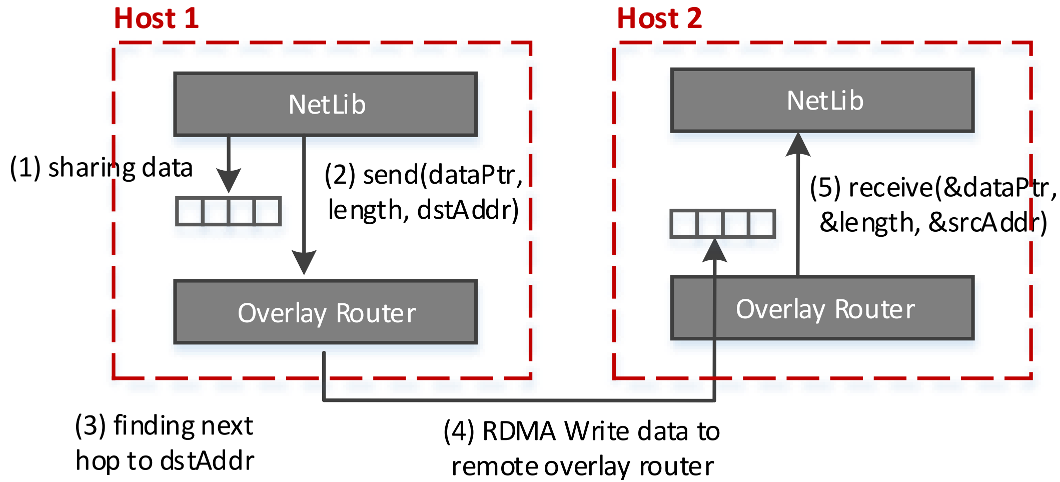


Figure 5.6: The working flow of sending data from one host to another via overlay routers in FreeFlow.

policies. It also exchanges routing information and compute routings to all containers in the cluster. FreeFlow inherits the control-plane behaviors from existing overlay network solutions.

In the data-plane, FreeFlow has its own implementation to make the data transfer more efficient. Figure 5.6 shows an example of how overlay routers deliver data from a sender container to a receiver container. As described in §5.4.1, the network library in the sender container will share the data with its local overlay router (Step 1) if the former finds that the receiver is on another host. After that, the network library will tell the overlay router to send the data to the receiver's IP address (Step 2). The overlay router will check its routing table to find the next hop router towards the receiver (Step 3) and (Step 4) it writes the data to the next hop overlay router via RDMA (or DPDK, if available). If RDMA or DPDK are not available, normal IP communication is used. If the next hop router finds the receiver is on its host, it will share the received data with the network library on the same host and notify the latter that the data is ready to fetch (Step 5).

Note that in this design, there is only a one time data copy from one host to another host, which is unavoidable. The communications between network library and overlay router on the same host are all performed via shared-memory.

5.4.3 The network orchestrator

The main functionality of the network orchestrator is to maintain the real-time information of container locations and VM locations (if needed). Since containers are typically started and managed by a cluster orchestrator (e.g. Mesos), the container to host mapping can be easily accessed from the cluster orchestrator. FreeFlow only adds a small module into existing clusters orchestrator to allow the network library modules to query the container-to-host mapping. Note that the orchestrator can either push the mappings to network libraries, or the libraries can pull it. The two choices have different scalability implications. We are investigating this tradeoff further.

5.5 PRELIMINARY IMPLEMENTATION

We have implemented a prototype of FreeFlow on a testbed of clustered bare-metal machines (Intel Xeon E5-2609 2.40GHz 4-cores CPU, 67 GB RAM, 40Gbps Mellanox CX3 NIC, CentOS 7). The prototype selects the most efficient data-plane mechanism based on the location of two containers: if the two containers are intra-host, shared-memory mechanism will be selected for data transfer, and if the two containers are inter-host, RDMA will be selected. We implemented the shared-memory via multiplexing `IPC namespace`, and enabled containers to use RDMA by using the `host mode`. We evaluated our prototype and compare it with state-of-the-art container overlay network - `Weave`. The results are shown in Figure 5.7. We see that the FreeFlow prototype achieves higher throughput and lower latency. The CPU utilization per bit/second is also significantly lower for FreeFlow compared to that of `Weave`.

5.6 RELATED WORK

Inter-VM Communication: The tussle between isolation and performance is not unique to containers. The issue has also been studied in the context of Inter-VM Com-

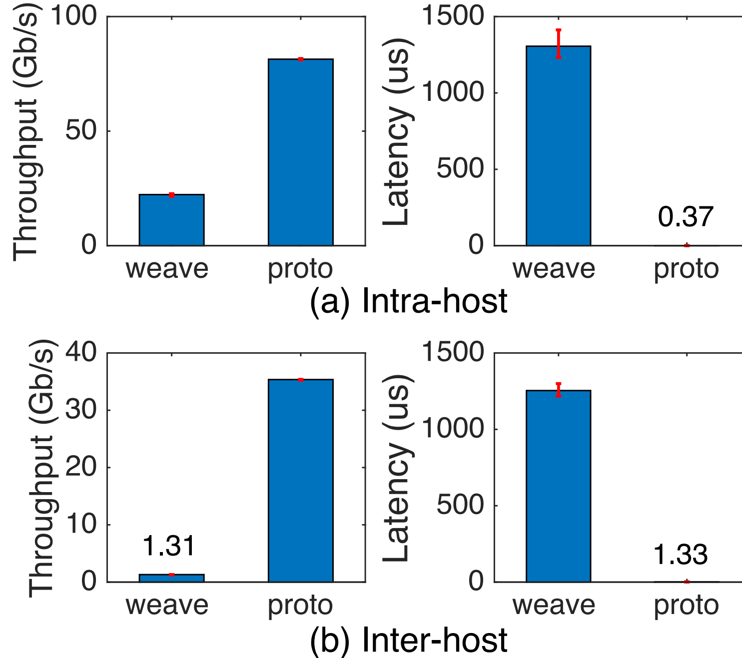


Figure 5.7: Compare FreeFlow prototype with weave.

munication. For example, NetVM[158] provides a shared-memory framework that exploits the DPDK library to provide zero-copy delivery between VMs. Netmap[155] and VALE[180] (which is also used by ClickOS[181]) are sharing buffers and metadata between kernel and userspace to eliminate memory copies. However, these systems cannot be directly used in a containerized setting. For example, the NetVM work is applicable only to intra-host setting, constrained by the possibility of shared memory. It does not handle inter-host communication. Similarly, the Netmap and VALE solutions are sub-optimal when the VMs/containers are located on the same physical machine: shared memory provides a much more efficient communication mechanism.

RDMA and HPC: RDMA originated from the HPC world, in the form of InfiniBand. The HPC community proposed RDMA enablement solutions for virtualization [182] and containerization [183] technologies. These solutions are addressing the challenges in exposing RDMA interfaces to virtualized/containerized applications, treating each VM/container as if it resides on a different node.

Table 5.1: Placement and data-motion optimization in FreeFlow

Requirements	**Placement**	**Data Motion**
- fast and high-performance container networking	- abstracting container placement and networking capabilities per location - transparent placement update	- optimizing data-motion based on location - opportunistic kernel bypassing based on hardware capabilities

The HPC community has also been using shared-memory based communication [184–186] for intra-node communication. These solutions are targeting MPI processes residing on a shared non-containerized, non-virtualized machine. They do not attempt to pierce the virtualization/containerization for additional performance.

The same concepts described for FreeFlow can also be applicable for MPI run-time libraries. This can be achieved either by layering the MPI implementation on top of FreeFlow, or by implementing a similar solution in the MPI run-time library.

General improvements: A significant amount of work has been spent attempting to improve the performance of the networking stack [178,187,188] in various scenarios. However, none of them were aiming at optimizing the performance of networking communications between co-residing containers. While the performance of intra-node communication for containers was identified as relevant before [189], to our knowledge there was no attempt at addressing this challenge.

5.7 CONCLUSION AND DISCUSSION

In this chapter, we discussed how to build a network solution for containers, named FreeFlow. It offers high performance, good portability and acceptable isolation. We sketched the design of FreeFlow which enables containers to communicate with the most efficient way according to their locations and hardware capabilities and keeps the decisions transparent to applications in containers. Table 5.1 summarizes the requirements, and placement and data-motion optimizations used in FreeFlow.

We are currently building a production version of FreeFlow, and we list a few important considerations below:

- **Live migration:** FreeFlow could be a key enabler for containers to achieve both high-performance and capability for live migration. It will require the network library to interact with the orchestrator more frequently, and may require maintaining additional per-connection state within the library. We are currently investigating this further.

- **Security and middle-box:** One valid concern for FreeFlow is how legacy middle-boxes will work for communication via shared-memory or RDMA, and whether security will be broken by using shared-memory or RDMA. We do not yet have complete answer to this issue. We envision that for security, FreeFlow would only allow shared-memory among trusted containers, for example, container belongs to the same vendor (e.g., running spark or storm). We are investigating how best to support existing middle-boxes (e.g. IDS/IPS) under FreeFlow.

- **VM environment:** So far our evaluation and prototype is based on containers running on bare-metal. But our design easily generalizes to containers deployed inside VMs. Some issues, such as efficient inter-VM communication (perhaps using NetVM [158]) need to be addressed, but we believe that it can be easily done within the context of FreeFlow design.

- **Scalability of FreeFlow :** Scalability of FreeFlow is a major focus of our ongoing work. The design proposed in §5.4 has a few potential scalability challenges. For example, overlay routers in FreeFlow may end up having to maintain per-flow state if they communicate with co-located containers using shared memory and with remote routers using RDMA – if done näively, the router may end up maintaining one queue pair for each pair of communicating containers. To solve this problem, we need to to multiplex a single queue pair on overlay routers for multiple container sessions.

CHAPTER 6: CONCLUSION AND FUTURE WORK

6.1 SUMMARY

In this thesis, we have shown how latency-driven designs for placement and data-motion can be leveraged to build production-quality interactive applications at a global scale, while also being able to address myriad challenges on heterogeneity, dynamism, state, and availability. In doing so, we have applied a latency-driven approach at all layers of storage, processing, and networking, developing four interactive systems.

Our first contribution, Ambry (a collaboration with LinkedIn) is a geo-distributed storage system. Ambry is the mainstream production system for storing LinkedIn's media content across the globe. Ambry's main goal is to minimize the user-perceived latency of upload and viewing content while using resources efficiently. Ambry uses a combination of locality-aware storage, background replication, load balancing and other techniques to place data in multiple datacenters around the world, while also minimizing the cross-datacenter traffic.

Our second system, Samza (a collaboration with LinkedIn) is a stream processing system capable of handling a large amount of state along with processing. Samza is an opensource project adapted by many companies, including LinkedIn, Uber, Netflix, etc. Samza aims at processing streams of events and generating actions/outputs, at a global scale, which in turn requires stateful processing (e.g., aggregations, averages, or joins) at a massive scale. We leverage data locality, and tiered storage along with background mechanism to store and replicate state with low latency.

Our third system, Steel (a collaboration with Microsoft), extends the traditional cloud-only computation to the emerging edge computing environment. Steel leverages the locality and proximity of edge devices to data sources towards building interactive applications (in a placement optimizer), along with a load-aware background edge-cloud networking mechanism (in a communication optimizer).

Finally, our fourth contribution, FreeFlow (a collaboration with Microsoft), is a container networking technology aiming at providing fast high throughput communication. Free Flow uses container locations, hardware capabilities, and opportunistic bypassing to minimize the data movement, and in turn the networking latency.

Overall, we have shown that with latency-driven designs in placement and data-motion, such as leveraging locality, background processing, tiered data access, partitioning and parallelism, opportunistic processing, load balancing and scaling, it is practical to develop large-scale interactive systems with hundreds of millions of users around the world. Furthermore, this latency-driven approach can be used at all layers of storage, processing, and networking.

6.2 FUTURE WORK

We have studied how to build interactive systems at a global scale in a number of production systems. We suggest several directions for future work related to this thesis.

6.2.1 Extending Global Interactivity

This thesis has shown how a latency-driven approach can be used at all layers of storage, processing, and networking towards reaching interactivity. This work has focused on each layer independently. An interesting future direction would be to study a more holistic and cross-layer approach. There are potential benefits of having a integrated design, e.g., the networking mechanism that has insight into the replication mechanism of the storage system can optimize towards it. The main challenge is finding the right level of sharing across layers such that each system is still operated and managed independently.

An extension direction to this thesis is reaching the perception of *one global system* operating seamlessly across the globe. While this thesis has focused on interactivity and latency of individual components/systems to build such a system, there is a need for a general and comprehensive view of this problem from other angles such as fault-tolerance and

continued availability. Achieving this unified global environment would require integrating existing large-scale frameworks and building new ones that are tailored for the access patterns of global applications. A first step toward this goal would be defining and analyzing the design patterns and best practices for latency-driven designs.

This work has focused on latency as the metric of interactiveness. However, other metrics can be used as well, e.g., Quality of Experience (QoE) that focuses on users to derive priorities of task. Additionally, with the current advancements in Artificial Intelligence (AI), an interesting next step would be to try and leverage AI toward building adaptive versions of the solutions discussed in this thesis.

6.2.2 Interactive Storage

The storage system design in this work (Ambry) focused on *immutable* objects. The immutable nature simplifies many requirements on consistency and enables lazy techniques like background replication. A future direction would be to employ a latency-driven approach for handling inconsistencies in mutable objects.

Not all objects are equally latency-sensitive, e.g., older objects become less popular and important. Another research direction is to create hybrid sensitivity-aware approaches for both sensitive and non-sensitive objects, where resources are better utilized. Smart and dynamic use of compression, erasure coding, and compaction along with reducing replication factor can be employed to reduce the resource usage of non-sensitive objects.

6.2.3 Interactive Processing

Similar to storage, not all paths in a job are equally critical (or influential on latency). This thesis has used this concept by moving most non-critical computation out of the main processing path. However, this has been done in per-system fashion. A thought provoking direction would be to *generalize these system-tailored techniques to fit many systems.* Ideally,

developers should be able to indicate sensitive and non-sensitive parts of a job/framework, and the underlying system should transparently optimize the sensitive paths.

Given the many sources of dynamism and heterogeneity in the cloud, and even more in the edge, a general idealistic direction is to move toward an *auto-pilot* system coping with changes automatically and dynamically. A well-desired example feature is adaptive scaling of resources. Despite the many research solutions in this area [84, 190–192], it still remains a fairly uncharted territory, especially when being solved in a reliable, low-overhead, and practical manner. There is an even greater need for this feature in the unreliable, varied bandwidth, and widely heterogeneous edge computing environment.

Another potential future direction is to study functional parallelism. Samza is a data parallel system, i.e., data is partitioned and processed in parallel, which is a great fit for its text-based workloads. An alternative option is functional parallelism, i.e., partitioning the processing across machines. Studying the trade-off of functional parallelism vs. data parallelism especially in the context of other workloads (e.g., media) is an interesting direction, particularly, given that the latency-consistency trade-off can have a different optimal result based on the workload (e.g., in case of a media workload).

6.2.4 Interactive Networking Mechanism

This thesis optimizes networking in a reactive manner, i.e., after a job's container placement is determined. An interesting direction would be to make this proactive. The networking layer can leverage its global view of all data flows along with historical application runs towards a better placement, e.g., placing a pair of heavily communicating containers in the same physical machine.

REFERENCES

[1] "The real cost of slow time vs downtime," http://www.webperformancetoday.com/2014/11/12/real-cost-slow-time-vs-downtime-slides/.

[2] Oracle, "How does load speed affect conversion rate?" https://blogs.oracle.com/marketingcloud/how-does-load-speed-affect-conversion-rate, 2016 (accessed Mar, 2018).

[3] Skilled, "Why faster page load time is better for your website," 2016 (accessed Mar, 2018).

[4] S. E. Land, "The need for speed: 7 observations on the impact of page speed to the future of local mobile search," 2016 (accessed Mar, 2018).

[5] Kissmetrics, "How loading time affects your bottom line," 2011 (accessed Mar, 2018).

[6] D. F. Galletta, R. Henry, S. McCoy, and P. Polak, "Web site delays: How tolerant are users?" *Journal of the Association for Information Systems*, vol. 5, no. 1, p. 1, 2004.

[7] J. Mars, L. Tang, and R. Hundt, "Heterogeneity in "homogeneous" warehouse-scale computers: A performance opportunity," *IEEE Computer Architecture Letters*, vol. 10, no. 2, pp. 29–32, 2011.

[8] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proceedings of the Third ACM Symposium on Cloud Computing.* ACM, 2012, p. 7.

[9] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," in *ACM SIGPLAN Notices*, vol. 48, no. 4. ACM, 2013, pp. 77–88.

[10] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics." in *NSDI*, vol. 2, 2017, pp. 4–2.

[11] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *Acm Sigact News*, vol. 33, no. 2, pp. 51–59, 2002.

[12] D. Abadi, "Consistency tradeoffs in modern distributed database system design: Cap is only part of the story," *Computer*, vol. 45, no. 2, pp. 37–42, 2012.

[13] S. A. Noghabi, S. Subramanian, P. Narayanan, S. Narayanan, G. Holla, M. Zadeh, T. Li, I. Gupta, and R. H. Campbell, "Ambry: Linkedin's scalable geo-distributed object store," in *Proc. SIGMOD.* ACM, 2016.

[14] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, "Streamz: Stateful stream processing at scale," *Proc. VLDB*, 2017.

[15] S. A. Noghabi, J. Kolb, P. Bodik, and E. Cuervo, "Steel: Simplified development and deployment of edge-cloud applications," in *Proc. HotCloud*. USENIX, 2018.

[16] S. A. Noghabi, J. Kolb, P. Bodik, E. Cuervo, I. Gupta, and R. H. Campbell, "Steel: Simplified management and optimization of edge-cloud iot applications," *(in progress) to be submitted to ACM Symposium on Cloud Computing (SoCC)*, 2018.

[17] Docker, "Docker community passes two billion pulls," https://blog.docker.com/2016/02/docker-hub-two-billion-pulls/, 2016.

[18] T. Yu, S. A. Noghabi, S. Raindel, H. Liu, J. Padhye, and V. Sekar, "Freeflow: High performance container networking," in *Proc. HotNet*. ACM, 2016.

[19] "Docker," http://www.docker.com/.

[20] "Kubernetes," http://kubernetes.io/.

[21] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *Proc. USENIX NSDI*, 2011.

[22] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar et al., "Apache hadoop YARN: Yet Another Resource Negotiator," in *Proc. SOSP*. ACM, 2013, p. 5.

[23] "CoreOS," https://coreos.com/.

[24] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," in *Proceeding of the ACM SIGOPS Operating Systems Review (OSR)*, no. 2, 2010.

[25] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proceeding of the ACM SIGOPS Operating Systems Review (OSR)*, 2007.

[26] A. Moniruzzaman and S. A. Hossain, "Nosql database: New era of databases for big data analytics-classification, characteristics and comparison," *arXiv preprint arXiv:1307.0191*, 2013.

[27] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh, "Consistency-based service level agreements for cloud storage," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 309–324.

[28] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, 2008.

[29] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser et al., "Spanner: Google's globally-distributed database," in *Proceeding of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[30] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "Pnuts: Yahoo!'s hosted data serving platform," *In Proceeding of the Very Large Data Bases Endowment (VLDB)*, vol. 1, no. 2, 2008.

[31] L. Qiao, K. Surlaker, S. Das, T. Quiggle, B. Schulman et al., "On brewing fresh espresso: Linkedin's distributed data serving platform," in *Proc. SIGMOD*. ACM, 2013, pp. 1135–1146.

[32] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann, "Consistency rationing in the cloud: pay only when it matters," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 253–264, 2009.

[33] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, 2015.

[34] Apache, "Flink," https://flink.apache.org.

[35] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman et al., "Millwheel: fault-tolerant stream processing at internet scale," *Proc. VLDB*, pp. 1033–1044, 2013.

[36] M. Zaharia, T. Das, H. Li et al., "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in *Proc. HotCloud*, 2012, pp. 10–10.

[37] "Trident," http://storm.apache.org/Trident-tutorial.html.

[38] Apache, "Hadoop," http://hadoop.apache.org/.

[39] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets." in *Proc. HotCloud*, 2010, p. 95.

[40] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino, "Apache tez: A unifying framework for modeling and building data processing applications," in *Proceedings of the 2015 ACM SIGMOD international conference on Management of Data*. ACM, 2015, pp. 1357–1369.

[41] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka et al., "Hive: A warehousing solution over a map-reduce framework," *Proc. VLDB*, vol. 2, no. 2, pp. 1626–1629, 2009.

[42] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in *Proc. SIGMOD*. ACM, 2008, pp. 1099–1110.

[43] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Proceeding of the IEEE Mass Storage Systems and Technologies (MSST)*, 2010.

[44] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and implementation of the Sun network file system," in *Proceeding of the USENIX Summer Technical Conference*, 1985.

[45] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *Proceeding of the ACM SIGOPS Operating Systems Review (OSR)*, 2003.

[46] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. Rosenthal, and F. D. Smith, "Andrew: A distributed personal computing environment," *Communications of the ACM (CACM)*, vol. 29, no. 3, 1986.

[47] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceeding of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[48] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel, "Finding a needle in Haystack: Facebook's photo storage." in *Proceeding of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[49] Hortonworks, "Ozone: An object store in HDFS," http://hortonworks.com/blog/ozone-object-store-hdfs/, 2014 (accessed Mar, 2016).

[50] A. Auradkar, C. Botev, S. Das, D. De Maagd, A. Feinberg, P. Ganti, L. Gao, B. Ghosh, K. Gopalakrishna et al., "Data infrastructure at LinkedIn," in *Proceeding of the IEEE International Conference on Data Engineering (ICDE)*, 2012.

[51] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar et al., "F4: Facebook's warm blob storage system," in *Proceeding of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[52] Twitter, "Blobstore: Twitter's in-house photo storage system," https://blog.twitter.com/2012/blobstore-twitter-s-in-house-photo-storage-system, 2011 (accessed Mar, 2016).

[53] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems." in *Proceeding of the USENIX Annual Technical Conference (ATC)*, 2010.

[54] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for Internet applications," in *Proceeding of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2001.

[55] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, "CRUSH: Controlled, scalable, decentralized placement of replicated data," in *Proceeding of the IEEE High Performance Computing, Networking, Storage and Analysis (SC)*, 2006.

[56] J. Kreps, N. Narkhede, J. Rao et al., "Kafka: A distributed messaging system for log processing," in *Proceeding of the USENIX Networking Meets Databases Workshop (NetDB)*, 2011.

[57] D. Stancevic, "Zero copy I: User-mode perspective," *Linux Journal*, vol. 2003, no. 105, 2003.

[58] "Bonnie++," http://www.coker.com.au/bonnie++/, 2001 (accessed Mar, 2016).

[59] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 1, 1992.

[60] M. Seltzer, K. Bostic, M. K. Mckusick, and C. Staelin, "An implementation of a log-structured file system for UNIX," in *Proceeding of the USENIX Winter Technical Conference*, 1993.

[61] G. R. Ganger and M. F. Kaashoek, "Embedded inodes and explicit grouping: Exploiting disk bandwidth for small files." in *Proceeding of the USENIX Annual Technical Conference (ATC)*, 1997.

[62] Z. Zhang and K. Ghose, "hFS: A hybrid file system prototype for improving small file and metadata performance," in *Proceeding of the ACM European Conference on Computer Systems (EuroSys)*, 2007.

[63] S. J. Mullender and A. S. Tanenbaum, "Immediate files," *Software: Practice and Experience*, vol. 14, no. 4, 1984.

[64] K. Ren, Q. Zheng, S. Patil, and G. Gibson, "Indexfs: Scaling file system metadata performance with stateless caching and bulk insertion," in *Proceeding of the IEEE High Performance Computing, Networking, Storage and Analysis (SC)*, 2014.

[65] E. K. Lee and C. A. Thekkath, "Petal: Distributed virtual disks," in *Proceeding of the ACM Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1996.

[66] Oracle, "Database securefiles and large objects developer's guide," https://docs.oracle.com/database/121/ADLOB/toc.htm, 2011(accessed Mar, 2016).

[67] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu et al., "Windows Azure storage: A highly available cloud storage service with strong consistency," in *Proceeding of the ACM Symposium on Operating Systems Principles (SOSP)*, 2011.

[68] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," *SIGMOD Record*, vol. 34, no. 4, pp. 42–47, 2005.

[69] N. Marz and J. Warren, *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*, 1st ed. Manning Publications Co., 2015.

[70] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *CACM*, vol. 51, no. 1, pp. 107–113, 2008.

[71] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel et al., “Storm@ twitter,” in *Proc. SIGMOD*, 2014, pp. 147–156.

[72] S. Kulkarni, N. Bhagat, M. Fu et al., “Twitter heron: Stream processing at scale,” in *Proc. SIGMOD*, 2015, pp. 239–250.

[73] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, “S4: Distributed stream computing platform,” in *Prod. ICDM Workshop*. IEEE, 2010, pp. 170–177.

[74] T. Akidau, R. Bradshaw, C. Chambers et al., “The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing,” *Proc. VLDB*, vol. 8, no. 12, pp. 1792–1803, 2015.

[75] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, “Integrating scale out and fault tolerance in stream processing using operator state management,” in *Proc. SIGMOD*. ACM, 2013, pp. 725–736.

[76] M. Pundir, L. M. Leslie, I. Gupta, and R. H. Campbell, “Zorro: Zero-cost reactive failure recovery in distributed graph processing,” in *Proc. SoCC*. ACM, 2015, pp. 195–208.

[77] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack et al., “The design of the Borealis stream processing engine.” in *Proc. CIDR*, 2005, pp. 277–289.

[78] Microsoft, “Azure event hub,” https://azure.microsoft.com/en-us/services/event-hubs/.

[79] Amazon, “Kinesis,” https://aws.amazon.com/kinesis/.

[80] “Databus,” https://github.com/linkedin/databus.

[81] “Powered by samza,” https://cwiki.apache.org/confluence/display/SAMZA/Powered+By.

[82] Apache, “Kafaka - powered by,” https://cwiki.apache.org/confluence/display/KAFKA/Powered+By.

[83] Oracle, “Package java.util.stream,” https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html.

[84] L. Xu, B. Peng, and I. Gupta, “Stela: Enabling stream processing systems to scale-in and scale-out on-demand,” in *Proc. IC2E*, 2016, pp. 22–31.

[85] “RocksDB,” http://rocksdb.org.

[86] L. Engineering, “Benchmarking apache kafka: 2 million writes per second (on three cheap machines),” https://engineering.linkedin.com/kafka.

[87] T. Rabl, S. Gómez-Villamor, M. Sadoghi et al., "Solving big data challenges for enterprise application performance management," *Proc. VLDB*, vol. 5, no. 12, pp. 1724–1735, 2012.

[88] E. P. Corporation, "Benchmarking top nosql databases," *Technical Report*, p. 19, 2015.

[89] A. AWS, "Lambda," https://aws.amazon.com/lambda/.

[90] Apache, "Beam," http://beam.incubator.apache.org.

[91] Amazon, "DynamoDB streams," http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Streams.html.

[92] P. Hintjens, *ZeroMQ: Messaging for Many Applications.* O'Reilly Media, Inc., 2013.

[93] "MongoDB," https://www.mongodb.com.

[94] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Making state explicit for imperative big data processing," in *Proc. ATC.* USENIX, 2014, pp. 49–60.

[95] T. A. S. Foundation, "Apache HBase," http://hbase.apache.org/.

[96] MySQL, "Mysql," http://www.mysql.com.

[97] Couchbase, "Couchbase," http://www.couchbase.com.

[98] S. Sanfilippo, "Redis," http://redis.io.

[99] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein et al., "Mapreduce online." in *Proc. NSDI*, 2010, pp. 20–25.

[100] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez, "Streamcloud: An elastic and scalable data streaming system," *TPDS*, vol. 23, no. 12, pp. 2351–2365, 2012.

[101] J. Meehan, N. Tatbul, S. Zdonik, C. Aslantas, U. Cetintemel et al., "S-store: Streaming meets transaction processing," *Proc. VLDB*, pp. 2134–2145, 2015.

[102] J.-H. Hwang, M. Balazinska, A. Rasin, U. Cetintemel et al., "High-availability algorithms for distributed stream processing," in *Proc. ICDE'05*, 2005, pp. 779–790.

[103] Y. Gu, Z. Zhang, F. Ye, H. Yang, M. Kim, H. Lei, and Z. Liu, "An empirical study of high availability in stream processing systems," in *Proc. Middleware*, 2009, p. 23.

[104] W. Lin, Z. Qian, J. Xu, S. Yang, J. Zhou, and L. Zhou, "Streamscope: continuous reliable distributed processing of big data streams," in *Proc. NSDI*, 2016, pp. 439–454.

[105] G. Jacques-Silva, F. Zheng, D. Debrunner, K.-L. Wu, V. Dogaru et al., "Consistent regions: Guaranteed tuple processing in ibm streams," *Proc. VLDB*, vol. 9, no. 13, pp. 1341–1352, 2016.

[106] R. Wagle, H. Andrade, K. Hildrum, C. Venkatramani et al., "Distributed middleware reliability and fault tolerance support in system s," in *Proc. DEBS*, 2011, pp. 335–346.

[107] K. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *TOCS*, pp. 63–75, 1985.

[108] Z. Sebepou and K. Magoutis, "Cec: Continuous eventual checkpointing for data stream processing operators," in *Proc. IEEE/IFIP DSN*, 2011, pp. 145–156.

[109] R. Fernandez, P. Pietzuch et al., "Liquid: Unifying nearline and offline big data integration," in *Proc. CIDR*, p. 8.

[110] B. Liu, Y. Zhu, and E. Rundensteiner, "Run-time operator state spilling for memory intensive long-running queries," in *Proc. SIGMOD*. ACM, 2006, pp. 347–358.

[111] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King et al., "SPC: a distributed, scalable platform for data mining," in *Proc. IWMSSP*. ACM, 2006, pp. 27–37.

[112] "Gartner Says 4.9 Billion Connected "Things" Will Be in Use," http://www.gartner.com/newsroom/id/2905717, accessed: 02/2018.

[113] D. Evans, "The Internet of Things. How the next evolution of the Internet is changing everything," Tech. Rep., 2011.

[114] "Azure Internet of Things," https://azure.microsoft.com/en-us/suites/iot-suite/, accessed: 02/2018.

[115] "Google Cloud IoT Platform," https://cloud.google.com/iot/platform, accessed: 02/2018.

[116] "AWS Internet of Things," https://aws.amazon.com/iot/, accessed: 02/2018.

[117] "C3 Internet of Things," https://c3iot.com/, accessed: 02/2018.

[118] "Azure Certified for IoT device catalog - Preview," https://catalog.azureiotsuite.com/, accessed: 08/2017.

[119] "Predictive maintenance preconfigured solution walkthrough," https://docs.microsoft.com/en-us/azure/iot-suite/iot-suite-predictive-walkthrough, accessed: 08/2017.

[120] "Google Cloud IoT Solutions," https://cloud.google.com/solutions/iot/, accessed: 02/2018.

[121] "Deploy Azure Function as an IoT Edge module - preview," https://docs.microsoft.com/en-us/azure/iot-edge/tutorial-deploy-function, accessed: 08/2017.

[122] "Azure IoT Edge," https://azure.microsoft.com/en-us/services/iot-edge/, accessed: 08/2017.

[123] C. Pahl and B. Lee, "Containers and clusters for edge cloud architectures–a technology review," in *Future Internet of Things and Cloud (FiCloud), 2015 3rd International Conference on*. IEEE, 2015, pp. 379–386.

[124] B. I. Ismail, E. M. Goortani, M. B. Ab Karim, W. M. Tat, S. Setapa, J. Y. Luke, and O. H. Hoe, "Evaluation of docker as edge computing platform," in *Open Systems (ICOS), 2015 IEEE Confernece on*. IEEE, 2015, pp. 130–135.

[125] K. Ha, Y. Abe, Z. Chen, W. Hu, B. Amos, P. Pillai, and M. Satyanarayanan, "Adaptive vm handoff across cloudlets," *Technical Report CMU-CS-15–113, CMU School of Computer Science*, 2015.

[126] B. Ottenwälder, B. Koldehofe, K. Rothermel, and U. Ramachandran, "Migcep: operator migration for mobility driven distributed complex event processing," in *Proceedings of the 7th ACM international conference on Distributed event-based systems*. ACM, 2013, pp. 183–194.

[127] S. Wang, R. Urgaonkar, M. Zafer, T. He, K. Chan, and K. K. Leung, "Dynamic service migration in mobile edge-clouds," in *IFIP Networking Conference (IFIP Networking), 2015*. IEEE, 2015, pp. 1–9.

[128] "AWS Greengrass," https://aws.amazon.com/greengrass/, accessed: 08/2017.

[129] "Datadog – Modern monitoring & analytics," https://www.datadoghq.com/, accessed: 08/2017.

[130] J. Kaldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O'Neill, K. W. Ong, B. Schaller, P. Shan, B. Viscomi et al., "Canopy: An end-to-end performance tracing and analysis system," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 34–50.

[131] "Stackdriver Monitoring," https://cloud.google.com/monitoring/, accessed: 08/2017.

[132] "Amazon CloudWatch," https://aws.amazon.com/cloudwatch/, accessed: 08/2017.

[133] "Applciation Insights," https://azure.microsoft.com/en-us/services/application-insights/, accessed: 08/2017.

[134] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, p. 18.

[135] B. Sharma, V. Chudnovsky, J. L. Hellerstein, R. Rifaat, and C. R. Das, "Modeling and synthesizing task placement constraints in google compute clusters," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 2011, p. 3.

[136] J. W. Park, A. Tumanov, A. Jiang, M. A. Kozuch, and G. R. Ganger, "3sigma: distribution-based cluster scheduling for runtime uncertainty," in *Proceedings of the Thirteenth EuroSys Conference*. ACM, 2018, p. 2.

[137] D. Vasisht, Z. Kapetanovic, J. Won, X. Jin, R. Chandra, S. N. Sinha, A. Kapoor, M. Sudarshan, and S. Stratman, "FarmBeats: An iot platform for data-driven agriculture," in *NSDI*, 2017, pp. 515–529.

[138] M. Satyanarayanan, G. Lewis, E. Morris, S. Simanta, J. Boleng, and K. Ha, "The role of cloudlets in hostile environments," *IEEE Pervasive Computing*, vol. 12, no. 4, pp. 40–49, 2013.

[139] "What is Azure IoT Suite," https://docs.microsoft.com/en-us/azure/iot-suite/iot-suite-what-are-preconfigured-solutions, accessed: 08/2017.

[140] K. Hong, D. Lillethun, U. Ramachandran, B. Ottenwälder, and B. Koldehofe, "Mobile fog: A programming model for large-scale applications on the internet of things," in *Proceedings of the second ACM SIGCOMM workshop on Mobile cloud computing*. ACM, 2013, pp. 15–20.

[141] "Multi-access Edge Computing," http://www.etsi.org/technologies-clusters/technologies/multi-access-edge-computing, accessed: 08/2017.

[142] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin, "Orleans: cloud computing for everyone," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 2011, p. 16.

[143] B. Chandramouli, J. Claessens, S. Nath, I. Santos, and W. Zhou, "Race: Real-time applications over cloud-edge," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 625–628.

[144] M. Satyanarayanan, P. Simoens, Y. Xiao, P. Pillai, Z. Chen, K. Ha, W. Hu, and B. Amos, "Edge analytics in the internet of things," *IEEE Pervasive Computing*, vol. 14, no. 2, pp. 24–31, 2015.

[145] Y. Elkhatib, B. Porter, H. B. Ribeiro, M. F. Zhani, J. Qadir, and E. Rivière, "On using micro-clouds to deliver the fog," *IEEE Internet Computing*, vol. 21, no. 2, pp. 8–15, 2017.

[146] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE pervasive Computing*, vol. 8, no. 4, 2009.

[147] E. A. Lee, B. Hartmann, J. Kubiatowicz, T. S. Rosing, J. Wawrzynek, D. Wessel, J. Rabaey, K. Pister, A. Sangiovanni-Vincentelli, S. A. Seshia et al., "The swarm at the edge of the cloud," *IEEE Design & Test*, vol. 31, no. 3, pp. 8–20, 2014.

[148] K. Ha and M. Satyanarayanan, "Openstack++ for cloudlet deployment," *School of Computer Science Carnegie Mellon University Pittsburgh*, 2015.

[149] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile edge computing: A key technology towards 5g," *ETSI white paper*, vol. 11, no. 11, pp. 1–16, 2015.

[150] K. Bhardwaj, S. Sreepathy, A. Gavrilovska, and K. Schwan, "Ecc: Edge cloud composites," in *Mobile Cloud Computing, Services, and Engineering (MobileCloud), 2014 2nd IEEE International Conference on*. IEEE, 2014, pp. 38–47.

[151] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *Proceedings of the 8th international conference on Mobile systems, applications, and services*. ACM, 2010, pp. 49–62.

[152] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *Proceedings of the sixth conference on Computer systems*. ACM, 2011, pp. 301–314.

[153] P. Bahl, R. Y. Han, L. E. Li, and M. Satyanarayanan, "Advancing the state of mobile cloud computing," in *Proceedings of the third ACM workshop on Mobile cloud computing and services*. ACM, 2012, pp. 21–28.

[154] Y. Zhu, H. Eran, D. Firestone, C. Guo et al., "Congestion control for large-scale RDMA deployments," in *ACM SIGCOMM*, vol. 45, no. 4, 2015.

[155] L. Rizzo, "Netmap: a novel framework for fast packet i/o," in *USENIX Security Symposium*, 2012.

[156] "Data plane development kit (DPDK)," http://dpdk.org/, accessed 2016.

[157] Mellanox, "RDMA aware networks programming user manual," http://www.mellanox.com/, Rev 1.7.

[158] J. Hwang, K. Ramakrishnan, and T. Wood, "Netvm: high performance and flexible networking using virtualization on commodity platforms," *IEEE Transactions on Network and Service Management*, vol. 12, no. 1, 2015.

[159] P. S. I. Group, "Single root I/O virtualization," http://pcisig.com/specifications/iov/single_root/, accessed 2016.

[160] "Kubernetes," http://kubernetes.io/.

[161] "CoreOS," https://coreos.com/, accessed 2016.

[162] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *ACM SIGCOMM*, 2014.

[163] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," in *ACM SIGCOMM*, 2011.

[164] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using mantri." in *USENIX OSDI*, 2010.

[165] M. Chowdhury and I. Stoica, “Efficient coflow scheduling without prior knowledge,” in *ACM SIGCOMM*, 2015.

[166] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, “Farm: fast remote memory,” in *USENIX NSDI*, 2014.

[167] M. Li, D. G. Andersen, A. J. Smola, and K. Yu, “Communication efficient distributed machine learning with the parameter server,” in *Advances in Neural Information Processing Systems*, 2014.

[168] Docker, “Devops page on docker website,” https://www.docker.com/use-cases/devops, accessed 2016.

[169] Iron.io, “Docker in production – what we’ve learned launching over 300 million containers,” https://www.iron.io/docker-in-production-what-weve-learned/, 2014.

[170] Datadog, “8 suprising facts about real Docker adoption,” https://www.datadoghq.com/docker-adoption/, 2016.

[171] FreeBDS, “chroot – FreeBDS Man Pages,” http://www.freebsd.org/cgi/man.cgi, FreeBDS 10.3 Rel.

[172] D. Merkel, “Docker: Lightweight linux containers for consistent development and deployment,” *Linux J.*, vol. 2014, no. 239, 2014.

[173] A. Madhavapeddy, T. Leonard, M. Skjegstad, T. Gazagnaire et al., “Jitsu: Just-in-time summoning of unikernels,” in *USENIX NSDI)*, 2015.

[174] “Weave Net,” https://www.weave.works/, accessed 2016.

[175] “Project calico,” https://www.projectcalico.org/.

[176] S. Hefty, “Rsockets,” in *OpenFabris International Workshop*, 2012.

[177] P. Balaji, S. Narravula, K. Vaidyanathan, S. Krishnamoorthy, J. Wu, and D. K. Panda, “Sockets direct protocol over infiniband in clusters: is it beneficial?” in *IEEE ISPASS*, 2004.

[178] M. Fox, C. Kassimis, and J. Stevens, “IBM’s Shared Memory Communications over RDMA (SMC-R) Protocol,” RFC 7609 (Informational), Internet Engineering Task Force, 2015.

[179] J. Liu, J. Wu, and D. K. Panda, “High performance RDMA-based MPI implementation over infiniband,” *International Journal of Parallel Programming*, vol. 32, no. 3, 2004.

[180] L. Rizzo and G. Lettieri, “Vale, a switched ethernet for virtual machines,” in *ACM CoNEXT*, 2012.

[181] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, “Clickos and the art of network function virtualization,” in *USENIX NSDI*, 2014.

[182] A. Ranadive and B. Davda, "Toward a paravirtual vRDMA device for VMware ESXi guests," *VMware Technical Journal, Winter 2012*, vol. 1, no. 2, 2012.

[183] L. Liss, "Containing RDMA and high performance computing," in *ContainerCon*, 2015.

[184] B. Goglin and S. Moreaud, "Knem: A generic and scalable kernel-assisted intra-node MPI communication framework," *Journal of Parallel and Distributed Computing*, vol. 73, no. 2, 2013.

[185] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. Barrett, R. Brightwell, W. D. Gropp, V. Kale, and R. Thakur, "Mpi + mpi: A new hybrid approach to parallel programming with mpi plus shared memory," *Computing*, vol. 95, 2013.

[186] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes," in *Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 2009.

[187] J. Brandeburg, "Reducing network latency in linux," in *Linux Plumbers Conference*, 2012.

[188] J. Wang, K.-L. Wright, and K. Gopalan, "Xenloop: A transparent high performance inter-vm network loopback," in *ACM HPDC*, 2008.

[189] J. Claassen, R. Koning, and P. Grosso, "Linux containers networking: Performance and scalability of kernel modules," in *IEEE/IFIP NOMS*, 2016.

[190] S. A. Noghabi, "Auto-scaling containers in samza," https://issues.apache.org/jira/browse/SAMZA-719.

[191] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, "Elastic scaling for data stream processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1447–1463, 2014.

[192] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*. ACM, 2013, pp. 725–736.