‘

Novel Strategies for Responsive Load Balancing in Cloud Applications

by Ratnadeep Bhattacharya

B.Tech in Electrical Engineering, August 2006, West Bengal University of Technology

A Dissertation submitted to

The Faculty of
The School of Engineering and Applied Science
of The George Washington University
in partial satisfaction of the requirements
for the degree of Doctor of Philosophy

February 23, 2024

Dissertation directed by

Dr. Gabriel Palmer
Associate Professor of Computer Science

The School of Engineering and Applied Science of The George Washington University certifies that Ratnadeep Bhattacharya has passed the Final Examination for the degree of Doctor of Philosophy as of February 23, 2024. This is the final and approved form of the dissertation.

Novel Strategies for Responsive Load Balancing in Cloud Applications

Ratnadeep Bhattacharya

Dissertation Research Committee:

Dr. Gabriel Palmer, Associate Professor of Computer Science, Dissertation Director

Dr. Rahul Simha, Professor, Committee Member

Dr. Timothy Wood, Associate Professor, Committee Member

Dr. Howie Huang, Professor, Dept of Electrical and Computer Engineering, The George Washington University, Committee Member

Dedication

Education is not the filling of a pail, but the lighting of a fire!

Acknowledgments

I would like to thank Dr. Timothy Wood for his guidance and patience over the last few years as I made my way to this point.

I would also like to thank my wife, Deepa Rajappa, without whose unwavering support, sacrifices and encouragement, this wouldn't have been possible. This achievement is as much hers as it is mine.

Finally, I would like to thank my kids, Ayush and Daksh Bhattacharya, who allowed us to skip innumerable tubs of ice cream.

Abstract

Novel Strategies for Responsive Load Balancing in Cloud Applications

Recent years have witnessed a substantial surge in cloud adoption, fueled by architectural shifts and the emergence of applications tailored for edge devices. Cloud-deployed applications, typically following the microservices architecture pattern, break down large monolithic structures into smaller, independent components known as microservices. While this approach enhances development flexibility and deployment agility, it introduces an internal network within the application. Consequently, **load balancing** becomes imperative among microservices, creating both a bottleneck and a potential single point of failure in communication.

To address these challenges, the distributed client-side load balancer has been introduced. This architectural shift prompts the adaptation of load balancing algorithms from single-node centralized configurations to the distributed client-side paradigm. However, this transition poses challenges, particularly for dynamic algorithms like Least Connection. While derived from well established and even nearly optimal algorithms like Join-the-Shortest-Queue (JSQ), **these algorithms mostly depend on the availability of accurate and current backend state information, which is missing in distributed client side algorithms**. This discrepancy hampers its effectiveness in distributed client-side scenarios.

In this work, we establish that load balancing algorithms, transposed from the centralized load balancer world, fail to hold up the same properties and actually hamper performance. We recognize that these challenges are unique and the obvious solution is to regain the information lost. The most common approaches for that are to either build centralized state or to pass

messages between the load balancers. However, both those solutions become nearly impossible to scale to applications spanning tens of thousands of microservices over data centers spread geographically. Thus we propose that **"distributed client-side load balancing" is a distinct subfield within the broader domain of load balancing** that requires further exploration. More specifically, in this work we:

- Build algorithms that use **feedback mechanisms** to rectify the information loss in distributed load balancing that most algorithms rely on.
- Enable upstream nodes to manage their own performance.
- Enable the system as a whole to quickly react to overload conditions.

Our results show that for most applications types, our algorithms can **reduce response time variability over current state-of-the-art by 2-4 times and 99^{th} percentile latency by about 2 times**.

Moreover, a **new wave of applications** operates on edge devices, where one set generates inputs interpreted by cloud-based services, triggering actions on other edge devices. Such applications demand asynchronous connectivity while adhering to predefined time-bound response requirements. Traditional asynchronous application designs, in the **absence of any temporal bound on message delivery**, struggle to ensure response time guarantees. While it is easy to introduce load balancing here to extract better performance, most of these applications need to maintain a state in the flow of their messages. Hence the goal in this part of our work is find a balance between loss of performance and the cost of rebuilding state in the flow of messages. Towards that goal, we propose and discuss:

- Methods that allow us to **incorporate synchrony within an asynchronous network**.
- Mechanisms to **rebuild state split from moving streams between multiple leaders**, that is required for the above.
- Algorithm that can handle dynamically changing load whose characteristics are unknown ahead of time.

Our results in this area show that **99^{th} percentile latency can be reduced by as much as 73%**. More importantly, we found that in our experiments the latency profile remained nearly constant when varying the service cost, whereas Apache Kafka's latency increased super-linearly.

Table of Contents

List of Figures

Preface

In the ever-evolving landscape of cloud computing and microservices architectures, this work embarks on a journey to unravel the intricacies of load balancing — a pivotal aspect that not only underpins the seamless operation of distributed systems but also plays a crucial role in the domain of traffic engineering. As organizations increasingly embrace cloud-first strategies and microservices patterns, the need for efficient load balancing becomes paramount for optimizing resource utilization by intelligently managing network traffic flows.

The genesis of this research lies in the recognition of a gap - traditional load balancing algorithms, honed in the monolithic era, failing to serve today's more varied and convoluted requirements. Modern distributed applications extend all the way from the almighty cloud to the puny embedded devices, often even connecting the two. They have forced software and infrastructure engineers to rethink their deployment strategies, giving rise to the "microservices" architecture and the proliferation of asynchronous communication alongside the more established client-server models. As distributed systems rose from obscurity to claim its righful place as the successor of the famed "Moore's Law", modern computing systems entered a brave new phase. Unfortunately though, it seems that not many noticed the age-old load balancing algorithms cracking under the pressure.

My thesis points a finger at some of these ever-widening cracks. Through a series of carefully crafted projects — Mu, BLOC, and SMALOPS — this work not only identifies the shortcomings in existing approaches but also charts a course toward novel solutions designed specifically for this new landscape and the intricacies of modern traffic engineering.

Acknowledging the collaborative nature of research, I extend my gratitude

to the mentors, colleagues, and contributors who have been instrumental in shaping this work. Their insights, feedback, and support have enriched the depth and breadth of the exploration undertaken.

The narrative unfolds through a detailed examination of each project — Mu, with its focus on optimizing load balancing in response to server heterogeneity; BLOC, navigating the challenges of distributed client-side load balancing and its implications; and SMALOPS, venturing into the intricate realm of load balancing in asynchronous applications. Each project contributes a unique perspective to the overarching theme, collectively presenting a holistic view of load balancing in the modern cloud services era.

As you traverse the pages that follow, I invite you to join me in this exploration, where challenges become opportunities, traditional boundaries are questioned, and innovative solutions emerge. May this work not only contribute to the academic discourse but also serve as a guidepost for practitioners navigating the dynamic landscape of modern computing and traffic engineering.

Ratnadeep Bhattacharya

Feb, 2024

Chapter 1: Introduction

Amidst the dynamic evolution of cloud computing, a transformative paradigm has emerged, reshaping the operational dynamics of organizations. This paradigmatic shift involves relinquishing the reins of infrastructure operation and management—encompassing critical elements such as compute, memory, network, and storage—from internal IT departments to nimble and specialized cloud providers. This strategic realignment is not merely a procedural adjustment but a pivotal transformation that liberates organizations, allowing them to channel their focus and resources toward the very core of their business competencies. Liberated from the intricacies of day-to-day infrastructure maintenance, organizations can now chart a course toward innovation, agility, and unparalleled efficiency. This delegation heralds a new era where the cloud becomes the cornerstone of operational prowess, enabling enterprises to navigate the complexities of the digital landscape with unprecedented flexibility and responsiveness.

The last decade has witnessed a formidable wave of innovation, with industry titans like Amazon Web Services (AWS), Google Cloud, and Microsoft Azure at the forefront, crafting resilient and impregnable cloud platforms. This technological evolution, coupled with the symbiotic integration of orchestration solutions such as Kubernetes [1], Knative [2], proxies like Envoy [3], and sophisticated service meshes exemplified by Istio [4] and Linkerd [5], has heralded a transformative era. This era seamlessly paves the way for the deployment of applications in the cloud, setting the stage for a paradigm shift in deployment strategies. Notably, major industry players such as Netflix, Pinterest, Twitter, and PayPal [6] have embraced this tide, adopting

"cloud-first" or even "cloud-only" deployment strategies. This strategic embrace marks a profound endorsement of the agility, scalability, and resilience afforded by the cloud, signaling a resounding shift in how organizations conceptualize and execute their digital strategies.

At the core of cloud environments lies a dynamic scalability that empowers infrastructure to gracefully expand or contract in response to the undulating tides of workload fluctuations. This inherent capability underscores the critical imperative for a meticulous approach to "right sizing" infrastructure — an artful calibration aimed at optimizing operational expenditure (opex). Operating within this operational range, where infrastructure is compelled to function at or near its maximum capacity, accentuates the need for an intricate dance of load distribution among service instances. This dance, a nuanced choreography, becomes the linchpin in orchestrating operational efficiency, ensuring resources are harnessed with precision to meet the demands of the ever-shifting landscape of workloads.

Microservices architecture emerges as a prevailing pattern for the deployment of cloud applications. Under this paradigm, each facet of business logic assumes the form of a standalone service, deployed as replicas using individual virtual machines (VMs) or containers. These services then intricately communicate to collectively create the overarching application functionality, a stark departure from earlier practices that amalgamated closely related functionalities into single applications.

The proliferation of edge devices, spanning from ubiquitous mobile phones to intricate IoT (Internet of Things) devices, has been instrumental in giving rise to a distinctive class of applications. Traditionally hosted on cloud infrastructure, these applications exhibit a data ingestion model wherein information is gathered from a specific set of sources. Subsequently,

this raw data undergoes a transformative process within the cloud service, culminating in the creation of actionable items tailored for consumption by an entirely different set of devices. In this context, source devices remain indifferent to the fate of data once dispatched to the cloud service. Given the minimal resource footprints of edge devices, the ultimate goal is often to maximize the handling of "events," necessitating the utilization of asynchronous communication. However, many of these applications have a stringent temporal dimension that delineates a critical requirement, stipulating a predefined time frame between the instantiation of data generation and the subsequent reception of the corresponding action item by the designated destination devices. This temporal imperative underscores the delicate balance that must be struck between the agility of asynchronous communication and the imperative for timely action.

Within the domain of asynchronous communication, dominant patterns invariably lean on a message bus to facilitate the exchange of information regarding events between source devices and the cloud application. Despite the widespread adoption of this paradigm, it is noteworthy that all major message buses pivot on the foundational concept of eventual consistency. In fact, asynchronous communication itself is prevalently concerned with “correctness" rather than temporal bounds on service. This prevailing approach, while offering advantages in certain contexts, particularly in terms of flexibility, introduces a notable challenge: the inability to provide definitive constraints on processing times. The consequence of this reliance on eventual consistency is the emergence of a critical challenge with regards to establishing a well-defined temporal boundary between the moment of data generation and the subsequent reception of action items. This challenge arises from the inherent variability in processing times dictated

by the eventual consistency model, which, by its nature, eschews rigid temporal guarantees. Consequently, reconciling the need for asynchronous communication with the imperative for timely action becomes a nuanced pursuit, calling for innovative solutions to navigate this intricate terrain.

This research endeavors to unravel the intricate dynamics of load balancing within microservices architectures, shedding light on the prevalent fallacies that permeate existing methodologies and their profound repercussions on response times across a spectrum of contexts. A critical aspect under scrutiny is the pervasive inclination to transplant successful algorithms from the centralized (monolithic) domain into the microservices realm. This practice, upon closer examination, unveils inherent inefficiencies that inflict severe degradation upon response times. The crux of this inefficiency is traced back to the highly distributed nature of information within the microservices domain, exacerbated by an augmented heterogeneity in the service capacity of backend systems. This heterogeneity, a consequence of various factors such as diverse hardware configurations in resource-constrained environments and interference stemming from a high VM/container-to-server ratio — referred to as the packing ratio — exerts a significant impact on the efficacy of load balancing algorithms. The ensuing challenge lies in addressing this distributed and heterogeneous landscape, necessitating innovative solutions tailored to the unique intricacies of microservices architectures.

In response to the distinctive demands posed by the microservices pattern, this research introduces a suite of specialized systems meticulously crafted to navigate the intricate challenges inherent in this architectural paradigm. A pivotal revelation surfaces during this exploration—the **recognition of "distributed load balancing" as a distinctive field in its own**

right, warranting dedicated and nuanced research to comprehend its complexities fully. Simultaneously, the research illuminates the preeminence of straightforward yet effective feedback-based load balancing algorithms over the established "state-of-the-art," marking a significant leap forward in optimizing microservices performance. **This dual emphasis on defining a new research frontier and presenting practical advancements underscores the research's comprehensive and forward-thinking contribution to the field of load balancing in microservices architectures**.

In a strategic shift towards the realm of asynchronous applications, the research reaches its apex with a groundbreaking project. This endeavor involves the construction of a sophisticated framework, strategically positioned atop Apache Kafka — a widely adopted message bus. The primary objective of this project is to **one, use intelligent load balancing algorithms to reduce message queuing times - effectively building a notion of synchrony into asynchronous systems, and two, to discuss some of the controls needed to reconcile stream state that was split by load balancing**. This final project, serving as the culmination of the research journey, not only adds a significant layer to the existing body of knowledge but also underscores the indispensable role of load balancing in optimizing the performance of asynchronous applications as well.

1.1 Background Knowledge

A significant chunk of modern applications are developed and deployed using the microservices pattern. These microservices communicate using some “service mesh" or “message bus", using what is known as the “event driven architecture". Since our work focuses on these patterns, we will focus on formally defining these and related ideas in this section.

1.1.1 Microservices Architecture

Microservices architecture is a design approach for developing software applications as a set of small, independently deployable services. In contrast to traditional monolithic architectures, where an entire application is developed as a single, tightly integrated unit, microservices break down an application into a collection of loosely coupled services. Each microservice represents a specific business capability and can be developed, deployed, and scaled independently.

Key characteristics of microservices architecture include:

Modularity: The application is divided into small, self-contained services, each responsible for a specific function or feature.

Independent Deployment: Microservices can be developed and deployed independently of each other. This allows for more frequent updates and releases, reducing the risk associated with changes.

Loose Coupling: Services are independent entities that communicate with each other through well-defined APIs (Application Programming Interfaces). This loose coupling makes it easier to replace or upgrade individual services without affecting the entire system.

Scalability: Each microservice can be scaled independently based on its specific usage patterns, allowing for more efficient resource utilization.

Resilience: Failure in one microservice does not necessarily bring down the entire application. The system can gracefully degrade, with other services continuing to function.

Technology Diversity: Different services within a microservices architecture can be implemented using different technologies, tools, and programming languages, based on the specific requirements of each service.

Autonomous Development Teams: Microservices architecture often

aligns with an organizational structure where small, cross-functional teams are responsible for individual microservices. This autonomy enhances development speed and flexibility.

Continuous Integration and Deployment (CI/CD): The independent nature of microservices facilitates the adoption of CI/CD practices, enabling automated testing, integration, and deployment.

Despite these advantages, microservices architecture also introduces challenges, such as increased complexity in managing distributed systems, inter-service communication, and data consistency across services.

Generally, the application is deployed as a group of services. Each service, in turn, is a cluster of instances or “pods" that run the same business logic. These services are glued together using either synchronous or asynchronous communication patterns. In synchronous applications, the services are ordered in “layers", containing one or more dependent services, with each layer of services making requests to the next layer. While the services within a layer may only be partially ordered, the layers themselves tend to have a total ordering. With the asynchronous application, services are far more loosely coupled and a total ordering may or may not exist.

Typically, entry to a microservices based application is through an “ingress" gateway that load balances incoming requests to a frontend service. In case of synchronous applications, the layers talk to each other directly sending a request and waiting for the corresponding response. In such cases, the “downstream" service nodes (the client or the requester) contain a “sidecar", a secondary node operating in the same network namespace, load balancer (client side). These load balancers select “upstream" service nodes (service provider instance or “pod") to which to direct individual requests. In the case of asynchronous applications, the services use a fire

and forget model. In this model, the services themselves do not care about the response and the request is simply sent to a message bus that is used to glue services together. Since the message bus forms its own separate application with its own ingress gateway, the client side distributed load balancer is done away with. The load balancing, however, moves inside the message bus application.

1.1.2 Service Mesh

A service mesh is a dedicated infrastructure layer that facilitates communication, management, and control between microservices in a distributed application. It acts as a configurable, low-latency communication fabric that allows services to communicate with each other reliably, securely, and efficiently. Service meshes provide a set of features to address common challenges associated with microservices architectures, such as service discovery, load balancing, security, observability, and traffic management.

Key features of a service mesh include:

Service Discovery: Service meshes automate the process of service discovery, allowing microservices to locate and communicate with each other without hard-coded addresses. This dynamic discovery simplifies the deployment and scaling of microservices.

Load Balancing: Service meshes implement load balancing strategies to distribute incoming requests across multiple instances of a service, ensuring optimal utilization of resources and improved fault tolerance.

Security: Service meshes enhance security by providing features like encryption (often through mutual TLS), authentication, and authorization. They can enforce policies to control which services are allowed to communicate and under what conditions.

Observability: Service meshes offer tools for monitoring and logging, allowing developers and operators to gain insights into the behavior and performance of microservices. Metrics, logs, and traces help diagnose issues and optimize system performance.

Traffic Management: Service meshes enable sophisticated traffic management capabilities, including A/B testing, canary releases, and blue-green deployments. These features allow controlled rollout of new features and updates to minimize risks.

Circuit Breaking: To prevent cascading failures, service meshes can implement circuit-breaking mechanisms. If a service becomes unhealthy or unresponsive, the service mesh can intelligently limit the flow of requests to that service, preserving overall system stability.

Retry Mechanisms: Service meshes can automatically handle retries for failed requests, reducing the impact of transient failures and improving the overall resilience of the application.

Service meshes are particularly valuable in large, complex microservices architectures, where the ability to manage communication and interactions between services is crucial for maintaining a reliable and secure application.

1.1.3 Event Driven Architecture

Event-driven architecture (EDA) is a software design paradigm that emphasizes the production, detection, consumption, and reaction to events that occur within a system. An event is a significant change in state or a notable occurrence that a system can detect and respond to. In an event-driven architecture, the flow of the application is determined by events, promoting loose coupling between components and allowing for better scalability, flexibility, and responsiveness.

Key components and concepts of event-driven architecture include:

Events: Events are notifications or signals that something of interest has happened. These can include user actions, system notifications, or changes in data. Events are typically categorized into two types: commands (indicating a request for a specific action) and events (indicating that something has happened).

Event Producers: These are components or services responsible for generating and emitting events when a relevant state change occurs. Event producers publish events to an event bus or a similar mechanism that facilitates communication.

Event Bus: An event bus is a communication channel that allows event producers to publish events, and event consumers to subscribe and receive those events. It acts as an intermediary through which events are transmitted between components.

Event Consumers: These are components or services that subscribe to specific events of interest. When an event occurs, the event consumer reacts by performing some action. Event consumers are often decoupled from event producers, allowing them to operate independently.

Decoupling: One of the core principles of event-driven architecture is decoupling, which means that components are independent and unaware of each other. Changes in one component do not directly affect others. This promotes modularity and flexibility, making it easier to modify or replace components without disrupting the entire system.

Scalability: Event-driven architectures are inherently scalable. New services can be added without impacting existing components, and the system can easily adapt to changes in load by distributing events efficiently.

Asynchronous Processing: Events are often processed asynchronously,

meaning that the sender and receiver are not required to interact in real-time. This asynchronous nature improves system responsiveness and allows for better handling of intermittent failures.

Event Sourcing: Event sourcing is a related concept where the state of an application is determined by a sequence of events rather than the current state. This approach provides a comprehensive audit trail of changes and enables the reconstruction of system state at any point in time.

In the microservices domain, event-driven patterns are common in microservices to enable communication between independently deployed services.

1.2 Projects

In the course of our work, we discovered that the load balancing algorithms, in both service meshes (synchronous communication) and event-driven architectures (asynchronous communication), lead directly to the over-provisioning of resources. When deployed in cloud environments, this in turn leads to higher operating cost and/or lower performance. Furthermore, to the best of our knowledge, even the research community seems oblivious to this particular impediment. Thus, it is our belief that **distributed load balancing in microservices** is an **hitherto unexplored field** of research.

The major problems that plague load balancing in the microservices domain are:

- **Server Heterogeneity**: In distributed systems, the “pods" (group of logically linked containers) can be scheduled on different physical server with varying resources and interference (due to the number of pods scheduled on the same physical server). This is especially true in an edge system which might deploy a bunch of low-cost servers whose

resource availability can vary wildly. In such cases, despite being deployed from the same configuration, pods can have very different performance profiles.

- **Distributed Information**: In service meshes, the load balancer is typically on the client or request side. This is done in order to avoid single point of failures and enable the system to scale. However, when multiple client load balancers make requests independently to the same set of backend pods, the information gets distributed between them. As such, client load balancers' ability to make correct decisions suffer severely. In fact, this degradation in system performance is proportional to the number of clien load balancers present in the system.

- **Eventual Consistency**: Beyond these, event driven applications further suffer from their eventual consistency designs. Eventual consistency asserts that a system will "eventually" converge to the same state by implementing some form of reconciliation. This is a weaker form of consistency, typically meant for systems where strong consistency poses significant challenges.

In order to intelligently discuss performance in any kind of system, we have to define performance. This entails selecting metric(s) to quantify performance. This is where the design of algorithms become tricky. Some of the major metric classes in this regard are **memory, CPU and I/O**. Most systems use one or the other, depending on the application and combining more than one into a single metric is believed to be tricky, though we are not aware of any specific study in this regard. This led us to investigate systems using **processing time** of an application, which we believe subsumes most

major metric classes. In our work we define processing time as a “deadline" before which an application can finish processing a particular “job". Practically though, this definition is dependent on a predefined notion of a “deadline". Hence, we use a simpler metric while building our systems - we aim to guarantee **each “job" equal time on the system**. We resolve this with the idea of **remaining capacity** on each backend node. This “remaining capacity" is defined as the difference between the number of concurrent jobs the backend can handle minus the number of jobs currently in its queue. Each project, though, implements this idea slightly differently:

- “Mu" estimates the time it takes for each backend node to process a job. It then uses this “capacity" estimate to predict the time it will take for that backend to process a new job, given the number of jobs already in its queue.

- “BLOC", on the other hand, assumes the “capacity" of each backend to be the same by virtue of being defined in a more standardized environment than “Mu". However, since BLOC operates in a more distributed environment, it passes the estimation responsibilities to the backend nodes themselves. The backend nodes then can indicate “remaining capacity" back to each load balancer they interact with.

- “SMALOPS" actually works slightly differently. It eschews the notion of “capacity" and simply focuses ensuring equalizing the load on each processor.

It should be noted though that this approach is not entirely unique to this work. The breakwater paper, [7], works with very small (nanosecond scale) requests that essentially uses the first request to a particular backend

node as a registration mechanism on that node. Thus allowing the node to provide feedback about “remaining capacity".

In our work we have proposed three systems, Mu [8], BLOC [9] and SMALOPS [10] to investigate the problems associated with each of the above discussed areas of load balancing and to propose possible solutions to such problems. The Mu system demonstrates issues with the load balancing of ingress load balancers. This is similar to traditional load balancing but the investigation focuses on edge computing systems where service capacities of the nodes vary. The BLOC system showcases problems with distributed client side load balancing identifying this area as a separate and new subspace of load balancing requiring further research. Both of these systems propose possible solutions that are based on incorporating feedback from backend nodes. We showcase that in both these areas lack of information about backend states cause significant deterioration of load balancing results. Our main contribution in this area is to show that the advent of different cloud computing environments and the microservices architecture have created a novel problem that existing load balancing algorithms and systems are ill-equipped to respond to.

In the synchronous microservices world, the Least Connection load balancing algorithm is considered to be state of the art. This algorithm works by maintaining a count of open connections to each of the backend nodes and then selecting the backend with the least amount of such open connections. A connection is considered open if a request has been sent and the client is waiting for a response. In case of distributed client side microservices, this algorithm can cause “herding" where every client side load balancer send requests to the same backend, identified as the one with the least amount of open connections, simultaneously. Herding has a large

negative impact on the overall response times. In order to protect against this issue, current implementations of the least connection algorithm also incorporate the power-of-two-random-choices [11], where two backends are selected at random before comparing the number of open connections. When compared against this version of the Least Connection algorithm, feedback based load balancing algorithms have been found to have significantly lower latencies.

We next turn our attention on asynchronous communication patterns. Traditionally, systems using asynchronous communication pattern are generally able to tolerate significant processing delays, mainly due to the design of modern message queues used to build asynchronous communication systems. However, for many modern applications, requiring asynchronous communication, this is not true anymore. While there has been significant research in this area, most of it has been confined to load with known characteristic over all time. Our contribution in this space is to demonstrate that message queuing delays in asynchronous communication systems can be significantly improved.

1.2.1 Mu: An Ingress Load Balancer

Our goal with the Mu load balancer was to improve the load balancer's response to server heterogeneity. Mu was tested against the Least Connection algorithm by running them both on ingress load balancers and thus both had access to complete information about all the backend nodes. Mu was built to directly use the service capacity and queue length values that was piggybacked from the backend nodes. The idea with Mu was to optimize each request for the fastest possible response time. Mu achieved the following:

- Optimize response time for each request thus improving tail latencies.
- Better load distribution leading to increase in system capacity.
- Ability to respond to underload conditions better.

While the greatest benefits of Mu are realized primarily in edge computing environments, even traditional cloud environments can experience server heterogeneity for a variety of reasons, thus making Mu applicable in cloud environments as well.

1.2.2 BLOC: A sidecare Load Balancer

BLOC explores the efficiency of distributed client side load balancers in synchronous applications. This pattern has emerged recently along with the microservices pattern. The initial BLOC paper, [9], demonstrates that state-of-the-art load balancing algorithms loose their desirable characteristics when adapted for distributed client-side load balancers. The main contribution of BLOC is in demonstrating a variety of methods by which backend cluster nodes can send feedback to the load balancers and the ways in which the load balancer can respond and incorporate that feedback. Incorporating feedback into the load balancer is a particularly hard problem since any information about the state of the backend cluster changes quickly in the presence of requests arriving from multiple load balancers randomly and continually altering the state of the backend cluster. The challenge that BLOC addresses in the process is to keep the information “reasonably" up-to-date without flooding the system with metadata messages and then to use the that information intelligently while being cognizant that the information becomes stale at unknown points of time.

BLOC’s focus is on correcting the loss of perspective in load balancers

that come from the information about backends becoming distributed. The major building blocks of BLOC are overload controls, namely, Active Queue Management (AQM) and Rate Limiting. AQM is used to modify the service capacity estimate which is then used to drive the rate limiter on each upstream service. With BLOC, we achieved the following:

- Built a feedback system between upstream and downstream services, allowing upstream load balancers a better understanding of downstream load conditions.
- Use the improved understanding through feedback to get upstream nodes rate limit themselves.
- React quickly to overload conditions and shift load away from such backends.

We further extended BlOC, [12], with the following:

- A detailed analysis of algorithms in centralized vs distributed client side load balancers providing a theoretical support to our claims.
- We improved upon our previous iteration of the BLOC algorithm.
- We also dove deeper into the parameter space of BLOC.

1.2.3 SMALOPS: A Load Balancer for Asynchronous Applications

Our final project is focused on load balancing load in asynchronous or event driven applications. In these applications, microservices take some action on their internal state, known as an “event", and generate a notification of each event, a “message". The microservice then sends the messages to a message bus. The message bus defines multiple “partitions"

(queues) to hold these message or event “streams" in order to allow for capacity planning. Other microservices can subscribe to receive messages from any other microservice from the message bus. A “stream" of such events, often idenfied by a “key" emebedded in the metadata of the messages, precipitate actions in subscriber microservices.

One of the main advantages of such applications is that the component microservices by being completely decoupled from each other can, in theory, scale indefinitely. With the message bus working as the glue between services, each service can process messages, generating events, at their own pace without choking up the pipeline. Such architectures, typically depend on the “eventual consistency" model in regards to the overall state of the application. This means that the typical asynchronous application has no in-built guarantees in term of processing times. For example, it quite common and acceptable for a retail banking application to take even a few days to update its state, like current balance.

Many modern applications have IoT devices as both the source and destination of messages. For example, an application might ingest data from mobile phones and vehicular sensors in an area and create safety recommendations for the different actors. The resource constrained nature of the sensors mean that the asynchronous communication pattern is the right architecture for this application. And it is obvious that such applications need a significantly tighter upper bound on processing times than eventual consistency.

Now due to this acceptance of eventual consistency, most message buses only provide for very rudimentary load balancing algorithms that are most often static in nature. Recent research, [13] [14] [15], has focused on algorithms that are more dynamic in nature and attempt to efficiently group

keys so that stream to partition assignment is efficient. However, these algorithms are load assignment algorithms rather than load balancing algorithms. They assume that one, the load in each stream is always unchanged and two, the number of streams remains constant throughout the lifetime of the application. Both of these assumptions do not hold for applications like the one described above. As such SMALOPS explores the ways in which better load balancing in asynchronous applications can lead to tighter processing times.

Chapter 2: Background

Traditionally software has always been developed and deployed as a few tiers, mostly 2 or 3, of applications, figure 2.1. These applications, known as “monoliths", themselves are developed as a single stack that incorporated the entire business logic, end-to-end. The tiers are often deployed as a cluster of individual nodes. As such, a load balancer needs to be deployed in front of these tiers so as to assign load to the “best" node.

2.1 Microservices

As applications moved to the cloud, companies realised that a more efficient deployment pattern is the microservices architecture. In this architecture, the monolith is broken down into components or **microservices**. Each microservice typically implements a single function. This enables each part of the application to be developed in a manner best suited to it. It also allows for independent deployment and scaling of only the required components leading to a much more efficient use of resources, figure 2.2.

However, moving from the monolith to microservices introduces the network inside the application as it the microservices need to communicate with each other. Generally that implies that microservices response times can be slower than the corresponding monolith. Furthermore, the centralized load balancer, between services, has been identified as both a choke point and a single point of failure in microservices architectures. As such, the load balancer deployment pattern has also changed fundamentally.

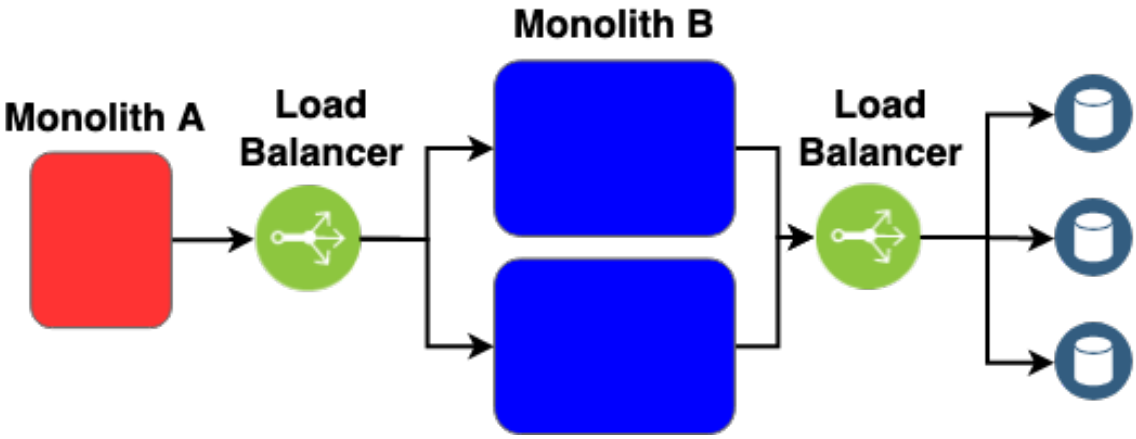


Figure 2.1: Monolith Architecture

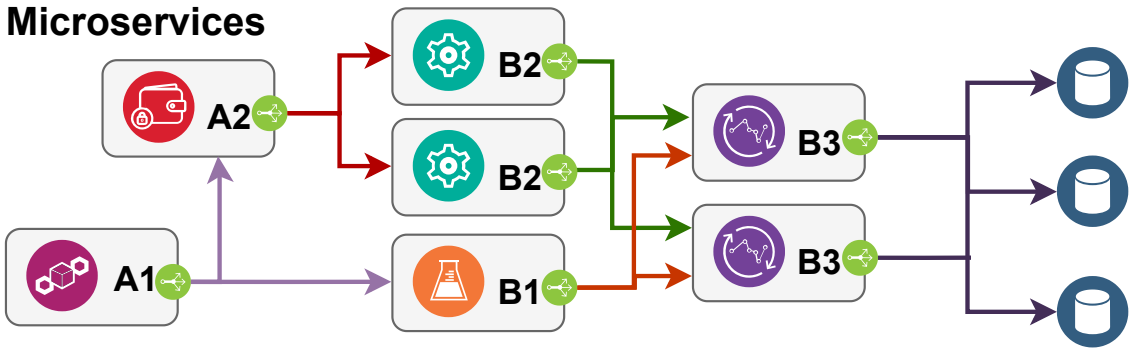


Figure 2.2: Microservices Architecture

2.1.1 Centralized Load Balancing vs Distributed Client Side Load Balancing

Load balancers in the microservices architecture are deployed in a distributed manner along with the services themselves in what is known as the “sidecar" pattern, figure 2.3. In the microservices world, applications are often deployed using the “container", [16], model. According to the container runtime company, Docker, [17], a container is a “standardized unit for development, shipment and deployment of software". For the purpose of understanding our research, a container can be thought of a network namespace, independent from that of the underlying physical or virtual machine. According to Linux [18], a network namespace provides isolation for networking resources [19]. Furthermore, multiple containers can share namespaces, like the Kubernetes “pod" [1]. This allows multiple containers to share the network, or other resources, between them. A “sidecar" is a container that sits next to an application container, sharing the same namespace in order to modify the application’s behavior in some way.

This change also essentially shifts the load balancer from “server side" to “client side". With the centralized load balancer, all traffic from clients would be directed to the load balancer. The load balancer would then redirect the traffic, using some algorithm, to a backend cluster node, figure 2.1. In the microservices architecture, this load balancer gets split up and moved alongside client services as sidecars. Now each load balancer operates only on the traffic outgoing from the service the load balancer is sitting next to, figure 2.2. Thus multiple load balancers are now choosing the backend service node independently and without sharing information between them.

These changes bring forth significant advantages:

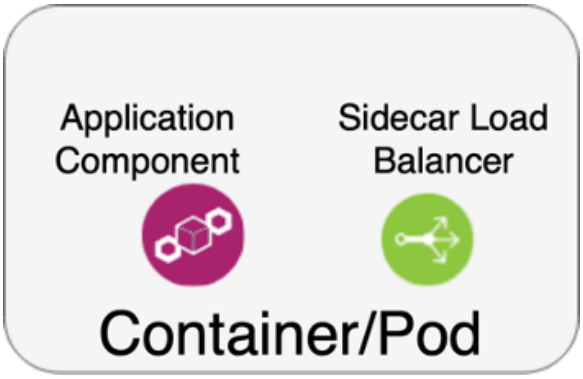


Figure 2.3: Sidecar Load Balancer

- Application architectures are no more constrained by the centralized load balancer which is both a single point of failure and a bottleneck.
- Application developers do not need to figure out how to subset the load balancers, (subsetting refers to each load balancer handling requests for only a subset of servers; this allows them to maintain long running connections to these backends optimizing response time and resource usage [20]), as each load balancer automatically maintains long term TCP connections to back ends based on information derived from the configuration.
- Application deployment is eased by packaging the application with the load balancer.

2.2 Optimal Centralized Load Balancing

Join the Shortest Queue (JSQ) is a nearly optimal algorithm for centralized load balancers. JSQ maintains a running count of all the open connections to each backend cluster node. Since it runs in a centralized location, it has perfect information and as such each request is correctly routed according to this policy, figure 2.4. This same policy has been

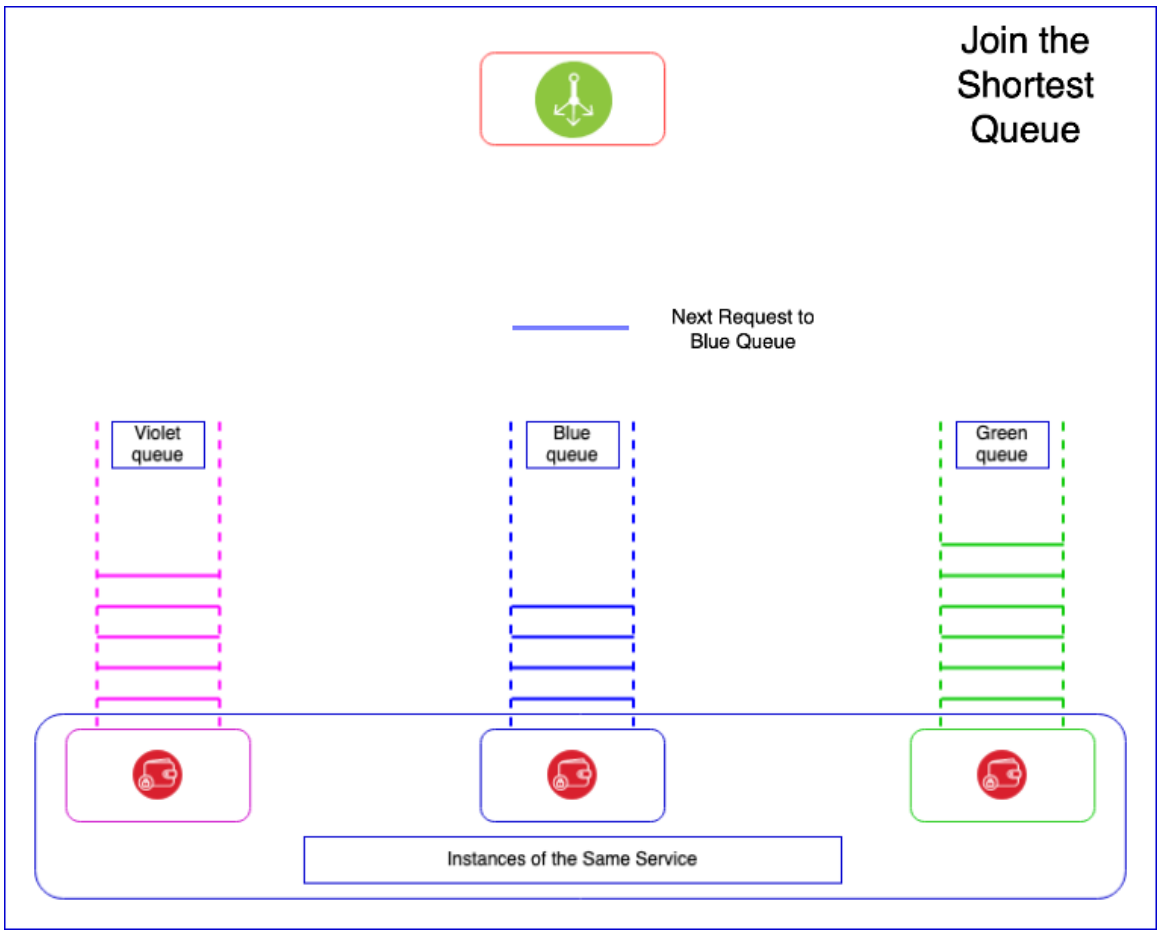


Figure 2.4: Join the Shortest Queue

adapted to the microservices world as the Least Connection (LC) algorithm, figure 2.5. Least Connection follows the same policy as JSQ. However, since LC is running at the client side, it can not see all the connections, from all the other clients, on any of the backend cluster nodes.

However, in the microservice and serverless domains, there are two deficits to the LC based approach:

- Even on, say, the ingress load balancer, which is effectively a centralized load balancer, under certain circumstances, like heterogeneous hardware or interefence due to a high packing ratio, the least loaded queue might not correspond to the fastest response time.

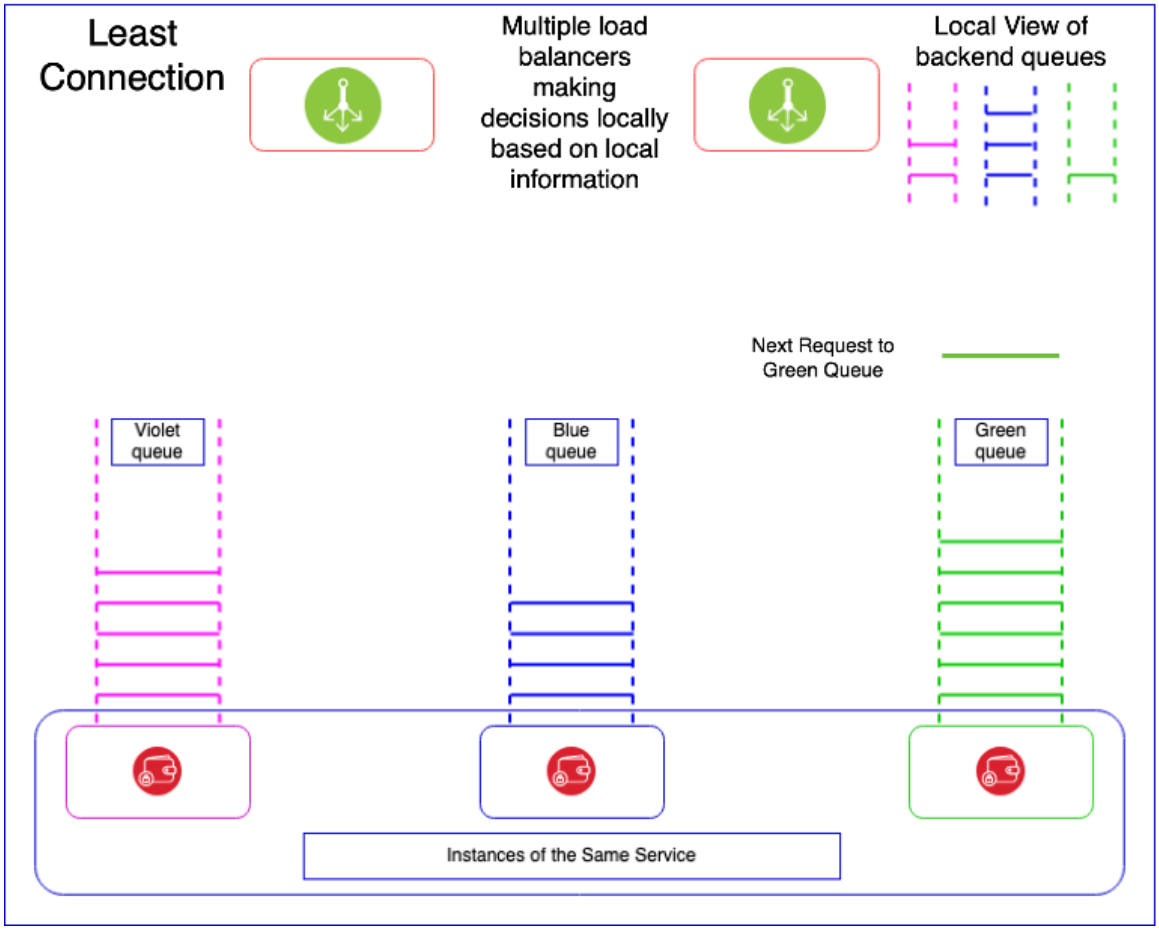


Figure 2.5: Least Connection

- Least Connection attempts to replicate JSQ using only locally available information – there are no synchronization of decisions or sharing of information. As such LC only operates on incomplete information and we show that the analysis for JSQ does not hold for LC since the same assumptions do not hold.s

While it is straightforward to realize the impact of non-homogeneous platforms on JSQ, the impact of local information on multiple load balancers, within the same service, using Least Connection is surprising.

Figure 2.5 shows that the availability of limited information can warp Least Connection leading to erroneous decisions. In Figure 2.4, we note

that there is a single centralized load balancer routing requests to the backends. In this case, this load balancer has global information about the entire backend cluster. This allows the load balancer to figure out which server is currently processing the least number of requests. However, in Figure 2.5, for Least Connection, we see that there are multiple load balancers routing requests to the same backend cluster. As such, each load balancer has only a limited view of the state of the backend cluster. Thus distributed load balancers, using algorithms that depend on global information, are much more prone to making inaccurate decisions. Figure 2.6 shows the response time distribution for the Least Connection and Random algorithms in the microservices architecuture. We can see that Least Connection performs much better than Random in terms of the response time distribution. Thus we argue that it is enough to compare our results against LC and demonstrate that despite being considered the state of the art, the Least Connection algorithm response time distribution is significantly wider than optimal.

2.2.1 Microservice Communication Patterns

Finally, microservices architectures come in two sub patterns:

- Synchronous or RESTful services based on a request-response pattern.
- Asynchronous or event driven services, following event driven architectures (EDA), based on messaging infrastructures.

2.3 Asynchronous Communication

Asynchronous applications follow a fire-and-forget model between their components. This is typically because in these applications, notification

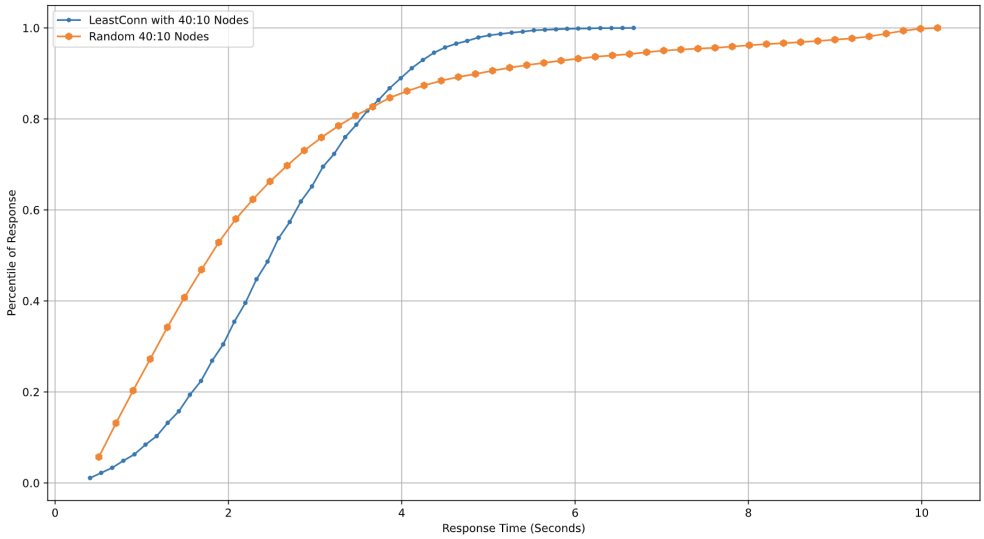


Figure 2.6: Least Connection vs Random

about certain changes get propagated through other components, if they exist. When such an application ingests some piece of external data, this event triggers an action, another event, at some component of the application. This event, in turn, sends out a notification, a message, about the component's state change caused by this event. Such events and messages can form a chain which ends when some predefined terminating condition is met or the chain propagates through a predefined path through the application. This logically connected chain of events and messages, from the initial trigger to the changed end state is known as a "transaction". Typically these applications operate on the principle of eventual consistency [21]. Generally, eventual consistency applications do not make any guarantees on transaction processing times. Let us consider one such traditional application - a retail banking application that given an external financial transaction updates the balance of the accounts involved. Anecdotal experience tells us that the account is updated any arbitrary time after the

financial transaction without a specific guarantee about the timeline.

The event driven or asynchronous services are especially lacking in infrastructure to load balance properly. Popular message buses used in EDA designs, like Kafka [22] and NSQ [23], only provide rudimentary load balancing algorithms like Round Robin and Random. The overwhelming assumption in message buses is a normally distributed workload and an equality of consumers. However, there are several cases, like [24] and [25], where services suffer from hot-key problems and need to devise custom solutions.

There are several differences between these message buses. An important one is that while some, like Kafka, support sticky sessions (a strict ordering of events for stateful messaging), others, like NSQ, distribute messages randomly to consumers. Simpler message queues like RabbitMQ [26] support slightly more advanced load balancing algorithms like "Fair Dispatch" [27], where the queue only sends one message at a time to any consumer. However, this only counters the differences in processing times in consumers and not hot keys.

In distributed caching/database systems, hot key issues are resolved by redirecting clients interested in the hot key to different replicas containing the key. In contrast, event buses shard requests to a service (topic) into partitions, with a key-value structure, and enforce a strict one-to-one mapping both between partitions and keys and between partitions and consumers (subscribers). The combination of these mappings force users to either write custom logic to assign specific partitions to specific consumers or redesign the subscriber service.

We anticipate that our work will demonstrate that the load balancing requirements of microservices are quite different from those of the mono-

lithic world. We also aim to show that the performance of load balancing algorithms ported over from centralized load balancers suffer in unexpected ways when deployed with microservices. We aspire to build a self-managing feedback system between synchronous microservices to inform load balancing decisions. Finally, we intend build smarter algorithms for EDA systems that can address hot key issues, especially in Kafka like event buses.

Chapter 3: Related Work

3.1 Load Balancing for Synchronous Services

Load Balancing is an old problem in computer science. There has been a lot of work that has been done in this area over the past few decades. The majority of this work has been focused on centralized load balancers for monolithic applications. Some of the more popular algorithms are “Random", which allocates resources based on a pseudo random number generator, “Round Robin" (RR), which routes requests in a round robin (sequentially with wrap-around) manner, “Least Loaded", which sends work to the server which is using less of its resources than others, and “Join the Shortest Queue" (JSQ), which allocates work to whoever has the least number of requests currently being processed.

Gilly et al. [28] provides more details for the different load balancing algorithms. Round Robin is a simple algorithm that works well for very simple deployments where the servers and job sizes are also homogeneous. However, it does not fare as well in the face of complex topologies and heterogeneity. “Weighted Round Robin" can address server heterogeneity through user configured weights but still cannot address heterogeneity in job sizes or complex topologies. “Least Connection", an adaption of the JSQ, popular in the microservices world, like JSQ, is a dynamic algorithm that measures the number connections to each backend, at runtime, to make its decisions. This ensures that Least Connection and JSQ take into consideration job size heterogeneity without any manual intervention.

In the centralized load balancer world, Gupta et al. [29] proves that JSQ is a nearly optimal algorithm and any other routing policy is unlikely to better

JSQ by more than 10%. This work assumes the backends are processor-sharing, that is, all incoming work is immediately dispatched. And while they consider backends serving static web pages, the processor sharing assumption holds true for several other systems. The work also shows that JSQ demonstrates near insensitivity to job size distribution. However, as we will see, these properties do not hold for distributed algorithms like Least Connection.

One more question we want to address here is that of the comparison between Random and Least Connection. After all, Random will distribute load between the backend servers irrespective of any heterogeneity concerns. This property of the Random load balancer, though, is only true asymptotically. Gupta et al. [29], Section 7, compares the first moments (average) of the queue length for different load balancing algorithms (simulated) and shows that Random consistently has a high queue average in their simulations. We also cite the results, [30], from a comparison done by the Envoy [3] team, where they show that while Random performs well asymptotically, they tend to have a high variance in how much load each of the servers receive.

However, as we move from monolithic architectures with centralized load balancers and global information to distributed microservices architectures with its sidecar load balancers, we explore if Least Connection, the JSQ derivative, is the best algorithm to use. We argue that as the centralized load balancers gets broken down into several sidecars, centralized information about the backends become distributed as well. This leads to distributed algorithms making bad decisions. In our work, we use feedback from the backend servers to correct this lack of perspective in distributed algorithms.

Kumar and Kumar, [31] points out, both overloading and underloading

are also big challenges in cloud computing. Furthermore, many cloud environments, like edge clouds, need to deal with a high variance in their capacities. Static load balancing algorithms require a deep knowledge of the workload and the infrastructure. This implies a stability of the infrastructure that is unavailable in cloud environments. As such, static, rule based, load balancing algorithms are unlikely to fare well in cloud environments. Dynamic load balancing techniques are more suitable to cloud environments. These algorithms are flexible enough to respond to the changing system dynamics and can also respond to overload situations by transferring load to relatively underloaded servers. The typical approaches with dynamic load balancers in cloud environments are to either maintain a global store of information, a central node or system, or to maintain a global state. In our work, we argue that both of these approaches limit the scalability of the environment.

There also has been some work that has been done in the cloud computing environment. Here we cite two papers, that we consider to be representative within the scope of large distributed heterogeneous clusters, HALO [32] and CHEETAH [33].

With the HALO load balancer, [32], the authors develop a system to estimate server speeds and cluster servers of similar speed together, essentially building mini homogeneous environments within the larger cluster. Once this distinction can be made, the load balancing problem is one of building a two level hierarchical load balancer using established simple algorithms. The top level load balancer distributes load to the individual clusters proportional to their mean server capacity. Within each of these mini-clusters, the second level load balancer can distribute load using a simple algorithm like Round Robin. The authors of this work correctly identify server capacity

to be an important parameter for dynamic distributed load balancers but bypass the estimation issue, which is notoriously hard, by assuming these capacities to be known.

In the CHEETAH load balancer, [33], the authors consider an additional problem of maintaining state for per connection consistency (PCC). Interestingly they do so by embedding “cookies" in the connection themselves piggybacking vital information without adding to the network load. This is an oft used technique within the larger cloud computing sphere but deciding the information to be piggybacked is much more specific to the problem being solved. CHEETAH forms a major discussion on the types of information that need to be encoded into the connections and the ways that information can be used. Another important contribution in this space comes from Netflix’s Zuul [34] which uses piggybacked information to build a distributed version of the Least Loaded algorithm.

Furthermore, our anecdotal experience suggests that both industry and the scientific community generally consider Least Connection (modified with Power of Two Random Choices [11]) and Random to be “good enough” algorithms in microservices systems. Our experiments show the failures of Least Connection, which have been, as mentioned earlier, surprising at times. Considering the random algorithm as a “magic pill” transcends beyond synchronous services into the EDA domain as evidenced by related infrastructure considering the requirement of non-normal workloads “niche use cases”. However, as we see in the Envoy team’s experiments [30] and the work of Gupta et al. [29], random, though providing for well distributed workloads asymptotically, suffers from an extremely high variance time series.

3.2 Load Balancing for Asynchronous Services

For our work with event driven architectures (EDA), we have chosen Apache Kafka to build the infrastructure and motivate the issue. This choice is justified as Kafka is the one of the more advanced event-buses available today, Sharvari and Nag [35].

Most of the work that has addressed load balancing in Apache Kafka based infrastructure has focused on load balancing the Kafka brokers between physical nodes, [36] and [37]. There also have been some work in predicting the performance of Kafka based systems but has assumed a correctly working system, [38]. There have been yet other works that look at autoscaling consumer groups in an attempt to match rate of messages produced to rate of messages consumed, [39]. However, looking at industry articles it seems that the assumption that hot keys are absent can be mistaken, [24] and [25].

In academia, there is ongoing research on this topic. However, as per our knowledge, most of that work assume:

- workload characteristics to be static, i.e. the load on individual streams do not change.
- number of streams to be constant over the lifetime of the application.

Gedik et al, [13], have used the lossy algorithm to track heavier flows and map those to partition explicitly. Other flows are mapped using the consistent hash function. This work is probably the most similar to ours. They use three lossy counters over tumbling windows to emulate a sliding window whereas we only use a single lossy counter over strictly demarcated windows.

Nasir et al, [14], has proposed PKG that uses power of two random choices to map each key to the least loaded partition selected by two different hash functions. This results in every flow, heavy and otherwise, being in a split state that requires reconciliation. Though like PKG, SMALOPS can split flows, we include a discussion of techniques to achieve generalized split state reconciliation.

Finally, Rivetti et al, [15], propose a solution that learns the distribution of the keys before using a global mapping function to achieve near-optimal load assignment. They map non-heavy keys to “buckets", where the number of buckets is user-defined and larger than the number of partitions, using a random hash function from two separate hash function families. Their solution maps the heavier flows, identified by the space saving algorithm, explicitly to specific partitions. SMALOPS on the other hand uses explicit maps for the heavier flows only while allowing consistent hashes to map others.

SMALOPS extends the existing research in the fundamental ways: firstly, it ensures that the load balancer can respond to dynamic workload changes while keeping the state overhead small. SMALOPS also proposes a generic mechanism to rebuild stream ordering allowing stateful processing on split state streams.

Chapter 4: Mu: Ingress Load Balancing in Edge Systems

Serverless computing platforms simplify development, deployment, and automated management of modular software functions. However, existing serverless platforms typically assume an over-provisioned cloud, making them a poor fit for Edge Computing environments where resources are scarce. In this paper we propose a redesigned serverless platform that comprehensively tackles the key challenges for serverless functions in a resource constrained Edge Cloud. Edge systems are small environments that are placed close to the end users. Edge environments aim to reduce latencies in time sensitive applications running on end user devices.

We built “Mu" as a set of extensions to the Knative serverless platform. Serverless computing is a good fit for this environment since the service profile can be swiftly reconfigured. However, the platform needs to be able to squeeze every bit of performance out of the environment. Due to the assumptions associated with cloud environments, the typical serverless platform falls significantly short of this goal. Our evaluations show that Mu improves 99th percentile response times by 62% through better load balancing.

4.1 Introduction

Serverlss platforms have gained popularity because they allow easy deployment of services in a highly scalable and cost-effective manner [40]. This should make serverless a perfect fit for Edge Computing, where tiny data centers are distributed throughout a geographic area, allowing users to access low latency services rather than relying on a distant, centralized

cloud. Each edge data center will be highly resource-constrained. Thus, the autoscaling "from zero" capabilities that allow serverless platforms to use no resources if there are no requests arriving, are highly desirable. Similarly, the fast instantiation of new functions ought to be a boon for Edge deployments with high user movement in and out of the area, as in a mobile edge cloud.

Unfortunately, current serverless platforms assume access to a more-or-less infinitely scalable cloud and pay little attention to resource wastage. When deployed at the edge, these characteristics lead to unacceptable performance such as high tail latency and unfair resource allocations under multi-tenancy because of the limited resources. As a result, current designs of serverless platforms are not yet a viable option for Edge environments.

In this work, we propose Mu, a resource management framework for serverless at the Edge that extends the open-source Knative platform. Mu tackles the load balancing issue in such environments, amongst other challenges. Mu's load balancer carefully assigns requests based on up-to-date statistics of backend load. This information is efficiently propagated through the system using 'piggybacking' of key measures to reduce monitoring overheads. We demonstrate that Mu's precise load balancing improves the 99th percentile response time by up to 62%, when nodes are heterogeneous or workloads are bursty—both common occurrences at the Edge.

4.2 Background

The rise of 5G has led to a network provider-centric Edge vision where cellular base stations or central offices provide a (relatively) small number of servers or racks of servers which can provide services for nearby users [41].

In this work, we consider an Edge cloud environment where limited

compute resources–likely on the scale of a single rack or less–are being made available to service requests for function execution from nearby network users. In this scenario, the Edge cloud needs to support a variety of different functions (since different users may have different needs), and it must manage its resources efficiently and fairly to simultaneously support all users.

Serverless Platforms: Cloud platforms provide compute and storage services at large scale and low cost through economies of scale and effective multiplexing. Serverless computing takes this multiplexing and scalability to the next level by allowing providers to commit just the required amount of resources to a particular application (as many instances as necessary, but only when needed) and utilize the resources for just the time needed to execute an invoked function [42].

In this work, we focus on Knative[2] and how it can be deployed in an Edge environment. In a Knative cluster, developers can write functions in a variety of languages, which are then deployed into backend worker pods. Each worker pod consists of two containers namely the 'queue proxy' and the 'function' itself. The 'queue proxy' is responsible for queuing incoming requests and forwarding them to the 'function' container for execution. Requests enter the system via an 'Ingress Gateway' that maintains metrics about active backend pods and routes requests to them. The platform is managed by an Autoscaler that dynamically adjusts the number of worker pods, a placement engine that places new pods, and a load balancer in the gateway that directs requests. In this work, we comprehensively consider all three of these aspects to enhance Knative's architecture and better adapt it to an Edge cloud environment.

There is a wide range of work on load balancing for web [28] and cloud

applications [31]. Our load balancing algorithm is inspired by the "join the shortest queue" (JSQ) approach [29], which has been shown to be nearly optimal, but only in an environment with homogeneous servers and workloads. JSQ also requires accurate information on queue length, and we show how we can efficiently acquire this through piggybacked metrics. We also draw inspiration from HALO [32], which focuses on heterogeneous environments, and we further show that serverless load balancers need to take special care when new pods are frequently added or removed.

There have been a number of measurement-driven efforts to understand the behavior of serverless platforms. Measurements on commercial serverless cloud platforms (AWS Lambda, Microsoft Azure and Google Cloud) [43, 44] while others [45, 46] show that it is important to consider throughput, scalability, memory footprint, etc. There have also been a number of measure-ment-based evaluations of open-source serverless frameworks such as Knative, OpenFaaS, OpenWhisk, Kubeless, etc. [47, 48, 49], which provide some preliminary understanding of the performance characteristics and sensitivity to configuration parameters of these platforms. We use these efforts to enhance our understanding of these open-source serverless frameworks, as we develop Mu.

4.3 System Design

Fig. 4.1 shows the architecture of Mu, which builds on the Knative, Kubernetes, and Istio tools. Mu extends the Istio Ingress Gateway to efficiently collect metrics that are "piggybacked" onto response headers by the Queue Proxy containers (§4.3.1) for timely feedback of critical information without resorting to periodic sampling.

All incoming traffic goes through the Ingress Gateway's Load Balancer,

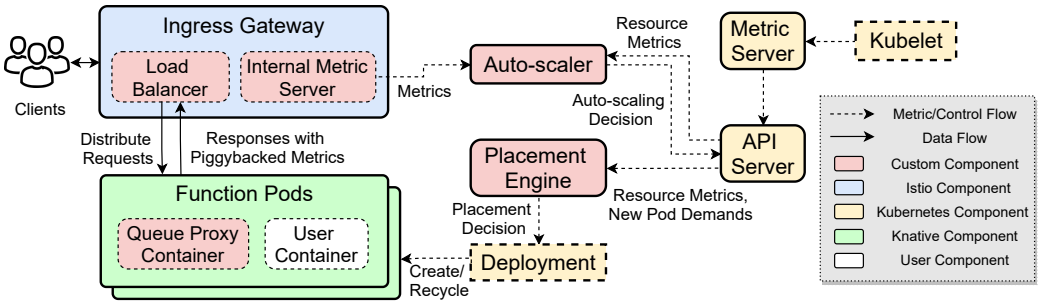


Figure 4.1: Mu Overview.

which factors in the gathered metrics to evenly load the function containers.

4.3.1 Metrics

A serverless platform relies on metrics such as the load on function containers to guide resource management. Some of these metrics, such as the load on each User Container are maintained by Knative in the Queue Proxy containers. The queue proxy is a sidecar container allocated for each user function container that buffers incoming requests. The queue proxy maintains a queue to throttle requests to the function container based on the container concurrency configuration parameter set by the administrator. To avoid high overhead when the number of function pods is large, Knative's Autoscaler periodically samples a subset of queue proxies to gather metrics, but we have found that this can lead to having an inaccurate view of important data.

To accurately monitor the status of function pods with low overhead, Mu extends the queue proxy at each pod to collect metrics about function processing and 'piggyback' those metrics in the response header to the ingress gateway to provide timely information. This allows the ingress gateway to maintain detailed per-pod statistics to guide its load balancing algorithm, while exporting aggregated information to the Autoscaler via its

Table 4.1: Summary of main notations

Notations	Definitions
T_c	time interval of capacity estimation
C_d	response count during T_c in pod
C_{dc}	count of responses whose confidence flag is 1 during T_c in pod
C_r	ongoing request count in user container
C_{cur}	current request count in cluster
C_{new}	new request count during scaling epoch
C_{pro}	processed request count during scaling epoch
$Queue$	queue size of the queue proxy
IR_{cur}	current incoming rate
IR_{pre}	predicted incoming rate
N_{cur}	current pod number
N_{des}	desired pod number to meet SLO
R_d	departure rate of pod
R_{sd}	Smoothed departure rate of pod
Cap_e	estimated pod capacity
$Ratio_c$	confidence ratio
RT_{avg}	average responding time
QT_{avg}	average queuing time
ET_{avg}	average execution time

Internal Metric Server.

The queue proxy gathers the following metrics:

Queue Length: The queue length metric shows the instantaneous size of the queue in the queue proxy, measured when the request is removed from the head of the queue to be executed. The load balancer uses this metric to determine the relative load across a group of worker pods and the Autoscaler uses aggregated queue length information to modulate the scaling decision and avoid potential Service Level Objective (SLO) misses.

Average Execution Time: The queue proxy measures the execution time of each request, which is the time between forwarding the request to the user container and receiving its response back. The average execution time ET_{avg} is the Exponentially Weighted Moving Average (EWMA) of the measured execution time. The function pod piggybacks this metric to the

ingress gateway, which passes on the average execution time across all function pods to the Autoscaler.

Departure Rate and Confidence Ratio: Ideally, the queue proxy would report the pod's maximum service capacity, but this metric can be difficult to estimate, particularly if the incoming rate is low. Instead, Mu has the queue proxy report its departure rate as well as a "confidence ratio" that indicates how fully loaded the server is. The calculation of these metrics is detailed in Algorithm 1. The queue proxy maintains a confidence flag for each request, revealing whether the user container is fully utilized (i.e., continuously has a queue of waiting requests) when processing this request. The default value of the confidence flag is 0. When a request arrives at the queue proxy, it sets the confidence flag to 1 if the queue size is larger than 0. During the processing of a particular request in the user container, the queue proxy resets the confidence flag of that request to 0 if the queue size drops to 0, implying that the user container is underloaded (departure rate is smaller than capacity).

Rather than choosing a fixed time interval for measuring estimated capacity, we adapt it based on the time scale of the request execution. The time interval for updating the estimated capacity is T_c. If the average execution time ET_{avg} increases, the time interval T_c increases accordingly, so as to collect sufficient responses in T_c for a more accurate departure rate estimate. When the average execution time ET_{avg} reduces, the time interval T_c drops, so as to update the departure rate quickly. When there are no requests in time interval T_c, then T_c will be reduced by half to react quickly for future requests, until T_c is back to its default value of 1 sec.

Every time interval T_c, the queue proxy computes the departure rate and confidence ratio. The departure rate is then smoothed using EWMA. The

confidence ratio is the ratio of the requests whose confidence flag is 1 to the total requests in the time interval T_c. If the user container is fully utilized in T_c, the confidence ratio is 1 and the actual capacity will be close to the departure rate. Both of these values are propagated to the load balancer, enabling it to make an estimate of a pod's maximum service capacity, i.e. Cap_e and share with the Autoscaler.

Algorithm 1 Capacity Estimation

On receiving a request in queue proxy:
if $Queue > 0$ **then**
    $request.confidence = 1$ ▷ $request.confidence$ is the confidence flag of this request
else
    $request.confidence = 0$

On arrival of a response from user container:
$C_d = C_d + 1$ ▷ update the response count
if $request.confidence == 1$ **then**
    $C_{dc} = C_{dc} + 1$
if $Queue == 0$ **then**
    for every request in the user container **do**
        $request.confidence = 0$
if $10 \cdot ET_{avg} > 2 \cdot T_c$ **then** ▷ increase the time interval
    $T_c = max\{10 \cdot ET_{avg}, 10\}$
if $10 \cdot ET_{avg} < T_c/2$ **then** ▷ decrease the time interval
    $T_c = min\{10 \cdot ET_{avg}, 0.1\}$

At every time interval T_c**:**
if $C_d == 0$ and $C_r == 0$ **then** ▷ the pod is idle
    $R_{sd} = 0$, $ET_{avg} = 0$
    if $T_c > 1$ **then**
        $T_c = max\{T_c/2, 1\}$ ▷ decrease the time interval
else
    $R_d = C_d/T_c$ ▷ the departure rate of this time interval
    if $R_{sd} == 0$ **then** ▷ update smoothed departure rate
        $R_{sd} = R_d$
    else
        $R_{sd} = \alpha \cdot R_{sd} + (1 - \alpha) \cdot R_d$ ▷ EWMA
    if $C_d > 0$ **then** ▷ update the confidence ratio
        $Ratio_c = C_{dc}/C_d$
    else
        $Ratio_c = 0$
$C_d = 0$, $C_{dc} = 0$ ▷ reset the counters

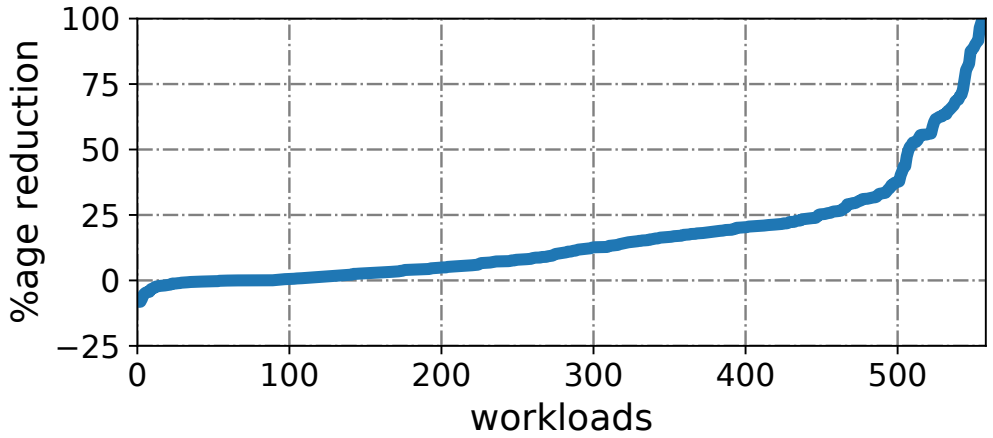


Figure 4.2: %age Reduction in absolute error for workloads with > 100K invocations

Predictor Accuracy: For validating the predictor we select all the workloads with more than 100K invocations for the first day, from the Azure Functions dataset [40]. The traces in the Azure dataset contained invocations per minute for each function. We select the top 555 workloads with at least 100K total invocations, and predict the number of invocations for the next minute. We took 50 prediction models with a combination of 5 different window sizes (10, 50, 100, 500, 1000) and 10 different learning rates (10^{-1} to 10^{-10}) for each function. Based on experiments across these functions, the value of EWMA coefficient was selected as 0.99 as it yielded better results than other values ranging from 0.5 to 0.999. For each workload, the average absolute error was calculated for the predictor and for the naive approach (which takes the current requests per minute as the prediction, like the default Knative which includes no prediction logic). The %age reduction in error for all the workloads is shown in Fig. 4.2. For 64 out of 555 workloads, the predictor performs slightly worse (-1.53% average degradation) than the naive approach, due to the random invocation pattern. Similarly, for 22 workloads, we see no improvement. For the remaining 469 workloads, the incoming rate is predicted fairly accurately, with the absolute error reduced

19.01% on average. The predictor executes ∼15.6K instructions for each prediction, taking ∼100 μ secs. For 200 workload streams, the predictor takes ∼20 ms every 2 seconds, an acceptably small 1% overhead.

4.3.2 Load Balancer

The load balancer resides in the ingress gateway and routes client requests across all pods to maximize utilization and ensure that no pod is overloaded. Load balancing requests in an Edge cloud serverless platform faces two primary challenges: resource heterogeneity and system dynamics. Unfortunately, the load balancers employed in existing serverless platforms fail to accurately account for either of these issues.

The first issue arises because an Edge cloud may be composed of a variety of hardware types, especially in “fog computing" environments where the cloud is composed of a mix of infrastructure nodes and resources pooled from mobile devices [50, 51]. Even if an Edge cloud is located in a more standardized environment such as a 5G base station, it is increasingly common for resource-constrained environments to use heterogeneity (e.g., ARM’s big.LITTLE architecture which combines high and low performance CPU cores on a single chip or accelerators like programmable NICs, GPUs, etc) to provide flexible trade-offs between performance, power utilization, and overall cost. Further, even if all hardware is identical, the dense consolidation of an Edge cloud may result in interference and resource contention which may cause some pods to execute functions more slowly than others, especially in the face of diverse workloads (IoT, ML, CDN, cellular functions, etc.). This heterogeneity can impact Knative’s “Least Connection” load balancer, which attempts to track the queue length at each backend pod by comparing the number of requests sent versus responses received. When deciding

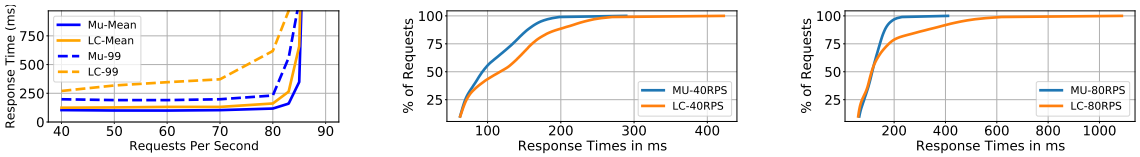
which pod to select for a new request, it only considers the queue length estimate, which we show can lead to poor decisions when backends have varying service capacities. Further, if the serverless platform runs multiple load balancer gateways, this queue length estimate may be inaccurate as it ignores queueing caused by other gateways.

The dynamic nature of Mu's autoscaling capabilities further complicates load balancing. The load balancer must be aware of newly added pods, and it should direct the appropriate amount of load to them – avoiding "herding" problems where too much load is shifted to a newly started pod, but also avoiding underloading it. In effect, a newly started pod represents a different type of heterogeneity since it will begin with an empty queue of requests, while other nodes may already have nearly full queues if scaling occurred due to approaching overload. The Knative Least Connection load balancer employs a power of two random choices algorithm [11] which means that it randomly selects two backends and then picks whichever has the smaller number of active connections. While this provides greater scalability as the cluster size increases, it comes at the expense of lower accuracy, which may not be the appropriate trade-off for a resource-constrained Edge cloud. As a result, a new pod in Knative has at most a $2/N$ chance of being selected in a cluster of N servers. Our evaluation shows that this limits Knative's ability to quickly shift load to new pods, leaving the system in an overloaded state despite idle resources.

4.3.3 Load Balancer Algorithm

A smart load balancer should recognize both differences in service capacity and pod queue length to appropriately route requests across new and existing pods. In Mu, we implement a new load balancer that leverages

the metrics gathered by function pods to make better decisions based on up-to-date information.



(a) 99%ile latency & mean latency

(b) Response time CDF with 40 RPS

(c) Response time CDF with 80 RPS

Figure 4.3: Mu's load balancer vs. Least Connection load balancer: Mu reduces tail latency across all load levels.

Estimating Pod Metrics: Most prior work on load balancing assumes access to service rate information for each backend; further, such rates are assumed to be static. In a serverless environment, the large number of different functions makes it impractical to assume all functions have been previously profiled to determine service rates, particularly for an edge cloud with hardware heterogeneity. A backend's capacity may also change over time, particularly in a densely packed Edge environment where resource contention can occur. Thus Mu must be able to accurately and dynamically determine both the service capacity of each pod, and its current load level. As described previously, Mu's Queue Proxies piggyback key metrics as part of each response header, providing the load balancer up-to-date information about each pod. However, further processing is required in order to produce accurate estimates of pod capacity and load.

When a function is deployed for the very first time, Mu has no information about its execution cost. However, once requests start to be processed, it quickly builds a model of each pod's service capacity as follows. On each response from backend pod i, we compare the piggybacked confidence, $pigRatio_i$, and departure rate, $pigR_i$, against previously saved values for the

pod, $savedRatio_i$ and $savedR_i$. If $pigRatio_i \geq savedRatio_i$ or $pigR_i \geq savedR_i$, then we update pod i's capacity estimate $Cap_i = pigR/pigRatio$ and update the saved confidence ratio and departure rate values to be equal to the piggybacked values. If the prior conditions are not met, then the saved values are not updated. A newly started pod with no data uses the maximum values seen by another pod of the same function type as a default.

The intuition behind this algorithm is that if the Confidence Ratio reported by the queue proxy is low, that indicates that the backend has had a low or empty queue, and thus it is safe to aggressively predict that the real service capacity is much higher than the departure rate. When the Confidence becomes 1, it means that the backend is consistently seeing a queue, which means its departure rate will be close to the actual maximum service capacity of the pod (otherwise the queue would have drained). Tracking a saved Confidence Ratio and Departure Rate ensures that the Load Balancer does not lose information over time, assuming that the service capacity drops simply because the arrival rate falls.

To track the load on each pod, the load balancer can use the piggybacked queue length values. Using the piggybacked value instead of a local counter at the load balancer ensures that the metrics are accurate even if there are multiple load balancers in the cluster. These metrics are aggregated and exposed to the Autoscaler, which uses them to determine when to scale up as described in the prior section. Further, we use the service capacity information to guide downscaling, causing the system to prefer to shut down slower pods when they are no longer needed. This not only helps ensure the downscaling won't cause unexpected overload, but also naturally makes the algorithm pick a pod with fewer requests in its queue, allowing its resources to be freed sooner.

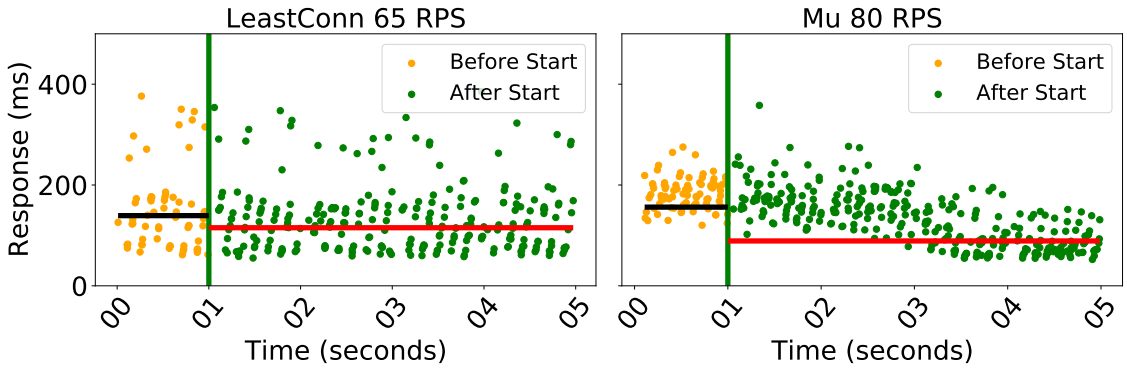


Figure 4.4: Mu takes advantage of a newly added pod more quickly: shifting load, improving both mean (horizontal lines) and variance in response time more

Selecting Pods: Using the above information about pod capacity and queue length, the Mu Load Balancer calculates the estimated response time, R_i, that a new request would see on each pod i in the cluster:

$$R_i = \frac{Q_i + 1}{Cap_i} \tag{4.1}$$

where Cap_i is the estimated service capacity and Q_i is the estimated queue length–we add one to account for the cost of processing the new request. The load balancer then selects the pod with the minimum R_i. This algorithm attempts to minimize the response times seen by all requests, and will naturally forward more requests to pods with higher service capacities or lower queue lengths (such as a newly started pod). It should be noted that since some functions may support concurrent processing of requests, this may be an inaccurate estimate of the request's actual response time; nevertheless, it represents both the service capacity and load on a function well, so we find it gives a good signal about what pod will be the best choice for the request.

Load Balancer Performance: To demonstrate the importance of using both queue length and service capacity to guide decision making, we run an experiment with two "fast" and two "slow" pods. To get a sense of what a reasonable level of heterogeneity is, we compared the service time of a CPU bound prime number calculating function on a high-performance AMD EPYC Rome 64 core Processor (3 GHz) and an Intel Xeon CPU X5650 running in a low power mode at 1.6GHz. The AMD system is roughly two times faster than the Intel one depending on the prime function parameter. Thus, in our experiments we set faster pods to be twice as fast as the slower ones; we use a function with a service time of about 100ms on a fast pod. We measure the response time when adjusting the client send rate. Fig. 4.3a shows how the mean and 99%ile latency change with a rising workload. We observe that Mu can support a higher request rate with lower response times, and that it particularly improves tail latency due to better accounting for the relative speeds of the different pods: at 80RPS, the 99%ile decreases from 618ms to 230ms, leading to a much narrower response time distribution as shown in Fig. 4.3b and 4.3c. To understand why Mu provides such a benefit, we examine the queue lengths of different pod types in each algorithm. Despite attempting to pick servers that have fewer active connections, Least Connection still tends to cause a higher queue build up on slow pods compared to fast pods. In contrast, Mu correctly recognizes it can safely queue more load on the faster pods, while still maintaining a low overall execution time.

Load Balancer Agility: We next demonstrate Mu's ability to more quickly adapt by leveraging its detailed pod information. We consider a scenario where four pods (two fast, two slow) are on the verge of overload. Fig. 4.4

shows the response time for requests immediately before and after a new fast pod begins (marked by the vertical line and color change). While the pod addition does help reduce the mean response time of Least Connection, it still shows a wide spread of response times due to the poor balancing of the load. In contrast, Mu provides a much tighter distribution of response times, and shows a clear downward trend as new requests are directed away from the heavily loaded pods and towards the new pod. Note that in order to cause Mu to hit the same overload point as Least Connection in this experiment we need to send it a higher workload (80RPS vs 65RPS), so Mu is not only handling a larger volume of requests, but it is able to do so while significantly reducing both tail and mean latency (horizontal lines).

4.4 Evaluation

Table 4.2: Experiment configuration

Parameter/Specification	Values	
Invocation Range	W-1	41-230 rps
	W-2	69-182 rps
Average invocations	W-1	154 rps
	W-2	146 rps
Container Concurrency	4	
Grace Flag (Mu only)	16	
Execution time	500ms	
Maximum pod capacity	48	
CPU and Mem. per pod	7 cores, 30GB	
Target	RPS	8
	CC	40
SLO	5 seconds	

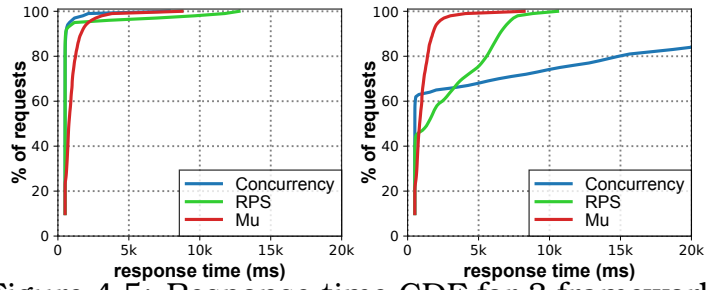
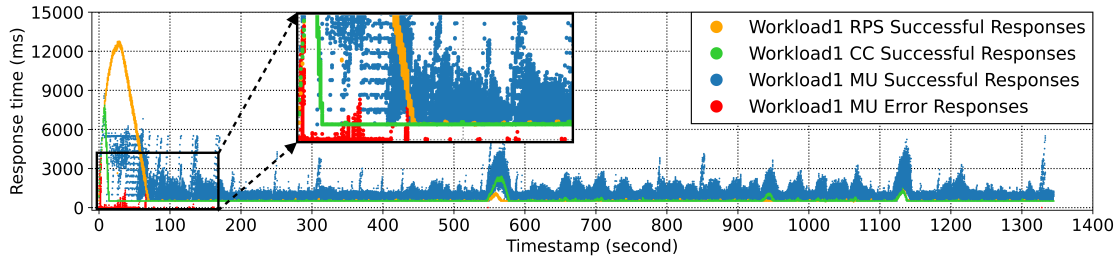


Figure 4.5: Response time CDF for 3 frameworks for Workload 1 (left); Workload 2 (right; only partial CDF for Concurrency)

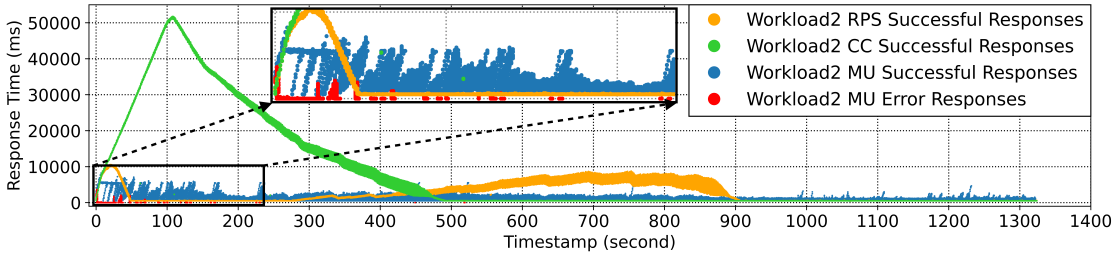
Table 4.3: Comparing Mu with the standard Knative build

		Average response time (ms)	99% response time (ms)	# 503 errors /total requests	Requests served within SLO	Requested Pods		Active Pods	
						Max	Avg.	Max	Avg.
Mu	Workload-1	952	3805	6779 / 221026	213437 (96.5%)	33	20.5	24	20.0
	Workload-2	1020	4073	5211 / 209905	203622 (97.0%)	26	19.4	24	18.9
RPS	Workload-1	880	11757	0 / 221026	213089 (96.4%)	38	29.3	26	25.1
	Workload-2	2605	8808	0 / 209905	158511 (75.5%)	32	27.9	22	20.9
Concurrency	Workload-1	588	2141	0 / 221026	220144 (99.6%)	141	41.4	40	24.5
	Workload-2	7765	49526	0 / 209905	142774 (68.0%)	136	62.3	24	21.2

We now integrate all the components of Mu, and evaluate it for a few large scale workloads. We compare Mu with the Knative default approaches.



(a) Workload 1



(b) Workload 2

Figure 4.6: Time series of Response Time for Mu, RPS, and Concurrency (Top: Workload 1; Bottom: Workload 2)

Implementation Details and Testbed Setup: Mu's implementation extends multiple components in the Knative ecosystem, including the Knative Queue-Proxy, Istio Gateway, Knative Autoscaler, and Kubernetes Scheduler (placement engine). We base our code on Kubernetes v1.17.0, Istio's Envoy Proxy v1.16.0, and Knative v0.13.0. Our extensions comprise ∼1,000 lines of code added for the Autoscaler, ∼500 lines for the load balancer and metrics server, ∼200 lines for the queue-proxy, and ∼800 lines for the placement engine.

We evaluate the serverless platforms on the Cloudlab testbed [52] consisting of one master and ten worker nodes, each of them equipped with Two Intel E5-2660 v3 10-core CPUs at 2.60 GHz (40 hyperthreads per host) and 160 GB ECC memory running Ubuntu 18.04.1 LTS. We do not add any extra pod heterogeneity in this experiment other than the natural fluctuations found on CloudLab.

4.4.1 Overall Mu Performance

To comprehensively evaluate Mu, we use the workloads received by functions in the Azure dataset [40]. We select 2 workloads with variable invocation patterns from the top 10 workloads sorted by maximum number of invocations for the first day in the dataset. We scale down these workloads by dividing the number of invocations by 100 for the experiment, treating each minute of the original trace as one second to add dynamics. The scaled down workload and the configuration of the serverless environment are in Table. 4.2. With the combined Autoscaler, Load Balancer, and Placement Engine, Mu achieves better overall performance for requests to serverless functions, even if the system is subject to a significantly heavy load, and more fairly allocates the limited edge cloud resources among the competing functions.

Latency and Fairness: The CDF of the response times for each workload and approach is shown in Fig. 4.5. Mu has good control over the response times and limits the tail latency that exceeds the specified SLO of 5 seconds for both workloads. For Workload 2, Mu provides a substantially tighter response time distribution than RPS or Concurrency. As shown in Table 4.3, the 99% response time for the two workloads are both below the 5 second SLO for Mu. Examining the response time distribution (Fig. 4.5), and the average and 99%iles (Table 4.3) and the time series of the response times (Fig. 4.6a, 4.6b), we see that Mu maintains fairness between the workloads for the entire length of the experiment.

In contrast, the standard Knative approaches result in much larger response time tails, and both unfairly treat one of the workloads. For Workload 1, both RPS and Concurrency (CC) achieve a lower average response time (except RPS has a relatively large number of requests experiencing high

delays at the start of the workload, resulting in its 99%ile being higher). However, for Workload 2, both RPS and CC behave quite poorly at different periods of the workload execution, as seen from the time series (Fig. 4.6b), with 25-32% of requests violating the SLO. Workload 2 sees an unacceptably large 99% latency with CC as seen in Table 4.3. Since Mu is conservative in its pod allocation for both Workload 1 and 2, it sees a slightly higher average response time for Workload 1 than RPS and CC, but better for Workload 2 than RPS and CC. The 99%ile for Mu is clearly better than the two alternatives.

SLO Performance: Overall, Mu provides a significant increase in the total number of requests served within the SLO (96.8%) compared to the RPS scaling policy (86.2%) and Concurrency scaling policy (84.2%), as shown in Table 4.3. Mu uses SLO-aware admission control and returns 503 errors for requests which it will not be able to serve within the SLO based on current queue lengths. This avoids the build up of a large queue with the arrival of a burst of requests. RPS and concurrency do not factor SLO into account, so when bursts occur, requests are buffered in the activator, and the queueing results in a large number of SLO misses. Throughout the experiment, Mu has relatively uniform response times, increasing only during bursts, when the system is under-provisioned (e.g., first 200 seconds of the experiment when we have to scale up from zero to a large number (~ 20) of pods). On the other hand, Concurrency and RPS see persistent queuing for long periods (> 400 seconds) and the response time grows substantially more than the desired target SLO of 5 seconds. There is also significant unfairness for Workload 1 vs. Workload 2 as seen in Fig. 4.6a, 4.6b.

As shown in Fig. 4.6, Mu returns 503 errors (indicated by red dots). Our view is that by having these failures (and potentially having those requests

be retransmitted) impacts a relatively small number (<5%) of requests, which is better than building up a large queue resulting in very long latencies for a large number of requests (25-30%, as seen for RPS and Concurrency) and likely to more seriously impact user Quality of Experience (QoE). These 503 errors are well correlated with the occurrence of bursts when resources are not yet provisioned by Kubernetes. This is mitigated somewhat by the predictor and proactive autoscaling. In fact, most of the 503 errors occur when the burst arrives at the beginning when the predictor has not yet learned the characteristics of the workload. Additionally, even though Mu's autoscaler requests allocation of a larger number of pods, Kubernetes can take a large amount of time to provision these pods, starting from an initial zero-scale system (as seen in the difference between pods being requested and active in the first 200 seconds for Mu.

Chapter 5: BLOC: Balancing Load with Overload Control in Microservices Architectures

The microservices architecture has become ubiquitous in the cloud environment. It simplifies application development by breaking monolithic applications into manageable micro services that can be developed and deployed independently of the whole. However, the move from a monolithic or simple multi-tier architecture to a distributed microservice "service mesh" leads to new challenges due to the more complex application topology.

A particular problem when automatically managing the performance of microservices is that since each service component scales up and down independently, it can easily create load imbalance problems on shared back-end services accessed by multiple components. Traditional load balancing algorithms were designed for centralized load balancers sitting between a group of clients and a server farm. These algorithms, however, do not port over well to a distributed microservice architecture where load balancers are deployed client-side. In this paper we propose a self managing load balancing system, BLOC, which provides consistent response times to users without using a centralized metadata store or explicit messaging between nodes.

We show that different service layers scaling independently can create unacceptably wide response time distributions and long tails, hurting client experience. This is because popular microservice load balancing algorithms, like Least Connection, only use a single component's view of the backend load to guide decisions. This limited perspective leads to an unevenly balanced system and the potential for incast problems where a large number of

frontend components can easily overload a shared backend. BLOC uses overload control approaches like rate limiting, active queue management and backpressure to provide feedback to the load balancers. The load balancers react to this feedback with techniques like backoff and retries. We show that this performs significantly better in solving the incast problem in microservice architectures.

Evaluating this framework, we found that BLOC improves the response time distribution range, between the 10^{th} and 90^{th} percentiles, by 2 to 4 times and the tail, 99^{th} percentile, latency by two times.

5.1 Introduction

Microservices have become increasingly popular due to a variety of advantages they provide like ease of deployment, continuous integration, independent development and others. However, it also brings the network inside the architecture as the monolith is broken into multiple independently deployed pieces. In most current scenarios, microservices are deployed as containers in clusters managed by an orchestrator like Kubernetes [**?**]. A pattern related to container clusters that has also become popular is a move away from single-node centralized load balancers. Instead, client-side load balancers are deployed alongside each upstream service container as a "sidecar", as illustrated in Figure 5.1. An advantage of using this pattern is that the load balancer is removed as a single point of failure or performance bottleneck.

Many microservice deployments are managed by service meshes like Istio [4]. Istio uses Envoyproxy [3] for load balancing, which uses a power of two random choices (P2C) [11] version of the Least Connection algorithm. Least Connection is based on Join the Shortest Queue (JSQ), which has been

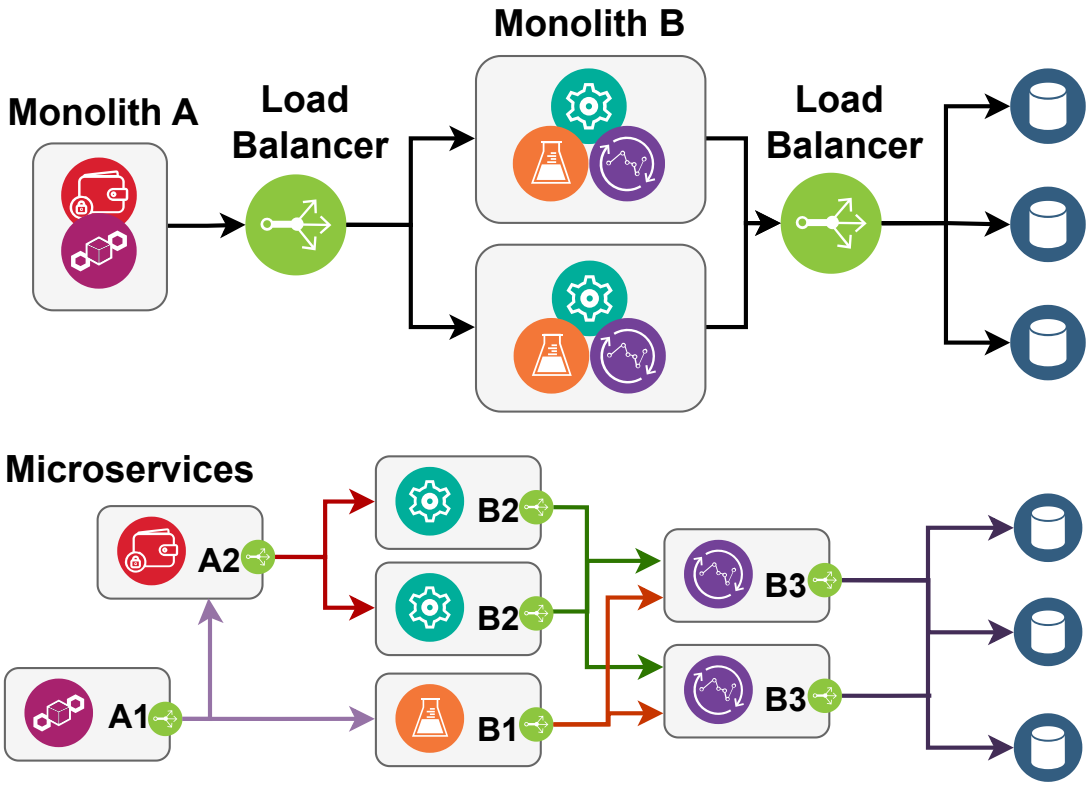


Figure 5.1: A multi-tier application built from Monolithic services (top) can be decomposed into microservice components (bottom), potentially improving development practices, but complicating the application topology. Sidecar load balancers (green circles) are deployed adjacent to each microservice component to route requests to downstream nodes.

proven to closely approximate the best possible load balancing algorithm by greedily selecting the backend which currently has the smallest queue of work [29]. However, JSQ's optimality depends on it being deployed in a centralized environment where all requests flow through a single load balancer, giving it a global view of backends' queues. In Least Connection, a sidecar based load balancer lacks this perspective, so it selects the backend to which it currently has the smallest number of open connections as the target for a request. In this case, the selected backend may not necessarily have the smallest queue since the policy only accounts for requests coming from the node attached to the sidecar.

In a microservice deployment, it is common for backend services to be shared by multiple upstream components, each of which may be replicated. In such a scenario, each upstream node sends only a small fraction of the total requests that each downstream node receives. This leads to a divergence between the actual load of the downstream nodes and the estimate of that load the upstream nodes have. As a result, the performance of the application can deteriorate quickly due to bad decisions made by such "local" algorithms.

In this work, we present BLOC[1], which makes the downstream nodes a part of the decision making without requiring expensive coordination. We compute the capacity of each service in terms of the number of requests one node of that service can handle while keeping end-to-end response times within the SLO (Service Level Objective). We then send each upstream node that we are currently interacting with "confidence chips" that will enable them to send requests in the future. The scheme also maintains

[1]The original article, published at ACSOS 2022 [9], has been extended by a submission to the ACM Transactions on Autonomous and Adaptive Systems (TAAS) journal, [12] which is still under review. Source code available at [53]

some capacity for upstream nodes that the downstream is not interacting with at the moment but might still send a request. Downstream nodes use active queue management to reject requests that push the number of active requests over its capacity. In response to such rejections, the upstream nodes backoff for a predefined amount of time. Upstream nodes also use power of two random choices to reduce the likelihood of immediately selecting a downstream node that just rejected a request.

We make the following contributions in this paper:

- The design of BLOC, a distributed load balancing system that uses admission control, backpressure, and piggybacked server information to effectively balance load, particularly in overload scenarios.
- BLOC's architecture is fully distributed, requiring no coordination between replicas or centralized load balancers that can be a bottleneck or single point of failure.
- BLOC's implementation uses ingress and egress proxies deployed as container sidecars, allowing its load balancing and admission control algorithms to be seamlessly integrated with existing applications without code modifications.

We implement BLOC as a Go based proxy and deploy it in a Kubernetes cluster. Our evaluations show that BLOC can improve the response time distribution from 10^{th} to 90^{th} percentile by 2 - 4 times and the 99^{th} percentile tail latency by two times.

5.2 Background

Microservices and Sidecars: Micorservices are a popular architecture pattern that breaks a monolithic application into multiple smaller services.

It allows for shorter development time, faster deployment cycles, usage of different technology stacks for different parts of the application, swapping entire parts of an application, and continuous integration without any impact on the operation of the overall system.

Microservices are typically deployed in containers with an orchestrator framework like Kubernetes. Just as microservices are the smaller parts of a decomposed monolithic service, container orchestration frameworks take this a step further and allow each microservice to be decomposed into several containers, e.g., one container might hold the application business logic, while others run monitoring components and load balancing proxies. These auxiliary containers are typically referred to as "sidecars", due to the way they are deployed adjacent to an application container and often process their incoming or outgoing requests. A group of application-specific and auxiliary containers that together form a logical service are grouped into a single namespace known as a "pod" by Kubernetes.

Since each pod can be replicated multiple times to scale up and down a microservice component, it is necessary to have load balancers that help route requests to the appropriate downstream node. The ability to easily glue together functional components has allowed for the move away from single-node centralized load balancers to distributed sidecar load balancers deployed as part of each pod. Each proxy sidecar thus handles load balancing all outgoing requests from the microservice component they are attached to across muliple downstream replicas. This distributes the load balancing work, giving a more scalable system, but it also means that each load balancer lacks the global view of a centralized approach.

Istio, Envoy, and Least Connection: As an example of industry deployments of microservices networking we cite Istio [4] and Envoy proxy [3]. Istio

is a popular example of what is known as a service mesh. A service mesh is a control plane that works with Kubernetes to deploy networking infrastructure throughout Kubernetes clusters. Typically, this is done through deploying a mesh of sidecar proxies, like Envoyproxy with Istio, that provides the networking data plane and implements components like load balancing, service discovery, backpressure and much more.

Envoy acts as both an ingress and egress proxy. The egress proxy implements load balancing and routing for any requests generated by the attached microservice component to downstream services. The ingress proxy intercepts all incoming requests from upstream services, and is used for monitoring, security management, etc. In our work we leverage this architecture so that downstream ingress proxies can provide feedback to upstream egress proxies, improving load balancing decisions. Since our changes are only within the proxy, no modifications need to be made to the microservice applications themselves.

Sidecar proxies typically, use traditional load balancing algorithms like the Power of 2 choices (P2C) version of Least Connection. In this algorithm, the proxy randomly considers two possible downstream nodes and selects the one that has the least number of outstanding requests from the current node. Unfortunately, the node being picked might actually be more heavily loaded than others since the proxy is unaware of requests forwarded by the proxies in other pods.

LeastConnection and similar algorithms that rely only on a sidecar load-balancer's local state can perform well when the number of service replicas is relatively low and workloads are evenly distributed across the upstream nodes. Yet in a microservice deployment, this may not be the case. For example, the applications provided by Deathstarbench, an open source

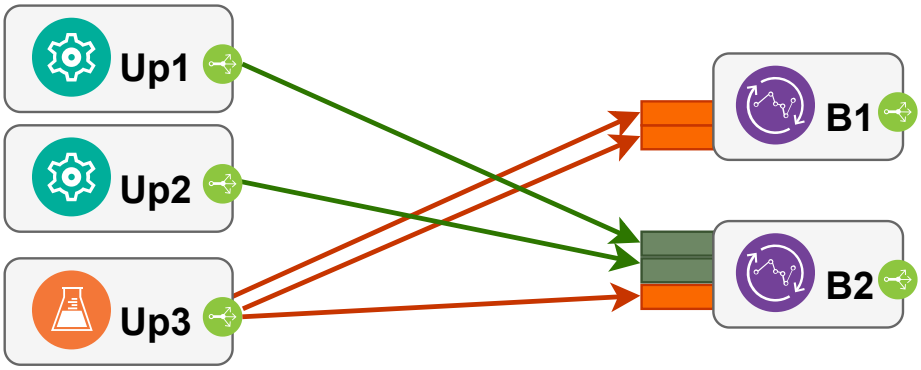


Figure 5.2: LeastConn only has information about outgoing requests leaving a sidecar, not the actual queue lengths at the backend nodes.

collection of microservices, each contain between 21 and 41 unique microservices, each of which may be replicated multiple times [54]. Netflix, an early adopter of microservice architectures, was reported to have over seven hundred different microservices deployed over tens of thousands of virtual machines as of 2015 [55]. These massive arrays of microservices form complex topologies with shared services being accessed by many different types of upstream components. Further, there might be geographical constraints in large clusters leading to different client pods sending requests at different rates to the backends. In such a dynamic environment, workloads can easily become skewed, leading to an inaccurate local view of downstream node load levels.

Least Connection Limitations: To see the intuition for why Least Connection can perform poorly, consider the situation in Fig 5.2, Upstream Node Up3 has two outstanding requests to Backend B1 and one to Backend B2. The other two upstream nodes each have one outstanding request to B2. Thus the total number of outstanding requests at B1 is two while that on B2 is three. If now a fourth request arrives at the load balancer of Up3, then the Least Connection algorithm on Up3's LB, will send the request to B2 instead of B1, which would have been the optimal solution. If a centralized

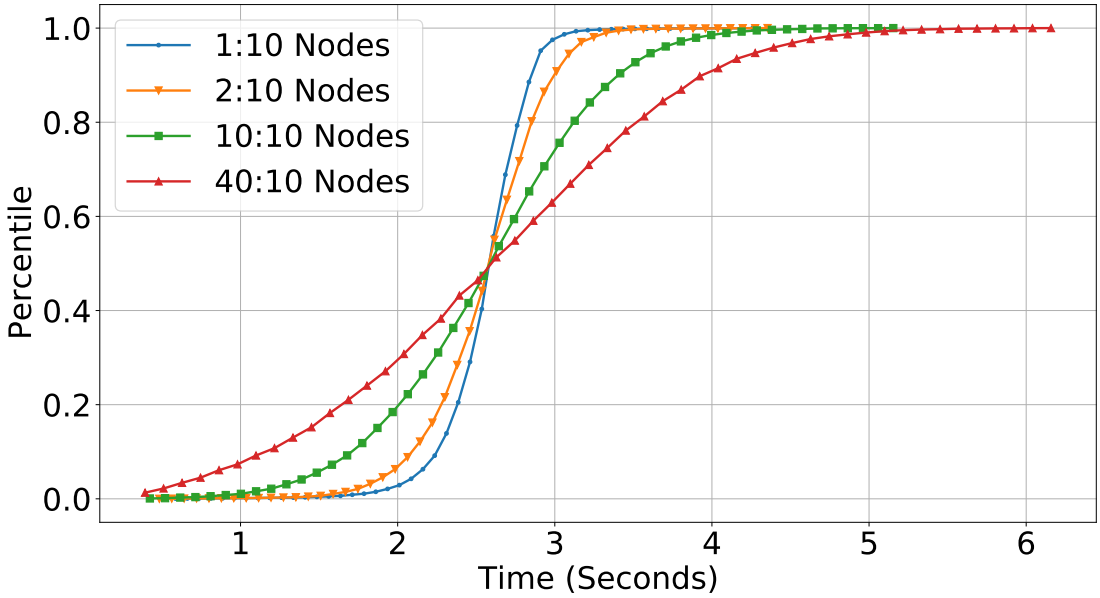
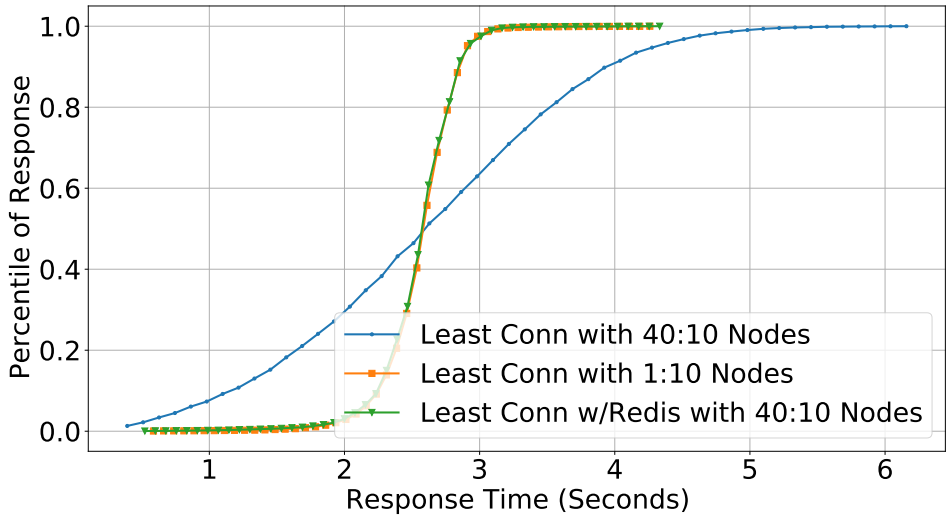


Figure 5.3: Changing from 1 to 40 frontends causes a significant increase in the range of response times and tail latencies.

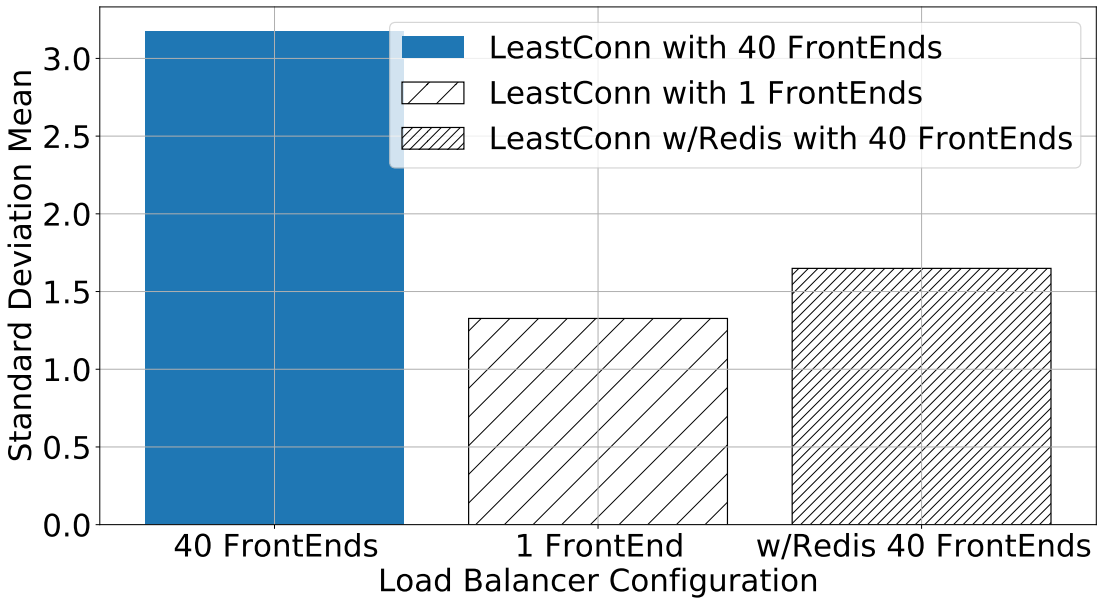
load balancer was being used, this issue would not arise.

Generally, with a small number of servers and clients where the clients are all receiving roughly the same number of requests, this is not an issue since the relative equivalence in the number of clients and servers mean that these discrepancies will be small so each sidecar's local view is a similar match to the global one. However, that may no longer be true when there is a large number of upstream nodes with different request characteristics to the downstream nodes.

To empirically measure this phenomenon, we deploy a pair of microservices and adjust the number of upstream nodes accessing a set of ten downstream replicas. In order to focus on the impact of load balancing across the downstream nodes, we configure the upstream service to be very lightweight, and make the downstream service expensive (consuming a 250 msec service time). We deploy a custom sidecar load balancer similar to Envoy running the Least Connection algorithm and use an HTTP load



(a) Redis Load Balancer Performance



(b) Least Connection Mean of Standard Deviation

Figure 5.4: Using Redis to provide a global view of backend state makes the response time distribution nearly identical to having a single centralized load balancer (green and orange lines overlap), and similarly reduces the variation in load across backends.

generator to stress test the system.

Figure 5.3 shows that as the number of upstream nodes increases, the response time distribution widens significantly. The case with only a single frontend (1:10) is representative of a traditional monolithic service deployment where a centralized load balancer sits between tiers of the application, while the other lines can represent distributed microservices. Interestingly, the median response time remains similar (about 2.5 seconds), but changing from 1 frontend to 40 frontends causes significant changes at the head and tail of the distribution. This result is somewhat unintuitive: one would typically expect adding more frontends to *improve* performance, not hurt it!

The explanation for these results is that the sidecar load balancers are making conflicting decisions due to lack of coordination – some requests are sent to very lightly loaded servers which are able to respond very quickly, while others queue up at overloaded servers, causing long delays. The impact can be quite large: the range between the 10^{th} and 90^{th} percentile increases by almost 5 times and tail latency degrades by more than 40% when going from 1 to 40 upstream nodes.

Diagnosing Least Connection: We determined that there are two factors that cause the response times of the system to degrade by such a large amount:

1. the metadata that each sidecar load balancer holds locally becomes stale much faster as the number of upstream nodes increase making the load balancing decisions progressively worse, and

2. a larger number of upstream nodes accessing backends with heavy requests can easily overload them, similar to the TCP incast problem

[56].

In order to prove the first point, we deployed a Redis service in our Kubernetes cluster to provide a global view of backend load. The Redis cache stored the active queue length of each downstream node. Before routing a request, a sidecar load balancer would fetch queue length data for all nodes from the caching service. The load balancer then updated the cache to increment the queue length for the selected downstream node. When receiving a response, the sidecar load balancer subtracted 1 for the downstream node that sent the response. This made the Redis service a global source of true backend queue lengths for all load balancers. With this simple addition of a caching service we found that the overall performance of a 40 upstream nodes is indistinguishable from that of using a single upstream node (Fig. 5.4a).

To show the level of imbalance between downstream nodes when the number of upstream nodes increases, we measured the total number of requests sent to each downstream node at 2 second intervals. With this data, we plotted (Fig 5.4b) to show the standard deviation across the ten backends during each interval, averaged over the entire experiment. We can see that the mean of the standard deviation of new requests received every sampling interval for the 40 frontend case is much higher than the 1 frontend node case.

We conclude that response time degradation is caused by the burstiness in the request profile which in turn is caused by inaccurate local data. This is exacerbated when backend requests are expensive (which is often the case), since even if all frontends send just one request to the same backend, they will cause it to be completely overloaded. Thus we must combine load balancing in the upstream nodes and overload control solutions in the

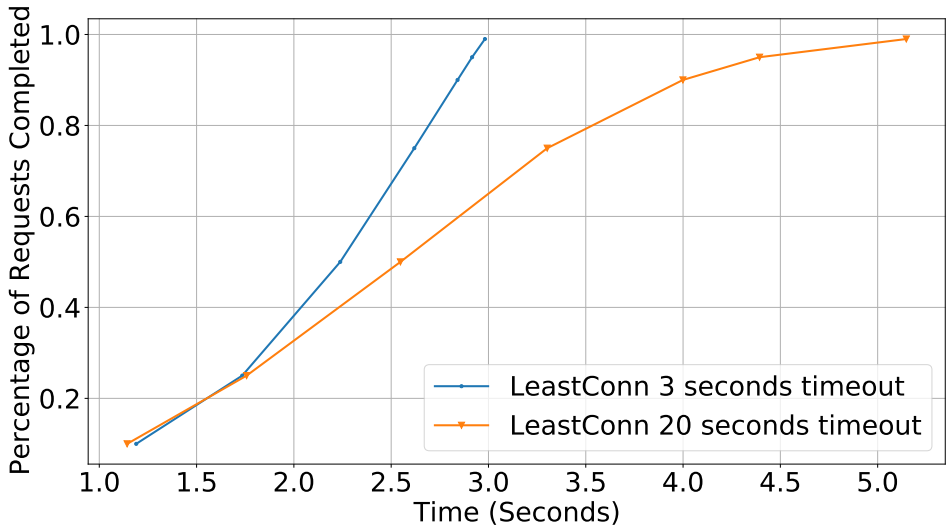


Figure 5.5: Using AQM to drop requests early helps the tail, but not the head of the distribution, suggesting backends are still not evenly utilized.

downstream nodes to solve this problem.

Overload Control Approaches: Two general techniques to implement overload control are:

- Rate limiting, where upstream nodes purposefully slow their requests to prevent backends from getting overloaded; often this is guided by backpressure algorithms where the server lets the client know that the server is overloaded.

- Admission control, where downstream nodes preemptively drop requests to avoid excessive queueing; Active Queue Management (AQM) algorithms try to intelligently drop requests or network packets to do this in a graceful way.

Unfortunately, naively applying backpressure has been shown to lead to system-wide hotspots and trick the system into upsizing or penalizing the wrong service [54]. Admission control, on the other hand, is extremely

useful in controlling the number of requests on the server, but it does so at the expense of “goodput" [54] directly affecting user experience.

To see the impact of an AQM approach that drops requests once they exceed a response time bound, we repeat our experiment with forty upstream nodes and ten backend nodes. In Figure 5.5 we show the impact of setting a 3 second timeout versus the default system with a 20 second timeout. Setting the timeout to a low value is similar to having an admission control system that will not allow any request into the queue if they would take longer than the timeout value. The results show that while the 3 second timeout puts a hard cap on the tail latency, it doesn't have much effect on the head latency, indicating that load is still not evenly distributed. Even worse, we find that the timeout-based system drops nearly 50% of the requests entering the system in order to achieve this, and that the load variation across backends is not significantly improved.

5.3 Least Connection Analysis

Join-the-Shortest-Queue, JSQ, has been proven to be a nearly optimal load balancing algorithm, [29]. In JSQ, the arrival rate into each queue is dependent on the length of the queue at that point in time, with the probability of a queue receiving a new request decreasing as the length of the queue increases. In the ideal case where all requests have the same cost, JSQ can guarantee that the most loaded server will have at most one more request in its queue than the least loaded server. For more complex scenarios, [57] provides an upper bound to the load imbalance between any two queues in the system. These properties ensure that load imbalances will be automatically corrected by JSQ by preferring backends with lower queue lengths. However, the JSQ algorithm assumes the load balancer

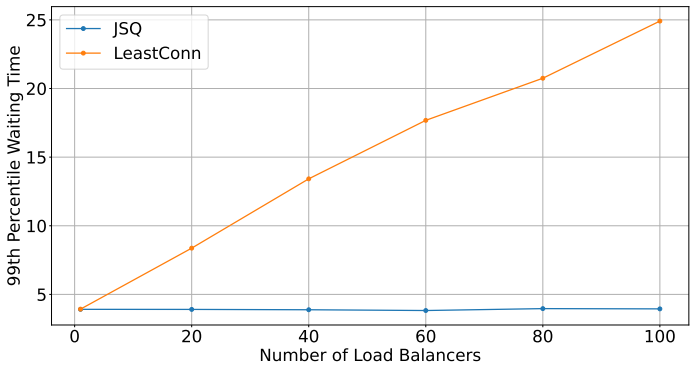


Figure 5.6: Simulating JSQ vs Least Connection shows how waiting time rises with the number of load balancers.

has a perfect view of the backend queue lengths, which is only feasible in a centralized situation with a single load balancer. Least Connection (LC), on the other hand, operates in distributed client side load balancers with incomplete knowledge of the backend queue lengths. In this section, we argue that LC cannot hold up the same guarantees as provided by JSQ and show why its performance degrades as the number of load balancers rises.

Simulation Analysis: We use a discrete event simulator to evaluate JSQ and LC performance. In Figure 5.6 we show how the 99th percentile of waiting time changes when simulating different numbers of load balancers with 10 backends. For the JSQ case, we assume that all load balancers have perfect information about the backend server queue lengths, so performance is steady with no impact as we adjust the number of load balacners. On the other hand, Least Connection sees a continual increase in tail response time as the number of load balancers rises.

Least Connection performs poorly for two reasons. First, each LC load balancer only has visibility into the subset of requests that it receives. On average, we expect N load balancers to each receive $1/N$ fraction of the total load. If $N = 1$, then LC is equivalent to JSQ, but as N rises, each load balancer is only seeing a smaller fraction of the total load. Intuitively, we can expect

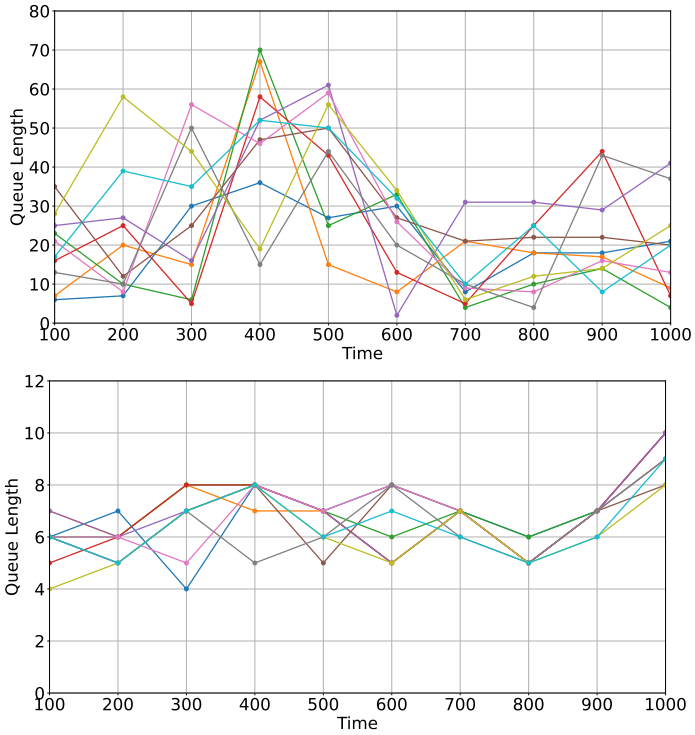


Figure 5.7: LC (top) sees both higher and more variable queue lengths than JSQ (bottom) over time

LC to make worse decisions as N rises since it has less useful information to base decisions on. Even if each load balancer were to try to account for this by multiplying the load it is aware of by a factor of N, we cannot expect this to be perfectly accurate due to the randomness in how requests reach each load balancer. Balls and Bins analysis can be used to show that as N rises, the difference between the most loaded and least loaded load balancer (and thus the imprecision of their estimates) will increase based on $log(N)$ [58]. Using our simulator, we measure the difference in queue length between the server selected by our LC load balancer for each request and the server that would have been selected if we had used JSQ. We find that with 20 load balancers, the median queue length difference is 3, and this rises to 9 when we increase to 100 load balancers.

The second challenge for LC is that each load balancer may only have

information about a subset of the backends. As N rises relative to the number of backends, it becomes more likely that a load balancer will have some backends for which it currently has no active requests, but this does not mean that the queue for those backends is empty. In these cases, LC simply picks a random backend with no active requests, leading it to behave more similarly to a random load balancer. To illustrate this challenge, we graph the queue length of each server over time under each load balancing scheme in Figure 5.7 when there are 100 load balancers. The graphs show that LC tends to have significantly higher queue lengths and that the spread between the most and least loaded backend is very large. In contrast, JSQ keeps a much lower average queue length (note the different y-axis), and the queue lengths for all servers are correlated, implying that they are changing based on the burstiness of the incoming workload, not based on random decision-making. Note that here there are 100 load balancers, yet the queue lengths on each server are on average around 30 for LC, implying that most servers have zero active connections from most of the load balancers. As a result, LC's behavior is tending towards being uniformly random instead of being able to intelligently use queue lengths.

Information Theoretic Analysis: To understand why LC does worse than JSQ, we also perform an information theoretic analysis using the concept of Shannon entropy. We compare the entropy equations for JSQ and LC to show that LC has significantly higher entropy, meaning we expect greater randomness in its output which will lead to poorer decision-making. Let us consider the following situation with M backend servers, L frontend load balancers, X requests arriving at the system in total, and at most Y requests being seen by any of the L load balancers.

- When JSQ is used, the load balancers are aware of the exact queue

size in each of the backend servers. This state can be represented as a vector, $S[M]$, where each entry $S[j]$ represents the queue size of the backend j, with the maximum value being X. Assuming all requests can be directed to any of the backends, the number of states in this case can be, $(X+1)^M$. If we assume, the probability of the system being in any state is uniformly distributed then:
$H[JSQ] = \log_2((X+1)^M) \implies H[JSQ] = M * \log_2(X+1)$

- When Least Connection is used, each of the L load balancers has its own view of the backends. The state in this case can be represented by a matrix, $S[L][M]$, where $S[i][j]$ is the i^{th} load balancer's view of the j^{th} backend. Considering each load balancer can only see at most Y requests, the number of states for each load balancer is $(Y+1)^M$. Across L load balancers then, the size of the state is $((Y+1)^M)^L \implies (Y+1)^{ML}$. Hence, the entropy of the system in this case is:
$H[LC] = \log_2((Y+1)^{ML}) \implies H[LC] = ML * \log_2(Y+1)$

Comparing the above values, we can see that $H[LC] \geq H[JSQ]$ since $ML * \log_2(Y+1) \geq M * \log_2(X+1)$, and that the uncertainty of LC grows linearly with N. While our model of entropy is imperfect since not all states will be uniformly distributed, the significant difference in entropy suggests LC will tend to make decisions based on greater randomness than JSQ.

5.4 System Design

In this work, our goal is to show that better load balancing can be done by combining AQM, backpressure, and a novel "confidence chip" distribution scheme that allows upstream load balancers to perform rate limiting in a self-organizing manner. The simple idea is that as requests flow downstream,

server metadata flows upstream to inform better load balancing. We wish to keep each server under its maximum capacity, distributing the load evenly through the system, without incurring overheads from explicit messaging or requiring global coordination which cannot scale to large microservice deployments.

Our framework is divided into two logical parts. First, we use “confidence chips" as a form of load information to make the load balancers’ decisions smarter. Confidence chips flow upstream from backend nodes, piggybacked in the response headers of successful requests. Rather than just use local information like the number of active connections, the load balancers use the number of confidence chips they have received from different backends as an indication of how likely they are to be able to handle additional requests at this time. This allows the backend to help load balancers coordinate request rates, without requiring any direct communication.

Second, BLOC uses overload control to restrict the number of active requests on the downstream nodes. Downstream nodes preemptively reject incoming requests if they will cause them to become overloaded. However, rather than simply dropping the requests, the upstream load balancer takes this as a hint both to back off from this server for some time and to retry the request on a different server.

A final key design consideration is that we seek to avoid adding complexity to the overall system deployment or adding centralized services that cannot scale well to large systems. Thus we eschew approaches such as the Redis-based global coordinator described previously. A centralized approach would be difficult to deploy in practice and could incur high overhead in terms of latency and resource cost if every request needed to access it in a large-scale system. Just as importantly, we seek to support legacy code by incorporating

BLOC into the sidecar proxies deployed alongside applications. This allows us to seamlessly add this functionality without any code modifications to the actual applications.

5.4.1 Confidence Chips

BLOC uses "confidence chips" as a way for upstream nodes to quickly learn which downstream nodes are above or below capacity. Each downstream node probabilistically returns a chip to upstream nodes piggybacked with the response header. An upstream node views the availability of a chip for a downstream as an indication that the particular node will have enough capacity to fulfill a request. The upstream spends a chip to make a request.

The probability of a downstream node returning a chip is related to how loaded the server is currently. This probabilistic distribution also serves as a hedge against requests from upstream nodes that the downstream is not talking to currently. We can reserve some capacity for upstream nodes for whom we do not have an active request right now but who might send a request to us soon. Also, since downstream nodes do not track chips granted, the probabilistic distribution protects the downstream from becoming oversubscribed.

Algorithm 2 details how BLOC processes each request. BLOC allows users to provide a capacity value in the configuration for the backends. This capacity value is used to limit the queue size in each individual backend. Lines 2-3 show that if adding another request would push the current queue size of the backend over the capacity configured then the request is rejected. Otherwise the request gets processed (line 4).

Lines 5-9 of algorithm 2 show our “confidence chips" calculation. This is a binary valued indicator sent along with each response to inform upstream

Algorithm 2 Backend Request Processing

```
1: function HANDLE(w: http.ResponseWriter, r: http.Request)
2:     if Capacity and QueueLength + 1 > Capacity then
           Respond(TooManyRequests)
           Exit
3:     ProcessRequest(r)
4:     if Random() < QueueLength/(0.8 * Capacity) then
5:         chip ← 0
6:     else
7:         chip ← 1
8:     Return chip bit with response
```

clients whether this backend has capacity available to handle more requests. In order to determine the value of the confidence chip, a random number, between 0 and 1, is compared against the ratio of capacity currently being used on the backend (line 5). A confidence chip returned with the response has a value of 1 (line 8) if the capacity ratio is lower than the random number. Otherwise the confidence chip has a value of 0 (line 6). Furthermore, the actual calculations are made against 80% of the defined capacity value to provide some protection against overloading. This means that as the queue length on the backend starts to approach 80% of its defined capacity, the backend will start sending less and less chips with a value of 1. It will send no chips with the value of 1 once queue length exceeds 80% of the defined capacity.

Every node has a "capacity" value defined at the service level which represents the number of requests it can have in its queue while meeting a target SLO. When responding to a request, the nodes decide whether or not to issue a chip to the upstream node based on the following formula:

$$r = \text{uniform random number} \in (0,1]$$

$$chip = \begin{cases} 0, \text{ if } r < \frac{q}{0.8 * ServiceCapacity} \\ 1, \text{ otherwise} \end{cases}$$

where q (*QueueLength* in backend request processing algorithm2) is the number of requests currently queued in the downstream node.

In this case, we guarantee that at least 20% of the downstream node's maximum capacity will always be reserved for the upstream nodes that are either not interacting with the downstream at the moment or have already been granted a chip. The probabilistic nature of the equation means that the number of chips generated during any given period of time will depend on the load on the downstream node during that time. Hence, when the server is lightly loaded it is more likely to send out chips whereas, during times of higher loads, it sends a lesser number of chips.

5.4.2 Client Side Backoff and Retries

Algorithm 3 describes how each client sends requests to backends. We saw in algorithm 2 that any request that pushes a backend's queue size beyond the defined capacity is rejected. In order to handle such cases, we also define the number of times each request can be retried. The while loop, lines 3-13 runs the overall algorithm. In line 4, we use the "PickServer" algorithm, described in the next section, to choose a possible backend to route the request.

If the server rejects the request, then it is reflected in the response from the server, line 5. When this happens, the upstream node increments the

Algorithm 3 Retry and Backoff

```
1: function ProcessRequest(request)
2:     retry_num ← 0
3:     while retry_num ≤ total_retries do
4:         server = PickServer(servers)
5:         response = server.Handle(request)
6:         if response == TooManyRequests then    ▷ Server becomes "inactive"
7:             retry++
8:             server.chip ← 0                                  ▷ Reset "chips"
9:             server.RcvTime ← GetCurrentTime()    ▷ Calculate ResetInterval
   (line 5 and 9, algorithm 4) from here
10:        else
11:            break
12:    return response
```

number of retries for that request (line 7), resets all chips for that backend (line 8) and sets the probe timer (line 9). At this point, that particular backend is considered to be "inactive".

If the request is successful or we have exhausted the number of allowed retries for the request, then the while loop (lines 3-13) is exited and the response is returned. Irrespective of the status of the response, we also note when the server received the last response. A request is dropped only if all retries have been exhausted.

5.4.3 Server Selection

BLOC's algorithm for server selection, Algorithm 4, starts by picking two backends at random, lines 2-9. However, the algorithm takes care to ensure that the servers selected are considered "active" by the upstream,

Algorithm 4 BLOC for Server Selection

```
1: function PickServer(servers)
2:     server1 ← RandomSelect(servers)
3:     // Either "ResetInterval" has passed since last response.  ▷ This backend can be probed.
4:     // Or the backend has a chip.  ▷ Backend is "active"
5:     while ElapsedTime(server1.RcvTime) ≤ ResetInterval and server1.chip == 0 do
6:         server1 ← RandomSelect(servers)
7:     server2 ← RandomSelect(servers)
8:     while ElapsedTime(server2.RcvTime) ≤ ResetInterval and server2.chip == 0 do
9:         server2 ← RandomSelect(servers)
10:    if server1.Reqs < server2.Reqs then  ▷ Select server with lower active requests from this upstream
11:        selectedServer ← server1
12:    else
13:        selectedServer ← server2
14:    if selectedServer.chip ≤ 0 and ElapsedTime(selectedServer.RcvTime) > ResetInterval then
15:        selectedServer.RcvTime ← GetCurrentTime()  ▷ Ensure this server is not probed too soon
16:    return selectedServer
```

or that a reset interval has passed since an overloaded server was last contacted. An upstream considers a backend "active" if and only if the upstream has a chip for that backend. Lines 5 and 9 of Algorithm 4 test for

this condition. Alternatively, lines 5 and 9 also test if the predefined reset interval has passed since a response was last received from that backend. If this condition is satisfied, then the upstream may select that server despite it not being “active". This is because we consider the state of the backend as unknown if at least the reset interval has passed since a response was last received. This backend, despite being inactive, is chosen because we want to probe inactive servers at regular intervals in order to update the upstream's metadata about them. However, when this happens we do not want to send another probe to the same backend too quickly. This is ensured by updating this backend's “RcvTime" to the current time (lines 17-19). Unless a chip for this backend is received, it will be rejected for all future iterations that run in the next “ResetInterval" units of time (lines 5 and 9).

When we have found two randomly selected servers that satisfy one of the conditions being tested in lines 5 and 9, we pick the one with the lower number of pending requests, as known to this particular client, (lines 12-16). Finally, this server is returned in line 4 of Algorithm 3. This is similar to the LeastConnection algorithm but BLOC ensures that the servers selected are unlikely to be overloaded.

5.4.4 Server Capacity

The capacity parameter plays an important role in determining the performance of a system. This parameter forms an upper bound on the size of the active queue of any upstream node in the cluster.

In the simplest case, a system administrator can specify a fixed Capacity value for each microservice based on its expected service time and SLO. The Capacity value times the service time gives an upper bound on request queueing time. For simple services, this may be feasible, but for large-scale

applications with many microservices, or deployments on heterogeneous hardware with different service costs, it may not be practical. Alternatively, BLOC has a simple method to dynamically estimate the capacity. This makes the system compute a cumulative average of the number of active requests in its queue for 30 seconds. The system then uses this average as the capacity value. We reset and recompute this average every 30 seconds. It admits all requests by default in the first 30 seconds where the capacity value is not defined yet. Improving this capacity estimate is an area needing further research.

5.5 Implementation and Experimental Setup

5.5.1 Customizable Microservice Generation

Our experiment testbed has been inspired by the Deathstarbench [54], which provides a set of premade microservice applications for system benchmarking. However, Deathstarbench is limited in its flexibility to only support its predefined applications. For BLOC, we built a customizable microservice generator that can define arbitrary microservice topologies [53]. Each microservice component is generated as a Python Flask service with a customizable request processing time and can optionally drive the input of many other services (fanout). The fanout is simulated by making parallel requests to each downstream service. Configuration files are generated to deploy the services in a Kubernetes cluster and automatically interconnect them to form the service mesh.

5.5.2 Sidecar Proxies

We also built the BLOCProxy reverse proxy framework [53] from the ground up to enable us to implement our algorithms with ease. The proxy handles ingress and egress traffic, allowing it to implement both admission control and load balancing. We redirect all incoming and outgoing traffic, except traffic to and from the proxy itself, to the proxy input and output ports respectively. The proxy maintains a local directory mapping pods to service types as well as the Active and Inactive lists. During each request, the proxy can select the next endpoint by using a load-balancing algorithm defined through an environment variable along with other metadata.

Currently, the system implements the following load-balancing algorithms:

- Random
- P2C Least Connection
- BLOC

The BLOC egress proxy modifies the HTTP headers generated by the microservice application to add a field indicating if confidence chips were generated. This is then interpreted and stripped out by the BLOC ingress proxy on the upstream node that generated the request. As of now, our implementation only supports HTTP1.1-based applications. However, our approach could easily be extended to support other protocols such as gRPC, broadening the types of applications that can make use of our design without any code modifications to the applications themselves.

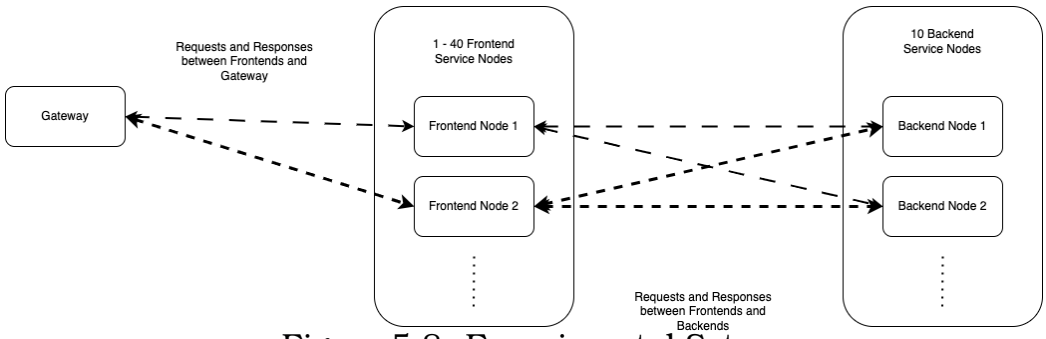


Figure 5.8: Experimental Setup

5.5.3 Control Plane

We also developed a simple control plane that uses the Kubernetes API to monitor live endpoints for each service that has been deployed. The proxies make REST API calls to the control plane pods, which run as a daemonSet in the Kubernetes cluster to populate their local service directories.

5.5.4 Test Bed Setup

In order to focus on load balancing between a pair of microservices, we use BLOC to run an application consisting of three layers of services (Fig 5.8) with the total number of pods ranging between 12 and 51 in a Kubernetes cluster running on 4 physical nodes.

The Gateway layer consists of a single pod that acts as the ingress gateway. All requests to the cluster are forwarded to this gateway and are distributed to the frontend layer. The frontend layer, in turn, is variably sized. It scales between 1 and 40 pods. This layer sends all requests to the backend layer. The backend layer has a constant size of 10 pods. We overprovision the gateway and frontend layers so they will not become the bottleneck.

5.5.5 Workload

Most of our experiments have been conducted with a basic backend service that simply sleeps for 250ms. However, we also test BLOC with backend service costs between 100 to 500 ms and there is provision for a variable service cost, which randomly selects a service cost uniformly in a range configurable through the environment variables.

For load generation, we use two open-source tools:

- hey [59], which is a closed-loop load generation tool that allows us to configure a concurrency for the requests we make
- a custom version of load test [60], that lets us define mean requests per second and generates load according to a Poisson distribution with this configured mean.

5.6 Evaluation

5.6.1 Experimental Setup

We ran our experiments on cloudlab [52] servers. A Kubernetes cluster was created with four Intel Xeon servers, each with 20 cores and 196GB of memory. We then deployed our control plane that ran a pod on each of the servers. These pods form the service that is queried to get information about the backends of services running in the cluster. We create an affinity between our services and the physical nodes, such that the gateway and frontend services run on 3 of the 4 physical nodes. The fourth physical node can only schedule pods of the backend service. This was done to ensure that the performance of the backend pods was not interfered with. We use hey [**?**] as our closed-loop generator. We use the tool to send requests to the

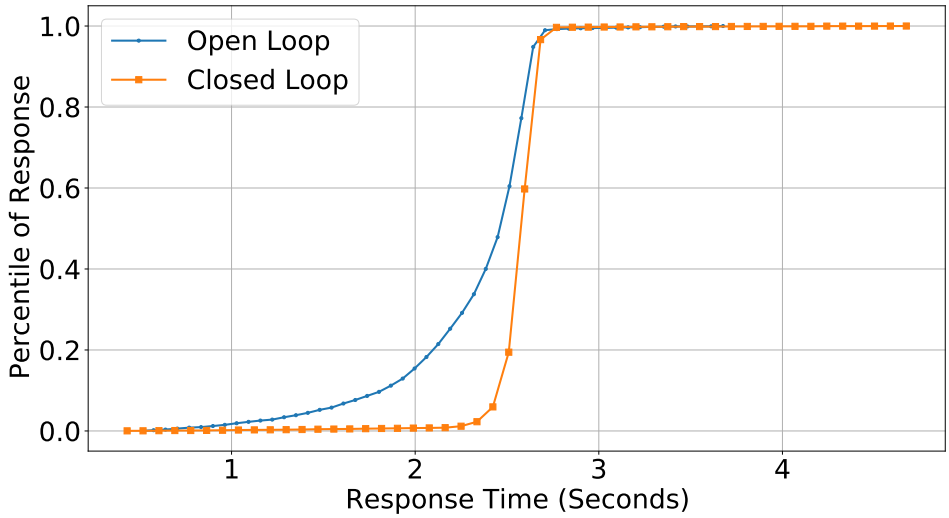


Figure 5.9: BLOC (Cap=10) provides a substantially tighter response time distribution by avoiding incast problems and applying careful admission control

gateway for a fixed amount of time (5 minutes) where every request starts a new TCP connection.

In most of our experiments we compare BLOC against Least Connection with 40 frontends and a centralized JSQ approach. We implement JSQ by using a single node in the frontend tier running the LC algorithm (since LC with 1 node is equivalent to JSQ). This allows us to see how BLOC compares against an algorithm with perfect load balancing information as a nearly optimal baseline. However, in some of our experiments, the single frontend node for the JSQ case can become a bottleneck leading to worse performance than BLOC; we note these cases as they arise.

5.6.2 BLOC Overall Performance

We first compare BLOC, Least Connection, and JSQ to evaluate our approach's impact on response time distribution. Figure 5.9 shows the

response time CDF of each approach when the upstream nodes are accessing a shared pool of ten backends. We compare forty BLOC upstream nodes with a fixed Capacity parameter of 10 against LeastConnection with either forty or one upstream node (which is equivalent to a centralized JSQ algorithm). While the median response times of all approaches are similar, there are dramatic differences in their response time distributions. When there are forty upstream nodes, Least Connection sees a very wide response time distribution, with the fastest 10% of requests finishing within 1 second and the slowest 10% of requests taking about 4 seconds, giving a 10-90%ile Range of 2.77 seconds. On the other hand, BLOC maintains a very narrow response time window, with a range of 0.97 seconds. In fact, BLOC achieves a tighter window than Least Connection running with a single upstream node (we ensure that the frontend is not the bottleneck in these experiments by using downstream backends with expensive service costs of 250 msec). Thus BLOC's distributed sidecars are able to effectively determine the relative loads on different servers, improving overall system utilization and providing very consistent response times.

Next, we vary the service cost of the backend nodes to understand the impact on load balancer performance. Figure 5.10 shows the improvement of BLOC over LC with forty frontends and 10 backends with a per request service cost ranging from 0.1 to 0.5 seconds. We use boxplots to show the median (black line), upper and lower quartiles (box edges), and 90%tile tail latency (whiskers). The response distribution of BLOC with 40 frontends is much closer to that of Least Connection with a single frontend compared to Least Connection with 40 frontends. The presence of outliers beyond the upper whiskers indicates a noteworthy observation: the 99th percentile tail latency of the BLOC when configured with 40 frontends, closely parallels

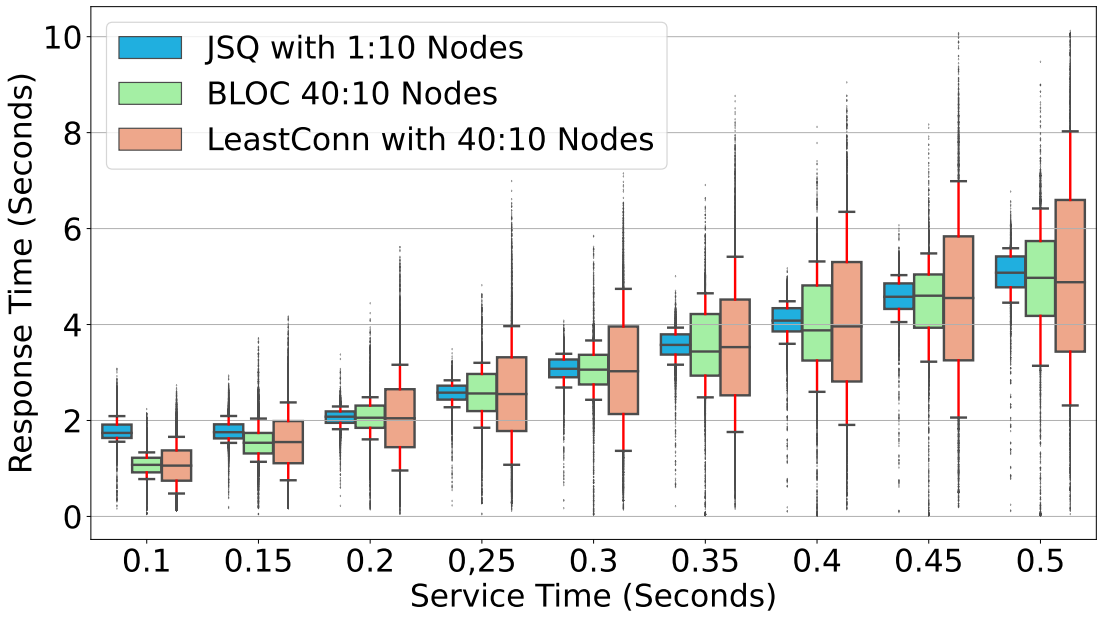
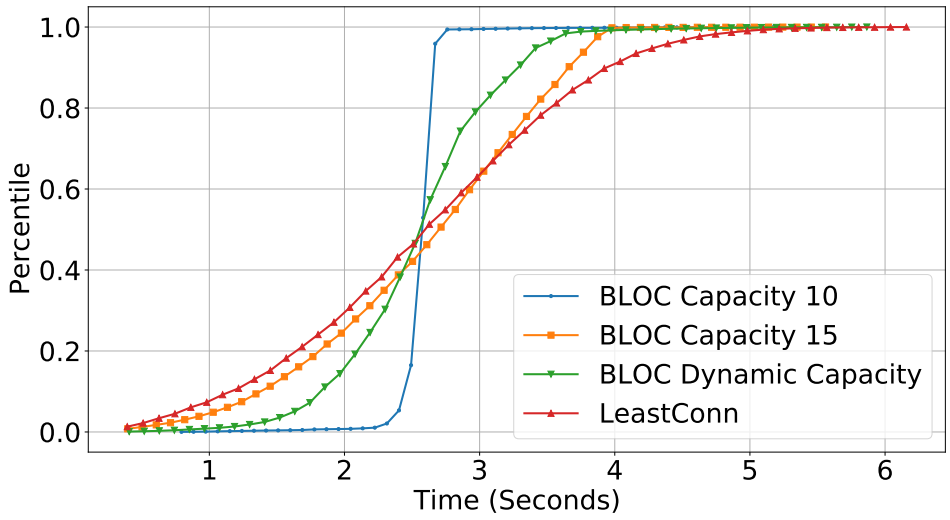


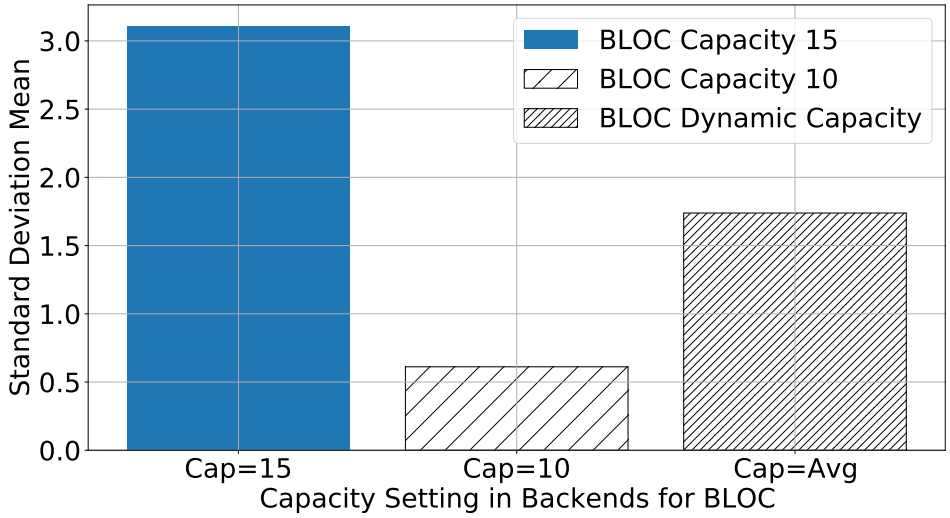
Figure 5.10: BLOC (Cap=10) provides a substantially tighter response time distribution by avoiding incast problems and applying careful admission control

that of the Least Connection when it operates with only a single frontend, whereas that of Least Connection with 40 frontends diverges from Least Connection with 1 frontend as the service cost increases. BLOC's performance is similar to a centralized JSQ load balancer, although since it lacks perfect information it cannot maintain as tight a responset ime distribution. However, with a very low backend service time (0.15 and below), we see that the single JSQ load balancer is the bottleneck, resulting in worse performance than BLOC.

BLOC relies on its estimate of downstream capacity to control its AQM algorithm and allocation of confidence chips. To illustrate the impact of the Capacity parameter, we evaluate several fixed settings and BLOC's dynamic capacity estimation technique. Fig 5.11a shows the difference in performance when we used a Capacity of 10 (which our tests suggest is optimal for this configuration) Capacity of 15 (which tends to too aggressively overload



(a) BLOC Capacity Sensitivity



(b) BLOC Capacity Burstiness

Figure 5.11: Sensitivity to capacity and impact on load imbalance

servers), and our dynamic Capacity value based on the observed average. All of these approaches provide an improvement over LeastConnection, but setting an appropriate value gives a tighter bound.

To further analyze the impact of Capacity, Figure 5.11b shows the level of imbalance on the downstream servers. This is measured by looking at the number of requests served by each node over time and calculating the standard deviation between them during each time interval; we then plot the mean of this variability. The results show that our hand-tuned Cap=10 setting provides the greatest benefit, but that using the dynamic averaging approach also keeps the variance relatively low.

5.6.3 Benefits of Different BLOC Components

BLOC employs several techniques to avoid overload and keep downstream nodes balanced, so in this experiment, we quantify the benefits of each approach. In Figure 5.12a, we show the CDF of AMQ, AMQ with BLOC techniques and Least Connection. We can see that using AQM to drop requests that exceed the downstream node's capacity (without the rest of BLOC's functionality), provides a substantial improvement in response time. However, this only shows the performance of requests that are successfully processed, and as shown in Figure 5.12a, AQM drops about 4,000 of the 12,000 requests sent during the experiment. Adding BLOC's backoff technique provides a further benefit to response time by reducing the chance that requests will be added to a long queue, however, it leads to an even higher drop rate. Adding support for Retries substantially improves the system, eliminating most of the drops and also providing a further reduction in the interquartile range. The final BLOC system that supports AQM, Retries, and Backoff provides a significant improvement to response times

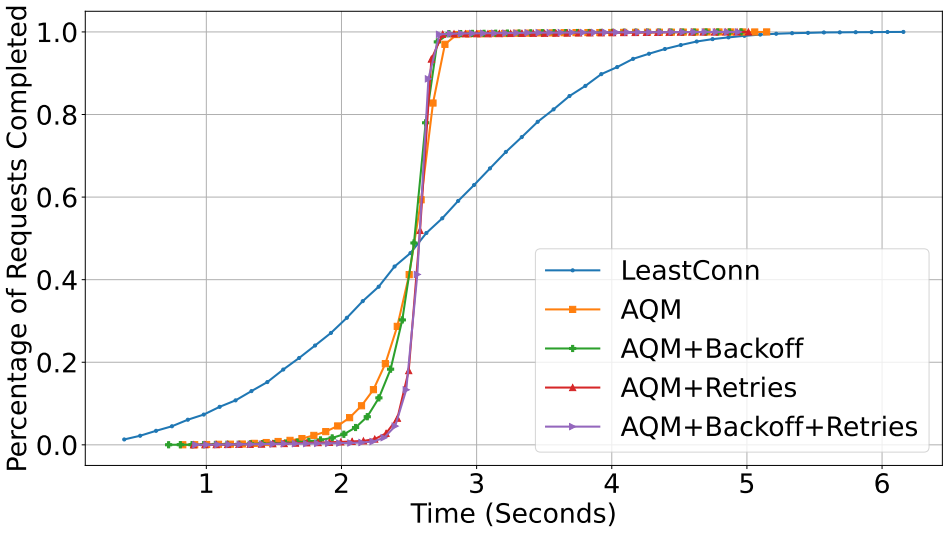
over LeastConn and reduces the number of failed requests by 22% (from 446 to 346) compared to the system with only AQM and Retries.

5.6.4 BLOC Under Bursty Workloads

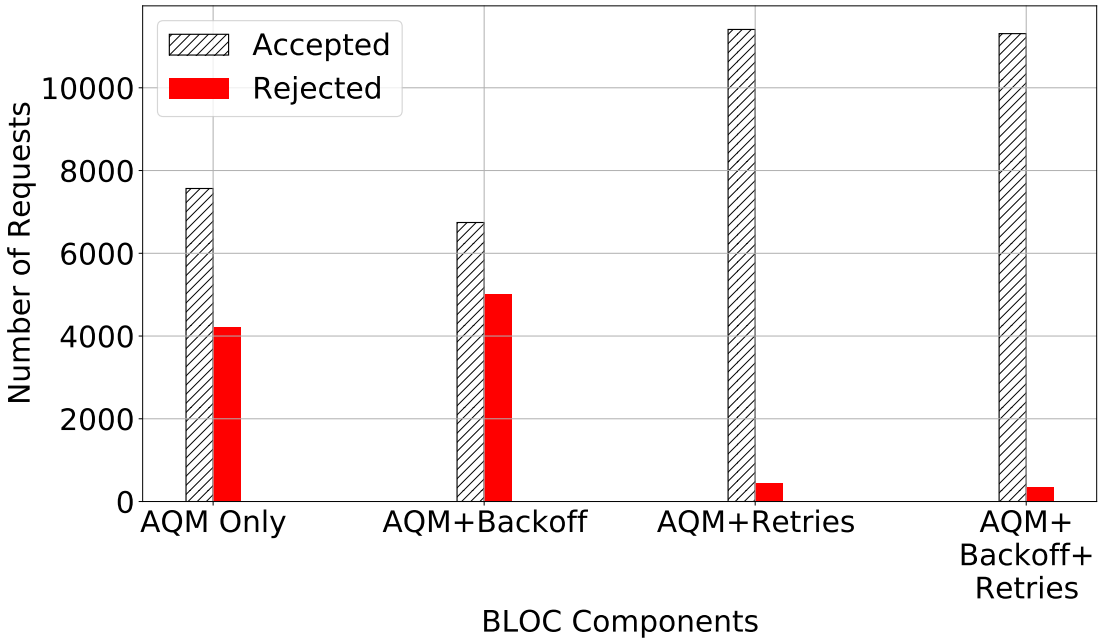
The prior experiments used the Hey benchmarking tool, which is a closed-loop load generator that seeks to continuously saturate the system. While this is an effective way to test the system on the brink of overload, it may not be representative of real web workloads which tend to have bursty periods of light and heavy load. In this experiment we use a customized version of loadtest [60], which is an open loop generator that can send requests at variable rates. While the official loadtest distribution follows a uniform distribution, our modified version sends requests following a Poisson distribution which gives a more realistic bursty arrival process.

In Figure 5.13 (Left-axis bars), we show the performance of BLOC relative to LeastConnection with forty upstream nodes and ten downstream nodes under increasingly intense request rates. The results show that BLOC provides a substantially better 90th percentile latency, allowing it to support a much larger incoming request rate than LeastConnection. LeastConnection becomes overloaded with very poor performance after a workload of 35 req/sec, whereas BLOC is able to gracefully handle loads as high as 47 req/sec.

While BLOC provides a dramatic improvement in response time distribution at high load, it is in part due to its preference to drop requests that will cause excessive queuing. To evaluate this, Figure 5.13 (Right-axis lines) shows the percent of requests dropped at each request rate for LeastConnection and BLOC. At lower request rates, BLOC still drops a small fraction of requests due to the bursty arrival pattern which can cause spikes in queue



(a) BLOC Components Response Time Comparison



(b) BLOC Components Throughput Comparison

Figure 5.12: The combination of all BLOC components ensures a tight response time distribution while minimizing request drops

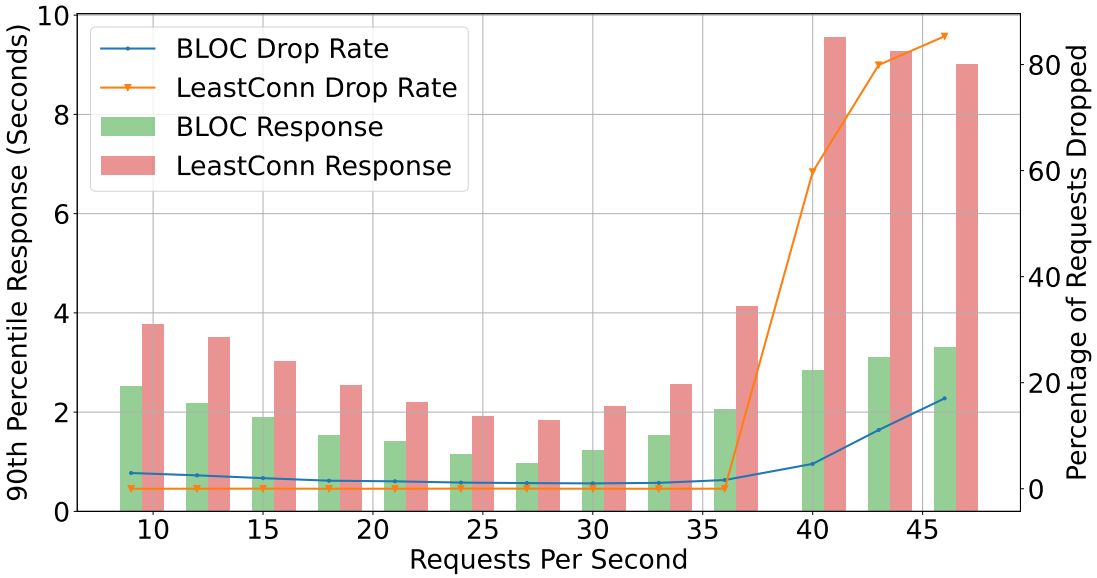


Figure 5.13: 90th percentile response time (Left-axis bars) and dropped requests (Right-axis lines) with Poisson load generated at different rates

length[2]. Nevertheless, BLOC's drop rate is reasonably low, and even when facing an overloaded system at the highest request rate, BLOC drops only 16% of requests compared to LeastConnection dropping more than 80%.

5.6.5 Handling of New Resources

In this experiment, we start out by sending requests with loadtest [60] (open loop) to a cluster with 40 frontend services and 10 backend service instances. We increase the scale of the backend service by one pod a minute into the experiment (total 5 minutes) and plot the response times in Figure 5.14. The load being sent to the cluster makes the cluster slightly higher than the capacity of the cluster. We see that under this amount of load, Least Connection's response times rise quite high very fast. Both Least Connection and BLOC have a mean response time of 5000ms in the first minute. But even after adding a new pod, Least Connection is still

[2]In fact, we believe BLOC's drops may be due to a bug causing the gateway node to incorrectly drop requests even though the downstream nodes are not full.

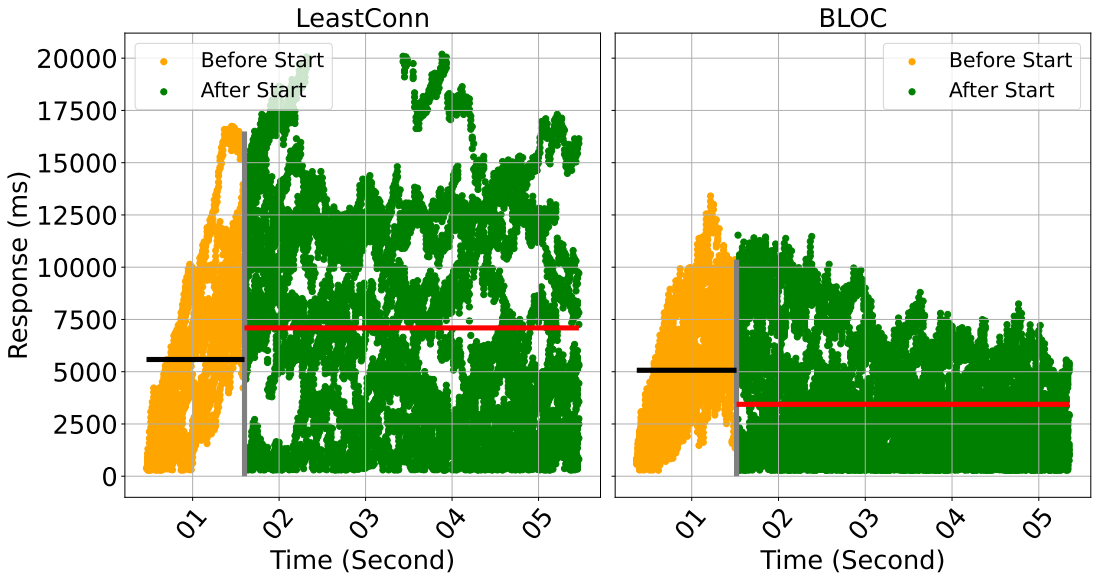
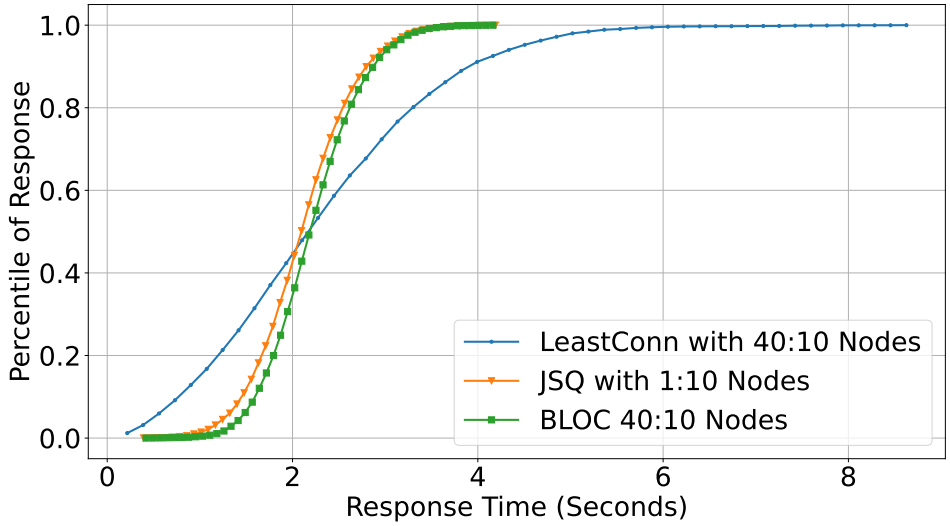


Figure 5.14: BLOC and Least Connection behavior when adding new resources to the cluster

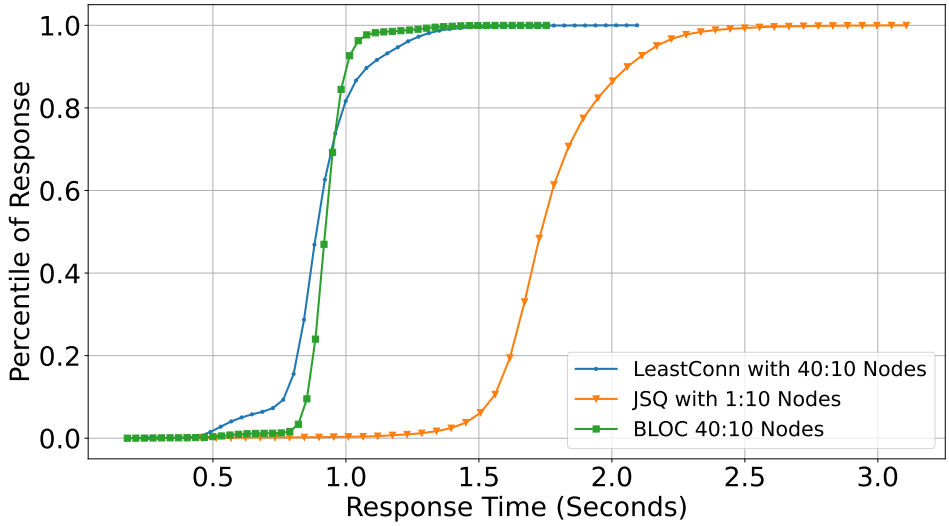
overloaded because it cannot effectively rebalance the queues on its servers. This would likely lead to LC needing to add another pod to stabilize the load. On the other hand, BLOC is able to utilize the newly added pod better and quickly reduces the mean response time to 3400ms, a reduction of 32% reduction in mean response time. This illustrates the importance of load balancing not just in reducing response times, but in reducing the total resources that must be assigned to the system.

5.6.6 A Real Variable Cost Backend Application

We use a real application that resizes images in response to incoming requests. This resizing application has a significant variance in its service cost, between 40 and 650ms. Figure 5.15a shows that BLOC is still able to make good decisions to optimize the response time of each request, despite the large variance, matching the performance of a single LC frontend load balancer.



(a) High, Variable Cost Application



(b) Low Service Cost Application

Figure 5.15: BLOC, JSQ vs Least Connection for applications with different service costs

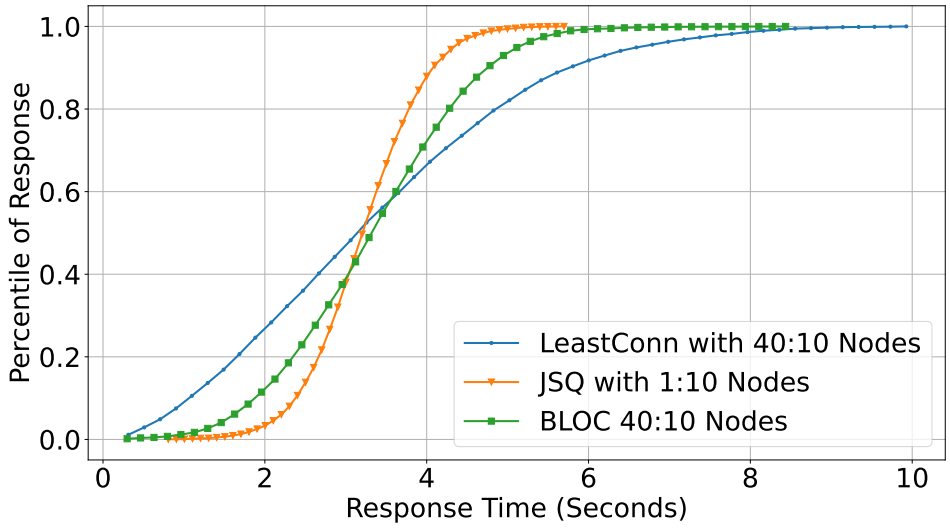


Figure 5.16: BLOC, JSQ vs Least Connection for a complete chain

5.6.7 Low Backend Service Cost

When the service cost of the backend layer is low, the frontend layer becomes the bottleneck. Scaling up the number of frontend layers improves performance. In this experiment, we have an app that rotates images in response to incoming requests and the service cost for this backend varies between 10 and 80ms. Figure 5.15b shows us that while both LeastConnection-with-40-frontends (LC40) and LeastConnection-with-1-frontend (LC01) have a narrow response time distribution, LC40 has a significantly better performance. Further, we see that BLOC-with-40-frontends has a response time distribution that is even narrower than LC40 and has a shorter tail than LC40.

5.6.8 BLOC vs Least Connection for a complete microservices chain

In this experiment, we combine image resizing (highly variable service cost) and image rotation (low service cost) components into a single backend

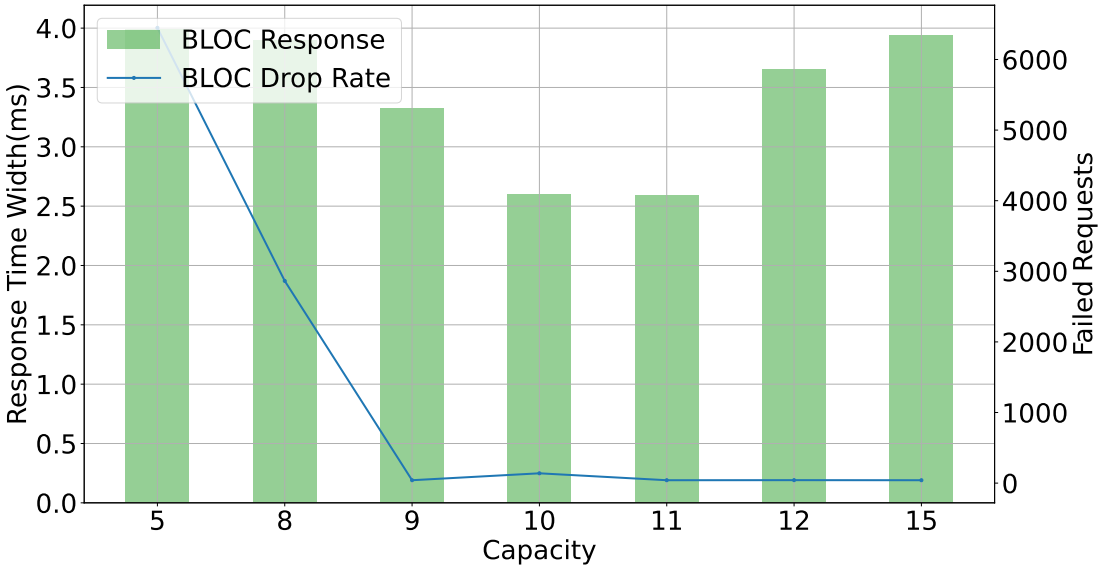


Figure 5.17: Impact of Different Values of Capacity - 90th percentile response time (Left-axis bars) and dropped requests (Right-axis lines) with Poisson load generated at different rates

application. Our experience dictates that estimating the capacity of such a variable and multi-layered backend is extremely difficult. In such cases, it is easier to use a dynamic capacity estimate. Figure 5.16 shows that BLOC with the dynamic capacity estimation algorithm can easily outperform Least Connection in such scenarios.

5.6.9 Impact of BLOC Parameters

In figure 5.17, we measured the width of the response time, the difference between 10 and 90 percentile responses, as we changed the static capacity values (Left-axis bars). This result indicates that results deteriorate quickly as we move away from the optimal capacity value. The dynamic capacity value, discussed in this paper, protects the system against such performance degradation by continually correcting the capacity by measuring real-time performance. We also measured how the number of failed requests changed as we changed the capacity value (Right-axis lines). This result is quite

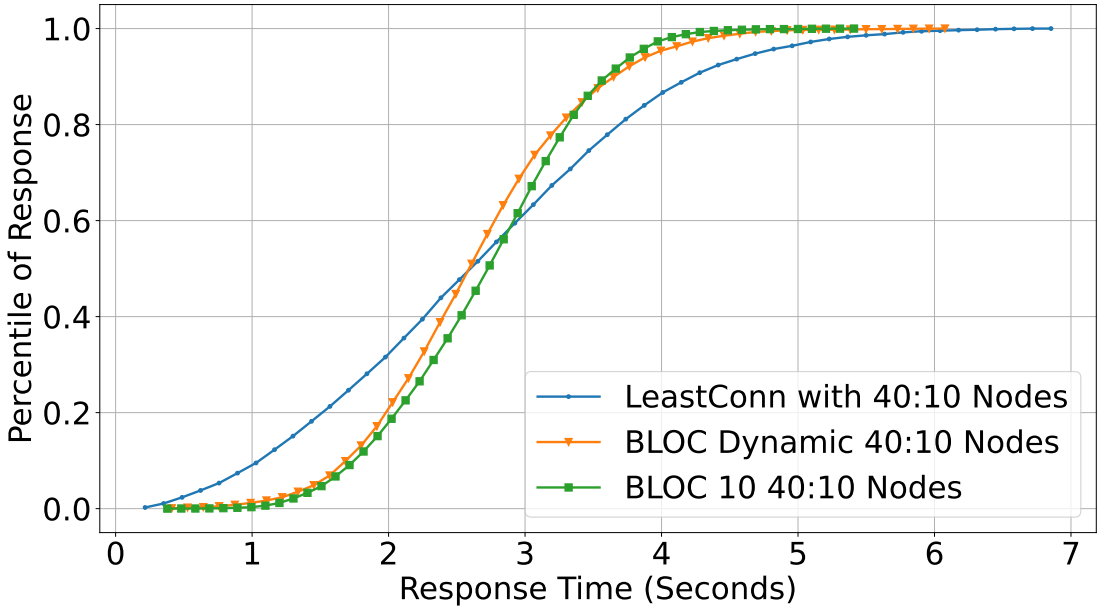


Figure 5.18: Dynamic and Static BLOC vs Least Connection Variable Service Cost

intuitive in that as we increase the capacity values, the number of errors decreases.

5.6.10 BLOC Performance with Variable Service Cost

Finally, we present a measure of the performance of BLOC vs. Least Connection under variable service cost. The service cost was chosen to be normally distributed with a mean of 250ms and a 125ms standard deviation, mimicking the parameters of the image processing application. In these experiments, refer to figure 5.18, we saw that both static and dynamic BLOC outperform Least Connection.

5.7 Related Work

In this work, we have combined load balancing with overload control:

- **Load Balancing** approaches typically attempt to solve issues related

to heterogeneity, performance, and uniform load distribution.

- **Overload Control** are admission control schemes that let servers control the rate at which clients can send requests.

5.7.1 Load Balancing

There is a wide range of work on load balancing for web [28] and cloud applications [31]. In microservices architectures, a load balancer plays an important role in terms of distributing workloads across multiple and various instances. There are two types of load balancers based on where the load balancer is placed, server-side and client-side. A study of trade-offs between server-side and client-side load balancers is presented in[61]. We have based our work on evaluating the LeastConnection algorithm, which is a client-side approximation of the JSQ algorithm [29], often used in the microservices environment.

Research on the performance of load balancers, recently, has generally looked at topics like handling heterogeneity[32], uniform load balancing with consistent connections[33], and so on. While it has been established that with centralized load balancing it is not possible to significantly improve JSQ [29], we find that this result does not port over to distributed client-side load balancing. In this work, we tweak these load-balancing algorithms to be aware that their data might be stale and to take overcommitment into account. As far as we know, there is no other work that takes a look at the load-balancing algorithms in microservices networks.

5.7.2 Overload Control

Overload on a system can cause catastrophic failures[62] and the idea behind overload control is to shed any excess load before it consumes any

resources[63]. In this work, we primarily use active queue management (AQM) to shed extra load. However, we do not want to sacrifice “goodput"[54] and as such build overcommitment and retries into the system. To our knowledge, there have been no prior attempts to use overload control toward load balancing in microservices.

5.7.3 Load Balancing with Server Feedback

There are two other systems [34], [7], that we know of, that incorporate feedback from the servers into how requests are distributed. In [34], the load balancer gets resource usage statistics from the servers to make its decisions. In our previous work [8], we have also used a similar feedback loop along with a measure of the server capacities. In a distributed load-balancing architecture like microservices, however, this leads to convergence issues. In [7], the authors use overload controls to ensure no backends are overloaded. However, the low target service cost of [7] enables communication between all clients and all servers, in the form of registration messages, allowing for a complete flow of information. BLOC works with a much higher service cost which implies that it needs to load balance and protect against overload without any node-to-node messaging.

5.8 Conclusions

Least Connection is a popular algorithm to balance load in microservices architecture and is based on Join the Shortest Queue, which has been proven to closely approximate optimal load balancing in a single node centralized load balancer. In the microservices world, the load balancer has moved from being a single centralized node to multiple instances each attached to a client service (upstream nodes). Here, Least Connection finds

it difficult to maintain the veracity of its metadata cache, which can atrophy quickly. This leads Least Connection to make bad load-balancing decisions in aggregation. This in turn leads to a significant widening of the response time distribution and the lengthening of the tail.

In our framework, BLOC, we show that using overload controls judiciously overcomes this problem and is a far simpler solution than maintaining a distributed state. We also show that BLOC significantly improves overall performance. In our experiments, response time distribution improved by 2 to 4 times and tail latency did so by nearly 2 times. Overall, our results show that carefully combining overload controls with load balancing can lead to consistent response time despite the presence of a large number of frontends sending requests to a shared set of backends. BLOC is able to guarantee this performance consistency without sacrificing either user experience (by dropping requests) or adding to the overall load and complexity of the system (by sending metadata messages or using centralized caching services). We also show that BLOC can work with systems with a wide range of service costs and can handle variable service costs given an appropriate capacity estimator function. Our goal for BLOC is for it to become a more generic load-balancing algorithm able to support systems in different operating ranges and our work here indicates that possibility. Further work on BLOC can be directed at exploring various capacity estimator functions that would further BLOC towards this goal.

Dynamic capacity estimation is a significant area of research in both industry and academia using techniques from queuing theory, machine learning, and dynamic programming among others. In this paper, we show that a simple running average method renders BLOC much more performant than state-of-the-art load-balancing algorithms. Thus investigating even

more powerful methods in the context of BLOC is one research direction worth exploring.

We have created a repository to enable anyone to refer to and run the code to verify our results at
https://github.com/MSrvComm/Experiments.

Acknowledgements: This work was supported in part by NSF Grant CNS-1837382.

Chapter 6: Load Balancing and Generalized Split State Reconciliation in Event Driven Systems

Event driven applications are often built with message queuing systems that provide no temporal upper bound on message delivery. However, many modern event driven applications, like a system inferring traffic conditions and generating recommendations to road users based on sensor data, are latency sensitive. Traditional message queuing systems use static load assignment algorithms that guarantee event ordering while mostly ignoring a temporal upper bound on message delivery. Another class of message queuing systems use stateless operators which deliver messages (events) quickly but pass the burden of stream state management to user applications. Synchronous communication patterns, on the other hand, provide an upper bound for message delivery while ensuring message ordering but unnecessarily bind limited resources reducing efficiency.

In this paper we explore load balancing choices in asynchronous systems and their impact on queuing delay. We then propose a load balancing framework, SMALOPS, for event driven applications with dynamically changing load and quick message delivery requirements. Our experiments confirm that with smarter load balancing, the 99%ile response times for events can be improved by as much as 73%, compared to traditional message queuing systems. SMALOPS introduces the following:

- A load balancing algorithm that can significantly reduce queuing delay in message delivery systems.
- Mechanisms enabling consumers to recover stream state when either the message delivery system does not support stateful operators or the

state has been split by moving streams between operators.

6.1 Introduction

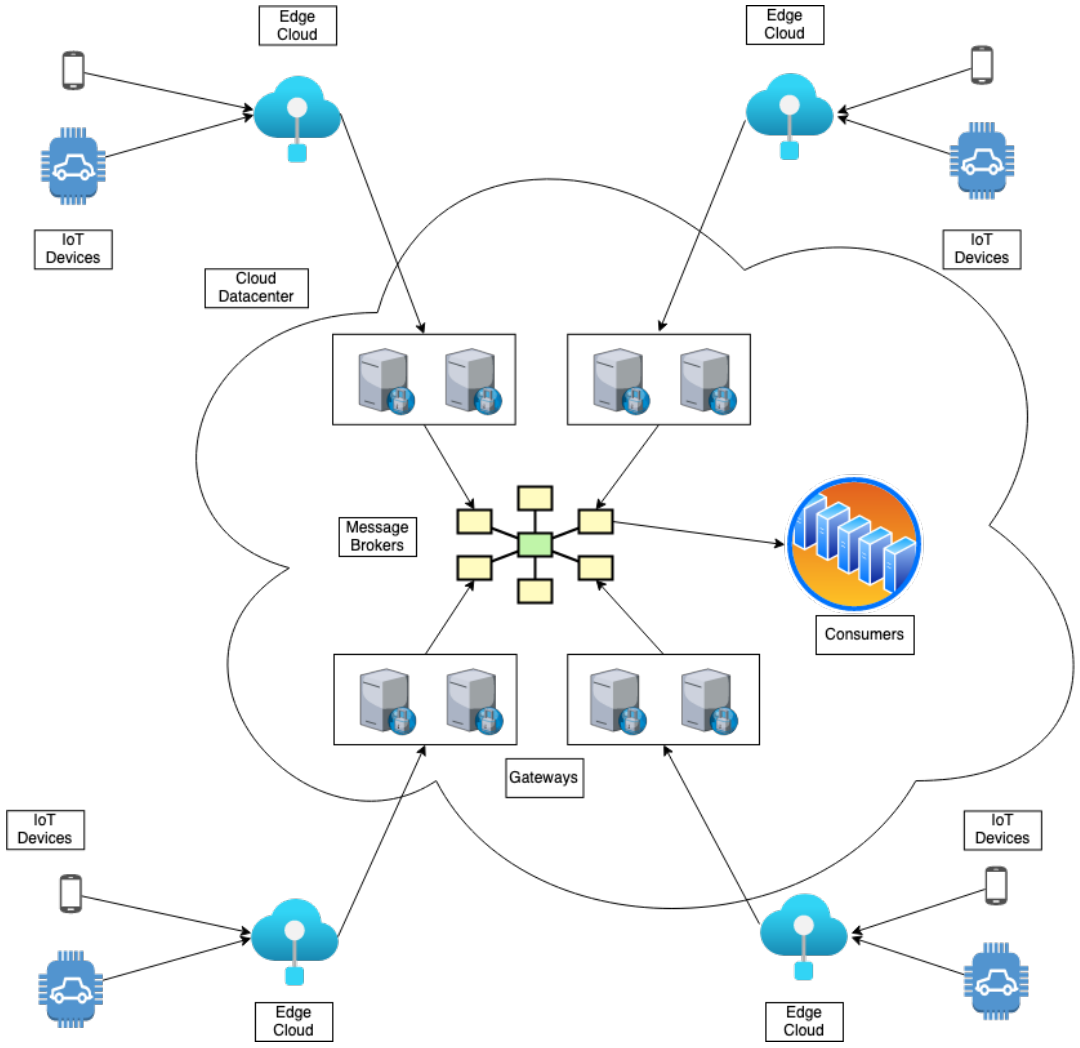


Figure 6.1: An example of an asynchronous architecture. IoT devices send messages to edge clouds. The edge clouds in turn forward those messages messages to cloud gateways (CGW) which forward them to Kafka brokers to be consumed by some consumer service.

Message queues are a popular tool for building event driven services. Traditional asynchronous services have a fire and forget model where messages tend to trigger a change of state of some downstream application(s). Usually, queuing delay is ignored since typical asynchronous applications have a

high tolerance for state change latencies. For example, when operating one's bank account, it is often acceptable for transactions to remain in "transit" for days without updating the state of the account. In other cases, the state change latency requirements are relaxed as such a state might be hidden from users, such as in log processing applications. As far as message delivery times are concerned, message queuing delay, message queuing systems make no guarantees.

Some message queuing systems provide strong message ordering guarantees leading such systems to disregard skewed load build-up. Further, most message queuing systems treat load balancing and stateful stream operations as orthogonal design choices. Systems like Apache Kafka assign load statically for the lifetime of the streams. While this prevents against split stream state, such a system is unable to handle a massive scale [64] or significantly skewed input [65]. Other message queuing systems like NSQ [23] offer a better scaling profile by foregoing stateful stream processing completely.

There is an emerging class of applications that use message queuing systems to integrate physical systems to data intensive applications. [64]. These applications , figure 6.1, operate on streams of data bounded by the notion of "freshness" imbibed in the messages that constitute the streams. Informally, we define this "freshness" to be a temporal bound beyond which the information carried by the message looses its relevance. Mostly, such freshness of messages can typically be guaranteed by splitting the streams and balancing them across multiple message queues, where the queues are statless operators. If an ordering of the meessages are required then it is up to the consumers of the messages to rebuild that ordering. Generally speaking, if the message originate from a multitude of sources and are split

across a variety of processors then finding a consistent, strict order is a very hard challenge. On the other hand, building data intensive cyber-physical systems using synchronous patterns, where the source of the data waits for a response to the request before sending the next request, is wasteful of the limited computing resources available in physical systems.

In this work, we present SMALOPS, which makes two primary contributions:

- A load balancing algorithm, focusing on the heavier flows (in terms of messages per unit time) thus reducing state tracking overhead, that is able to respond to dynamic workload changes.
- Such an algorithm affects a split in the streams that it operates on, thus we also discuss a mechanism to rebuild message ordering to support stateful stream processing.

SMALOPS optimizes for queuing delay while being cognizant of the desirability of stream state and fully aware of possible migrations of local application state stored in the downstream services. Finally, SMALOPS is also biased towards keeping larger streams pinned and migrates them only when absolutely required. As per our knowledge, SMALOPS is the first framework of its kind.

We implemented SMALOPS as a Go based framework on top of Apache Kafka v3.3.1 and deployed it in a Kubernetes cluster. Our evaluations show that SMALOPS can improve the 99^{th}%ile queuing latency for streams by as much as 73%. Though SMALOPS is built on top of Apache Kafka, it is not dependent on any Kafka specific features. We demonstrate the principles behind SMALOPS in a manner completely agnostic of the underlying system.

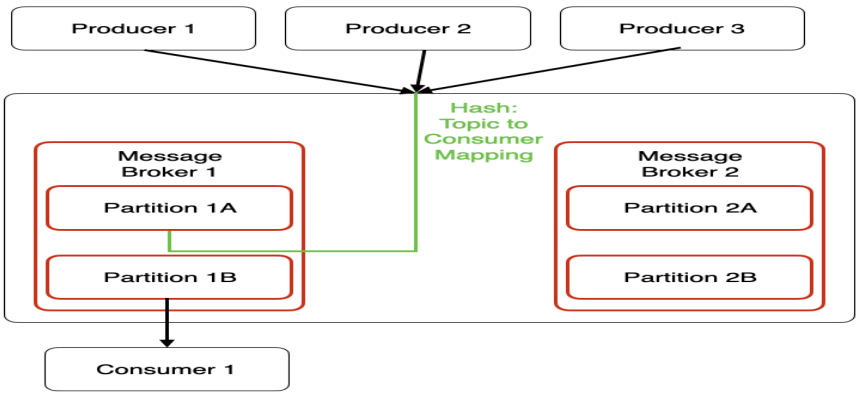


Figure 6.2: Message Queuing System

6.2 Background and Motivation

Asynchronous Services: Services can be integrated using asynchronous communication, typically using a message bus. When integrated in such a fashion, the component services can be truly autonomous and are not tightly coupled with any other service. In asynchronous services, any operation that changes the “state" of a service generates an “event". Events trigger a “message" being sent out from that service to a message queuing system. Other services in the application, can register an interest with, “subscribe to", the message queuing system, figure 6.2, to be notified when messages related to one or more event types, “topics", are received. Topics can be further subdivided into “streams", which are streams of related messages, “key"-value tuples, within a topic.

Asynchronous services operate solely by propagating event notifications (messages) through the system. Upstream services communicate with downstream ones by sending messages to a topic. When downstream services receive these messages, they update their state and may generate their own events and associated messages. This communication pattern makes no guarantees about the transmission latency of these messages.

Traditional systems that use such communication patterns are typically applications that can tolerate these chains of state changes to complete in arbitrary amounts of time.

A great advantage of asynchronous services is that these systems can scale much better compared to synchronous systems, since the services are not coupled and thus unencumbered by the demands of dedicated resources waiting for responses.

Message Queuing Systems are, at their core, producer-consumer systems. They accept incoming messages from a producer, store them in a queue, inform any consumers of newly arrived messages and track how much each consumer has processed amongst other possible features. Most messaging systems have a format that allows the producer to specify the “topic" the message belongs to. Consumers also specify the topics they are subscribing to. Messaging systems further subdivide all messages received on a topic into “partitions" (internal queues). A partition is a messaging system construct to assist the messaging system to scale. The algorithm for allocating messages to partitions vary by implementation, are generally rudimentary in practice and is the subject of investigation of this paper.

In order to understand this topic further, we will use two messaging system examples, Apache Kafka [22] and NSQ [23]. In case of Apache Kafka, the messages can contain a “key". When the messages do not contain a key, they are randomly allocated to a partition. When the messages contain a key, the partition allocated corresponds to a hash of the key. Thus messages with the same key on the same topic always end up in the same partition. This approach allows Apache Kafka to guarantee a total ordering of messages within each partition (a partial order in the system). NSQ, on the other hand, uses a simpler approach to partitioning, in that a copy of every message is

sent to all partitions and one of those copies are sent to a randomly selected consumer, completely ignoring any “key". Thus NSQ offloads the ordering problem to the consumer. While Kafka is overly restrictive in associating keys to partitions, NSQ is overly flexible allowing for a per message decision making. We aim to find the optimal balance between the approaches.

Load Balancing: We also have investigated this partitioning problem from a load balancing point of view. Both NSQ and Kafka take the approach that given enough incoming messages, random allocation and/or hashing would balance the number of messages on each partition and thus the load on each consumer. However, there are several examples in industry, [66] [25], where static load balancing policies resulting in skewed load assignments have led to complex system architectures. Related work, [8] [9], demonstrate that the accepted load balancing algorithms are inadequate in modern service deployment patterns. The random algorithm, amongst others, has been specifically investigated in [29]. In this paper, we take a look at the hashing algorithm as load balancing has become important in asynchronous services due to the emergence of a new class of applications.

Let us consider the earlier example of a traffic application that upon receiving signals from different sensors in geographical vicinity infers traffic conditions and relay traffic related recommendations to road users. This system has to be built as an asynchronous system because we do not want to 1) unnecessarily tie up the limited resources available in the sensors sending the signals and 2) as the data generated from an area increases, we would want to dynamically scale the application up. At the same time though, we also care about the queuing delay through the system. As discussed above, current messaging systems are incapable of supporting this new breed of applications. The primary reason is that both random and

hash-based allocation policies do not consider current and/or dynamically changing load conditions. Hence this work focused on load balancing in asynchronous systems.

Message Order and Consumer State: There are a couple of other factors that are typical to these applications that will need to be considered when discussing migrating load between partitions of messaging systems. Many of these new applications, an example is discussed above, depend on maintaining a strict order of messages. In this paper, we focus on the problem of migrating the streams but also discuss mechanisms that enable the consumers to maintain ordering of messages when streams are migrated.

In many of these cases, the consumer might also maintain a local state corresponding to each key (“stream") and we discuss at length our reasoning behind how we make our algorithm cognizant of the requirements of migrating that state.

6.3 System Design

The primary idea behind SMALOPS is detecting and managing the “heavier" streams (ones that are sending a relatively higher number of messages per unit time). The intuition is that the majority of the load can be balanced by balancing the heavier streams. We use a slightly modified version of the lossy counting algorithm [67] to detect these heavy streams. Next, we define a “size" for each partition proportional to the rate of messages incoming to it. SMALOPS then uses the power-of-two random choices algorithm, [11], to select a partition to which a new “hot key" is assigned. The rebalancing problem is posed as a linear integer programming problem, which is then solved using a heuristic. This heuristic is biased towards keeping the larger streams pinned to the same partitions as much as possible. The reasoning

behind this is that when a stream is migrated to a new partition and thus possibly to a new consumer, the messages need to be queued for a while to ensure all older messages belonging to the streams are processed first. Thus it makes sense to migrate many smaller streams than few larger streams. Smaller streams have more infrequent messages implying that they already can tolerate some delay relative to larger streams. Also, more streams, with lesser number of messages per stream, means lower average waiting time for each stream. Another reason for keeping larger streams pinned as long as possible, is that size of locally stored application state on consumers might be related to the “size" of the stream. Thus such a bias might help reduce state migration costs.

We propose our framework to consist of two parts - the **input** side and the **output** side. The input side consists of gateways that accept incoming messages and apply the load balancing algorithm and the output side consists of the consumers that operate on the stream of messages. In between the two, there is a Kafka cluster integrating the two sides.

The input side gateways are hierarchically arranged into two sets of gateways:

- This first set of gateways simply accept all messages and map each key in the message metadata to a gateway in the second layer.
- The gateway in the second layer then applies the load balancing algorithm.

The input side is split into two layers mainly to avoid having to implemented distributed algorithms to detect “heavy" streams. These algorithms, [68] [69], depend on being able to talk to different nodes in the layer, which would increase complexity by further necessitating a node discovery algo-

rithm.

Instead, our scheme combines messages belonging to the same key from different layer 1 gateways to the same layer 2 gateway. The layer 1 gateways could themselves use a load balancing scheme, for example based on weighted distributed hash tables [70]. However, we leave that for future research. Our current paper assumes that this can be done efficiently and focuses on reducing queuing delays in message queues.

While we focus on the input side in this paper, the consumers (the output side) are ultimately responsible for rebuilding the ordering guarantees using a novel protocol that requires them to synchronize to some extent or by stitching together the streams further downstream.

A major goal of this paper is to improve on the load balancing algorithm which assigns streams statically. SMALOPS introduces a finer-grained stream balancing algorithm that requires us to maintain some state about the stream. However, the design of SMALOPS aims to minimize the amount of state that needs to be stored.

Once the heavier streams are identified, a significant next step is to correctly identify the partitions that are candidate for migration. SMALOPS is biased towards migrating smaller streams, keeping the larger ones on the same partition as much as possible. Consumers block messages on a stream when it is migrated allowing older messages of the stream to be processed first. Let us consider a stream $S1$ whose messages are ordered as $S1M(1)$, $S1M(2)$, $S1M(3)$ and so on. Let us further consider that stream $S1$ is mapped to Kakfa partition $P1$ and the consumer $C1$ is listening on the partition $P1$. Now let us suppose that stream $S1$ gets migrated to another partition $P2$ and a different consumer $C2$ is listening on partition $P2$. Further, suppose the first message of $S1$ received by partition $P2$ after this migration is $S1M(n+1)$.

The header of this message will contain information that will tell partition $P2$ that $S1$ has been migrated from partition $P1$. On the other hand, when consumer $C1$, listening on partition $P1$, receives the message $S1M(n)$, it will contain header information that tells $C1$ that stream $S1$ has been migrated to partition $P2$ and thus consumer $C2$ will now receive newer messages of stream $S1$. At this point consumer $C1$ will send a synchronization message to consumer $C2$, letting it know that $C1$ has received the lass message of stream $S1$ that was sent to $C1$. $C2$ will queue the $S1M(n+1)$ message and process it only after receiving the synchronization message from $C1$. Towards this end, SMALOPS extends the stream metadata with the “Message Set" concept to enable downstream applications to rebuild stream state.

As the load of the streams across the system change, SMALOPS' stream rebalancing algorithm attempts to achieve better load assignment to partitions relative to rudimentary algorithms currently in use. The random load balancing algorithm can perfectly balance load given enough time but suffers from large skews in load assignment in the short term [29]. Furthermore, the random algorithm also splits the streams. The key-based-hashing load balancing policy prevents the streams from splitting, but it is a stateless algorithm that statically assigns keys to partitions without considering the actual load associated with each key. SMALOPS aspires to assign load equally to the partitions while providing a general approach to reconciling split streams..

The overall goal of SMALOPS is that, given a set of partitions, $P = \{p_1, p_2, ..., p_m\}$ and a set of keys assigned to these partitions $K_P = \{\{K_{p_1}\}, \{K_{p_2}\}, ..., \{K_{p_m}\}\}$, the aim of SMALOPS is to achieve:

$$R_{\{k_{p_i}\}} \approx R_{\{k_{p_j}\}} \forall i, j \in \{1, ..., M\} \wedge i \neq j \tag{6.1}$$

where $R_k = \sum_j P_k$, that is R_k is the sum of sizes of the hot keys mapped to the partition P_k and $R_{\{k_{p_j}\}}$ is the rate of incoming messages for all hot keys on partition P_j, the "size" of the partition.

6.3.1 Hot Key Analysis

We use the lossy counting algorithm to detect the heavier streams, without needing to maintain state for every stream. SMALOPS uses an error tolerenace of $\varepsilon = 0.001$ and the corresponding counting window width of $w = \frac{1}{\varepsilon}$. This means that the upper bound of the error is $err \leq \varepsilon N$, where N is the number of messages received so far. SMALOPS deviates from the lossy count algorithm by resetting N after every counting interval w. We do this with the assumption these systems are both long running and the actual keys are dynamic, thus we do not want to track any key that is not "hot" in the current interval. This allows us to save on the amount of state we need to store in the gateways. However, this is easily extensible to use a rolling window or maintain a small amount of history.

6.3.2 Stream Balancing

Stream balancing when there are no keys involved can simply use the power-of-two-random-choices, [11], algorithm to balance each message independently for better performance than random [29]. The challenge lies in balancing the streams where each stream is identified with a key. In order to do this, we define an absolute error function to determine how well the streams are balanced:

$$E_{LB} = \sum_j |R_{P_j} - R_{avg}| \tag{6.2}$$

where E_{LB} is the load balancing error, R_{P_j} is the rate of incoming messages into the P_j partition and R_{avg} is the average rate of incoming messages into the gateway across all partitions. The goal of SMALOPS is to migrate as few and as small streams as possible in order to minimize the error. In other words, we want the total number of messages arriving at any partition to be nearly equal at any point of time.

We decided to use equation 6.2 since it would allow us to use linear integer programming as a possible solution approach. However, linear integer programming solutions are notoriously hard, belonging to the NP hard class of problems. Thus we have used the heuristic described here in our solution.

The stream balancing algorithm in SMALOPS starts with dividing the partitions into two sets - the “underloaded set", where the “size" of all the partitions in the set is lesser than average of the system, and the “overloaded set", where the size of all partitions are greater than the system average. For each partition in the “overloaded set", we select a set of streams, $K_{\{M\}}$, for possible migration to one of the partitions in the “underloaded set". We define $K_{\{M\}} = \{k_j : k_j \in K_J \wedge R_{k_j} \leq R_{gateway} - R_{avg}\}$, where K_J is the set of all hot keys on that particular gateway, $R_{gateway}$ is the total rate of incoming messages (for hot keys) on the gateway and R_{k_j} is the rate of incoming messages for the key k_j. That is all streams selected for possible migration have a “size" that is lesser than the difference between the overall size of the partition (sum of sizes of all streams mapped to that partition) and the average of the partition sizes as seen by that gateway. When selecting streams from this set $K_{\{M\}}$, we first sort the keys according to descending size. Thus we migrate the largest stream we need to but no larger. This creates a bias in the system towards keeping the larger streams pinned to the same partition while still

reducing the number of streams that need to be migrated, reducing both stream balancing and state migration costs, if any, in the consumers.

6.3.2.1 Migration Target

When selecting a target partition, P_{target}, for migrating a stream identified by the key $k_m \in K_{\{M\}}$, we follow the heuristic:

$$min_{\delta}\delta = |(R_m + \sum R_{P_{target}}) - R_{avg}| \tag{6.3}$$

The partition, P_{target}, is chosen to minimize δ.

6.3.2.2 Stopping Condition

We stop migrating keys from the set $K_{\{M\}}$ when the following condition is satisfied:

$$\begin{gathered} \forall k_m \in K_{\{M\}} \wedge \forall target \neq source : \\ R_{source} - R_{target} < |(R_{target} + R_m) - (R_{source} - R_m)| \end{gathered} \tag{6.4}$$

where P_{source} is the partition from which streams are being migrated. Thus we stop migrating keys from P_{source} when migrating any partition from the “underloaded set" will increase the absolute error function.

When migration candidates are not found, that partition is ignored even if it has a higher than system average load. This can happen when the hot keys mapped to this partition are too large and possibly has a stiff state migration cost associated.

6.3.3 Stream Order

As mentioned earlier, many messaging system do not provide any message ordering guarantees within the stream which allows them to use random or round robin load balancing for all messages. For other messaging systems, migrating a stream from one partition to another violates the ordering guarantees. Thus SMALOPS requires a generic mechanism for guaranteeing message ordering. For this purpose, we propose two mechanisms in SMALOPS:

- A novel protocol that allows only the two consumers involved in the migration to synchronize for maintaining message ordering.
- A subdivision of a stream called a "message set" that allow applications downstream to reconstruct only the portions they need.

6.3.3.1 Message Sets

When talking about ordering, we are mainly interested in streams being totally ordered. We mark each stream with a message set metadata. This metadata is simply a numerical identifier that is incremented every time the stream is migrated to a different partition, figure 6.3.

Message sets can be forwarded to downstream applications without any processing in the consumers. This allows independent applications to rebuild the stream from the message sets without any processing required at the consumers. When the stream is migrated from a source to a destination partition, this information is also embedded in the message set header, figure 6.4.

When a partition is migrated from, say, partition 1 to partition 2, the message set number is incremented and the source/destination partition

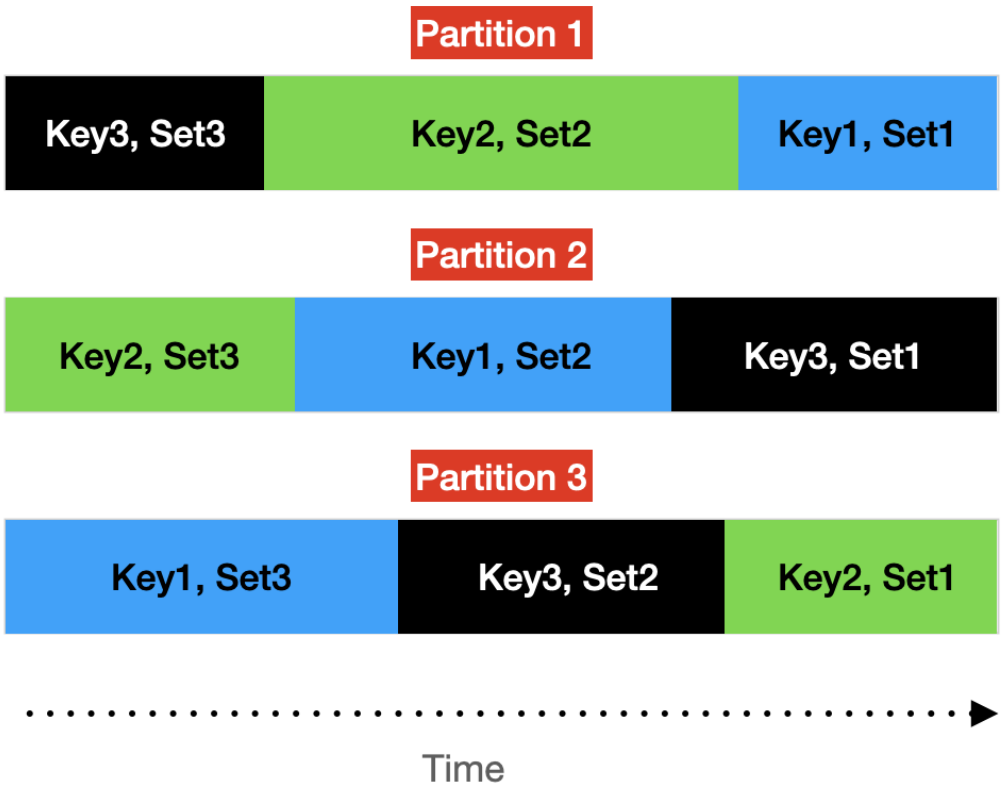


Figure 6.3: Message Sets

data is recorded in the header of the new message set that is being sent to partition 2. This information can be used further downstream to stitch the streams together. However, before the new message set is created and sent, a last message with the updated migration information is sent to current partition, and thus the current consumer (figure 6.5). Consumers can choose to use this information to trigger the consumer-side partition migration protocol or to pass it for the streams to be stitched together further downstream.

In figure 6.3, we see that the "Key2" stream migrates from partition 3 to partition 1 and partition 2. In figure 6.5, the last message of the "Key2" stream on partition 1 and the first message of the "Key2" stream on partition 2 have the exact same header. This indicates that the stream is migrating from partition 1 to partition 2.

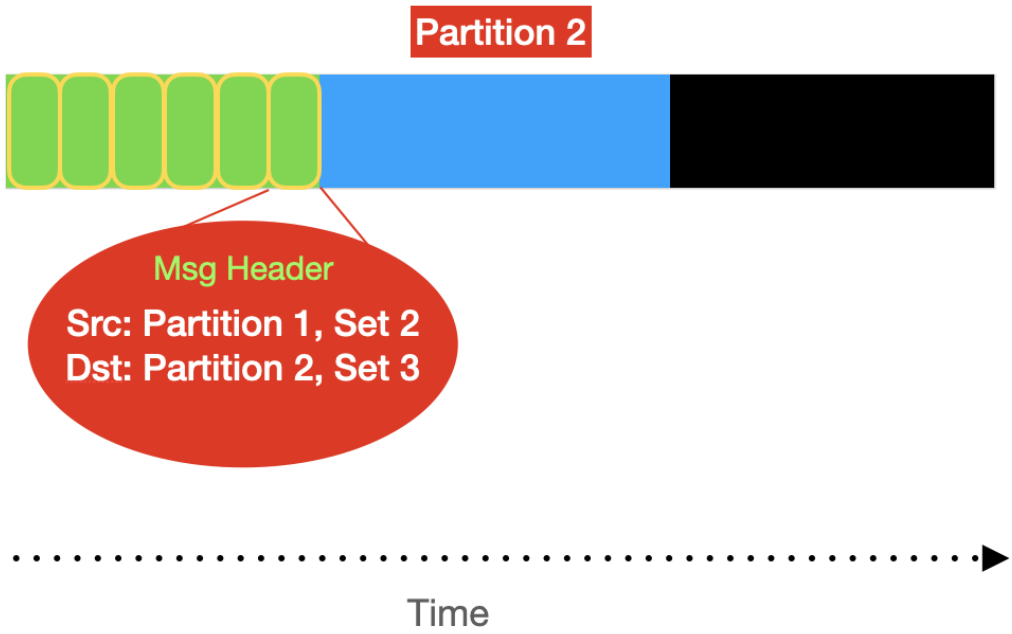


Figure 6.4: Message Set Header

6.3.3.2 Stream Ordering Protocol

This protocol works on the consumer side and describes a mechanism for two consumers to synchronize between themselves to maintain message ordering.

Figure 6.6 shows the timeline of the gateway and the two consumers, C1 and C2, involved in the migration. When the gateway decides to migrate key K1 from consumer C1 (partition P1) to consumer C2 (partition P2), it sends one last message to the consumer C1, denoted by **LK1C1** in the diagram. This message contains a new message set header, refer to figure 6.5, that tells consumer C1 that the key is going to be migrated to consumer C2. The next message, denoted by **FK1C2**, is then sent to consumer C2, which determines that this stream has been migrated from consumer C1 from the message set header. The new consumer C2, then queues any incoming message belonging to key K1 in some internal structure. Once consumer C1

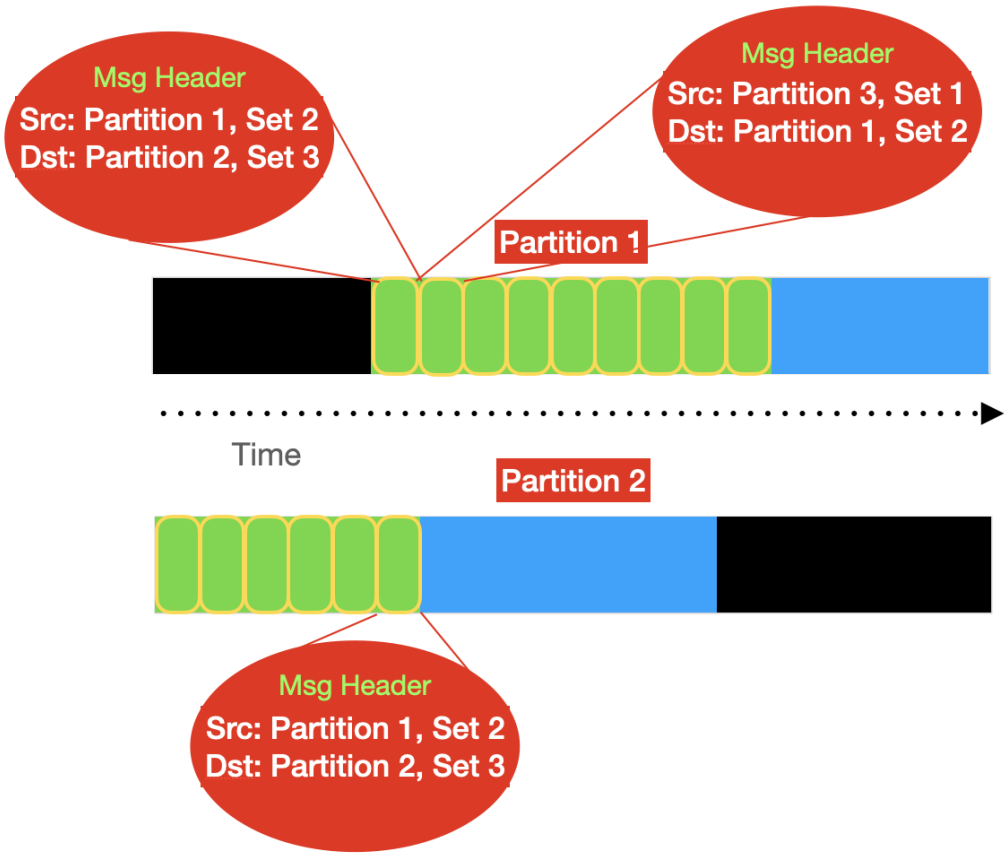


Figure 6.5: Message Set Migration

finishes processing the **LK1C1** message, it sends notification, denoted by **PK1** to consumer C2. Once C2 receives this notification, it can immediately start processing messages belonging to the key K1.

It is trivial to work out that in SMALOPS, the **FK1C2** message is guaranteed to be processed earlier on C2 than it would have been on C1. According to our stream balancing algorithm, described above, the conditions under which the key K1 gets migrated to the consumer C2 from the consumer C1, is when consumer C1 is more loaded than consumer C2. Thus the average size of the queue on C2 is guaranteed to be smaller than the size of the queue on C1.

In terms of queuing time on C2 for **FK1C2**, we have to consider two different cases:

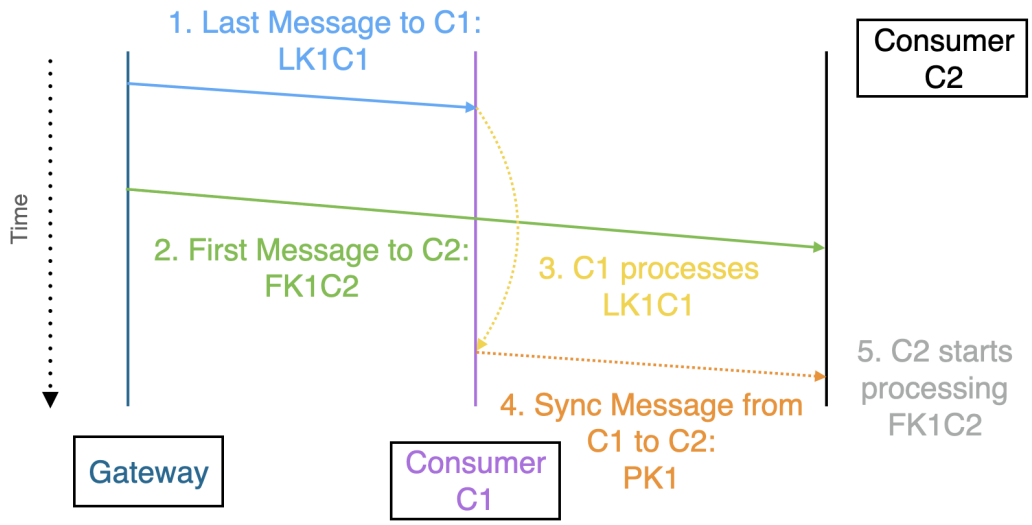


Figure 6.6: Consumer side protocol

- C2 receives **FK1C2** before C1 receives **LK1C1**.
- C2 receives **FK1C2** concurrently as C1 receives **LK1C1**.

In the second case, ignoring time required to send and receive **PK1**, maximum queuing time for **FK1C2** on C2 is $O(t_{msg})$, the average time required to process a message. In the first case, the maximum queuing time for **FK1C2** on C2 is $O(R_{C1} - R_{C2})$, the difference in load between the two partitions. However, since **FK1C2** arrives at the system only after **LK1C1**, the effective queuing time is still $O(t_{msg})$. Now it might be possible that we migrate K1 again from C2 to another partition while messages for K1 are still queued on C2. Our propensity of migrating smaller keys, amongst the hot keys, means there is some protection against too many messages queuing up on C2 in the first place as $1/R_{k_j}$ gets larger. For further protection against this issue, SMALOPS makes migration decisions at discrete intervals only. In choosing the interval for making migration decisions, we see that the maximum queuing time of **FK1C2** on C2 is $O(t_{msg})$ in all cases. As such SMALOPS uses a decision making interval defined as Ct_{msg} where C is a

constant. In our design, we have chosen C such that Ct_{msg} is orders of magnitudes greater than $O(t_{msg})$ for simplification ($t_{msg} = 10ms$ and decision making interval $= 10seconds$).

6.3.4 Stateful Consumers

SMALOPS also considers the fact that consumers may maintain local application state, “appstate", associated with each stream. Migrating a stream to a different partition probably migrates it to a different consumer as well, potentially triggering an appstate migration. In such cases, SMALOPS needs to be aware of the state migration costs. Let us consider an example. Let us assume that processing time for a message from any stream is t_{msg}. Let K_{P_1} be the set of all keys whose messages are sent to partition P_1 and thus to the consumer C_1. Also, let R_{K_i} be the incoming rate of key $K_i \in K_{P_1}$. Then the processing and queueing time for a message belonging to the stream identified by key K_1 can be approximated by:

$$t_{msg}(1 + (1 - \frac{R_{K_1}}{\sum_i R_{K_i}}))$$

where the last term is the waiting time given rate of incoming messages belonging to the non-K_1 keys.

Now let us consider if the stream identified by key K_1 is migrated to another partition P_2 and thus consumer C_2. Let K_{P_2} be the existing set of hot keys on partition P_2 and R_{K_j} be the corresponding rates of incoming messages. Thus the processing and queueing time for a message belonging to the stream identified by key K_1 on partition P_2 can be approximated by:

$$t_{msg}(1 + (1 - \frac{1}{\sum_j R_{K_j}}))$$

Finally, let the cost of migrating the appstate corresponding to key K_1 be t_{SM}. Then migrating the stream makes sense only if,

$$t_{msg}(1+(1-\frac{R_{K_1}}{\sum_i R_{K_i}})) > t_{SM} + t_{msg}(1+(1-\frac{1}{\sum_j R_{K_j}})) \tag{6.5}$$

There is a special case to be considered. If $t_{SM} = O(t_{msg})$, then then equation 6.5 reduces to $\frac{R_{K_1}}{\sum_i R_{K_i}} > \frac{1}{\sum_j R_{K_j}}$.

In order to handle local consumer state, the gateways need to track:

- **Rate** of incoming messages for all keys (on that gateway).
- Possibly also the **cost of state migration** and message processing cost.

This tracking would significantly bloat the state stored in the gateways. This **state bloat** can be completely avoided if the stream migrations are made at discrete time intervals greater than some threshold that ensures that all state migrations have completed before new migrations are started. Combining our rational for selecting $t_{threshold}$ from section 6.3.3.2 with our logic described here, we define this threshold as $t_{threshold} = Cmax(O(t_{SM}), O(t_{msg}))$, where C is a constant. This threshold also allows SMALOPS $t_{threshold}$ time to recalculate the next migrations.

6.3.5 Hierarchical Gateways

In SMALOPS, we arrange the gateways in a hierarchical architecture where the second layer gateways are arranged in a hash ring. The first level server hashes the key and sends it to the second layer. This means that messages in the same stream are always routed to the same second level gateways. This allows us to avoid the following complexities:

- Distributed hot key detection.
- Distributed state as each gateways would need to know how to route the hot keys.

The first level gateways route the messages using only a few metadata and the key. Potentially this can be offloaded to virtual network functions, like OpenNetVM [71], and programmable NICs.

6.3.6 State

SMALOPS gateways, independently, maintain a state in order to better load balance hot keys. This state involves tracking each hot key in the system and which partition it is mapped to. We sample the incoming streams, given a sampling threshold (0.5 in our experiments), and send the sample to the lossy counting algorithm. The algorithm works in discrete measuring windows (buckets). During each bucket the algorithm maintains a list of the keys seen during that bucket, the load associated with that key and the bucket during which the key was first added. The algorithm then checks if any of the keys are “heavy" according the pre-defined error and support thresholds (0.01 and 0.001 in our experiments). At the start of each bucket, we carry over a little history from the old bucket by continuing the keys that are still heavy while dropping ones that are not “heavy" anymore from the list. The algorithm also resets the number of messages seen to 0. These two approaches allow us to determine keys that are hot during the current bucket and to drop keys that are not hot anymore.

In a long running experiment where keys sending 100+ messages in any counting window were defined as “hot" and $|K|$ was defined to be 28M (million), we found that our custom power-law, open-loop generator actually

used 3.2M keys while sending a total of 28M messages. Out of these 3.2M keys, 14K (thousand) were detected as being “hot" over the course of the experiment. These 14K keys accounted for 20M+ of the 28M total messages sent by the generator. The SMALOPS gateway tracked no more than 1-1.5K keys in any of their counting windows. Thus we can reasonably expect SMALOPS to balance a significant portion, $\frac{\sum_{j=1}^{H} R_j T}{mc_p} \approx \frac{3}{4}$, of the load on the messaging system while only maintaining a very small state, tracking only $\sim$0.05% of the number of keys seen. Here H is the number of hot keys in the gateway, m is the number of partitions, R_j is the size of each of the hot keys, T is the weight threshold and c_p is the average load generated by the non-hot keys.

6.4 Implementation and Experimental Setup

Our experiments are based on testing the tail latencies involved in messaging systems delivering events to consumers. We use Kafka batch processing as our baseline performance. Since we use a batch processing baseline, there is a dependence on the time taken to process individual messages. For our experiments, we assume every message takes the same processing time. In this section we discuss some of the implementation details of SMALOPS and the test cases that we are most interested in.

6.4.0.1 Custom Open Loop Load Generator

For this experiment, we needed a generator that would send traffic consisting of several flows at a configurable rate. The generator also needed to follow power-law distributions for selecting the key that identified the flow. Unfortunately, we did not find a reliable, well-documented open-loop generator with those characteristics. As such, we developed our own open

loop load generator. We used the zipf distribution to generate load due to its prevalance in different workload like Twitter hashtags [72] and web caches [73]. We define the zipf distribution as per $k \in [0, num_{keys}] : P(k) \propto (v+k)^{-s}$, where $s > 1$ and $v \geq 1$. Our generator uses $s = 1.1$ and $v = 2.72$ [74].

6.4.1 Hierarchical Gateways

We arranged our gateways into a two-layer hierarchy. The first layer of gateways receive streams from our open loop generator. However, instead of building a hash ring, the first level gateway maintains a map of streams that have already been seen and mapped to a second level gateway. In case a new stream is received, it is assigned to a second level gateway using the round robin load balancing policy. Since we ensure that the gateways are never the bottlenecks, we assume that this approach is indistinguishable from a hash ring and other more advanced approaches for our purposes.

6.4.2 Second Level Gateway

We built the SMALOPS gateways from the ground up to enable us to implement our algorithms with ease. The gateway handles all incoming flows which allows it to track the hot flows and rebalance them according load on the partitions. Each gateway locally tracks all hot keys, its “size" and which partition that flow is mapped to. It also tracks all the partitions it has outstanding hot flows on and the current load on each of those partitions. SMALOPS gateways need some initial configuration to set up the system. The primary amongst them is the threshold that identifies a hot flow. Any flow with at least that number of messages in the counting window, also configurable, is considered as hot and actively tracked.

The SMALOPS gateway implements the following mechanisms:

- Detect hot keys using the lossy counting algorithm.
- Calculate the size of each flow given the hot flow identifier threshold.
- Track the total size assigned to each partition from this gateway.
- Decide which flows to migrate and which target partition is best candidate to accept the flow.
- Finally, attach a metadata span with each message that can be later used to identify time spent by each message within the messaging system.

6.4.3 A Side Note on the implementation of the Lossy Counting Algorithm

The lossy counting algorithm uses two parameters - support (s) and error tolerance (ε). “Support", s, is a measure of how frequently an item is seen while ε is the algorithm's error tolerance of how accurately support is measured ($\varepsilon \ll s$). In our experiments, we use the values $s = 0.01$ and $\varepsilon = 0.001$. So our system tracks all keys that appear more than $(s-\varepsilon)N$ times, where N is the number of messages seen so far.

We deviate from the lossy counting algorithm in that we reset N after every $1/\varepsilon$ messages. This is done so that SMALOPS can react quickly to changes in load. This also helps SMALOPS to avoid tracking keys that were hot in the past but not during the current window.

6.4.4 Control Plane

We also developed a simple control plane that uses the Kubernetes API to monitor live endpoints for the second level gateway service. The first

level gateways make REST API calls to the control plane pods, which run as a daemonSet in the Kubernetes cluster to populate their local service directories.

6.4.5 Consumers

We have developed consumers that calculate a number in a predetermined number of loops to mimic processing of messages. The consumers also complete the Jaeger spans that allows us to identify the transmission latency of each message.

6.5 Evaluation

6.5.1 Experimental Setup

We ran our experiments on cloudlab [52] servers. A Kubernetes cluster was created with four Intel Xeon servers, each with 20 cores and 196GB of memory. We then deployed our control plane that ran a pod on each of the servers. These pods form the service that is queried to get information about the backends of the gateway service running in the cluster.

We use an in-house power-law based open loop generator. We use the tool to send requests to the gateway for a fixed amount of time (5 minutes) where every request is part of a flow, identified by a key selected using the Zipf distribution. We also experimented with messages without any key metadata that could be routed on a per message basis to get a measure of what an optimal performance on a single gateway might look like.

The core motivation for designing the experimental setup was to track the queuing delay of individual events in Apache Kafka given a workload that follows a power law distribution.

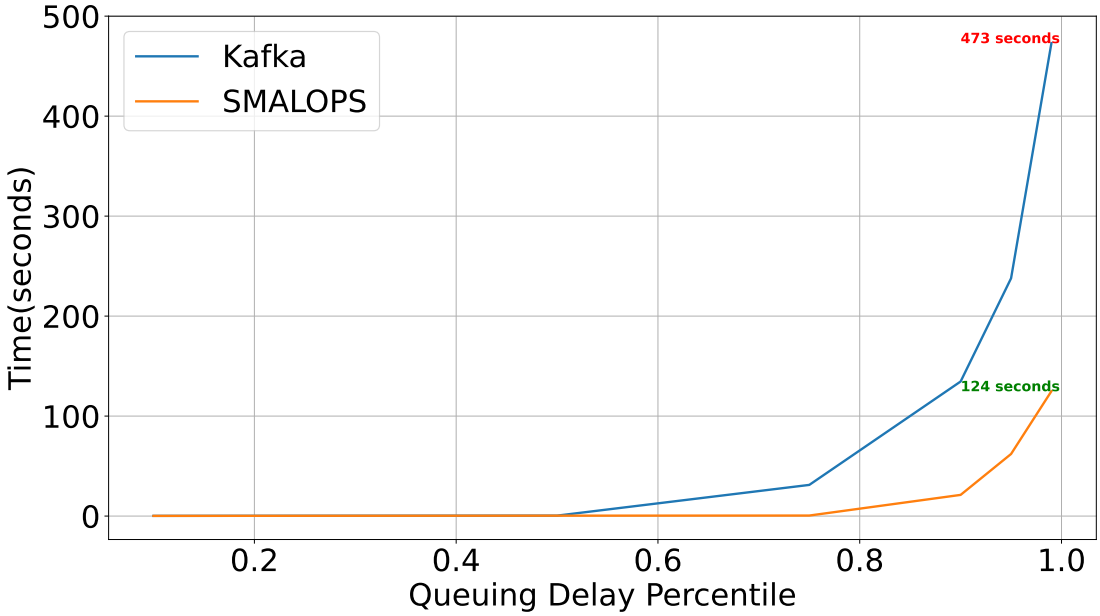


Figure 6.7: Queuing Delay Comparison

6.5.2 SMALOPS Overall Performance

In our first experiment, we set a processing time of 50 microseconds for each message on the consumers and compare the queuing delay distribution achieved through Kafka and SMALOPS respectively. Here we see that SMALOPS improves overall native Kafka performance, with the 99 %ile queuing delay improving by 73%, figure 6.7.

We repeated this experiment varying consumer processing times between 10 and 50 microseconds. The comparison between the 99%ile queuing delay can be seen in figure 6.8. We note that as the system starts getting overloaded, the 99%ile latency of Kafka keeps increasing while the improvements achieved by SMALOPS remain stable. The 99%ile improvement at 50 microseconds consumer processing time is 73% while the 99%ile queuing delay improvement at 20 microseconds consumer processing time is 57%. In our experiment, 50ms processing time/message was just before the system is overloaded (tail latencies become really long).

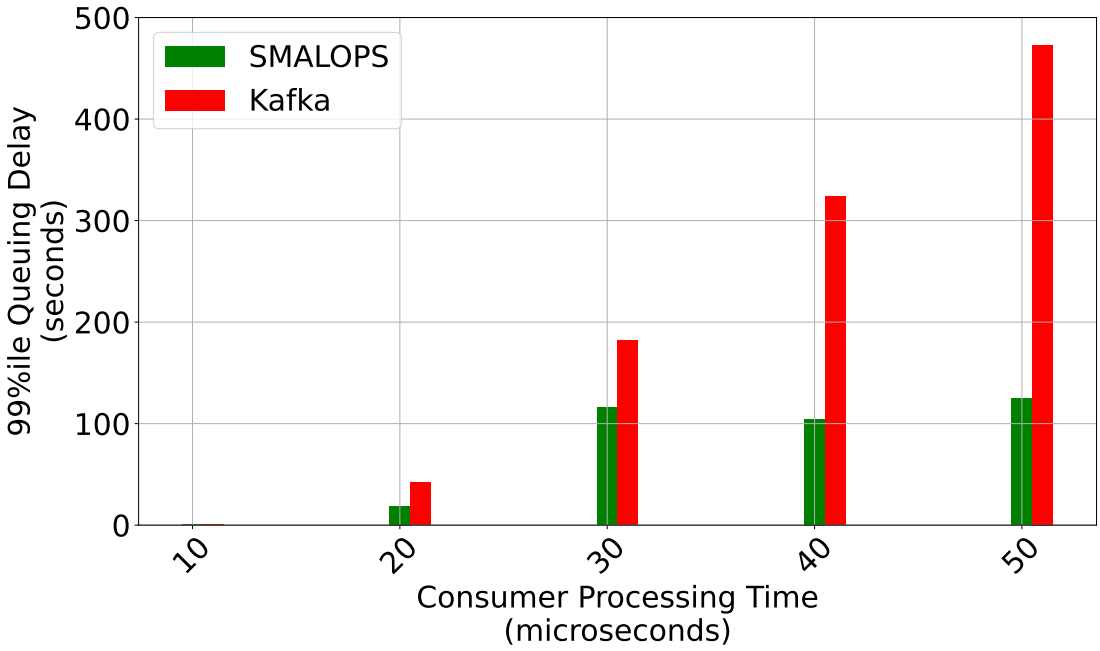
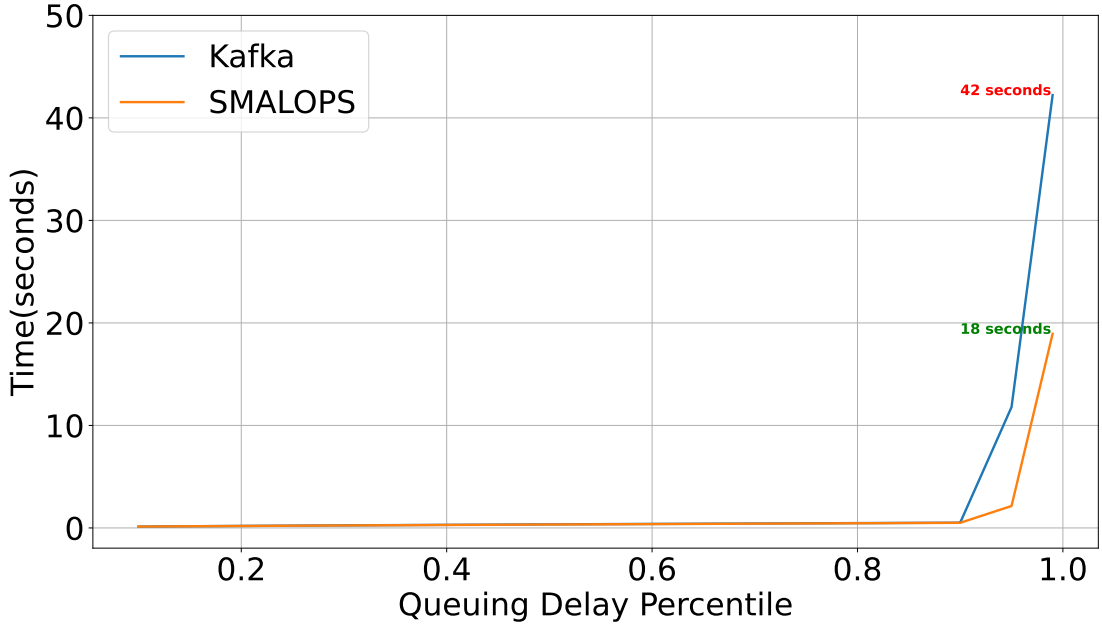


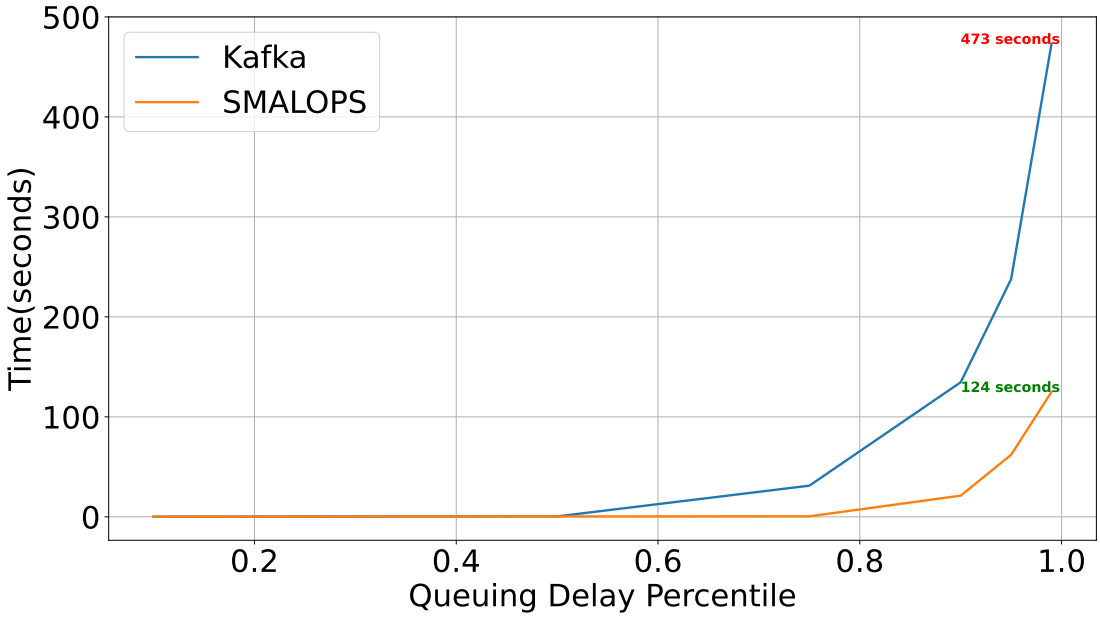
Figure 6.8: 99%ile Queuing Delay Comparison

Furthermore, we see that as the system starts getting overloaded, Kafka's performance start degrading earlier. For example, for a 20 microseconds consumer processing time, the performance of Kafka and SMALOPS are indistinguishable till 90%ile latency. However, for a 50 microseconds consumer processing time, Kafka starts degrading around the 50%ile mark while SMALOPS's performance remains nearly identical. This comparison between the performances of SMALOPS and Kafka can be seen in Figure 6.9.

In figure 6.10 we experiment with the threshold, as a percentage of total number of messages, to be used to classify a flow as "heavy". We see that when we classify flows that account for 10% or more of the messages received as "heavy", SMALOPS' tail latency drops sharply. We also see that further increasing the threshold has no significant impact on our results. Rather keeping this threshold as low as possible allows us to keep the number of keys being tracked lower.



(a) Queuing Delay Distribution Improvement at 20 μs



(b) Queuing Delay Distribution Improvement at 50 μs

Figure 6.9: Performance improvement with SMALOPS is realised earlier

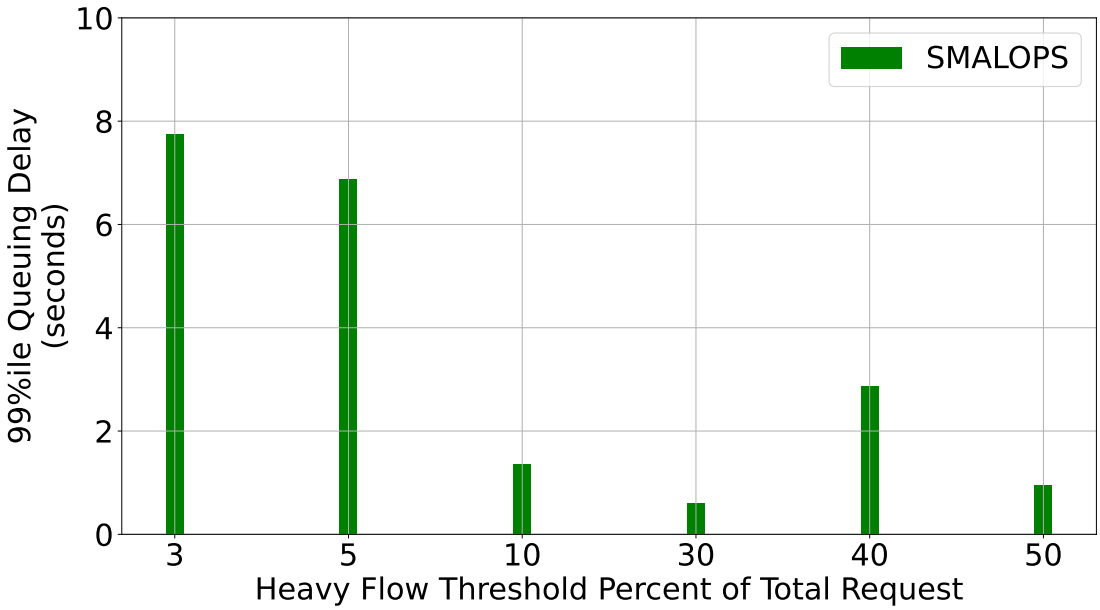


Figure 6.10: Impact of Threshold Definition on Latency

Finally, we ran SMALOPS against “dynamic load", where the load is generated by shifting the keys through our zipf based generator every minute. This ensures that:

- A few new streams are introduced to the system regularly.
- A few streams stop altogether.
- Load on remaining streams change.

Figure 6.11 shows that SMALOPS can significantly improve streaming system performance even in the face of these new challenges. Each message in this experiment takes 20ms to be processed.

6.6 Related Work

Load balancing is a well known problem that has been extensively studied for a long time. Advent of modern distributed systems has renewed interest

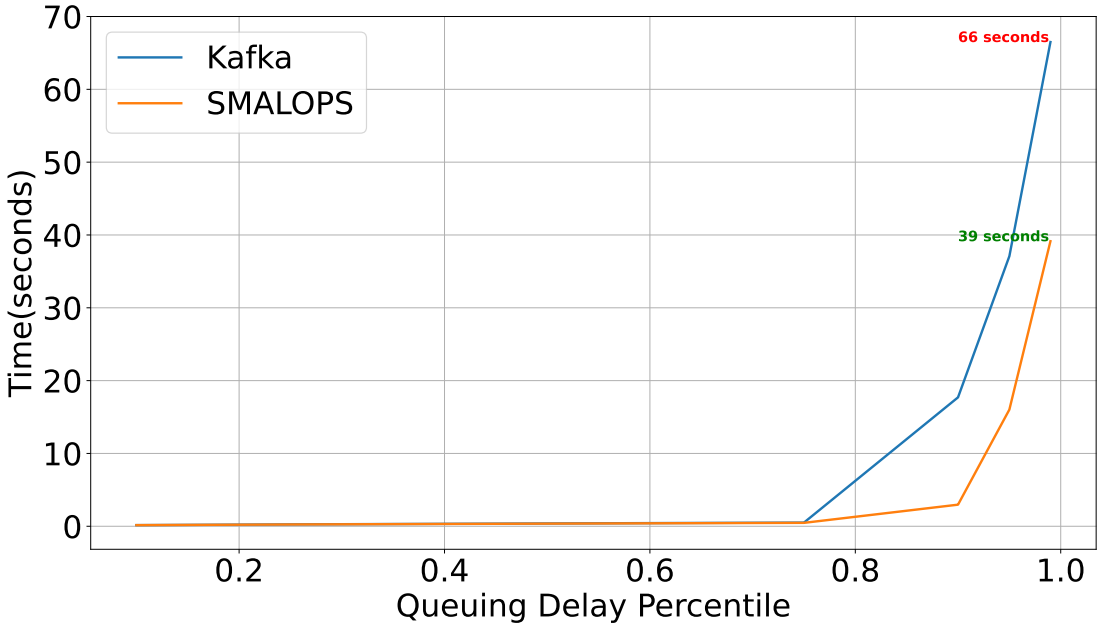


Figure 6.11: Kafka vs SMALOPS under dynamic load conditions

in the area. Load balancing in the topic based pub-sub systems like Apache Kafka have generally been in two forms:

- Balance the partitions themselves within the brokers [75].
- Assign topics to partitions as the topics enter the system [76], [37].
- Build overlay networks [77], [78] to connect relevant nodes directly.

Gedik et al, [13], have used the lossy algorithm to track heavier flows and map those to partition explicitly. Other flows are mapped using the consistent hash function. This work is probably the most similar to ours. They use three lossy counters over tumbling windows to emulate a sliding window whereas we only use a single lossy counter over strictly demarcated window.

Nasir et al, [14], has proposed **PKG** that uses power of two random choices to map each key to the least loaded partition selected by two different hash

functions. This results in every flow, heavy and otherwise, being in a split state that requires reconciliation.

Finally, Rivetti et al, [15], proposed **DKG** that learns the distribution of the keys before using a global mapping function to achieve near-optimal load assignment. They map non-heavy keys to “buckets", where the number of buckets is user-defined and larger than the number of partitions, using a random hash function from two separate hash function families. Their solution maps the heavier flows, identified by the space saving algorithm, explicitly to specific partitions.

SMALOPS extends existing research in fundamental ways:

- SMALOPS only tracks and load balances only the heavy hitters since with the zipfian distribution the majority of the system load come from a small number of heavier flows.
- SMALOPS focuses on dynamic workload distributions in two important ways:
  - SMALOPS accounts for the fact that different keys produce heavy streams at different times.
  - SMALOPS realizes that not every heavy hitter is active at the same time. This allows SMALOPS to significantly reduce the number of flows it is tracking.
- SMALOPS balances the gains from migrating streams against the cost of rebuilding stream state or migrating applications state.

6.7 Conclusions and Future Work

In this paper we propose a load balancing strategy for streaming systems that is able to deal with dynamic changes in the incoming streams. We argue that current streaming systems are ill-equipped to increasingly real-time requirements of modern applications. Current research assumes load on streaming systems to be static.

In contrast, SMALOPS significantly improves load imbalance in streaming systems. To the best of our knowledge, SMALOPS is the first system to consider dynamic input load and propose a generalized split state reconciliation. For our future work, we plan to expand on the generalized split state reconciliation.

We realized that the system, may be shifting around a lot of flows unnecessarily, "**load thrashing**", due to our definition of load imbalance, equation (6.2). This is because even an infinitesimal value for E_{LB} can trigger a flow redistribution phase. Hence we built a third variant of SMALOPS, that defines a new parameter, tol, as a load imbalance tolerance threshold and change equation (6.2) to

$$E_{LB} = \sum_j |R_{P_j} - R_{avg}| > tol \tag{6.6}$$

Our experiments show that in this case the performance of this variant fails to improve over the performance of SMALOPS. However, this may be due to small measuring window sizes and is a possible direction to explore in our future work.

Acknowledgements: This work was supported in part by NSF Grant CNS-1837382.

Chapter 7: Thesis Conclusion

7.1 Summary

In our work, we have explored two main themes regarding load balancing in distributed systems (microservice based). The first was to challenge an established perception that load balancing as a research topic has been thoroughly explored. The second is to show that intelligent load balancing can be a great tool to build partially synchronous networks that are required by a class of modern applications. The latter, however, also breaks certain guarantees regarding message ordering. This prompts our work to also discuss certain aspects of generic split state reconciliations of ordered message flows or streams.

We found that for server-client based applications, the norm is to adapt traditional load balancing algorithms with established good performance metrics to the microservices world. Our search found no study into the the efficacy of these adaptations in microservices. Most contemporary research, like [33], [32] etc, that we found tackled the problem of building efficient load balancing frameworks using established algorithms. However, in doing so we felt most of them made assumptions that should be researched themselves. In our work, we focused on two main issues that arise when applications transcend over to microservices architectures from the monolithic world:

- When using distributed client side load balancers, none of the load balancers are aware of the server queues anymore.
- In certain scenarios, the different servers have different service capacities.

In our research, we have tackled both these problems from a systems perspective by showing how simple feedback mechanisms along with proper handling of stale information can automatically account for both loss of a central source of truth and differing service capacities in the backend systems. Our techniques, not only help load balancers outperform contemporary benchmarks, it also demonstrates a strong need to further investigate load balancing in microservices architectures.

In case of asynchronous applications, only rudimentary algorithms like random, round robin or hashing are used in the industry. More often than not, the primary focus of these systems are persistence and/or ordering of the messages. Though there have been research into more advanced load balancing algorithms, they have mostly assumed a static load profile for the lifetime of the system. Our research demonstrates that it is possible to build systems that successfully track and load balance high density streams without tying up an inordinate amount of system resources and achieve significant gains in message delivery times. However, these gains come at the cost of splitting the streams into different partitions. Thus our research also discusses schemes that would enable reconciliation of these streams.

7.2 Future Work

We believe that there is much work that need to be done to establish the exact characteristics of load balancing in these modern systems. We believe the following areas to most likely have most impact on **distributed client side load balancing** research:

- mechanisms that effectively share state between client-side load balancers enabling them to make better decisions.

- mechanisms to characterize and track workloads with judicious use of system resources.
- explore efficient means of reconciling split state in message streams that result from migrating streams to remove hot spots.

Bibliography

[1] Production-Grade Container Orchestration. Kubernetes. [Online]. Available: https://kubernetes.io/

[2] Home - Knative. [Online]. Available: https://knative.dev/docs/

[3] Envoy Proxy - Home. [Online]. Available: https://www.envoyproxy.io/

[4] Istio. Istio. [Online]. Available: https://istio.io/latest/

[5] The world's lightest, fastest service mesh. | Linkerd. [Online]. Available: https://linkerd.io/

[6] Top 10 Companies Using Cloud and Why | CustomerThink. [Online]. Available: https://customerthink.com/top-10-companies-using-cloud-and-why/

[7] I. Cho, A. Saeed, J. Fried, S. J. Park, M. Alizadeh, and A. Belay, "Overload control for {µs-scale} {RPCs} with breakwater," pp. 299–314. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/cho

[8] V. Mittal, S. Qi, R. Bhattacharya, X. Lyu, J. Li, S. G. Kulkarni, D. Li, J. Hwang, K. K. Ramakrishnan, and T. Wood, "Mu: An efficient, fair and responsive serverless framework for resource-constrained edge clouds," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '21. Association for Computing Machinery, pp. 168–181. [Online]. Available: https://doi.org/10.1145/3472883.3487014

[9] R. Bhattacharya and T. Wood, "BLOC: Balancing Load with Overload Control In the Microservices Architecture," in *2022 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pp. 91–100.

[10] ——, "Load Balancing and Generalized Split State Reconciliation in Stream Processing Systems," vol. Under Review @ IEEE ACSOS 2024.

[11] M. Mitzenmacher, "The power of two choices in randomized load balancing," vol. 12, no. 10, pp. 1094–1104.

[12] R. Bhattacharya, Y. Gao, and T. Wood, "Dynamically Balancing Load with Overload Control for Microservices," vol. Accepted @ ACM Transactions on Autonomous and Adaptive Systems.

[13] B. Gedik, "Partitioning functions for stateful data parallelism in stream processing," vol. 23, no. 4, pp. 517–539. [Online]. Available: https://dl.acm.org/doi/10.1007/s00778-013-0335-9

[14] M. A. U. Nasir, G. De Francisci Morales, D. García-Soriano, N. Kourtellis, and M. Serafini, "The power of both choices: Practical load balancing for distributed stream processing engines," in *2015 IEEE 31st International Conference on Data Engineering*, pp. 137–148.

[15] N. Rivetti, L. Querzoni, E. Anceaume, Y. Busnel, and B. Sericola, "Efficient key grouping for near-optimal load balancing in stream processing systems," in *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '15. Association for Computing Machinery, pp. 80–91. [Online]. Available: https://dl.acm.org/doi/10.1145/2675743.2771827

[16] J. Watada, A. Roy, R. Kadikar, H. Pham, and B. Xu, "Emerging Trends, Techniques and Open Issues of Containerization: A Review," vol. 7, pp. 152 443–152 472.

[17] Docker: Accelerated Container Application Development. [Online]. Available: https://www.docker.com/

[18] Linux.org. [Online]. Available: https://www.linux.org/

[19] Network_namespaces(7) - Linux manual page. [Online]. Available: https://man7.org/linux/man-pages/man7/network_namespaces.7.html

[20] Better Load Balancing: Real-Time Dynamic Subsetting. Uber Blog. [Online]. Available: https://www.uber.com/blog/better-load-balancing-real-time-dynamic-subsetting/

[21] W. Vogels, "Eventually consistent," vol. 52, no. 1, pp. 40–44. [Online]. Available: https://dl.acm.org/doi/10.1145/1435417.1435432

[22] Apache Kafka. Apache Kafka. [Online]. Available: https://kafka.apache.org/

[23] NSQ Docs 1.2.1 - A realtime distributed messaging platform. [Online]. Available: https://nsq.io/

[24] Effective Strategies for Kafka Topic Partitioning | New Relic. [Online]. Available: https://newrelic.com/blog/best-practices/effective-strategies-kafka-topic-partitioning

[25] Building an Adaptive, Multi-Tenant Stream Bus with Kafka and Golang | by Xinyu Liu | Lyft Engineering. [Online]. Available: https://eng.lyft.com/building-an-adaptive-multi-tenant-stream-bus-with-kafka-and-golang-5f1410bf2b40

[26] Messaging that just works — RabbitMQ. [Online]. Available: https://www.rabbitmq.com/

[27] RabbitMQ tutorial - Work Queues — RabbitMQ. [Online]. Available: https://www.rabbitmq.com/tutorials/tutorial-two-go.html

[28] K. Gilly, C. Juiz, and R. Puigjaner, "An up-to-date survey in web load balancing," vol. 14, no. 2, pp. 105–131. [Online]. Available: https://doi.org/10.1007/s11280-010-0101-5

[29] V. Gupta, M. Harchol Balter, K. Sigman, and W. Whitt, "Analysis of join-the-shortest-queue routing for web server farms," vol. 64, no. 9, pp. 1062–1081. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0166531607000624

[30] Examining Load Balancing Algorithms with Envoy | by Tony Allen | Envoy Proxy. [Online]. Available: https://blog.envoyproxy.io/examining-load-balancing-algorithms-with-envoy-1be643ea121c

[31] P. Kumar and R. Kumar, "Issues and Challenges of Load Balancing Techniques in Cloud Computing: A Survey," vol. 51, no. 6, pp. 120:1–120:35. [Online]. Available: https://doi.org/10.1145/3281010

[32] A. Gandhi, X. Zhang, and N. Mittal, "HALO: Heterogeneity-Aware Load Balancing," in *2015 IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pp. 242–251.

[33] Cheetah: A High-Speed Programmable Load-Balancer Framework With Guaranteed Per-Connection-Consistency | IEEE Journals & Magazine | IEEE Xplore. [Online]. Available: https://ieeexplore.ieee.org/document/9552525

[34] N. T. Blog. Rethinking Netflix's Edge Load Balancing. Medium. [Online]. Available: https://netflixtechblog.com/netflix-edge-load-balancing-695308b5548c

[35] S. T and S. N. K. A study on Modern Messaging Systems-Kafka, RabbitMQ and NATS Streaming. [Online]. Available: http://arxiv.org/abs/1912.03715

[36] A. K. Y. Cheung and H.-A. Jacobsen, "Load Balancing Content-Based Publish/Subscribe Systems," vol. 28, no. 4, pp. 9:1–9:55. [Online]. Available: https://doi.org/10.1145/1880018.1880020

[37] D. Dedousis, N. Zacheilas, and V. Kalogeraki, "On the Fly Load Balancing to Address Hot Topics in Topic-Based Pub/Sub Systems," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pp. 76–86.

[38] H. Wu, Z. Shang, and K. Wolter, "Performance Prediction for the Apache Kafka Messaging System," in *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pp. 154–161.

[39] D. Landau, X. Andrade, and J. G. Barbosa. Kafka Consumer Group Autoscaler. [Online]. Available: http://arxiv.org/abs/2206.11170

[40] M. Shahrad, R. Fonseca, Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider," pp. 205–218. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/shahrad

[41] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella, "On Multi-Access Edge Computing: A Survey of the Emerging 5G Network Edge Cloud Architecture and Orchestration," vol. 19, no. 3, pp. 1657–1681. [Online]. Available: https://doi.org/10.1109/COMST.2017.2705720

[42] J. Schleier-Smith, V. Sreekanti, A. Khandelwal, J. Carreira, N. J. Yadwadkar, R. A. Popa, J. E. Gonzalez, I. Stoica, and D. A. Patterson, "What serverless computing is and should become: The next phase of cloud computing," vol. 64, no. 5, pp. 76–84. [Online]. Available: https://dl.acm.org/doi/10.1145/3406011

[43] G. McGrath and P. R. Brenner, "Serverless Computing: Design, Implementation, and Performance," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pp. 405–410. [Online]. Available: https://ieeexplore.ieee.org/document/7979855

[44] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '18. USENIX Association, pp. 133–145.

[45] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless Computing: An Investigation of Factors Influencing Microservice Performance," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 159–169. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8360324

[46] How does language, memory and package size affect cold starts of AWS Lambda? [Online]. Available: https://www.pluralsight.com/resources/blog/cloud/does-coding-language-memory-or-package-size-affect-cold-starts-of-aws-lambda

[47] S. K. Mohanty, G. Premsankar, and p. u. family=Francesco, given=Mario, "An Evaluation of Open Source Serverless Computing Frameworks," in *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 115–120. [Online]. Available: https://ieeexplore.ieee.org/document/8591002

[48] J. Li, S. G. Kulkarni, K. K. Ramakrishnan, and D. Li, "Understanding Open Source Serverless Platforms: Design Considerations and Performance," in *Proceedings of the 5th International Workshop on Serverless Computing*, ser. WOSC '19. Association for Computing Machinery, pp. 37–42. [Online]. Available: https://dl.acm.org/doi/10.1145/3366623.3368139

[49] A. Palade, A. Kazmi, and S. Clarke, "An Evaluation of Open Source Serverless Computing Frameworks Support at the Edge," in *2019 IEEE World Congress on Services (SERVICES)*, vol. 2642-939X, pp. 206–211. [Online]. Available: https://ieeexplore.ieee.org/document/8817155

[50] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, ser. MCC '12. Association for Computing Machinery, pp. 13–16. [Online]. Available: https://dl.acm.org/doi/10.1145/2342509.2342513

[51] M. Satyanarayanan, Z. Chen, K. Ha, W. Hu, W. Richter, and P. Pillai, "Cloudlets: At the leading edge of mobile-cloud convergence," in *6th International Conference on Mobile Computing, Applications and Services*, pp. 1–9. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7026272

[52] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The Design and Operation of {CloudLab}," pp. 1–14. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/duplyakin

[53] "BLOCProxy Performance," MSrvComm. [Online]. Available: https://github.com/MSrvComm/Experiments

[54] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19.

Association for Computing Machinery, pp. 3–18. [Online]. Available: https://dl.acm.org/doi/10.1145/3297858.3304013

[55] Amazon Web Services, "AWS re:Invent 2015: A Day in the Life of a Netflix Engineer (DVO203)," Oct. 2015. [Online]. Available: https://www.youtube.com/watch?v=-mL3zT1iIKw

[56] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph, "Understanding TCP incast throughput collapse in datacenter networks," in *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*, ser. WREN '09. Association for Computing Machinery, pp. 73–82. [Online]. Available: https://dl.acm.org/doi/10.1145/1592681.1592693

[57] P. Berenbrink, A. Czumaj, and A. Steger, "Balanced Allocations: The Heavily Loaded Case."

[58] M. Raab and A. Steger, ""Balls into Bins" — A Simple and Tight Analysis," in *Randomization and Approximation Techniques in Computer Science*, M. Luby, J. D. P. Rolim, and M. Serna, Eds. Springer Berlin Heidelberg, vol. 1518, pp. 159–170. [Online]. Available: http://link.springer.com/10.1007/3-540-49543-6_13

[59] J. Dogan, "hey - open loop load generator," original-date: 2016-09-02T10:24:09Z. [Online]. Available: https://github.com/rakyll/hey

[60] Custom loadtest: Open loop poisson load generator. [Online]. Available: https://github.com/lyuxiaosu/loadtest

[61] M. Autili, A. Perucci, and L. De Lauretis, "A hybrid approach to microservices load balancing," *Microservices: Science and Engineering*, pp. 249–269, 2020.

[62] H. Adkins, B. Beyer, P. Blankinship, P. Lewandowski, A. Oprea, and A. Stubblefield, "Building secure and reliable systems," p. 557.

[63] J. C. Mogul and K. K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel," vol. 15, no. 3, pp. 217–252. [Online]. Available: https://doi.org/10.1145/263326.263335

[64] T. P. Raptis and A. Passarella, "A Survey on Networked Data Streaming With Apache Kafka," vol. 11, pp. 85 333–85 350. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10213406

[65] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, "Benchmarking Distributed Stream Data Processing Systems," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pp. 1507–1518. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8509390

[66] B. Feldmann. Solving my weird Kafka Rebalancing Problems. bakdata. [Online]. Available: https://medium.com/bakdata/solving-my-weird-kafka-rebalancing-problems-c05e99535435

[67] G. S. Manku and R. Motwani, "Approximate frequency counts over data streams," vol. 5, no. 12, p. 1699. [Online]. Available: https://doi.org/10.14778/2367502.2367508

[68] H. Dai, M. Li, A. X. Liu, J. Zheng, and G. Chen, "Finding Persistent Items in Distributed Datasets," vol. 28, no. 1, pp. 1–14. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8935435

[69] A. Manjhi, V. Shkapenyuk, K. Dhamdhere, and C. Olston, "Finding (recently) frequent items in distributed data streams," in *21st International Conference on Data Engineering (ICDE'05)*, pp. 767–778. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/1410191

[70] R. Lösch, J. Schmidt, and N. G. Felde, "Weighted Load Balancing in Distributed Hash Tables," in *Proceedings of the 21st International Conference on Information Integration and Web-based Applications & Services*, ser. iiWAS2019. Association for Computing Machinery, pp. 473–482. [Online]. Available: https://doi.org/10.1145/3366030.3366069

[71] "openNetVM," sdnfv. [Online]. Available: https://github.com/sdnfv/openNetVM

[72] J. A. Pérez Melián, J. A. Conejero, and C. Ferri Ramírez, "Zipf's and Benford's laws in Twitter hashtags," in *Proceedings of the Student Research Workshop at the 15th Conference of the European Chapter of the Association for Computational Linguistics*, F. Kunneman, U. Iñurrieta, J. J. Camilleri, and M. C. Ardanuy, Eds. Association for Computational Linguistics, pp. 84–93. [Online]. Available: https://aclanthology.org/E17-4009

[73] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and Zipf-like distributions: Evidence and implications," in *IEEE INFOCOM '99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future Is Now (Cat. No.99CH36320)*, vol. 1, pp. 126–134 vol.1. [Online]. Available: https://ieeexplore.ieee.org/document/749260

[74] S. T. Piantadosi, "Zipf's word frequency law in natural language: A critical review and future directions," vol. 21, no. 5, pp. 1112–1130. [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4176592/

[75] D. . Burato, “Load balancing and fault early detection for Apache Kafka clusters.” [Online]. Available: http://dspace.unive.it/handle/10579/15159

[76] T. P. Raptis and A. Passarella, “On Efficiently Partitioning a Topic in Apache Kafka,” in *2022 International Conference on Computer, Information and Telecommunication Systems (CITS)*, pp. 1–8. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9832981

[77] C. Chen, H.-A. Jacobsen, and R. Vitenberg, “Algorithms Based on Divide and Conquer for Topic-Based Publish/Subscribe Overlay Design,” vol. 24, no. 1, pp. 422–436. [Online]. Available: https://ieeexplore.ieee.org/document/6971250

[78] V. Turau and G. Siegemund, “Scalable Routing for Topic-Based Publish/Subscribe Systems Under Fluctuations,” in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pp. 1608–1617. [Online]. Available: https://ieeexplore.ieee.org/document/7980098