TECHNISCHE UNIVERSITÄT BERLIN
FAKULTÄT FÜR ELEKTROTECHNIK UND INFORMATIK
LEHRSTUHL FÜR INTELLIGENTE NETZE
UND MANAGEMENT VERTEILTER SYSTEME

On predictable performance for distributed systems

vorgelegt von
Lalith Suresh Puthalath

Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades
DOKTOR DER INGENIEURWISSENSCHAFTEN (DR.-ING.)
genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr. Volker Markl, Technische Universität Berlin
Gutachterin:	Prof. Anja Feldmann, Ph. D., Technische Universität Berlin
Gutachter:	Prof. Dr. Marco Canini, Université Catholique de Louvain
Gutachter:	Prof. Dr. Jon Crowcroft, University of Cambridge
Gutachter:	Prof. Dr. Willy Zwaenepoel, École Polytechnique Fédérale de Lausanne

Tag der wissenschaftlichen Aussprache: 27. Juni, 2016

Berlin 2016
D 83

Abstract

Today, we increasingly rely on a plethora of interactive online services for social networking, e-commerce, video and audio streaming, email, and web search. These services need to deal with billions of visits per day while handling unprecedented volumes of data. At the same time, these services need to deliver fluid response times to their users. Failure to do so impacts the quality of experience for users and directly leads to revenue losses as well as increases in operational costs.

Furthermore, the massive compute and storage demands of these large online services have necessitated a shift to the cloud. These services run inside large data-centers, across hundreds to thousands of heterogeneous servers, leading to a paradigm called warehouse-scale computing (WSCs). A complex ecosystem of fault-tolerant distributed systems back these services, each of which run on clusters of hundreds to many thousands of servers. These distributed systems include data-analytics stacks (*e.g.*, Hadoop), distributed storage systems (*e.g.*, Cassandra), cluster schedulers (*e.g.*, Borg), as well as web-applications architected as micro-services (*e.g.*, Netflix). These systems present operational challenges that occur due to several dynamic conditions, such as complex inter-server request-response patterns, server-side performance fluctuations, skews, and hot-spots in data access patterns, as well as multi-tenancy. Several of these dynamic conditions manifest at very small timescales, making operator intervention and tuning an infeasible means of addressing them. As a consequence, end-to-end system performance becomes notoriously difficult to predict and guarantee. This necessitates robust mechanisms by which distributed systems can adapt to these dynamic conditions that occur at very short timescales.

To address this challenge, the goal of this dissertation is to develop adaptive mechanisms to achieve performance predictability for distributed systems. We focus our attention on two important classes of distributed systems that permeate the WSC landscape. The first is that of low-latency distributed storage-systems, which are often in the critical path of online services today. Storage servers in these settings experience performance fluctuations due to a multitude of reasons, and this significantly harms application performance. We demonstrate the need for adaptive replica selection, wherein storage clients route requests to replica servers in response to performance heterogeneity among different servers. We then present the design and implementation of the C3 system, which embodies a novel adaptive replica-selection mechanism. In C3, clients, with minimal assistance from servers, adaptively prefer faster replicas while avoiding degenerate behaviors such as load oscillations. In experiments conducted on Amazon EC2, in production settings at Spotify and SoundCloud, as well as through simulations, C3 significantly reduces tail latencies (up to 3x at the 99.9th percentile) and improves system throughput (up to 50%). The second class of systems we focus on are service-oriented architectures (SOA) and micro-services, an increasingly popular architectural approach to building fault-tolerant online services. We present how complex interactions between different workloads can lead to significantly degraded system performance, potentially leading to widespread application outages. We discuss the challenges in meeting end-to-end performance objectives of interest in these settings, such as efficient and fair resource utilization, meeting deadlines, as well as reducing latencies. We present the design and implementation of Cicero, a system that embodies adaptive, distributed algorithms to enforce end-to-end resource management policies in SOAs and micro-services. In system evaluations wherein we emulated a production SOA, we demonstrated how Cicero can be used to achieve performance isolation among tenants, avoid cascading failures, and meet a high fraction of end-to-end deadlines.

Zusammenfassung

Wir verlassen uns heutzutage zunehmend auf eine Fülle von interaktiven Onlinediensten wie beispielsweise Social Networking, E-Commerce, Video und Audio-Streaming, E-Mail und Web-Suche. Diese Dienste müssen täglich oft mehrere Milliarden Anfragen beantworten und verarbeiten. Zur gleichen Zeit müssen die Dienste schnellstmögliche Antwortzeiten für die Endnutzer liefern. Geschieht dies nicht, so leidet die Quality of Experience für die Endbenutzer und führt unweigerlich zu Einnahmeverlusten sowie gestiegenen Betriebskosten.

Weiterhin haben die massiven Berechungs- und Speicheranforderungen dieser großen Onlinedienste eine Verlagerung eben dieser Dienste in die Cloud geführt. Die Dienste laufen in großen Rechenzentren auf hunderten bis tausenden heterogenen Servern und haben somit zu dem Paradigmenwechsel hin zum Warehouse-Scale Computing (WSC) geführt. Die Dienste bauen weiterhin auf einem komplexen Ökosystem von fehlertoleranten verteilten Systemen auf, wobei jeder dieser Dinste wiederum auf einem Cluster von mehreren hundert oder tausend Servern ausgeführt wird. Zu diesen verteilten Systemen gehören Systeme zur Datenanalyse (z.B. Hadoop), verteilten Speicherlösungen (z.B. Cassandra), Cluster Schedulern (z.B. Borg) sowie Web-Anwendungen, welche auf Mikrodiensten aufbauen (z.B. Netflix). Aufgrund verschiedener dynamischer Aspekte, wie z.B. komplexen Anfrage-Antwort Mustern zwischen Servern, serverseitigen Performanzfluktuationen und variierenden Anfrage-Verteilungen, bergen diese Systeme viele Herausforderungen im Betrieb. Viele dieser dynamischen Aspekte manifestieren sich in kurzen Zeiträumen, so dass für die Betreiber keim eine Möglichkeit des Einreifens oder der der Feinjustierung gibt. Als Konsequenz wird die Vorhersage und die Garantie der Performanz von Ende-zu-Ende Systemen außerordentlich schwierig. Somit bedarf es robuster Mechanismen, damit sich verteilte Systeme an die dynamischen Aspekte des Betriebs anpassen können.

Um obigen Herausforderungen zu begegnen, ist es das Ziel dieser Dissertation adaptive Mechanismen zu entwickeln, mit welchen die Vorhersage der Performanz in verteilten Systemen ermöglicht wird. Konkret betrachten wir zwei wichtige Klassen von verteilten Systemen, welche durchgängig in der WSC-Landschaft anzutreffen sind. Als ersten Punkt untersuchen wir verteilte Speichersysteme mit geringen Antwortzeiten, welche für viele heutigen Onlinedienste eine kritische Komponente darstellen. Die Performanz der Server zum Speichern von Daten unterliegt hierbei aufgrund vieler Gründe oft Fluktuationen, welche die Performanz der Anwendung verschlechtert. So demonstrieren wir den Bedarf eines adaptiven Replika-Auswahl Algorithmus, in welchem die Clients des Speicherdienstes Anfragen an verschiedene Replikas in Abhängigkeit der jeweiligen Performanz richten. Wir präsentieren anschließend das Design und die Implementierung des C3 Systems, welches einen solchen neuartigen Replika-Auswahl Algorithmus umfasst. Im C3-System wenden sich Clients ohne große serverseitige Unterstützung auf dynamische Art und Weise an Replikas, ohne dass dies zu unerwünschtem Verhalten wie Last-Oszillationen

führt. In unseren Experimenten, welche wir auf Amazon EC2 ausgeführt haben, und welche auf Betriebslasten von Spotify und SouncCloud sowie Simulationen beruhen, zeigen wir, dass C3 die Endlast der Latenzverteilungen signifikant (bis zu 3-fach im 99,9ten Perzentil) und den System-Durchsatz (um bis zu 50%) steigert. Als zweites System betrachten wir Serviceorientierte Architekturen (SOA) und Mikrodienste, welche zunehmends an Popularität als Architektur zur Entwicklung von fehlertoleranten Diensten gewinnen. Wir zeigen auf, wie komplexe Interaktionen zwischen verschienen Arbeitslasten zu signifikanten Einbußen der Systemperformanz führen kann, was wiederum zum Ausfall von Anwendungen führen kann. Wir diskutieren die Herausforderungen, bestimmte Ende-zu-Ende Performanzgarantien wie effiziente und faire Resourcenverwendung, die Einhaltung von Fristen, sowie die Reduktion der Antwortzeiten, zu realisieren. Wir stellen das Design und die Implementierung von Cicero vor, einem System, welches adaptive und verteilte Algorithmen verwendet, um Richtlinien des Rourcenmangements in SOAs und Mikrodiensten umzusetzen. In unseren Systemevaluationen, in denen wir ein Produktions-SOA emulieren, zeigen wir, wie Cicero dazu verwendet werden kann, die Leistungen, welche verschiedenen Endkunden gegenüber erbracht wird, zu entkoppeln und dabei kaskadierende Fehler zu vermeiden sowie einen möglichst hohen Anteil der Ende-zu-Ende Fristen einzuhalten.

Acknowledgements

"So long, and thanks for all the fish!"

I am extremely grateful to my advisors Anja Feldmann and Marco Canini, who guided me throughout my PhD. They encouraged me to work harder, to keep shooting higher, and held me to very high standards; without them, I would not have delivered this thesis. Not only have I benefited from Anja's extensive knowledge about all things networked, but also her vast experience with the many dimensions of the research process. Your advice on "finding my hammer" remains etched in my head for life. Marco's breadth and depth of knowledge on computer systems were critical to all projects in this thesis, and it was an honor to have been mentored by him over the years. I will also take with me Marco's expertise on other subjects, like food, Internet memes, and virtualizing governments. Anja and Marco, you have both set the bar extremely high as I embark on building my own career as a computer scientist, and for that, I am forever indebted.

I also extend my sincere thanks to Jon Crowcroft and Willy Zwaenepoel for their valuable feedback on my research. It was truly an honor to have had you both on my committee.

The close feedback cycle with Spotify and SoundCloud greatly benefited the C3 project. I am indebted to Sean Braithwaite for the fruitful collaboration with SoundCloud and for making my time spent there enjoyable (especially with all those philosophical conversations we had). I am grateful to Axel Liljencrantz, Jimmy Mårdell and Radovan Zvonček from Spotify for their valuable input and suggestions. Special thanks go to Sofie Thorsen for her contributions to the C3 project, including the experiments she conducted at Spotify. I also greatly benefited from the summer I spent at Microsoft Research in Redmond, which is where the Cicero project began. Peter Bodik, Ishai Menache, and Ganesh Ananthanarayanan, it was a great privilege to have learnt from all of you.

During the course of my research, I also benefited greatly from interactions with my colleagues at INET. Julius Schulz-Zander, thank you for all the great times we had working on the Odin project and for changing my life forever by introducing me to the rolling median. Dan Levin, thank you for all the sharp feedback over the years and the many gems on the Internet you introduced me to. Matthias Rost, Iam grateful to have had you as an office mate over the past many years, and hope that your relentless discipline and hard work have rubbed off on me. Ramin Khalili, the many discussions on performance evaluation concepts with you have no doubt influenced my research a lot, and I also cherish our time spent as rivals over table football. It was also a pleasure to supervise many bright and talented students over the years like Tobias Brandt and David Herzog, who I am confident will build great careers for themselves. And to the rest of the group, thank you for the many times you went through my dry runs.

My six years across Europe would not have been what they were if not for the amazing friends I have had here. I extend my heartfelt gratitude to Marcus, Navaneeth, Eva, Mariano, Wasif, Vasia, Joost, Christian, and Janine for having been a great source of support throughout this journey. Franzi, thank you for having been with me through both the ups and downs, and for being an endless source of strength for me.

Last, but by no means least, I thank my father, mother, and sister for their unconditional support and encouragement, without which none of this would have been possible.

Eidesstattliche Erklärung

Ich versichere an Eides statt, dass ich diese Dissertation selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Datum Lalith Suresh Puthalath

Papers

Parts of this thesis are based on the following peer-reviewed papers that have already been published, as well as papers currently under submission. All my co-authors are acknowledged as scientific collaborators of this work.

Pre-published papers

International conferences

Suresh, L., Canini, M., Schmid, S., and Feldmann, A. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (Oakland, CA, May 2015), USENIX Association, pp. 513–527.

Papers under submission

International journals

Suresh, L., Canini, M., Schmid, S., Feldmann, A.,Thorsen, S., Brandt, T.,and Braithwaite, S. Cutting tail latency in cloud data stores via adaptive replica selection.

International conferences

Suresh, L., Bodik, P., Menache, I., Ananthanarayanan, G., and Canini, M. Distributed Resource Management for Multi-tenant Micro-services.

Contents

1 Introduction

Today, we increasingly rely on a plethora of interactive online services for social networking, e-commerce, video and audio streaming, email, and web search. These online services are high-traffic websites, with providers such as Google and Facebook handling tens of billions of visits per day that operate upon petabytes of data [1, 25].

The compute and storage requirements for operating at such scales necessitate that more and more online services run inside large data centers [43]. This trend is further exacerbated by the exponential growth of mobile devices and tablets as the end point for accessing these online services [24]. This shift towards the "cloud" has led to the paradigm of *warehouse-scale computers* (WSCs) [43]. In WSCs, a single program is an *entire Internet Service* such as Netflix that runs across thousands of commodity server machines. In this paradigm, an Internet service is typically architected as a complex mix of inter-communicating distributed systems. As an example, consider a load of a Facebook page. A user accesses Facebook over a browser which is intercepted by a web server. The web server interacts with several back-end systems, such as TAO [50] for the social graph, Haystack [45] for photos, MySQL [28] and Memcached [17] for storing objects, as well as a range of sub-systems for Facebook's chat service [27], serving advertisements, performing type-ahead [29] and other site features.

More generally, the distributed systems backing WSCs include various flavors of file systems [83], data-stores [5,63], cluster schedulers [15,142], data-analytics stacks [83], monitoring infrastructure [42], and micro-services [109,113], wherein distributed applications implementing business logic are realized as loosely coupled clusters of communicating processes. A key characteristic of the aforementioned distributed systems is that their architectures follow the *scale-out approach*, wherein a system's capacity is increased by adding more servers to an existing cluster, as opposed to upgrading the compute and storage capacities of individual servers. This paradigm of system design is visible in several classes of distributed systems typical of WSCs today. For instance, with distributed data-stores and data-analytics platforms, storage capacity and bandwidth is increased by adding more storage servers (and disks). Similarly, applications architected as micro-services comprise several inter-communicating clusters of servers, each of which can be scaled out independently to support higher demands and compute capacities.

Many of the systems discussed above need to work in concert to realize even a single online service such as Netflix. At the same time, users of these online services also expect fluid response times [77]. Several studies have shown how end users are sensitive to even seemingly nominal increases in web page loading times, and how this directly impacts revenues. For instance, Google reported that slowing down the page-load time for Google Search results by 100ms to 400ms led to 0.2% to 0.6% fewer searches [51]. In fact, users exposed to 400ms delays for six weeks in that study did 0.21% fewer searches for five weeks *after* the study ended, indicating that poor user experiences have ramifications for people's long-term behaviors. A study from Bing found that a two-second slowdown in page-load times reduced the number of queries per user by 1.8% and revenue per user by 4.3% [127]. Furthermore, Shopzilla [127] discussed how a five-second speedup in page loads resulted not only in revenue improvements but also facilitated a 50% reduction in hardware. This underlines the fact that such performance improvements have important ramifications not only for user satisfaction and thus revenue, but also for operating costs.

Thus, two factors intersect to form the motivation for this thesis: *i)* the ever-increasing compute and storage requirements for online services mandate a shift towards WSCs, a setting that is widespread today and is expected to remain so in the years to come, and *ii)* both users and providers of these online services stand to gain from predictable performance for distributed systems typical of WSCs.

1.1 Challenges to predictable performance

While the scale-out approach has enabled building highly fault-tolerant systems with its emphasis on redundancy, achieving predictable end-to-end performance in such systems is challenging. This is due to two key characteristics of scale-out distributed systems, namely, complex request-response characteristics as well as performance fluctuations:

Complex request-response characteristics. A common characteristic of scale-out distributed systems is that even a single operation performed by the system requires processing across several servers in concert. For example:

- With distributed data-stores, even a single query from an application server may require fetching tens to thousands of records from a cluster of storage nodes in parallel. For instance, a single load of a Facebook page by a user aggregates and filters hundreds of items from the social graph [50].
- In a micro-services application, a single invocation of a service-level API may require the cooperation of several tiers of servers, each of which may comprise hundreds of servers.
- Within Microsoft Bing, a single web search request involves multiple sequences of processing stages, such as spell checking and ad generation. Each of these stages may involve tens to thousands of servers [88].
- A single analytics job in Apache Hadoop or Flink may involve a complex DAG of compute activity and data transfers with dependencies [83].

These characteristics imply that even serving a single end-user request (e.g., to return a web page) may involve calls to tens or hundreds of servers [63, 88]. The slowest of these calls to complete determines the response time of the end-user request.

Performance fluctuations. At the same time, individual servers of these large distributed systems suffer from *performance fluctuations* over time [62]. These performance fluctuations are caused by many factors, such as contention for shared resources, background activities like garbage collection and log compaction, skewed access patterns, as well as imperfect performance isolation in multi-tenant settings. Some of these effects, such as garbage collection, occur at sub-second timescales [62, 99]. Given that a single service-level request involves processing across tens to hundreds of servers, the slowest server determines the overall request completion time. This implies that significant delays at any of these servers inflate the latency observed by end users and reduce system throughput. These sources of performance fluctuation cannot be eliminated in large and complex systems, and thus, make end-to-end performance notoriously difficult to predict and guarantee. Thus, when operating at scale, even rare episodes of performance variability affect a significant fraction of all requests in large-scale distributed systems.

To summarize the challenges:

- Even a single user-level operation in a scale-out distributed system requires request-response exchanges involving several servers.
- Delays affecting a small subset of these requests impact the overall completion time of the user-level operation.
- Performance fluctuations at individual servers therefore impact a large fraction of user-level operations.

1.2 The need for adaptive mechanisms

As distributed systems in WSCs scale to accommodate growing user demands and data volumes, complex inter-server dependencies lead to interactions that may compromise system performance. As an example, consider the Visual Studio Online outage in 2014 [85]. The multi-tiered system architecture involved a front-end service (Shared Platforms Services, SPS) making calls to an intermediary tier (Azure Service Bus), which then made calls to a cloud database (SQL Azure). A *single request type* from SPS was accessing a slow database on SQL Azure. These calls from SPS were made using blocking remote procedure calls, which blocked a thread each until a response arrived. Since the responses were delayed because of the slow database, the thread pools on SPS servers eventually ran out of threads, starving completely unrelated request types. This led to all requests to another service, Team Foundation Server (TFS), being blocked because of dependencies on SPS, which subsequently brought down Visual Studio Online.

Similarly, even in well-provisioned distributed storage clusters, unpredictable events such as garbage collection on individual hosts can lead to latency spikes [99]. Furthermore, background activities such as *compaction* [5, 52] lead to significantly increased I/O activity, which in turn affects the storage system's latencies as well as throughput.

Given the *sub-second* timescales at which these inter-component interactions and performance fluctuations occur, operator intervention and manual tuning of myriad system configuration parameters are infeasible means to address the challenge of performance predictability. This is further complicated by the fact that as systems evolve over time, performance characteristics and bottlenecks change, and manually tuned configuration parameters can thus quickly become outdated.

Growing system complexity and the inevitability of variable performance necessitates *adaptive* mechanisms with which systems can quickly react to dynamic conditions that occur on small timescales. These adaptive mechanisms are necessary to build systems that deliver predictable end-to-end performance.

1.3 Thesis goal and scope

Goal. In this thesis, we focus on the question of designing adaptive mechanisms for two important and pervasive classes of scale-out distributed systems: cloud data-stores and micro-services applications. Our focus is on online services, which require low end-to-end latencies, as opposed to offline services such as batch computing where jobs run on the timescales of minutes and hours, if not days.

Distributed data-stores. A broad range of low-latency data-storage systems are in the critical path of online services today. They include distributed key-value stores, relational databases, in-memory caches as well as block storage systems. As explained earlier, these data-stores suffer from many sources of performance fluctuations over time, which make it challenging for them to deliver consistently low latencies, especially at higher system loads. An important characteristic of these systems is that data is often replicated on multiple nodes for performance and fault-tolerance reasons. Clients in such systems can then perform *replica selection*, wherein a client selects one out of multiple replica servers to service a request. This presents an opportunity for clients to respond to performance variability among servers. In this thesis, we first discuss the fundamental challenges involved in designing a replica selection scheme that is robust in the face of performance fluctuations across servers. We illustrate these challenges through performance evaluations of the Cassandra distributed database on Amazon EC2. We then present the design and implementation of an adaptive replica selection mechanism, C3, that is robust to performance variability in the environment. We demonstrate C3's effectiveness in reducing the latency tail and improving throughput through extensive evaluations on Amazon EC2, through simulations, and through evaluations against production workloads from Spotify and SoundCloud. Our results show that C3 significantly improves latencies along the mean, median, and tail (up to 3 times improvement at the 99.9^{th} percentile) and provides up to 50% higher system throughput.

Service-oriented architectures (SOAs) and micro-services. In Service-oriented architectures (SOA) and micro-services, a large distributed application is broken up into collections of smaller, inter-communicating services, typically aligned with development team boundaries. The focus on loose coupling and modularity allows several teams of developers to evolve their respective services independently of each other. However, these systems raise new challenges in attaining high performance and efficient resource utilization, especially in the face of multi-tenancy (wherein tenants may map to different external customers or consumers of the service, but also internal product groups, applications, or various differentiated background tasks). In these systems, request execution happens across tens to even thousands of processes. The execution paths and resource demands of different types of requests across different services are generally not known a-priori. This leads to the fundamental challenge of managing resources and scheduling requests *end-to-end*, to meet various performance and fairness objectives. In this thesis, we present a fully distributed framework for end-to-end resource management and scheduling in multi-tenant distributed systems, named Cicero. Cicero uses distributed admission control mechanisms that dynamically adapt the rate of requests allowed in the system according to bottlenecks detected along request execution paths, quickly reacting to overload in the face of changing workloads. By propagating minimal metadata through requests as they flow through the system, Cicero also enforces the execution of scheduling policies to meet deadlines and minimize average completion times. In system evaluations against production as well as synthetic workloads, Cicero successfully achieves various end-to-end performance objectives such as reducing average latencies, meeting deadlines, providing fairness and isolation as well as regulating system-wide load.

1.4 Contributions

The contributions of this thesis are as follows:

- **Adaptive replica selection for cloud data-stores:** We highlight the challenges of delivering consistent and low tail-latencies in the context of replicated and partitioned cloud data-stores. We present the need for adaptive replica selection schemes, and highlight the challenges in designing them for practical settings. We present the design and implementation of **C3**, an adaptive replica selection mechanism that is robust to performance variability in the environment.
- **End-to-end resource management and scheduling for micro-services:** Given the emerging nature of the micro-services approach to architecting large and complex applications, we discuss the challenges in achieving end-to-end performance objectives of concern for both tenants and the operator. We then present the design and implementation of **Cicero**, a system that allows enforcing a diverse range of end-to-end resource management policies in a micro-services setting.

1.5 Outline

Chapter 2 discusses the background setting for this thesis, namely, *warehouse-scale computers* (WSC). It highlights the challenges involved in achieving predictable performance for distributed systems typical of WSCs.

Chapter 3 positions the work in this thesis with respect to related work around end-to-end performance predictability in different systems domains.

Chapter 4 focuses on cloud data-stores, and highlights the potential for replica selection as a means to reducing latencies in the face of server-side performance fluctuations. It then presents the design, implementation and evaluation of the C3 system.

Chapter 5 moves beyond the two-tiered setting with storage clients and servers as discussed in C3. It discusses the more general setting of service-oriented architectures and micro-services (SOAs). It presents distributed algorithms embodied within the Cicero system, that enable the enforcement of a diverse range of resource-management policies in an SOA, such as performance isolation as well as meeting end-to-end deadlines. We then validate our design through evaluations.

Chapter 6 concludes this thesis and outlines future work.

2 Warehouse scale computing: background, challenges and objectives

A complex mix of *distributed* systems software running inside data-center environments power online services today. These systems have specific characteristics and challenges that motivate this thesis. In this chapter, we present relevant background information to aid the reader in contextualizing this thesis.

In Section 2.1, we first discuss trends in building online services that motivate server-centric computing. We highlight how this trend has led to the paradigm of warehouse-scale computers (WSCs), wherein a single "user application" is a highly distributed Internet service like Google Search, running atop clusters of thousands of servers (Section 2.2). In Section 2.3, we then highlight the commodification of WSCs, implying that they are currently – and will remain – the paradigm of choice to build online services for years to come.

Next, we move on to a discussion of the software stack within WSCs and introduce important terminology. In Section 2.4, we introduce scale-out distributed systems in the context of the WSC software stack. Subsequently, we highlight important challenges involved in achieving predictable performance in the context of these distributed systems in Section 2.6. We then outline the scope of this thesis in terms of the class of techniques used to tackle these challenges in Section 2.7.

2.1 From client-centric to server-centric computing

The proliferation of Internet services such as search, e-mail, and social networking have accelerated the trend towards server-side and cloud computing. There are several advantages to this trend. Users no longer need to regularly configure or update all the software they rely upon, and simply access these services over a browser. At the same time, service providers need not deal with myriad kinds of operating systems, different versions of software as well as hardware from various generations, and this significantly accelerates development lifecycles. Thus, instead of pushing software updates to hundreds of millions of users, services such as Google or Facebook can rapidly innovate and evolve their systems "behind the scenes," benefiting both users and providers. In addition, the ever-rising number of mobile devices, which is already numbering in the billions today [24], further accelerates the trend of pushing computing to the server side and having thin client-side software for efficiency reasons. Furthermore, as online services themselves become richer and more complex, the economics of increasing storage and computing requirements also mandate this shift towards server-centric computing. In fact, these services are prohibitive to perform on clients. For instance, services such as web search, on-demand video streaming, and image processing require and are best served by massive computing infrastructures.

2.2 Towards warehouse-scale computers

The data centers powering many of today's large Internet services are no longer a miscellaneous collection of hardware in a co-location center, running workloads that could be served off a single or few machines (AltaVista famously ran on five computers in 1996). They are instead a large array of physical (and/or virtualized) servers, where even the smallest unit of software deployment for a service such as web search may require hundreds to thousands of machines. This kind of computing platform powering large online-services today is referred to as a warehouse-scale computer (WSC) [43].

WSCs are a key departure from the computing model wherein a program runs on a single machine. In a WSC, a program is instead an entire Internet service (such as Netflix's streaming service, Google Search or Amazon's e-commerce site), and instead of a single machine, thousands of servers with heterogeneous capabilities run myriad software components. A WSC may involve a single private entity owning the hardware, the platform software as well as the applications that run atop the infrastructure (*e.g.* Google owning all its hardware and software). Alternatively, WSCs may involve a tenant (Netflix) renting massive hardware infrastructure as well as software services (such as access to a shared database) from a provider such as Amazon.

2.3 Commodification of WSCs

Barroso *et al.* [43] argued that the economics of server-class computing puts clusters of hundreds of nodes within the reach of a wide band of corporations and research organizations. Indeed, we find this stance to be vindicated. For instance, as of January 2014, Spotify had over 5,000 physical servers and well over 1,000 virtual servers across their public and private clouds [33]. Airbnb has around 5,000 EC2 instances running on AWS, with about 1,500 of those catering to the user-facing portion of their application [34]. Depending on the popularity of the site, Netflix's world wide streaming service used as many as 10,000 machines in 2012 [30]. The availability of easy-to-use public clouds such as Amazon AWS and Google Compute Engine have enabled small and medium sized enterprises to quickly scale up their infrastructure to hundreds and thousands of servers to meet business needs without having to commit to building and maintaining data centers of their own. Furthermore, organization continue to migrate their systems to the cloud [11], suggesting that the omnipresence of WSCs is expected to grow even further in the years to come.

2.4 Software for WSCs: From scale-up to scale-out distributed systems

For reasons of cost efficiency, WSCs today are typically built using large clusters of low-end servers rather than a smaller number of high-end servers [44]. The lower reliability and higher failure rates of individual hardware components is compensated for using a highly fault-tolerant software stack comprising a mix of *scale-out distributed systems*. As demand for a service grows, capacity in scale-out distributed systems is increased not by upgrading individual servers, but by adding more server and hardware resources to existing clusters. This also implies being able to scale down cluster sizes during times of lower demand to reduce costs and manage system efficiency. For instance, it is common practice to adapt the number of servers in response to diurnal patterns in site traffic [9]. The need for such elasticity as well as the cost efficiency of using large clusters of low-end hardware resources mean scale-out distributed systems are *essential* to the paradigm of WSCs. A typical WSC thus comprises a complex stack of systems software, which Barroso *et al.* [43] classify into three broad categories:

- **Platform-level software:** This is software present on individual servers that abstracts away hardware resources. This includes firmware, an operating system kernel, as well as a variety of libraries.
- **Cluster-level infrastructure:** A collection of distributed systems that simplify the usage of resources distributed across thousands of servers. Barroso *et al.* [43] consider the union of these services as an operating system for a data-center. Such systems include distributed file systems (*e.g.* HDFS [83],

GFS [72]), cluster schedulers (*e.g.* Borg [142], Apollo [48]), and distributed storage systems (*e.g.* Amazon's Dynamo [63], Cassandra [5]). These services shield programmers building WSC applications from resource-management and data-distribution complexity.

- **Application-level software:** This is software that implements a specific service (such as Google Search, Netflix's streaming service or Amazon's e-commerce site). Application-level software comprises both online and offline services, each of which has different requirements. Online services are typically user facing, and thus have tight requirements on end-to-end latencies (*e.g.* Google search, Gmail, Spotify, SoundCloud *etc.*). On the other hand, offline compute frameworks are typically used for large-scale data analysis or as part of a pipeline that generates data used in online services (*e.g.* recommendations for online shopping services, or the index for Google Search).

Scale-out distributed systems thus permeate the cluster-level and application-level software in WSCs. Furthermore, the commodification of WSCs has driven the rapid growth of an ecosystem of open-source software systems at the platform-, cluster- and application- level. The Hadoop stack, NoSQL databases, application server platforms such as Tomcat, and libraries for building service-oriented architectures such as Hystrix [108] and Finagle [137] have allowed developers to quickly have warehouse-scale Internet services up and running.

This thesis focuses on parts of cluster-level infrastructure, namely, distributed data-stores (such as Dynamo [63] or Cassandra [5]), as well as architectures for building application-level software (micro-services and service-oriented architectures).

2.5 Predictable performance: Metrics of concern

It is useful to think about the above-mentioned scale-out distributed systems as presenting a "service" to their users. Users view these systems as black boxes that expose a certain service through APIs. We refer to invocations of these APIs as *service-level requests*. For instance, users of a data-store are often application software that perform reads and writes against the data-store through a client library. A service-level request in such a scenario might be a query by an application server to fetch all records corresponding to a list of keys.

In this thesis, we are primarily interested in predictable performance for service-level requests as described by two key metrics:

- ***i)* End-to-end latency**, in particular the tail of the latency distribution, such as the 99.9th percentile latency.
- ***ii)* Throughput**, expressed in requests per second.

These two metrics are often used by providers to define service-level objectives (SLOs). For instance, the strict latency requirements for Amazon's services are measured by the 99.9th percentile of the latency distribution under a peak value for client throughput [63].

2.6 Challenges to predictable performance in WSCs

Internet services backed by WSCs today need to deliver fluid response times to their users [51, 127] while simultaneously serving a high volume of traffic. At the same time, for cost and power efficiency reasons, WSCs need to run at high levels of system utilization as well.

To meet these goals, the distributed systems across the WSC stack must deliver consistent and predictable performance in terms of end-to-end latency and throughput. However, the combination of two key characteristics of these systems make this goal challenging. First, individual servers and software components comprising these distributed systems consistently experience performance fluctuations due to various factors. Second, a basic unit of work submitted to one of these distributed systems requires the cooperation of tens to thousands of servers (like a search query to a distributed database). The combination of these two factors implies that performance variability at the granularity of individual servers and software components will quickly impair end-to-end performance for the distributed systems in WSCs.

We will now discuss both of these factors in detail.

2.6.1 Performance fluctuations are the norm rather than the exception

When operating WSCs involving hundreds to thousands of servers, a key challenge is the performance fluctuations exhibited by individual servers over time. Several studies have confirmed the degree of performance variability that plague individual servers and components in WSCs. Dean and Barroso [62] list various sources of performance variability in Google's WSCs, including interference due to contention for shared resources within different parts of and among applications (also discussed in detail by Melanie *et al.* [93]), garbage collection, maintenance activities (such as log compaction for log-structured storage systems), and background daemons performing periodic tasks (a related example of which is [122]). Xu *et al.* [150] performed an analysis of tail latencies on EC2, and discuss how scheduling intricacies of the Xen virtualization hypervisor leads to latency variability. Their measurement study shows that co-scheduling of latency-sensitive and CPU-bound tasks can inflate tail latencies by factors between two and four. In a study of production workflows in Bing, Jalaparti *et al.* [88] point out performance variability due to hardware-related issues (a subset of machines consistently exhibiting higher service times), sporadic congestion events, time-dependent events such as rolling upgrades throughout a cluster, and input data related issues wherein requests accessing a specific part of the

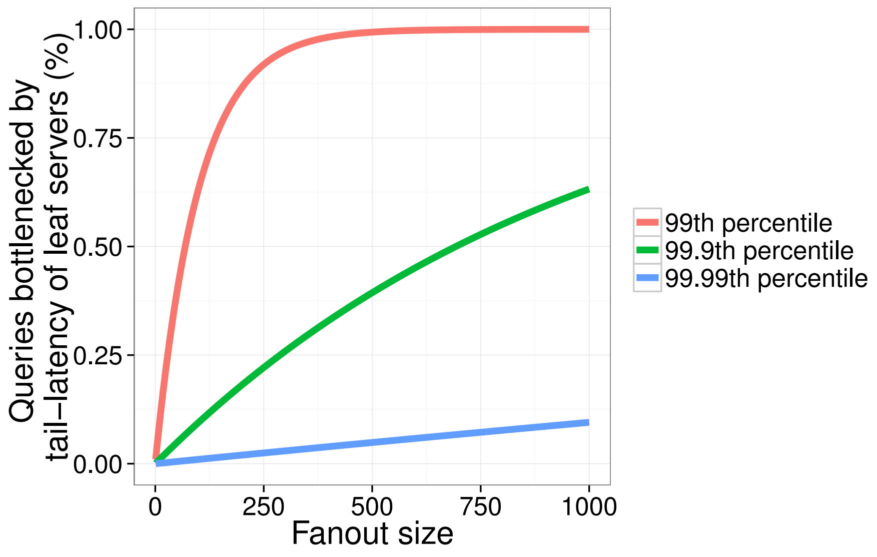


Figure 2.1: Fraction of queries that will be dominated by tail latency of individual leaf servers as a function of fanout size. 63% of queries with a fanout size of 100 will experience latencies higher than the 99th percentile latency of individual leaf servers. (Reproduced from [62])

search index were slower. In a large-scale study of disk and SSD performance over 87 days, Hao *et al.* [84] show that storage performance instability is commonplace: disk and SSD based RAIDs suffer from at least one slow drive (that affects tail latencies) 1.5% and 2.2% of the time respectively. Chronos [94] discusses latency spikes that are caused by the OS network stack (NIC interactions and socket handling), application lock contention inside the kernel, and application-layer straggler threads. Furthermore, co-location of interactive, performance-sensitive production workloads with batch workloads, coupled with imperfect operating system-level performance isolation also contributes to performance variability [93, 142]. In the context of big data analytics stacks built using memory-managed languages (such as Java), Ionel *et al.* [75] show how garbage collection increases the run time of data processing jobs by up to 40%. Similarly, Maas *et al.* [99] show how GC pauses in an Apache Spark cluster compromise job completion times by up to 15%.

2.6.2 Request processing involves tens to thousands of servers

A common architectural pattern in the way online services are built today is to have large request fanouts, wherein a single service-level request triggers tens to thousands of sub-operations in parallel [62]. Typically, this happens via a root server aggregating responses from a large number of leaf nodes: all these sub-operations must complete for the parent request to make progress. Similarly, cloud storage systems typically support batch queries that request several records in parallel.

While fanouts are one dimension of the problem, service-level requests may also require *sequences* of processing involving several stages. This is further exacerbated by the recent trend towards architecting large application- and cluster-level systems as micro-services and service-oriented architectures (SOA). For instance, the web-search workflow in the Bing SOA involves multiple tiers (such as a document lookup tier, spell-checking and snippet generation) which are accessed in a sequence for even a single request [88]. Each of these tiers further triggers large request fanouts as well. For instance, the median request in the Bing SOA requires processing across sequences of 15 tiers and 10% of these tiers process a query in parallel across thousands of servers [88].

2.6.3 Performance variability exacerbates at scale

We illustrate how performance variability at the level of individual servers affect end-to-end performance at scale using the simple, idealized model from [62]. Consider the classic web service setting wherein a *service-level request* arriving at an application server triggers a single lookup request to a database server. Assume that a fraction of requests p to the database server experience a high-latency episode due to background activity (such as a garbage collection pause). If $p = 1\%$, 1 in a 100 requests made to the database will be slow. In effect, an equivalent fraction of service-level requests will experience higher than usual latencies.

Now consider the kind of setting described in Section 2.6.2, wherein we have large request fanouts. Here, a single service-level request at an application server triggers f requests in parallel to f different database servers. The overall service-level latency now depends on the slowest of the f parallel requests. Because of this property, sporadic component-level performance variability bottlenecks a significant fraction of queries in the presence of large request-fanouts. To see why, assume again that a fraction of requests p to each individual database server experiences a high-latency episode. The overall service-level latency will be high if even a single request out of the f parallel requests hits a slow server. The probability for such an event occurring depends on the values of p and f, and is given by:

$$P(Latency(Q) > p^{th} \text{ percentile latency of leaf servers}) = (1 - (p)^f)$$

According to this equation, with a fanout size of 100 requests, 63% of queries will experience response times greater than or equal to the 99th percentile latency of the individual leaf servers. Figure 2.1 visually represents this probability as a function of the fanout size for the 99th, 99.9th and 99.99th percentile latencies.

A similar line of reasoning holds when service-level requests require processing across sequences of stages as described in the previous section.

To summarize, as service-level requests need to traverse multiple tiers of processing, each of which can have high request fanouts, performance fluctuations at individual servers significantly impact end-to-end performance.

2.7 Goal: Short-term adaptations to performance variability for online services

The central theme of this thesis is the following: what adaptive mechanisms can distributed systems typical of WSCs today employ to be robust in the face of performance fluctuations that individual servers experience?

In light of that, there are two broad classes of mechanisms to cope with performance variability: short-term adaptations and long-term adaptations [62].

- **Short-term adaptations.** These techniques typically operate at very short timescales of milliseconds to tens of milliseconds, or at the granularity of each individual request. These approaches typically work with already provisioned resources and servers in a system (as opposed to spawning new server instances dynamically). Examples of such approaches include duplicating/re-issuing requests [62, 143], or performing adaptive replica selection and resource scheduling [126].
- **Long-term adaptations.** These techniques react to coarser-grained phenomena, such as re-balancing data partitions in response to persistent workload skews [62], or consistent cross-application interference in the case of co-located workloads [140]. The class of solutions that provision new hardware resources/capacity in the system based on predictive models fall into this category [131].

Goal. In this thesis, we will focus on designing short-term adaptation techniques for online, interactive services (as opposed to offline batch services). The key constraint we impose on the solution space is practicality. This means that we desire solutions that operate on minimal assumptions about the nature of the workload. Furthermore, we do not assume a-priori knowledge about system behavior based on models developed offline (such as the optimization-framework approach used by [88]).

2.8 System types considered in this thesis

In the previous sections, we presented a high-level overview of the kinds of systems permeating WSCs, and prior work targeting performance predictability in the context of different system types. In this thesis, we will focus on two important classes of distributed systems present in WSCs, namely distributed data-stores and microservices. We now present some relevant background on these system types before concluding the chapter.

2.8.1 Cloud data stores

Storage systems are often in the critical path of any application-level software. In fact, several online services rely on a mix of different storage backends, with each geared for different workloads. For instance, a page-load by a Facebook user will aggregate responses from TAO [141], Memcached [17], Haystack [45] and f4 [106]. Application-level software require low-latencies and high throughput from these data-stores in order to support the tens to hundreds of millions of users of the respective online service.

These data-stores come in a wide-variety of flavors, each of them typically specialized towards specific workloads. Block storage systems typically store and serve Binary Large Objects (blobs) of data. These blobs are either stored unstructured as in Facebook's Haystack [45] and f4 [106], or in organized hierarchically as in GFS/Colossus [73] and HDFS [83]). Other storage systems store structured data (typically organized as tables), examples of which include BigTable [52], HBase [7], Spanner [59], TAO [141], and Cassandra [5]). More generally, a plethora of systems fall under the banner of NoSQL data-stores, some examples of which are MongoDB [18], and Amazon's Dynamo DB [63]). In-memory data stores are often used as caches and to store ephemeral data in large online services, such as session state. This includes systems such as Memcached [17], and Redis [21]. Lastly, a lot of organizations also use traditional relational databases, such as MySQL [28]. Evidence also exists for building large clusters of MySQL instances and handling data-sharding, request routing and aggregation at the application layer [69].

Some salient characteristics of the environment in which these systems are used are:

- **Use of replication for availability and performance:** for reasons of fault-tolerance, data is typically replicated across multiple storage nodes. Replication factors in such settings are typically low (3 is a common value within a data-center). Furthermore, many workloads typical of online services tend to be read-mostly [49, 92]. This allows clients or middleware to exploit the combined read capacities of multiple replica servers and thus take advantage of replication for performance reasons as well.
- **High fan-ins and fanouts:** Given the push towards micro-services, and a large number of application servers, it has become common for data-stores to have high-fanins from many clients. Given the lack of centralized request dispatching, these requests often go directly from the client to the storage node that dispatches the responses. Furthermore, as discussed in § 2.6.2, high request fanouts are commonplace [62, 148]. In such workloads, clients need to operate upon queries that involve reads to multiple keys, each of which may hit different nodes.

Quoted verbatim from [151]:

- All teams will henceforth expose their data and functionality through service interfaces.
- Teams must communicate with each other through these interfaces.
- There will be no other form of inter-process communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.
- It doesn't matter what technology they use. HTTP, Corba, Pub-Sub, custom protocols — doesn't matter.
- All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.

Figure 2.2: The SOA mandate from Steve Yegge's Google+ post [151]

- **Lack of centralized request scheduling:** Data-stores in the critical path of online-services, wherein low-latency reads are necessary, often eschew centralized request scheduling for scalability reasons. This lack of centralized request scheduling means a single, consistent, global view of how all outstanding requests have been assigned to storage nodes does not exist.

2.8.2 Service-oriented architectures and micro-services

Given stringent performance and availability requirements, applications deployed on WSCs are typically built as distributed systems comprising multiple tiers of loosely coupled, communicating clusters of processes. In such *service-oriented architectures*, a large distributed application is broken down into modular and reusable components. In 2011, Steve Yegge accidentally leaked a controversial Google+ post where he discusses an alleged mandate from Jeff Bezos in 2002 to all Amazon engineers. This mandate serves as a useful anecdote that highlights the core ideas behind using SOAs (Figure 2.2).

Service-oriented architectures stress re-usability and loose-coupling. They enable organizations to scale out developer effort, with each team responsible for a single service (or a few at most). This is done by ensuring that each service is exposed using well defined APIs, accessible over a wide array of transports such as HTTP. Thus, a service A that depends on a service B simply invokes B's external APIs over the network. This allows developer teams to evolve their respective code bases more quickly than is otherwise possible with large software monoliths.

A trend in this space is on keeping individual services leaner, leading to *micro-services*. In the micro-services approach, a single complex application may be comprised of tens to hundreds of smaller services, each of which are clusters of servers on their own. Several companies have adopted the micro-services approach, including Netflix [109,113], SoundCloud [128], Spotify [32], Uber [138], and Amazon [105]. The micro-services approach to building systems is not only evident at the application-level (as in the Bing SOA [88] and all the examples listed above), but also at the cluster-level. For instance, the Kubernetes [15] architecture comprises a core that exposes APIs around which cluster management logic is built as a collection of micro-services [142].

The salient characteristics of SOAs and micro-services of interest in this thesis are as follows:

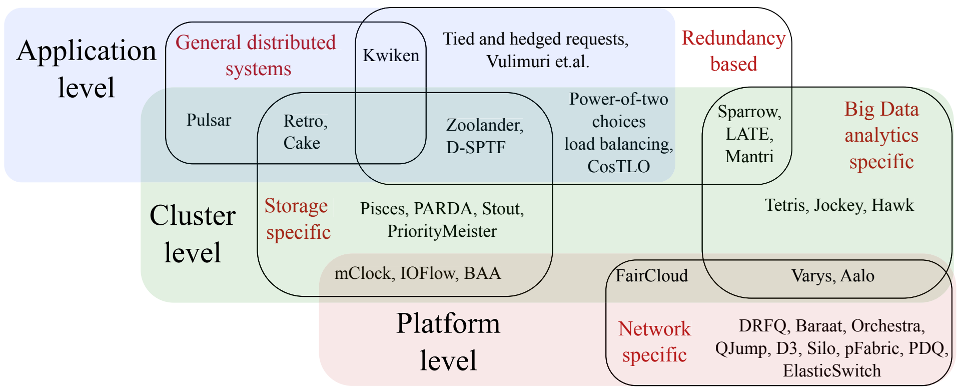
- **Large number of services:** Micro-services from different organizations easily involve 10s to 100s of independent services [109, 113]. Each of these services may be realized by tens to thousands of servers [88].
- **Complex execution DAGs:** A single external request to a system built as a micro-service triggers requests and responses between different services. In the Bing SOA, the median request involves sequences of 15 services (*e.g.* snippet generation and spell checking), and 10% of services involve aggregating responses from up to 10 other services [88].
- **Diverse processing times across stages:** A key characteristic of the micro-services setting is that processing times and throughputs of different service tiers may be vastly different between each other. This complicates end-to-end performance engineering for different types of requests [88, 90].

2.9 Summary

In this chapter, we presented the necessary background for this thesis. We introduced warehouse-scale computers and their historical background. We gave an overview of the scale-out distributed systems that permeate the data-center landscape. Next, we outlined the key obstacles in the way of achieving predictable performance for these distributed systems: namely, performance variability at the granularity of individual servers and components, as well as how this variability is amplified at scale. Next, we laid out the design space of solutions we consider within this thesis: short-term adaptations in the context of online, interactive services. Lastly, we presented the necessary background on cloud data-stores as well as SOAs/Micro-services, the two classes of distributed systems we will focus on in this thesis.

3 Related work

In this section, we present an overview of approaches related to work presented in this thesis. Prior efforts around performance predictability for WSCs have focused on different layers of the stack (across the application, cluster and platform level). We present a broad categorization of these solutions in Figure 3.1, and will now discuss them in detail.



Application level
General distributed systems
Kwiken
Tied and hedged requests, Vulimuri et.al.
Redundancy based
Pulsar
Retro, Cake
Zoolander, D-SPTF
Power-of-two choices load balancing, CosTLO
Sparrow, LATE, Mantri
Big Data analytics specific
Cluster level
Storage specific
Pisces, PARDA, Stout, PriorityMeister
Tetris, Jockey, Hawk
mClock, IOFlow, BAA
FairCloud
Varys, Aalo
Platform level
Network specific
DRFQ, Baraat, Orchestra, QJump, D3, Silo, pFabric, PDQ, ElasticSwitch

Figure 3.1: Categorization of related work on performance predictability

3.1 Using redundancy to mitigate outliers

Dean and Barroso [62] described techniques employed at Google to tolerate latency variability. They discuss short-term adaptations in the form of request reissues, along with additional logic to support preemption of duplicate requests to reduce unacceptable additional load. In D-SPTF [98], a request is forwarded to a single server. If the server has the data in its cache, it will respond to the query. Otherwise, the server forwards the request to all replicas, which then make use of cross-server cancellations to reduce load as in [62]. Vulimiri *et al.* [143] also make use of duplicate requests. They formalize the different threshold points at which using redundancy aids in minimizing the tail. In the context of Microsoft Azure and Amazon S3, CosTLO [148] also presents the efficacy of duplicate requests in coping with performance variability. Kwiken [88] decomposes the problem of minimizing end-to-end latency over a processing DAG into a manageable optimization over individual stages, wherein one of the latency reduction techniques used at each stage is request reissues. Zoolander [131] is a key-value store that uses replication to mask response times from outliers. It uses an analytical model to scale out via replication when system resources are under-utilized, and reverts to scaling out via traditional approaches during periods of heavy-utilization.

All the above approaches propose the use of additional system resources in order to mitigate outliers. The use of redundant requests serves to protect against outliers, but presents a trade-off between reducing tail-latency and improving system throughput. Chapter 4 instead raises the question of how to *select* replica servers, a first-order concern that is not addressed by the above discussed works. We show that replica selection does not involve a similar latency-throughput trade-off. Relevant literature on replica selection includes the work by Mitzenmacher [104], who showed that allowing a client to choose between two randomly selected servers based on queue lengths exponentially improves load-balancing performance over a uniform random scheme (discussed further in Chapter 4). This approach is embodied within systems such as Sparrow [116], which, however, operate in an offline cluster computing setting where jobs run on the order of several hundreds of milliseconds, if not minutes and hours. Chapter 4 instead presents the challenge of replica selection in the context of low-latency data stores that are typically in the critical path of web serving workloads.

3.2 Resource allocation and scheduling for storage systems

Pisces [126] is a multi-tenant key-value store architecture that provides fairness guarantees between tenants. It is concerned with fair-sharing the data-store and presenting proportional performances to different tenants. Pisces [126] recognizes the problem of weighted replica selection and employs a round robin algorithm. PARDA [79] is also focused on the problem of sharing storage bandwidth according to proportional-share fairness. Stout [101] uses congestion control to respond to storage-layer performance variability by adaptively batching requests. PriorityMeister [154] focuses on providing tail latency QoS for bursty workloads in shared networked storage by combining priorities and rate limiters. mClock [80] discusses I/O resource allocation inside a hypervisor. The proposed technique supports proportional-share fairness across virtual machines using mechanisms such as reservations, shares and limits. That is, each VM receives a resource allocation according to its share, subject to its reservation and limits. PARDA [79] adopts a flow-control approach to achieving proportional allocation of resources in a distributed storage setting without support from the storage array itself. PARDA leverages latency measurements to detect overload, and uses a FAST-TCP [146] inspired control mechanism to regulate the number of IO requests issued per storage client accordingly. IOFlow [135] uses a logically centralized architecture to dictate service and routing properties for IO requests that traverse the multiple layers between VMs and a storage backend. Using these "data plane" hooks, a centralized controller enforces various policies such as differentiated treatment, and minimum bandwidth guarantees. Wang *et al.* [145] discusses fair resource allocation across storage clients when accessing multi-tiered storage systems composed of heterogenous devices.

The above literature does not directly address the concern of adapting to performance variability, and are orthogonal to the work presented in Chapter 4. Chapter 4 subsequently compares against the round-robin replica selection scheme proposed by Pisces [126].

3.3 Resource allocation and scheduling for general distributed systems

Beyond storage specific solutions (§ 3.2), solutions have been proposed for general distributed systems. Retro [100] proposes a centralized controller for distributed systems, which adjusts rates of different workflows to achieve global performance objectives. Using periodic reports of resource usage by different tenants (or more generally, workflows), Retro's controller applies various policies to determine rate-limits for different workflows at various control points throughout the system (which will then regulate the load at the overloaded resources). Pulsar [38] provides an abstraction of a virtual datacenter where a tenant runs on VMs and accesses several *appliances*. A centralized scheduler allocates rates which are then enforced at the level of *network flows*. Pulsar treats each appliance as a blackbox and regulates rates for different tenants based on periodically collected statistics about the load at each appliance and whether each tenant is meeting their desired performance objectives. Chapter 5 instead presents a fully distributed architecture, and thus avoids scalability bottlenecks latent in a centralized approach.

Cake [144] makes use of coordinated, multi-resource scheduling for HBase/HDFS with the goal of achieving both high throughput and bounded latencies. Cake uses two levels of schedulers: first-level schedulers that schedule access to individual resources and second-level schedulers which run a slower-timescale feedback loop that adjusts resource allocations at the first-level schedulers with the objective of maximizing SLO compliance and system utilization. In the specific context of solutions for distributed storage systems (discussed above in section 3.2): mClock [80], PARDA [79], Pisces [126], IOFlow [135], Wang *et al.* [145] provide high-level scheduling policies and fair sharing; however, these systems typically consider a two-tiered system with clients and servers instead of a general DAG.

Chapter 5 instead discusses resource allocation and scheduling in the context of more general distributed systems architected as service-oriented architectures and micro-services. Lastly, Kwiken [88] considers interactive systems where requests execute in a DAG of services and tunes the request retry timeouts to achieve the best trade-off between resource usage and tail latency. Unlike Kwiken, the work presented in Chapter 5 does not involve training and building an offline model of system behaviors. Our work extends beyond Kwiken by addressing challenges of multi-tenancy and distributed resource management.

3.4 Offline, cluster computing

Predictable performance in the context of offline cluster has been an active research area. Systems such as LATE [153] and Mantri [37] have looked at the problem of skew-tolerance, wherein straggler tasks in data-parallel applications can delay the overall job completion time. These systems rely on task retries (§ 3.1) in order to handle outliers. Another angle of attack has been on resource allocation and scheduling, typically informed via pre-computed models of job characteristics. Jockey [71] precomputes statistics using a simulator, which is informed about a job's characteristics, in order to predict the completion time of the job. Tetris [76] uses prior knowledge of a job's characteristics to compute heuristics with which to perform multi-resource packing of jobs on a cluster. Hawk [64] addresses the challenge of short jobs faring poorly due to competition for resources with large jobs. Hawk instead proposes a hybrid scheduling approach, where short jobs are scheduled in a fully distributed way, and long jobs are scheduled using a centralized scheduler.

Chapter 4 and Chapter 5 instead focuses on challenges for interactive online services, wherein request processing occurs at much shorter timescales than in offline batch processing systems.

3.5 Data-center networking

Lastly, a lot of research efforts have focused on predictable performance with regards to the data-center network that supports WSCs. With the flexibility allowed in designing data center networks, several recent works have focused on resource allocation and scheduling problems related to quickly completing one or multiple transfers (e.g., Varys [56], Aalo [54], Baraat [66], Orchestra [55]), achieving low latency (e.g., Qjump [78], D3 [147], Silo [89], pFabric [36], PDQ [87]) and fairly sharing the network bandwidth (e.g., FairCloud [118], ElasticSwitch [119]). We believe that these approaches are orthogonal to the solutions proposed in this thesis, which operate at the application layer.

3.6 Summary

In this chaper, we discussed the landscape of solutions that focused on performance predictability in the context of WSCs. We classified these approaches according to the specific layer of the WSC stack that they target, the classes of distributed systems for which they apply, as well as the techniques used to achieve predictable performance. We then positioned the work in this thesis with respect to the literature discussed above.

4 Reducing tail-latency for cloud data-stores

In this chapter, we raise the question of performance predictability in the context of cloud data-stores, distributed systems that are often in the critical path of every online service today [63]. We then make the case for adaptive replica selection as a means to address server-side performance fluctuations in cloud data-stores.

Several studies [62, 88, 150] indicate that latency distributions in Internet-scale systems exhibit long-tail behaviors. That is, the 99.9^{th} percentile latency can be more than an order of magnitude higher than the median latency. Recent efforts [35,62,71,88,116,131,154] have thus proposed approaches to reduce tail latencies and lower the impact of skewed performance. These approaches rely on standard techniques including giving preferential resource allocations or guarantees, reissuing requests, trading off completeness for latency, and creating performance models to predict stragglers in the system.

A recurring pattern to reducing tail latency is to exploit the redundancy built into each tier of the application architecture. In this chapter, we show that the problem of *replica selection* — wherein a *client* node has to make a choice about selecting one out of multiple *replica servers* to serve a request — is a first-order concern in this context. Interestingly, we find that the impact of the replica selection algorithm has often been overlooked. We argue that layering approaches like request duplication and reissues atop a poorly performing replica selection algorithm should be cause for concern. For example, reissuing requests but selecting poorly-performing nodes to process them increases system utilization [143] in exchange for limited benefits.

As we show in Section 4.1, the replica selection strategy has a direct effect on the tail of the latency distribution. This is particularly so in the context of data stores that rely on replication and partitioning for scalability, such as key-value stores. The performance of these systems is influenced by many sources of variability [62, 95] and running such systems in cloud environments, where utilization should be high and environmental uncertainty is a fact of life, further aggravates performance fluctuations [93].

Replica selection can compensate for these conditions by preferring faster replica servers whenever possible. However, this is made challenging by the fact that servers exhibit performance fluctuations over time. Hence, replica selection needs to quickly adapt to changing system dynamics. On the other hand, any reactive scheme in this context must avoid entering pathological behaviors that lead to load imbalance among nodes and oscillating instabilities. In addition, replica selection should not be computationally costly, nor require significant coordination overheads.

In this chapter, we present C3, an adaptive replica selection mechanism that is robust in the face of fluctuations in system performance. At the core of C3's design, two key concepts allow it to reduce tail latencies and hence improve performance predictability. First, using simple and inexpensive feedback from servers, clients make use of a replica ranking function to prefer faster servers and compensate for slower service times, all while ensuring that the system does not enter herd behaviors or load-oscillations. Second, in C3, clients implement a distributed rate control mechanism to ensure that, even at high fan-ins, clients do not overwhelm individual servers. The combination of these mechanisms enable C3 to reduce queuing delays at servers while the system remains reactive to variations in service times.

Our study applies to any low-latency data store wherein replica diversity is available, such as a key-value store. We hence base our study on the widely-used [12] Cassandra distributed database [5], which is designed to store and serve larger-than-memory datasets. Cassandra powers a variety of applications at large web sites such as Netflix and eBay [6]. Compared to other related systems (Table 4.1), Cassandra implements a more sophisticated load-based replica selection mechanism as well, and is thus a better reference point for our study. However, C3 is applicable to other systems and environments that need to exploit replica diversity in the face of performance variability, such as a typical multi-tiered application or other data stores such as MongoDB or Riak.

In summary, we make the following contributions:

1. Through performance evaluations on Amazon EC2, we expose the fundamental challenges involved in managing tail latencies in the face of service time variability (§4.1).

2. We develop an adaptive replica selection mechanism, C3, that reduces the latency tail in the presence of service time fluctuations in the system. C3 does

Cassandra	*Dynamic Snitching*: considers history of read latencies and I/O load
OpenStack Swift	Read from a single node and retry in case of failures
MongoDB	Optionally select nearest node by network latency (does not include CPU or I/O load)
Riak	Recommendation is to use an external load balancer such as Nginx [16]

Table 4.1: Replica selection mechanisms in popular NoSQL solutions. Only Cassandra employs a form of adaptive replica selection (§4.1.3).

not make use of request reissues, and only relies on minimal and approximate information exchange between clients and servers (§4.2).

3. We implement C3 (§4.3) in the Cassandra distributed database and evaluate it through experiments conducted on Amazon EC2 (for accuracy) (§4.4) and simulations (for scale) (§4.4.2). We demonstrate that our solution improves Cassandra's latency profile along the mean, median, and the tail (by up to a factor of 3 at the 99.9th percentile) whilst improving read throughput by up to 50%. Furthermore, we also present results of running C3 against production workloads from SoundCloud and Spotify.

The rest of the chapter is organized as follows. Section 4.1 motivates the need for adaptive replica selection and outlines the challenges involved. Section 4.2 explains C3's design and its components. Section 4.3 discusses the implementation of C3 within the Cassandra distributed database. In Section 4.4, we evaluate C3 under various conditions. Section 4.5 presents a closing discussion of our work, including open questions. Section 4.6 concludes the chapter.

4.1 The Challenge of Replica Selection

In this section, we first discuss the problem of time-varying performance variability in the context of cloud environments. We then underline the need for load-based replica selection schemes and the challenges associated with designing them.

4.1.1 Performance fluctuations are the norm

Servers in cloud environments routinely experience performance fluctuations due to a multitude of reasons. Citing experiences at Google, Dean and Barroso [62] list many sources of latency variability that occur in practice. Their list includes, but is not limited to, contention for shared resources within different parts of and between

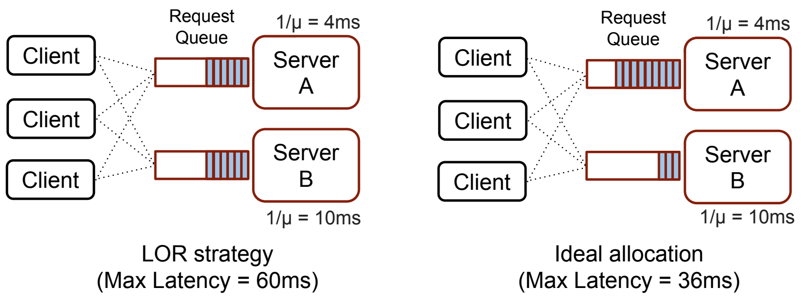


Figure 4.1: *Left*: how the least-outstanding requests (LOR) strategy allocates a burst of requests across two servers when executed individually by each client. *Right:* An ideal allocation that compensates for higher services time with lower queue lengths.

applications (further discussed in [93]), periodic garbage collection, maintenance activities (such as log compaction), and background daemons performing periodic tasks [122]. Recently, an experimental study of response times on Amazon EC2 [150] illustrated that long tails in latency distribution can also be exacerbated by virtualization. A study [88] of interactive services at Microsoft Bing found that over 30% of analyzed services have 95^{th} percentile of latency 3 times their median latency. Their analysis showed that a major cause for the high service performance variability is that latency varies greatly across machines and time. Lastly, a common workflow involves accessing large volumes of data from a data store to serve as inputs for batch jobs on large-scale computing platforms such as Hadoop, and injecting results back into the data store [132]. These workloads can introduce latency spikes at the data store and further impact on end-user delays.

As part of our study, we spoke with engineers at Spotify and SoundCloud, two companies that use and operate large Cassandra clusters in production. Our discussions further confirmed that all of the above mentioned causes of performance variability are true pain points. Even in well provisioned clusters, unpredictable events such as garbage collection on individual hosts can lead to latency spikes. Furthermore, Cassandra nodes periodically perform *compaction*, wherein a node merges multiple SSTables [5,52] (the on-disk representation of the stored data) to minimize the number of SSTable files to be consulted per-read, as well as to reclaim space. This leads to significantly increased I/O activity.

Given the presence of time-varying performance fluctuations, many of which can potentially occur even at sub-second timescales [62], it is important that systems gracefully adapt to changing conditions. By exploiting server redundancy in the system, we investigate how replica selection effectively reduces the tail latency.

4.1.2 Load-based replica selection is hard

Accommodating time-varying performance fluctuations across nodes in the system necessitates a replica selection strategy that takes into account the load across different servers in the system. A strategy commonly employed by many systems is the *least-outstanding requests* strategy (LOR). For each request, the client selects the server to which it has the least number of outstanding requests. This technique is simple to implement and does not require global system information, which may not be available or is difficult to obtain in a scalable fashion. In fact, this is commonly used in load-balancing applications such as Nginx [19] (recommended as a load-balancer for Riak [16]) or Amazon ELB [4].

However, we observe that this technique is not ideal for reducing the latency tail, especially since many realistic workloads are skewed in practice and access patterns change over time [41]. Consider the system in Figure 4.1, with two replica servers that at a particular point in time have service times of 4 ms and 10 ms respectively. Assume all three clients receive a burst of 4 requests each. Each request needs to be forwarded to a single server. Based on purely local information, if every client selects a server using the LOR strategy, it will result in each server receiving an equal share of requests. This leads to a maximum latency of 60 ms, whereas an ideal allocation in this case obtains a maximum latency of 36 ms. We note that *LOR* over time will prefer faster servers, but by virtue of purely relying on local information, it does not account for the existence of other clients with potentially bursty workloads and skewed access patterns, and does not explicitly adapt to fast-changing service times.

Designing distributed, adaptive and stable load-sensitive replica selection techniques is challenging. If not carefully designed, these techniques can suffer from "herd behavior" [103, 121]. Herd behavior leads to load oscillations, wherein multiple clients are coaxed to direct requests towards the least-loaded server, degrading the server's performance, which subsequently causes clients to repeat the same procedure with a different server.

Indeed, looking at the landscape of popular data stores (Table 4.1), we find that most systems only implement very simple schemes that have little or no ability to react quickly to service time variations nor distribute requests in a load-sensitive fashion. Among the systems we studied, Cassandra implements a more sophisticated strategy called Dynamic Snitching that attempts to make replica selection decisions informed by histories of read latencies and I/O loads. However, through performance analysis of Cassandra, we find that this technique suffers from several weaknesses, which we discuss next.

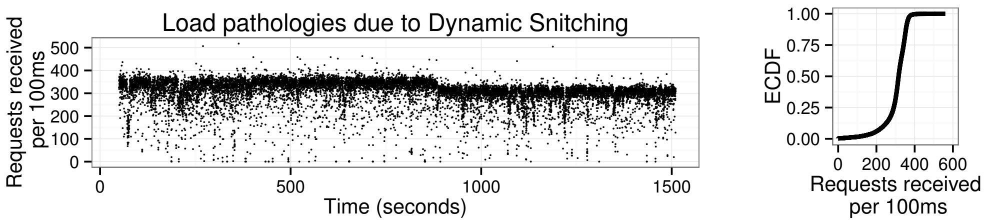


Figure 4.2: Example load oscillations seen from the most heavily utilized Cassandra node due to Dynamic Snitching, in measurements obtained on Amazon EC2. We show a timeseries (**left**) and the corresponding ECDF (**right**), for the number of requests processed in a 100 ms window by the Cassandra node. We note that the number of requests processed in a 100 ms window by a node ranges from 0 up to 500, which is symptomatic of herd behavior.

4.1.3 Dynamic Snitching's weaknesses

Cassandra servers organize themselves into a one-hop distributed hash table. A client can contact any server for a read request. This server then acts as a *coordinator*, and internally fetches the record from the node hosting the data. Coordinators select the best replica for a given request using *Dynamic Snitching*. With Dynamic Snitching, every Cassandra server ranks and prefers faster replicas by factoring in read latencies to each of its peers, as well as I/O load information that each server shares with the cluster through a gossip protocol.

Given that Dynamic Snitching is load-based, we evaluate it to characterize how it manages tail-latencies and if it is subject to entering load-oscillations. Indeed, our experiments on Amazon EC2 with a 15-node Cassandra cluster confirm this (the details of the experimental setup are described in § 4.4). In particular, we recorded heavy-tailed latency characteristics wherein the difference between the 99.9^{th} percentile latencies are *up to 10 times* that of the median. Furthermore, we recorded the number of read requests individual Cassandra nodes serviced in 100 ms intervals. For every run, we observed the node that contributed most to the overall throughput. These nodes consistently exhibited synchronized load oscillations, example sequences of which are shown in Figure 4.2. Additionally, we confirmed our results with the Spotify engineers, who have also encountered load instabilities that arise due to garbage-collection induced performance fluctuations in the system [96].

A key reason for Dynamic Snitching's vulnerability to oscillations is that each Cassandra node re-computes scores for its peers at fixed, discrete intervals. This interval based scheme poses two problems. First, the system cannot react to time-varying performance fluctuations among peers that occur at time-scales less than the fixed-

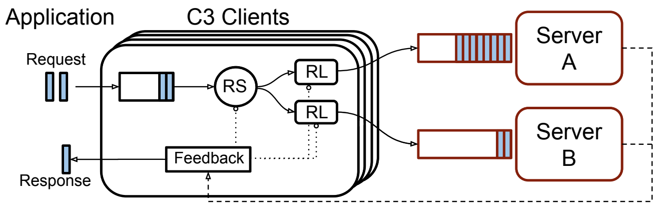


Figure 4.3: Overview of C3. **RS:** Replica Selection scheduler, **RL:** Rate Limiter of server $s \in [A, B]$.

interval used for the score recomputation. Second, by virtue of fixing a choice over a discrete time interval (100 ms by default), the system risks synchronization as seen in Figure 4.2. While one may argue that this can be overcome by shortening the interval itself, the calculation performed to compute the scores is expensive, as it is also stated explicitly in the source code; a median over a history of exponentially weighted latency samples (that is reset only every 10 minutes) has to be computed for each node as part of the scoring process. Additionally, Dynamic Snitching relies on gossiping one second averages of `iowait` information between nodes to aid with the ranking procedure (the intuition being that nodes can avoid peers who are performing compaction). These `iowait` measurements influence the scores used for ranking peers heavily (up to two orders of magnitude more influence than latency measurements). Thus, an external or internal perturbation in I/O activity can influence a Cassandra node's replica selection loop for extended intervals. Together with the synchronization-prone behavior of having a periodically updated ranking, this can lead to poor replica selection decisions that degrade system performance.

4.2 C3 Design

C3 is an adaptive replica selection mechanism designed with the objective of reducing tail latency. Based on the considerations in Section 4.1, we design C3 while keeping in mind these two goals:

i) *Adaptive:* Replica selection must *cope and quickly react to heterogeneous and time-varying service times* across servers.

ii) *Well-behaved:* Clients performing replica selection must *avoid herd behaviors* where a large number of clients concentrate requests towards a fast server.

At the core of C3's design are the two following components that allow it to satisfy the above properties:

1. **Replica Ranking:** Using minimal and approximate feedback from individual servers, clients rank and prefer servers according to a scoring function. The

scoring function factors in the existence of multiple clients and the subsequent risk of herd behavior, whilst allowing clients to prefer faster servers.

2. **Distributed Rate Control and Backpressure:** Every client rate limits requests destined to each server, adapting these rates in a fully-distributed manner using a congestion-control inspired technique [82]. When rate limits of all candidate servers for a request are exceeded, clients retain requests in a backlog queue until at least one server is within its rate limit again.

4.2.1 Replica ranking

With replica ranking, clients individually rank servers according to a scoring function, with the scores serving as a proxy for the latency to expect from the corresponding server. Clients then use these scores to prefer faster servers (lower scores) for each request. To reduce tail latency, we aim to minimize the product of queue size (q_s) and service time ($1/\mu_s$, the inverse of the service rate) across every server s (Figure 4.1).

Delayed and approximate feedback. In C3, servers relay feedback about their respective q_s and $1/\mu_s$ on each response to a client. The q_s is recorded after the request has been serviced and the response is about to be dispatched. Clients maintain Exponentially Weighted Moving Averages (EWMA) of these metrics to smoothen the signal. We refer to these smoothed values as $\bar{q_s}$ and $\bar{\mu_s}$.

Accounting for uncertainty and concurrency. The delayed feedback from the servers lends clients only an approximate view of the load across the servers and is not sufficient by itself. Such a view is oblivious to the existence of other clients in the system, as well as the number of requests that are potentially in flight, and is thus prone to herd behaviors. It is therefore imperative that clients account for this potential concurrency in their estimation of each server's queue size.

For each server s, a client maintains an instantaneous count of its outstanding requests os_s (requests for which a response is yet to be received). Clients calculate the queue size estimate ($\hat{q}_s$) of each server as $\hat{q}_s = 1 + os_s \cdot w + \bar{q_s}$, where $w >= 1$ is a weight parameter. We refer to the $os_s \cdot w$ term as the *concurrency compensation*.

The intuition behind the concurrency compensation term is that a client will always extrapolate the queue size of a server by an estimated number of requests in flight. That is, it will account for the fan in from multiple clients concurrently submitting requests to the same server. Furthermore, clients with a higher value of os_s will implicitly project a higher queue size at s and thus rank it lower than a client that has sent fewer requests to s. Using this queue size estimate to project the $\hat{q}_s/\bar{\mu_s}$ ratio results in a desirable effect: a client with a higher demand will be more likely to rank s poorly compared to a client with a lighter demand. This hence provides a degree of robustness to synchronization. In our experiments, we set w to the number

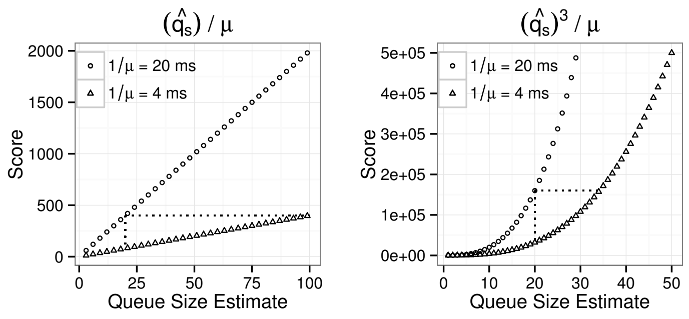


Figure 4.4: A comparison between linear (**left**) and cubic (**right**) scoring functions. For differing values of $1/\mu$, the difference in queue size estimates required for the scores of two replicas to be equal is smaller for the cubic function (thus penalizing longer queues).

of clients in the system. This serves as a good approximation in settings where the number of clients is comparable to the expected queue lengths at the servers.

Penalizing long queues. With the above estimation, clients can compute the $\hat{q}_s/\bar{\mu}_s$ ratio of each server and rank them accordingly. However, given the existence of multiple clients and time-varying service times, a function linear in $\hat{q}$ is not an effective scoring function for replica ranking. To see why, consider the example in Figure 4.4. The figure shows how clients would score two servers using a linear function: here, the service time estimates are 4 ms and 20 ms, respectively. We observe that under a linear scoring regime, for a queue size estimate of 20 at the slower server, only a corresponding value of 100 at the faster server would cause a client to prefer the slower server again. If clients distribute requests by choosing the best replica according to this scoring function, they will build up and maintain long queues at the faster server in order to balance response times between the two nodes.

However, if the service time of the faster server increases due to an unpredictable event such as a garbage collection pause, *all requests* in its queue will incur higher waiting times. To alleviate this, C3's scoring function *penalizes longer queue lengths* using the same intuition behind that of delay costs as in [47, 139]. That is, we use a non-decreasing convex function of the queue size estimate in the scoring function to penalize longer queues. We achieve this by raising the $\hat{q}_s$ term in the scoring function to a higher degree, b: $(\hat{q}_s)^b/\bar{\mu}_s$.

Returning to the above example, this means the scoring function will treat the above two servers as being of equal score if the queue size estimate of the faster server ($1/\mu = 4$ ms) is $\sqrt[b]{20/4}$ times that of the slower server ($1/\mu = 20$ ms). For higher values of b, clients will be less greedy about preferring a server with a lower μ^{-1}. We use $b = 3$ to have a cubic scoring function (Figure 4.4), which presents a good trade-off between clients preferring faster servers and providing enough robustness to time-varying service times. We evaluate the sensitivity of this parameter in Section 4.4.2.

Cubic replica selection. In summary, clients use the following scoring function for each replica:

$$\Psi_s = R_s - 1/\bar{\mu_s} + (\hat{q}_s)^3/\bar{\mu_s}$$

where $\hat{q}_s = 1 + os_s \cdot n + \bar{q_s}$ is the queue size estimation term, os_s is the number of outstanding requests from the client to s, n is the number of clients in the system, and R_s, $\bar{q_s}$ and $\bar{\mu_s}^{-1}$ are EWMAs of the response time (as witnessed by the client),[1] queue size and service time feedback received from server s, respectively. The score reduces to R_s when the queue size estimate term of the server is 1 (which can only occur if the client has no outstanding requests to s and the queue size feedback is zero). Note that the $R_s - \mu_s^{-1}$ term's contribution to the score diminishes quickly when the client has a non-zero queue size estimate (see Figure 4.4).

4.2.2 Rate control and backpressure

Replica selection allows clients to prefer faster servers. However, replica selection alone cannot ensure that the combined demands of all clients on a single server remain within that server's capacity. Exceeding capacity increases queuing on the server-side and reduces the system's reactivity to time-varying performance fluctuations. Thus, we introduce an element of rate-control to the system, wherein every client rate-limits requests to individual servers. If the rates of all candidate servers for a request are saturated, clients retain the request in a backlog queue until a server is within its rate limit again.

Decentralized rate control. To account for servers' performance fluctuations, clients need to adapt their estimations of a server's capacity and adjust their sending rates accordingly. As a design choice and inspired by the CUBIC congestion-control scheme [82], we opt to use a decentralized algorithm for clients to estimate and adapt rates across servers. That is, we avoid the need for clients to inform each other about their demands for individual servers, or for the servers to calculate allocations for potentially numerous clients individually. This further increases the robustness of our system; clients' adaptation to performance fluctuations in the system is not purely tied to explicit feedback from the servers.

Thus, every client maintains a token-bucket based rate-limiter for each server, which limits the number of requests sent to a server within a specified time window of δ ms. We refer to this limit as the *sending-rate* (*srate*). To adapt the rate limiter according

[1]Note R_s implicitly accounts for network latency but we consider that network congestion is not the source of performance fluctuations.

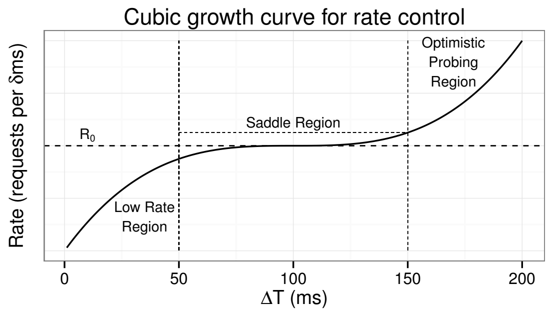


Figure 4.5: Cubic function for clients to adapt their sending rates

to the perceived performance of the server, clients track the number of responses being received from a server in a δ ms interval, that is, the *receive-rate* (*rrate*). The rate-adaptation algorithm aims to adjust *srate* in order to match the *rrate* of the server.

Cubic rate adaptation function. Upon receiving a response from a server s, the client compares the current *srate* and *rrate* for s. If the client's sending rate is lower than the receive rate, it increases its rate according to a cubic function [82]:

$$srate \leftarrow \gamma \cdot \left(\Delta T - \sqrt[3]{(\frac{\beta \cdot R_0}{\gamma})}\right)^3 + R_0$$

where ΔT is the elapsed time since the last rate-decrease event, and R_0 is the "saturation rate" — the rate at the time of the last rate-decrease event. If the receive-rate is lower than the sending-rate, the client decreases its sending-rate multiplicatively by β. γ represents a scaling factor and is chosen to set the desired duration of the saddle region (see § 4.3 for the values used).

Benefits of the cubic function. While we evaluated multiple rate adaptation functions (see § 4.4.2), we were attracted to a cubic growth function because of its property of having a saddle region. The functioning of the *cubic* rate adaption strategy caters to the following three operational regions (Figure 4.5): (1) *Low-rates:* when the current sending rate is significantly lower than the saturation rate (after say, a multiplicative decrease), the client increases the rate steeply; (2) *Saddle region:* when the sending rate is close to the perceived saturation point of the server (R_0), the client stabilizes its sending rate, and increases it conservatively, and (3) *Optimistic probing:* if the client has spent enough time in the stable region, it will again increase its rate aggressively, and thus probe for more capacity. At any time, if the algorithm perceives itself to be exceeding the server's capacity, it will update its view of the server's saturation point and multiplicatively reduce its sending rate. The parameter γ can be adjusted for a desired length of the saddle region. Lastly,

Algorithm 1 On Request Arrival (Request *req*, Replicas $\mathcal{R}$)

```
1:  repeat
2:      $\mathcal{R} \leftarrow sort(\mathcal{R})$                    ▷ sort replicas by cubic score function
3:      for Server $s$ in $\mathcal{R}$ do
4:          if $s$ within $srate_s$ then
5:              $consume\_token(srate_s)$
6:              $os_s \leftarrow os_s + 1$                    ▷ update outstanding requests
7:              $send(req, s)$                              ▷ send to server $s$
8:              return
9:      if $req$ not sent then
10:         wait until token available                      ▷ Backpressure
11: until $req$ is sent
```

given that multiple clients may potentially be adjusting their rates simultaneously, for stability reasons, we cap the step size of a rate increase by a parameter $s_{\max}$.

4.2.3 Putting everything together

C3 combines distributed replica selection and rate control as indicated in Algorithms 1 and 2, with the control flow in the system depicted in Figure 4.3. When a request is issued at a client, it is directed to a replica selection scheduler. The scheduler uses the scoring function to order the subset of servers that can handle the request, that is, the replica group ($\mathcal{R}$). It then iterates through the list of replicas and selects the first server s that is within the rate as defined by the local rate limiter for s. If all replicas have exceeded their rate limits, the request is enqueued into a backlog queue. The scheduler then waits until at least one replica is within its rate before repeating the procedure. When a response for a request arrives, the client records the feedback metrics from the server and adjusts its sending rate for that server according to the cubic-rate adaptation mechanism. After a rate increase, a hysteresis period is enforced (Algorithm 2, line 3) before another rate-decrease so as to allow clients' receive-rate measurements enough time to catch up since the last increased sending rate at T_{inc}.

4.3 Implementation

We implemented C3 within Cassandra. For Cassandra's internal read-request routing mechanism, this means that every Cassandra node is both a C3 client and server (specifically, coordinators in Cassandra's read path are C3 clients). In vanilla Cassandra, every read request follows a synchronous chain of steps leading up to an eventual enqueuing of the request into a per-node TCP connection buffer. For C3, we modified this chain of steps to control the number of requests that would be

Algorithm 2 On Request Completion (Request *req*, Server *s*)

```
1: $os_s \leftarrow os_s - 1$                                  ▷ update outstanding requests
2: update EWMA of $q_s$, $\mu_s^{-1}$ feedback
3: if ($srate_s > rrate_s$ && $now() - T_{inc} > hysteresis\_period$) then
4:     $R_0 \leftarrow srate_s$
5:     $srate_s \leftarrow srate_s \cdot \beta$
6:     $T_{dec} \leftarrow now()$
7: else if ($srate_s < rrate_s$) then
8:     $\Delta T \leftarrow now() - T_{dec}$
9:     $T_{inc} \leftarrow now()$
10:    $R \leftarrow \gamma \cdot \left(\Delta T - \sqrt[3]{(\frac{\beta \cdot R_0}{\gamma})}\right)^3 + R_0$
11:    $srate_s \leftarrow \min(srate_s + s_{\max}, R)$
```

pushed to the TCP buffers of each node. Recall that C3's replica scoring and rate control operate at the granularity of replica groups. Given that in Cassandra, there are as many replica groups as nodes themselves, we need as many backpressure queues and replica selection schedulers as there are nodes. Thus, every read request upon arrival in the system needs to be asynchronously routed to a scheduler corresponding to the request's replica group. Lastly, when a coordinator node performs a remote read, the server that handles the request tracks the service time of the operation and the number of pending read requests in the server. This information is piggybacked to the coordinator and serves as the feedback for the replica ranking.

There are challenges in making this implementation efficient. For one, since a single remote peer can be part of multiple replica groups, multiple admission control schedulers may potentially contend to push a request from their respective backpressure queues towards the same endpoint. Care needs to be exercised that this does not lead to starvation. To handle this complexity, we relied upon the Akka framework [3] for message-passing concurrency (*Actor* based programming). With Akka, every per-replica group scheduler is represented as a single actor, and we configured the underlying Java thread dispatcher to fair schedule between the actors. This design of having multiple backpressure queues also increases robustness, as one replica group entering backpressure will not affect other replica groups. The message queue that backs each Akka actor implicitly serves as the backpressure per-replica group queue. At roughly 600 bytes of overhead per actor, our extensions to Cassandra are thus lightweight. Our implementation amounted to 398 lines of code.[2]

For the rest of our study, we set the cubic rate adaptation parameters as follows: the multiplicative decrease parameter β is set to 0.2, and we configured γ to set the saddle region to be 100 ms long. We define the rate for each server as a number of permissible requests per 20 ms (δ), and use a hysteresis duration equal to twice

[2]Based on a Cassandra 2.0 development version.

the rate interval. We cap the cubic-rate step size (s_{max}) to 10. We did not conduct an exhaustive sensitivity analysis of all system parameters, which we leave for future work. Lastly, Cassandra uses read-repairs for anti-entropy; a fraction of read requests will go to all replicas (10% by default). This further allows coordinators to update their view of their peers.

4.4 System Evaluation

In evaluating C3, we are interested in answering the following questions across various conditions:

1. Does C3 improve the tail latency without sacrificing the mean or median?
2. Does C3 improve the read throughput (requests/s)?
3. How well does C3 load condition the cluster and adapt to dynamic changes in the environment?

We evaluate C3 under several settings. First, we stress our Cassandra implementation of C3 using workloads from the Yahoo Cloud Serving Benchmark (YCSB) [58], a closed-loop workload generator (§ 4.4.1). Next, we use simulations to study the C3 scheme independently of the Cassandra implementation (§ 4.4.2). Lastly, we present results conducted against production workloads at Spotify and SoundCloud (§ 4.4.3).

4.4.1 Evaluation using synthetic benchmarks

Experimental Setup: We evaluated C3 on Amazon EC2. Our Cassandra deployment comprised 15 m1.xlarge instances. We tuned the instances and Cassandra according to the officially recommended production settings from Datastax [10] as well as in consultation with our contacts from the industry who operate production Cassandra clusters.

On each instance, we configured a single RAID0 array encompassing the four ephemeral disks which served as Cassandra's data folder (we also experimented on instances with SSD storage as we report on later). We used the industry-standard Yahoo Cloud Serving Benchmark (YCSB) [58] to generate datasets and run workloads while stressing Cassandra in a closed-loop. We assign tokens to each Cassandra node such that nodes own equal segments of the keyspace. Cassandra's replication factor was set to 3. We inserted 500 million 1KB-sized records generated by YCSB, which served as the dataset. The workload against the cluster was driven from three instances of YCSB running in separate VMs, of identical EC2 instance type as the Cassandra nodes. Each YCSB instance ran 40 request generators, for a total of 120 generators. This leads to the baseline Cassandra cluster operating at average latencies of ∼15ms for read requests for different workloads. Each generator has a TCP

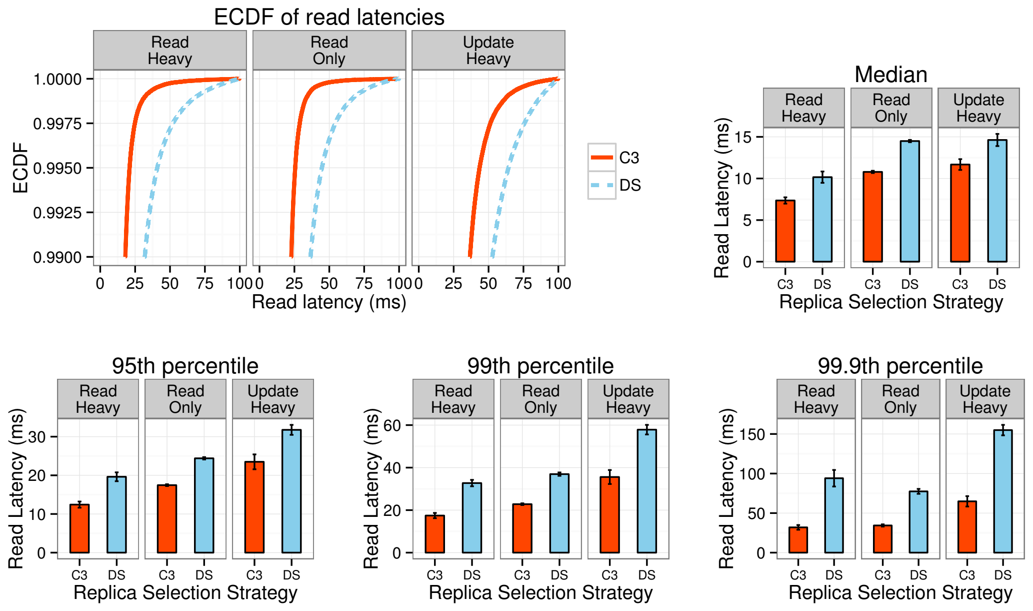


Figure 4.6: Cassandra's latency characteristics when using Dynamic Snitching and C3. C3 significantly improves the tail latency under different workloads without compromising the median.

connection of its own to the Cassandra cluster. Generators create requests for keys distributed according to a Zipfian access pattern prescribed by YCSB, with Zipf parameter $\rho = 0.99$, drawing from a set of 10 million keys. We used three common workload patterns for Cassandra deployments to evaluate our scheme: read-heavy (95% reads – 5% writes), update-heavy (50% reads – 50% writes) and read-only (100% read). These workloads generate access patterns typical of photo tagging, session-store and user-profile applications, respectively [58]. The read and update heavy workloads in particular are popular across a variety of Cassandra deployments [68, 92]. Each measurement involves 10 million operations of the workload, and is repeated five times. Bar plots represent averages and 95th percentile confidence intervals. All experiments use a read consistency level (CL) of ONE unless noted otherwise (we also conduct experiments using QUORUM consistency). Note, the consistency level determines the number of replicas that need to acknowledge a request.

Impact of workload on latency: Figure 4.6 indicates the read latency characteristics of Cassandra across different workloads when using C3 compared to Dynamic Snitching (DS). Regardless of the workload used, C3 improves the latency across all the considered metrics, namely, the mean, median, 99^{th} and 99.9^{th} percentile latencies. Since the ephemeral storage in our instances are backed by spinning-head

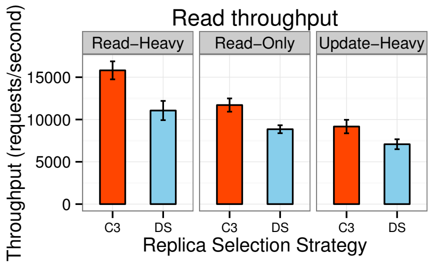


Figure 4.7: Throughput obtained with C3 and with Dynamic Snitching. C3 achieves higher throughput by better utilizing the available system capacity across replica servers.

disks, the latency increases with the amount of random disk seeks. This explains why the read-heavy workload results in lower latencies than the read-only workload (since the latter causes more random seeks). Furthermore, C3 effectively shortens the ratio of tail-latencies to the median, leading to a more predictable latency profile. With the read-heavy workload, the difference between the 99.9th percentile latency and the median is 24.5 ms with C3, whereas with DS, it is 83.91 ms: *more than 3x improvement*. In the update-heavy and read-only scenarios, C3 improves the same difference by a factor of 2.6 each. Besides the different percentiles, C3 also improves the mean latency by between 3 ms and 4 ms across all scenarios.

Impact of workload on read throughput: Figure 4.7 indicates the measured throughputs for C3 versus DS. By virtue of controlling waiting times across the replicas, C3 makes better use of the available system capacity, resulting in an increase in throughput across the considered workloads. In particular, C3 improves the throughput by between 26% and 43% across the considered workloads (update-heavy and read-heavy workloads respectively). We also note that the difference in throughput between the read- and update-heavy workloads of roughly 75% (across both strategies) is consistent with publicly available Cassandra benchmark data [68].

Impact of workload on load-conditioning: We now verify whether C3 fulfills its design objective of avoiding load pathologies. Since the key access pattern of our workloads are Zipfian distributed, we observe the load over time of the node that has served the highest number of reads across each run, that is, the most heavily utilized node. Figure 4.8 represents the distribution of the number of reads served per 100 ms by the most heavily utilized node in the cluster across runs. Note that *despite improving the overall system throughput*, the most heavily utilized node in C3 serves fewer requests than with DS. As a further confirmation of this, we present an example load profile as produced by C3 on highly utilized nodes (Figure 4.9). Unlike with DS, we do not see synchronized load-spikes when using C3, evidenced by the

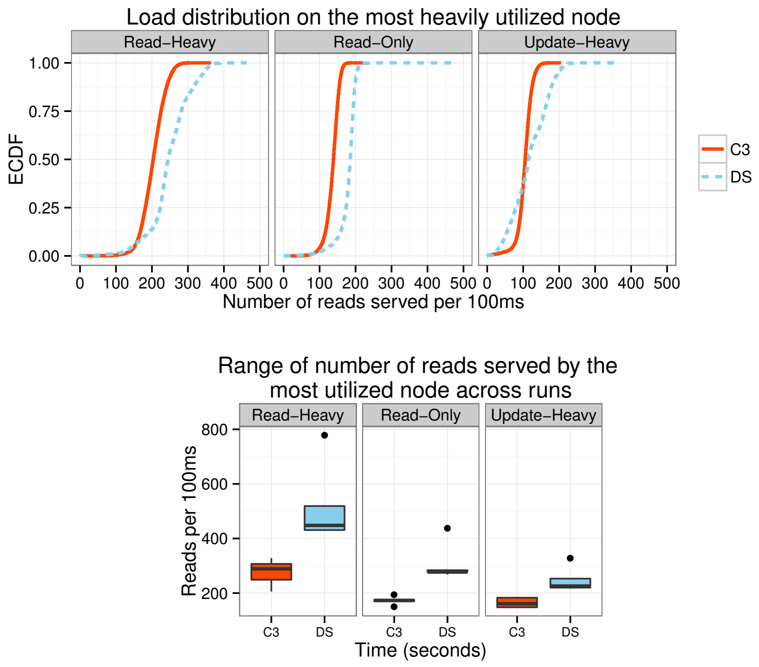


Figure 4.8: Aggregated distribution of number of reads serviced per 100 ms, by the most heavily loaded Cassandra node per run. With C3, the most heavily utilized node has a lower range in the load over time, wherein the difference between the 99th percentile and median number of requests served in 100 ms is lower than with Dynamic Snitching.

lack of oscillations and synchronized vertical bursts in the time-series. Furthermore, given that C3's rate control absorbs and distributes bursts carefully, it leads to a smoother load-profile wherein samples of the load in a given interval are closer to the system's true capacity unlike with DS.

Performance at higher system utilization: We now compare C3 with DS to understand how the performance of both systems degrade with an increase in overall system utilization. We increase the number of workload generators from 120 to 210 (an increase of 75%). Figure 4.10 presents the tail latencies observed for the read-heavy workload. For a 75% increase in the demand, we observe that C3's latency profile, even at the 99.9^{th} percentile, degrades proportionally to the increase in system load. With DS, the median and 99.9^{th} percentile latencies degrade by roughly 82%, whereas the 95^{th} and 99^{th} percentile latencies *degrade by factors of up to 150%*. Furthermore, the mean latency with DS is *70% higher* than with C3 under the higher load.

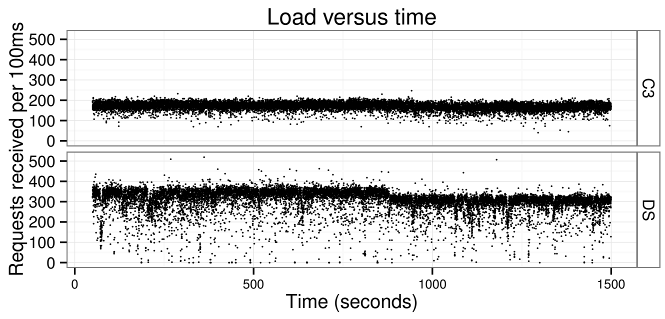


Figure 4.9: Example number of reads received by a single Cassandra node, per 100ms. With C3 (top), Cassandra coordinators internally adjust sending rates to match their peers' perceived capacity, leading to a smoother load profile free of oscillations. The per-server load is lower in C3 also because the requests are spread over more servers compared to DS (bottom).

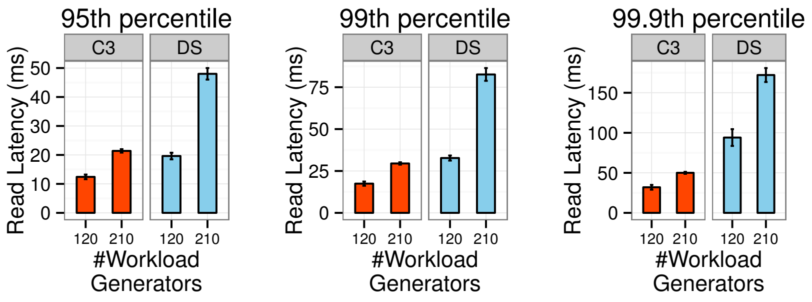


Figure 4.10: Overall performance degradation when increasing the number of workload generators from 120 to 210.

Adaptation to dynamic workload change: We now evaluate a scenario wherein an update-heavy workload enters a system where a read-heavy workload is already active, and observe the effect on the latter's read latencies. The experiment begins with 80 generators running a read-heavy workload against the cluster. After 640 s, an additional 40 generators enter the system, issuing update-heavy workloads. We observe the latencies from the perspective of the read-heavy generators around the 640 s mark. Figure 4.11 indicates a time-series of the latencies contrasting C3 versus DS. Each plot represents a 50-sample wide moving median[3] over the recorded latencies. Both DS and C3 react to the new generators entering the system, with a degradation of the read latencies observed at the 640 s mark. However, in contrast to

[3]A moving median is better suited to reveal the underlying trend of a high-variance time-series than a moving average [40]

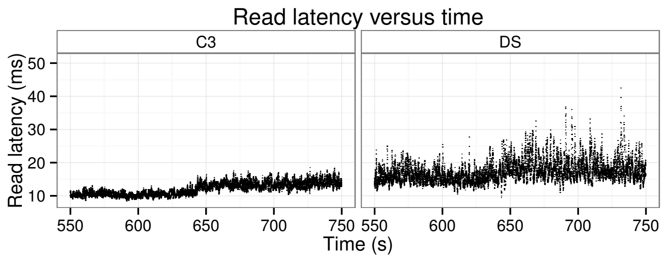


Figure 4.11: Dynamic workload experiment. The moving median over the latencies observed by the read-heavy generators from a run each involving C3 (**left**) and DS (**right**). At time 640 s, 40 new generators join the system and issue update-heavy workloads. With C3, the latencies degrade gracefully, whereas DS fails to avoid latency spikes.

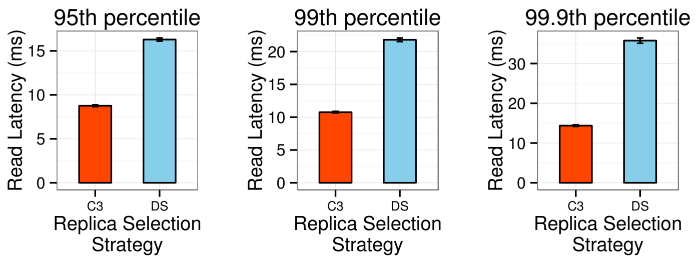


Figure 4.12: Results when using SSDs instead of spinning-head disks.

DS, C3's latency profile degrades gracefully, evidenced by the lack of synchronized spikes visible in the time-series as is the case with DS.

Skewed record sizes: So far, we considered fixed-length records. Since C3 relies on per-request feedback of the service times in the system, we observe whether variable length records may introduce any anomalies in the control loop. We use YCSB to generate a similar dataset as before, but where field sizes are Zipfian distributed (favoring shorter values). The maximum record length is 2KB, with each record comprising the key, and ten fields. Again, C3 improves over DS along all the considered latency metrics. In particular, with C3, the 99^{th} percentile latency is just under 14 ms, whereas that of DS is close to 30 ms; *more than 2x improvement.*

Performance when using SSDs: As a further demonstration of C3's generality, we also perform measurements with m3.xlarge instances, which are backed by two 40 GB SSD disks. We configured a RAID0 array encompassing both disks. We reduced the dataset size to 150 million 1KB records in order to ensure that the dataset fits the reduced disk capacities of all nodes. Given that with SSDs, the system can

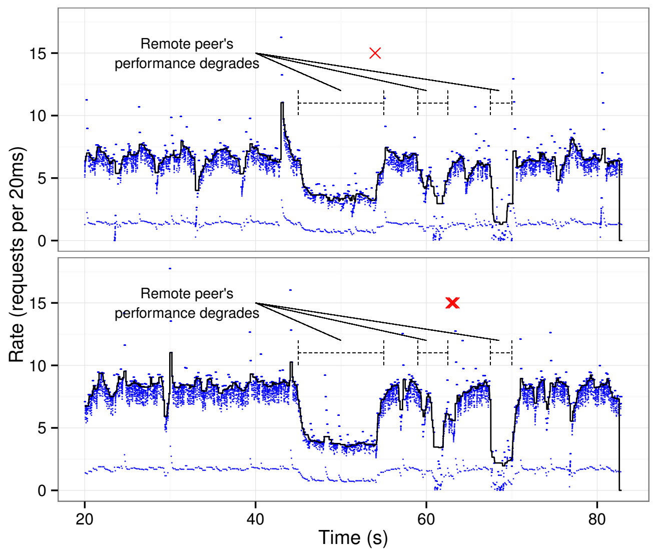


Figure 4.13: Sending rate adaptation performed by two coordinators (top and bottom) against a third common server. The receiving server's latency is artificially inflated thrice. The blue dots represent the sending-rates as adjusted by the cubic rate control algorithm, the black line indicates a moving median of the sending rates, and the red X marks indicate moments when affected replica group schedulers enter backpressure mode.

sustain a higher workload, we used 210 read-heavy generators (70 threads per YCSB instance). Figure 4.12 illustrates the latency improvements obtained when using C3 versus DS with SSD backed instances. Even under the higher load, both algorithms have significantly lower latencies than when using spinning head disks. However, C3 again improves the 99.9^{th} percentile latency by *more than 3x*. Furthermore, the difference between the 99^{th} and 99.9^{th} percentile latencies in C3 is *under 5 ms*, whereas with DS, it is on the order of 20 ms. Lastly, C3 also improves the average latency by roughly 3 ms, and increases the read throughput by 50% of that obtained by DS.

Sending rate adaptation and backpressure over time: Lastly, we turn to a seven-node Cassandra cluster in our local testbed to depict how nodes adapt their sending rates over time. Figure 4.13 presents a trace of the sending rate adaptation performed by two coordinators against a third node (*tracked node*). During the run, we artificially inflated the latencies of the tracked node thrice (using the Linux `tc` utility), indicated by the drops in throughput in the interval (45, 55) s, as well as the two shorter drops at times 59 s and 67 s. Observe that both coordinators' estimations of their peer's capacity agree over time. Furthermore, the figure depicts all three rate regimes of the cubic rate control mechanism. The points close to 1 on the y-

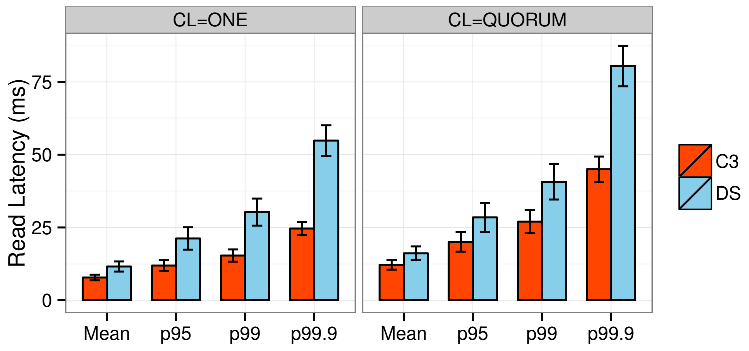


Figure 4.14: Impact of consistency level: Mean, 95th, 99th and 99.9th percentile read latencies for read consistency levels ONE and QUORUM. The prefix p in the x-axis labels refers to a percentile.

axis are arrived at via the multiplicative decrease, causing the system to enter the low-rate regime. At that point, C3 aggressively increases its rate to be closer to the tracked saturation rate, entering the saddle region (along the smoothened median). The stray points above the smoothened median are points where C3 optimistically probes for more capacity. During this run, the backpressure mechanism fired 4 times (3 of which are very close in time) across both depicted coordinator nodes. Recall that backpressure is exerted when *all* replicas of a replica group have exceeded their rate limits. When the tracked node's latencies are reset to normal, the YCSB generators throttle up, sending a heavy burst in a short time interval. This causes a momentary surge of traffic towards the tracked node, forcing the corresponding replica selection schedulers to apply backpressure.

Impact of consistency level: We also present an evaluation to demonstrate the impact of different consistency levels in Cassandra. Recall that Cassandra clients can decide, based on their requirements, how many replicas need to acknowledge each read and write request. While all the measurements shown so far made use of consistency level ONE, we also present results when using consistency level QUORUM. With the QUORUM consistency level and a replication factor of three, two replicas are read from for each client-initiated read request. These experiments were conducted on an eight node test cluster at Spotify (cluster configuration appears in § 4.4.3). We used eight Cassandra nodes with 250 million 1KB-sized records, leading to 100GB per node for this experiment. As with the previous experiments, we used three YCSB instances, each with 40 workload generator threads. Figure 4.14 shows the read latency measurements for mean, 95th, 99th and 99.9th percentiles when using a consistency level of ONE and QUORUM. We find that C3 improves the latency distribution across both consistency levels. At consistency level ONE, with this smaller cluster size, we find that the 99.9th percentile latency with DS is *2.2* *times* that of C3's. For QUORUM reads, the 99.9th percentile latency when using DS is *1.78 times* higher than when using C3.

4.4.2 Evaluation Using simulations

We turn to simulations to further evaluate C3 under different scenarios. Our objective is to study the C3 scheme independently of the intricacies of Cassandra and draw more general results. Furthermore, we are interested in understanding how the scheme performs under different operational extremes. In particular, we explore how C3's performance varies according to *(i)* different frequencies of service time fluctuations, *(ii)* lower utilization levels, and *(iii)* under skewed client demands. We also use simulations to analyze the impact of concurrency compensation, the queue-penalization parameter, and the choice of the rate control algorithm.

Experimental setup: We built a discrete-event simulator, wherein workload generators create requests at a set of clients, and the clients then use a replica selection algorithm to route requests to a set of servers. A request generated at a client has a uniform probability of being forwarded to any replica group (that is, we do not model keys being distributed across servers according to consistent hashing as in Cassandra). The workload generators create requests according to a Poisson arrival process, to mimic arrival of user requests at web servers [114]. Each server maintains a FIFO request queue. To model concurrent processing of requests, each server can service a tunable number of requests in parallel (4 in our settings). The service time each request experiences is drawn from an exponential distribution (as in [143]) with a mean service time $\mu^{-1} = 4$ ms. We incorporated time-varying performance fluctuations into the system as follows: every T ms (*fluctuation interval*), each server, independently and with a uniform probability, sets its service rate either to μ or to $\mu \cdot D$, where D is a range parameter (thus, a bimodal distribution for server performance [123]). We set the D parameter to 3 (qualitatively, our results apply across multiple tested values of D). The request arrival rate corresponds to 70% (high utilization scenario) and 45% (low utilization scenario) of the average service rate of the system, considering the time-varying nature of the servers' performance (that is, as if the service rate of each server's processor was $(\mu + D \cdot \mu)/2$). As with our experiments using Cassandra, we use a read-repair probability of 10% and a replication factor of 3, which further increases the load on the system. We use 200 workload generators, 50 servers, and vary the number of clients from 150 to 300. We set the one-way network latency to 250 μs. We repeat every experiment 5 times using different random seeds. 600,000 requests are generated in each run.

We compare C3 against three strategies:

1. **Oracle (ORA):** each client chooses based on perfect knowledge of the instantaneous q/μ ratio of the replicas (no required feedback from servers).

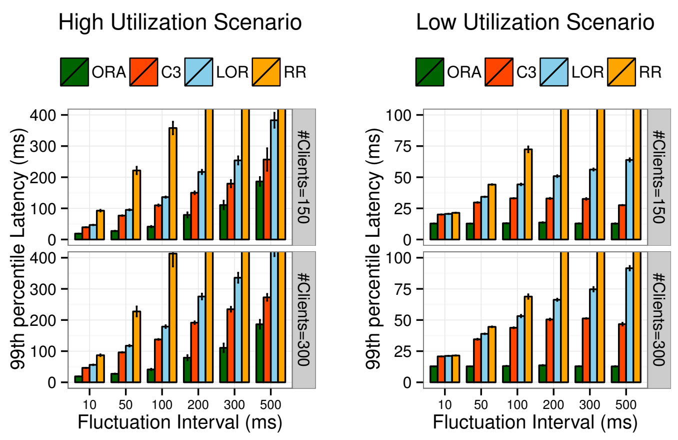


Figure 4.15: Impact of time-varying service times at high-utilization *(left)* and low-utilization *(right)* scenarios. Bars exceeding 400 ms are not shown.

2. **Least-Outstanding Requests (LOR):** each client selects a replica to which it has sent the least number of requests so far.

3. **Round-Robin (RR):** as in C3, each client maintains a per-replica rate limiter. However, here it uses a round-robin scheme to allocate requests to replicas in place of C3's replica ranking. This allows us to evaluate the contribution of just rate limiting to the effectiveness of C3.

We also ran simulations of strategies such as uniform random, least-response time, and different variations of weighted random strategies. These strategies did not fare well compared to *LOR*; hence, we do not present results for them. We do not model disk activity in the simulator, and thus avoid comparing against Dynamic Snitching (since it relies on gossiping disk `iowait` measurements).

Impact of time-varying service times: Given that C3 clients rely on feedback from servers, we study the effect of the service time fluctuation frequency on C3's control over the tail latency. Figure 4.15 *(left)* presents the 99^{th} percentile tail-latencies under a high utilization scenario for C3 and other compared strategies with 150 and 300 clients. Here, the arrival rate is set to match 70% of the system utilization. When the average service times of the servers in the system change every 10 ms, C3 performs similarly to *LOR* and *RR*. This is expected, because at such a high frequency of performance variability, clients can make use of one round-trip's worth of feedback for at most another request, before that information is stale. However, as the interval between service time changes increases, *LOR*'s performance degrades more compared to that of C3. Furthermore, the performance of *RR* suggests that rate-limiting alone does not improve the latency tail. This

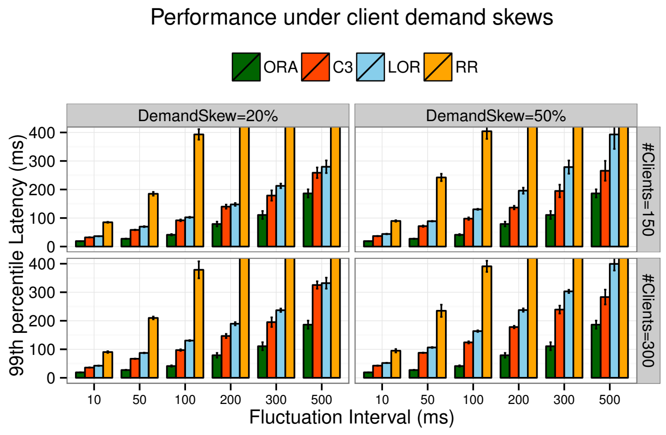


Figure 4.16: Impact of demand skews: 20% and 50% of the clients generate 80% of the requests to the servers. Bars exceeding 400 ms are not shown.

is because *RR* does not proactively prefer faster servers.. We also note that C3's performance remains relatively close to that of *ORA*.

Performance at low utilization: While C3 is geared towards high-utilization environments with a number of requests in flight [114, 126], we now demonstrate the efficacy of C3 under low-utilization settings as well. We set the arrival rate to match a 45% system utilization. Figure 4.15 *(right)* presents the 99th percentile tail-latencies under this low utilization scenario. While the performances of *LOR* and *RR* degrade with higher fluctuation intervals, C3's performance begins to plateau instead. This is because a client using *LOR*, will allocate requests to slow servers as long as it has assigned more requests to other replicas (or by chance when the number of outstanding requests is zero and/or identical across all replicas). This leads to poor allocations as initially explained in Figure 4.1. In contrast, C3 monitors service times presented by different servers and uses servers in proportion to their products of queue size and service times to balance the latency distribution. Thus, unlike C3, the longer a server remains slow, the higher the chance that it will receive requests when clients use the *LOR* strategy. This weakness of *LOR* results in higher tail latencies than C3.

Performance under heavy demand-skews: We study the effect of heavy demand skews on the observed latencies. Figure 4.16 presents results when 20% and 50% of C3 clients generate 80% of the total demand towards the servers, respectively. Again, we find C3 at an advantage because of its ability to measure and react to relative performance differences among servers. Thus, despite the demand skew, C3 outperforms *LOR* and *RR*.

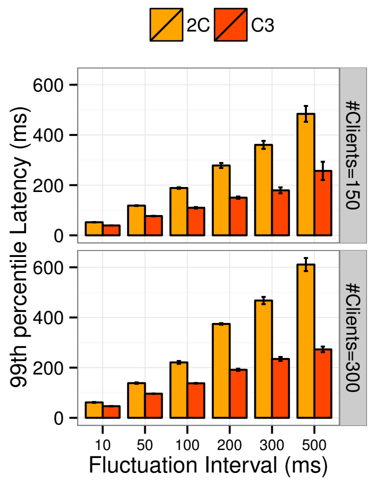


Figure 4.17: Alternative approach to concurrency compensation. A replica selection heuristic that uses a randomized, two-choices selection based on the queue size and service time feedback from the server (2C), fares poorly compared to C3's approach of desynchronizing based on the outstanding requests from a client to a server.

Impact of concurrency compensation: To deal with the challenge of concurrency from high fan-ins, C3's replica ranking heuristic makes use of the number of outstanding requests from a client to a server as a desynchronization mechanism (§ 4.2). An alternative approach we evaluated was to use the queue size and service time feedback from the server as in § 4.2, but instead, avoid the concurrency compensation part of C3's heuristic. Instead, inspired by [104], we use a randomized two choices selection to desynchronize clients. In this case, for each replica ranking decision, we randomly select two servers from the replica list and then pick the replica with the best score according to the ranking heuristic. In Figure 4.17, we compare C3 against such a heuristic (labeled *2C* in the figure) for the high-utilization scenario described earlier. The results indicate that the *2C* heuristic fares poorly compared to C3's use of the number of outstanding requests. We attribute this two factors. First, is C3's use of additional information in the form of outstanding requests to each server. Second, a two choices selection presents limited opportunity for desynchronization in settings with low replication factors, which is typical of cloud data stores (recall that we use a replication factor of 3). Indeed, in a setting without performance fluctuations (that is, service time averages of servers remain constant), we find that both schemes perform comparably (Figure 4.18).

Impact of the queue penalization parameter, *b*: We perform a sensitivity analysis of the queue length penalization factor used for the replica ranking heuristic

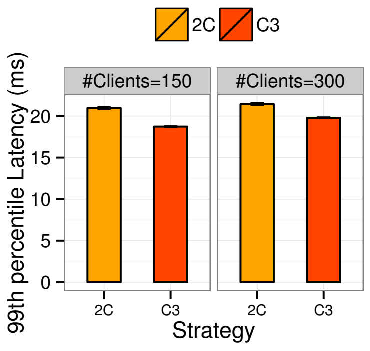


Figure 4.18: Alternative approach to concurrency compensation. In the absence of performance fluctuations, the 2C heuristic (see Figure 4.17) fares comparably to C3.

(b parameter). In all of our reported results so far, we have used a fixed value of $b = 3$. We make use of the high-utilization scenario described above, and vary this parameter through values from one to four. Figure 4.19 indicates the 99th percentile latencies across different values for the b parameter choice. We note that the 99th percentile latency is fairly robust to the choice of b in the scenarios involving 150 clients. With 300 clients however, the fan-in per-server for the same demand is higher, and each client now operates on more uncertainty in selecting servers. With this higher degree of uncertainty, clients benefit from switching replicas sooner (and thus, higher values of b). At 300 clients, we see improvements exceeding 100ms at fluctuation intervals of 500ms, when moving from $b = 1$ to higher values. This is because a long queue length at a server before it slows down for an extended interval will harm the latency tail more. Given that this effect is more prominent at higher fluctuation intervals, penalizing longer queue lengths with higher values of b leads to improvements in the latency tail.

Choice of rate-control algorithm: We also present an evaluation involving alternate rate-adaptation algorithms. We experimented with the following rate adaptation schemes in place of C3's TCP-CUBIC inspired approach: *(i)* simple additive increase, multiplicative decrease (AIMD), *(ii)* an adaptation of TCP BIC [149] (which is similar to CUBIC), *(iii)* the FAST-TCP based approach used by PARDA [79]. All three approaches in TCP literature are window-based, which were adapted to be rate-based (as we have done for C3 in § 4.2.2). AIMD and BIC use the same congestion signal as C3 (that is, periodically, if the receive rate does not match the sending rate, it is treated as congestion). These three approaches are thus demand clocked. FAST-TCP on the other hand is not demand clocked. It sets a "target latency", and increases the rate as long as the response time is less than the target latency.

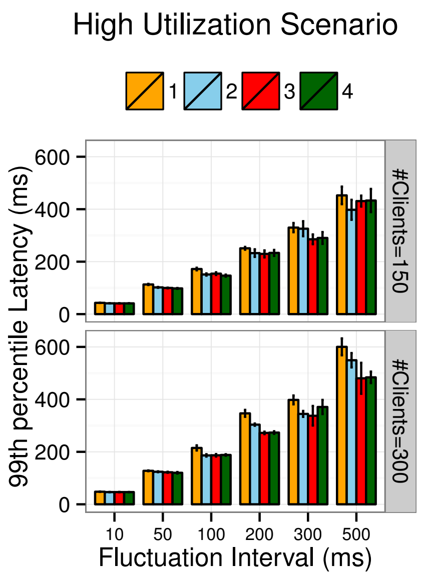


Figure 4.19: Impact of the b parameter in the replica ranking function. The penalization of longer queue lengths affects performance when the fluctuation intervals are longer. We see improvements when switching from $b = 1$ to higher values, with more than 100ms of improvement at fluctuation intervals of 500ms.

Our objective is to study how a sudden increase in system demand affects the queue evolution at the server-side. We simulate a scenario involving ten clients and ten servers, operating at a low overall utilization of 15%. 10 seconds into the simulation, three clients receive a surge of requests that drives system utilization to 60%. We refer to these clients as the *high-rate clients*, whereas the remaining clients are the *low-rate clients*. We observe the evolution of the server's queue length when using the different rate adaptation algorithms. Figure 4.20 depicts the queue length over time for one server from the moment the request surge appears at the high-rate clients (all servers exhibited qualitatively identical profiles). We note that with AIMD and BIC, the server's queue length exhibits a sustained spike. C3's CUBIC approach leads to a short-lived spike up to 20 requests, whereas with FAST, we do not note any perturbation in the timeseries. The reason these approaches exhibit different outcomes is because of their convergence properties, which were in turn a function of their parameter choices. With AIMD and BIC, the high-rate clients that receive the traffic surge grow their client-side backlog queues while the rate-control algorithm is yet to fully ramp up. This is evidenced by the lack of server-side queue occupancy seen between $t = 10s$ and $t = 11s$ for AIMD and BIC (which C3 and FAST do not have). What follows after $t = 11s$ is a sustained drain of these backlog

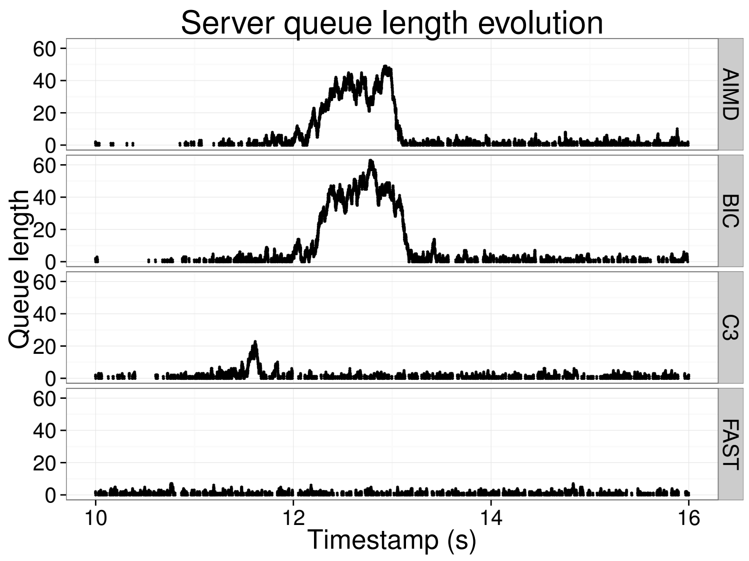


Figure 4.20: Impact of rate-adaptation algorithm: Server queue length evolution.

queues when the sending rates converge, causing the queue size spikes observable in Figure 4.20. C3's CUBIC approach however, ramps up faster than AIMD and BIC, leading to lower backlog queue sizes at the clients. Due to the lower backlog queue sizes, as the clients' rate limits converge to match the increased demand, the drain phase has a less severe impact on the server's queues. On the other hand, we find FAST to be extremely sensitive to the target latency setting. For the results shown in Figure 4.20, we set the target latency to be high (out of reach of the load at which the system was running). This means that clients never exert any backpressure, and do not exhibit a ramp-up phase unlike the other approaches. Hence, the backlog queues never increase at the clients, and this "instantaneous convergence" at the clients receiving the traffic surge leads to the lack of spikes at the server (since the overall utilization of 60% is still well under the system capacity). Indeed, due to these effects, the 99th percentile latencies for the low-rate clients are lower with C3 and FAST (19.43ms and 19.05ms respectively), than for AIMD and BIC (26ms and 25.75ms).

We do not interpret these differences as merits of the C3 and FAST scheme over AIMD or BIC. On the contrary, we believe that any of these approaches can be tuned and adapted to target either rapid-convergence or a slower ramp-up, depending on system requirements. What we emphasize instead is that the *convergence properties* of the rate adaptation algorithm is the dominating factor in determining queue evolution at the server.

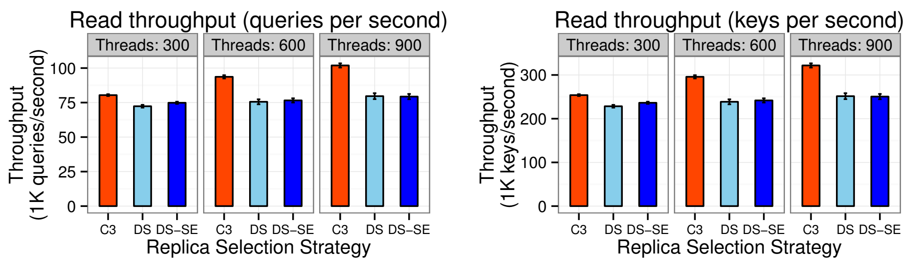


Figure 4.21: SoundCloud experiment: read throughput at different loads (number of workload generator threads).

4.4.3 Evaluations against production workloads

To further validate C3, we turned to experimentation against production workloads. We will now discuss results from experiments conducted at two companies that operate Cassandra clusters: SoundCloud and Spotify. The experiments in the next two sections were conducted on Cassandra clusters running on dedicated physical machines.

Evaluation at SoundCloud

In the evaluations conducted so far, we observed C3's efficacy when performing single-key reads. In this section, we discuss a performance evaluation of C3 conducted at SoundCloud using a production workload that involved *multi-key reads*. We will refer to a multi-key read as a *query*.

Experimental Setup: We used a 16-node Cassandra cluster at SoundCloud. Each server had 32 CPU cores (Intel(R) Xeon(R) CPU @ 2.00GHz), 64 GB RAM, a 150GB SSD which hosted the root filesystem and a separate 750GB SSD which hosted Cassandra's data folder. The servers hosted production datasets of approximately 230 GB per node, spread across a large number of column families each with a replication factor of 3. We used a home-built workload replay tool in order to generate read-requests against the cluster that followed the characteristics of production traffic. The SoundCloud service whose read pattern was replayed generates multi-key read queries that request between one and twenty keys. Clients issue a single query to a Cassandra coordinator picked in a round-robin manner. The coordinator retrieves all requested records before returning a response to the client. Thus, the overall query time will be bottlenecked by the slowest read request executed as part of the query. The column family against which we tested

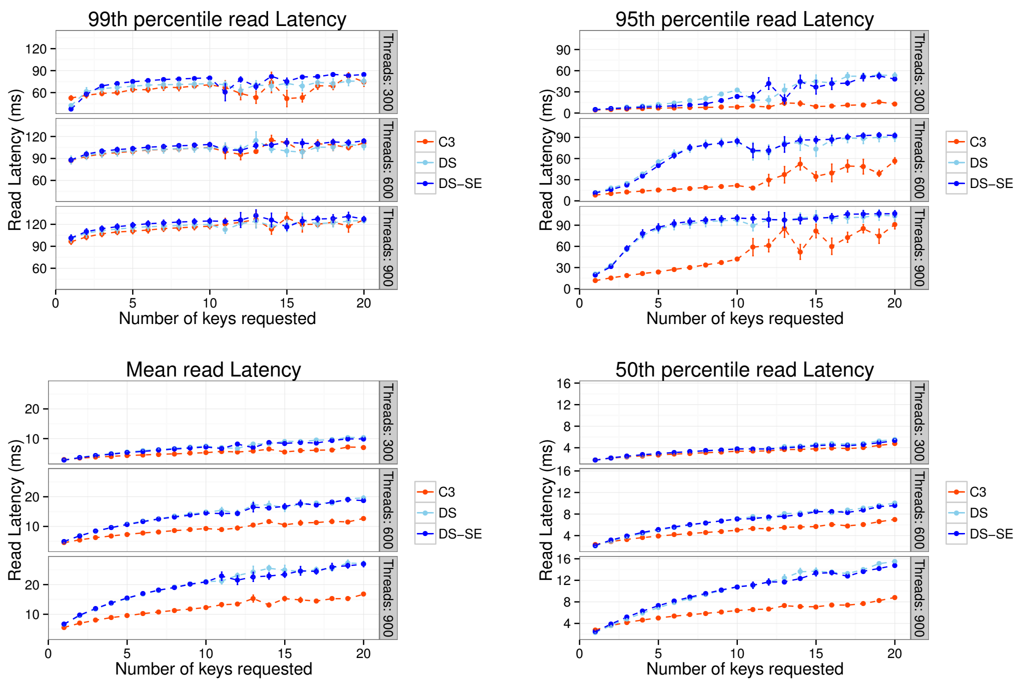


Figure 4.22: SoundCloud experiment: query latency by number of keys requested across different loads (number of workload generator threads).

our workload amounted to roughly 130GB per node. Given that the column family hosts timeseries data, the data has variable sized records. We used four machines to drive the workload generator against the Cassandra cluster. Reported latency measurements are recorded from one of the workload generators machines, whereas throughput measurements are recorded from all workload generators. All reported metrics correspond to the overall completion time and throughputs for queries from the perspective of the clients. We conducted experiments using a total of 300, 600 and 900 client threads. We compare C3 against Dynamic Snitching (DS), and Dynamic Snitching with speculative execution enabled at the 99th percentile (DS-SE). With speculative execution, the coordinator retries a request if it does not receive a response within the 99th percentile latency of the column family being queried. We use a consistency level of ONE for all results.

Results: We find that C3 improves overall system throughput across the various levels of system load. At a workload generator thread count of 900, C3 improves the query read throughput by 25% (Figure 4.21). This translates to an additional 25,000 queries per second (75,000 keys per second). Furthermore, while sustaining the higher throughput, we find that C3 improves the mean, median and 95th percentile latencies (Figure 4.22). For queries that requested 20 keys, at a load of 300 client threads, we observe that DS and DS-SE experience roughly *3.5 times* the 95th percentile latency as that of C3. Indeed, in the setting with 300 client threads, we see DS and DS-SE experience higher 95th percentile latencies with an increase in the number of keys requested because the system is nearing saturation (whereas C3 exploits the combined throughputs of available replicas effectively).

However, we note that for all the tested loads, C3, DS and DS-SE present similar 99th percentile query-latencies. To understand why, note that all the algorithms work at the granularity of individual key fetches, as opposed to entire queries. The query latency is bottlenecked by the slowest read request executed as part of the query. That said, DS-SE reissues requests to another replica if it does not receive a response within the 99th percentile latency of the concerned column family. Yet, we observe that at all the loads tested, DS-SE did not significantly improve latencies at the 99th percentile and above over DS. The fact that DS-SE fails to improve the query latency despite using request reissues suggests that the *choice of replica* is not the bottleneck for queries that appear at the 99th percentile of the query latency distribution. Given that the dataset being accessed comprises timeseries data, records have variable lengths. Thus, if a high latency episode occurs due to reading a larger record size (as opposed to sporadic performance variability), the request will experience a similar latency regardless of the choice of replica. This explains why, with the available dataset and workload, all schemes deliver similar latency profiles at the 99th percentile and above (we observed this for the 99.9th percentile as well).

Evaluation at Spotify

Lastly, we report on an evaluation of C3 against a production workload at Spotify.

Experimental Setup: The Cassandra cluster used for the evaluation comprised 8 nodes, with 16 cores, 32GB of RAM and spinning disks in a RAID 10 configuration. The Cassandra cluster hosted a production dataset from a service with a replication factor of 3, resulting in roughly 130GB of data per node. A sampled subset of production workload from 6 application servers were redirected to the C3-enabled Cassandra cluster hosting the production dataset. The machines sending the workload had 32 cores and 64GB of RAM each. The workload is thus open-loop; we observed it led to a disk utilization of 50-60% on the Cassandra nodes. The requests have a read-write ratio of 97% reads and 3% writes. The accessed column family comprises a variable number of fields (each 30 to 40 characters long), the distribution for which is indiciated in Figure 4.24. All read requests fetch all fields in the row corresponding to a single given key. The application servers used the Astyanax-based [8] Cassandra clients with a consistency level of ONE, and routed requests to Cassandra nodes in a round-robin manner.

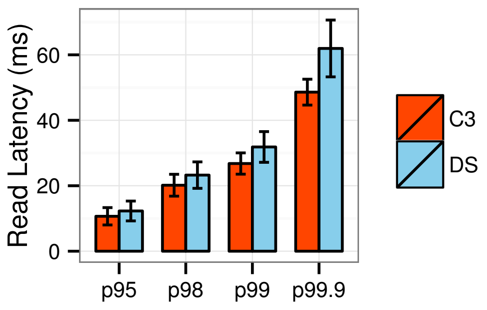


Figure 4.23: Spotify experiment: read latency at various percentiles with CL=ONE. The prefix *p* in the x-axis labels refers to a given percentile.

Results: Figure 4.23 presents results observed at the various latency tails. C3 improves the latency tail at the 99.9th percentile by up to 27% while also presenting lower variability than DS. At the other observed percentiles, C3 presents only a marginal improvement. We attribute this to two factors. First, the load at which the clusters are operated is modest with only six application servers generating traffic, leading to low queuing delays, thus presenting limited opportunity for C3 to improve latencies at the lower percentiles. Second, the high variability of record sizes may influence the service time measurements that C3's replica ranking relies upon.

During this study, we also evaluated the possibility of introducing C3 into token-aware Cassandra clients (instead of the coordinator running the algorithm). The clients thus send requests directly to the node hosting the key. Running C3 on the Cassandra coordinators while using token-aware clients in this setting yields limited opportunity for improvements. This is because the workload uses single-key reads, and the coordinator node hosting the data is very likely to serve the request without forwarding it (thus, not invoking the C3 mechanism at all). For instance, Figure 4.25 shows the same experiment as above, repeated with token-aware clients performing QUORUM reads. The key obstacle in the way of implementing C3-based token-aware clients is the modification of the client-server wire protocol, which prevented us from further exploring this direction at Spotify.

4.5 Discussion

How general is C3? C3 combines two mechanisms in order to carefully manage tail latencies in a distributed system: ***(i)*** a load-balancing scheme that is informed by a continuous stream of in-band feedback about a server's load, and ***(ii)*** distributed rate-control and backpressure. We believe that these insights can be applied to

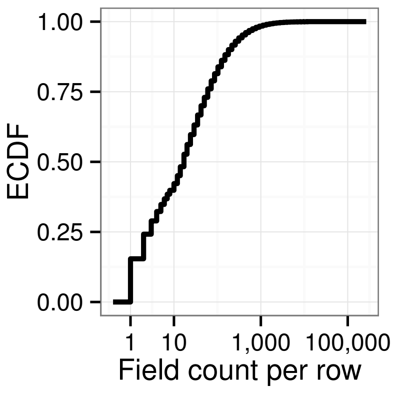


Figure 4.24: Spotify experiment: distribution of number of fields per row in the accessed column family.

any low-latency data store that can benefit from replica diversity. Furthermore, our simulations compared C3 against different replica selection mechanisms, and allowed us to decouple the workings of the algorithms themselves from the intricacies of running them within a complex system such as Cassandra. Lastly, insights from C3's replica ranking mechanism has influenced the design of ELS [14], a latency-based load balancer built at Spotify for use against their backend services. ELS uses a similar heuristic as C3, albeit without server side feedback. Load balancers use the product of the number of outstanding requests to each server (q) and an estimation of the end-to-end latency (L) to rank replicas. Similar to C3's b parameter, the ELS heuristic also raises the q term to a higher exponent to penalize longer queues ([14] reported using $b = 3$ as well). This further points to the generality of our approach.

Long-term versus short-term adaptations: A common recommended practice among operators is to over-provision distributed systems deployed on cloud platforms in order to accommodate performance variability [22]. Unlike application servers, storage nodes that handle larger-than-memory datasets are not easily scaled up or down; adding a new node to the cluster and the subsequent re-balancing of data are operations that happen over timescales of hours. Such questions of provisioning sufficient capacity for a demand is orthogonal to our work; our objective with C3 is to carefully utilize *already provisioned* system resources in the face of performance variability over short timescales.

Strongly consistent reads: Our work has focused on selecting one out of a given set of replicas, which typically applies to environments where eventual consistency is used. This maps to common use-cases at large web services today, including Facebook's accesses to its social graph [141] and most of Netflix's Cassandra usage [91].

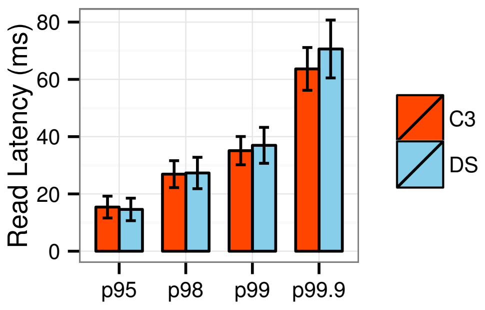


Figure 4.25: Spotify experiment: ready latency at various percentiles when using token-aware clients with CL=QUORUM. The prefix *p* in the x-axis labels refers to a given percentile.

However, it remains to be seen how our work can be applied to strongly consistent reads as well. In particular, the gains in such a scenario depend on the synchronization overhead of the respective read protocol, and the effect of a straggler cannot be easily avoided.

Multi-tenancy and differentiated services: A dimension that we do not cover in this work is that of multi-tenancy, as well as differentiated services for traffic from different tenants or applications. There are several challenges on the way to supporting multi-tenancy for systems such as Cassandra. This includes delivering isolation at various layers of the stack including request handling, thread-pool usage, sharing of various caches, as well as isolation of and protection from background activities triggered by different tenants. At the time of this study, the open-source NoSQL data-stores we considered did not have full support for multi-tenancy (including Cassandra [39]). Not surprisingly, for this reason, most Cassandra operators we spoke to deploy one cluster per-application type. Assuming, however, that a data-store does provide strong isolation properties, one can expect per-tenant request queues and fair scheduling at each server (as in [126]). In such settings, clients of a given tenant can receive queue size and service time feedback corresponding to these per-tenant queues, and perform replica selection accordingly.

Stability: To understand the stability properties of the C3 scheme, we considered the use of fluid models [97]. Our approach to modeling replica selection using a fluid model involved defining a set of differential equations that describe the evolution of queues at the servers, as a function of *i)* the service rates of each server, and *ii)* how clients distributed requests (the fluid) across the servers. Once this set of equations is defined, they can be numerically evaluated to study the queue size evolution at each server (as in Figure 4.20). There are several challenges in applying this approach to studying C3. The key difficulty is in capturing the behavior of C3 clients, since they track several metrics *over time* in order to perform replica ranking. This includes, for each server, the end-to-end response time, the service time and queue size, as well as the number of outstanding requests from the client. Expressing the time-dependent, stateful, and discrete nature of these metrics accurately in a fluid model proved to be challenging. We therefore leave this for future work.

4.6 Summary

In this chapter, we highlighted the challenges involved in making a replica selection scheme explicitly cope with performance fluctuations in the system and environment. We presented the design and implementation of C3. C3 uses a combination of in-band feedback from servers to rank and prefer faster replicas along with distributed rate control and backpressure in order to reduce tail latencies in the presence of service time fluctuations. Through comprehensive performance evaluations, we demonstrate that C3 improves Cassandra's mean, median and tail latencies (by up to 3 times at the 99.9^{th} percentile), all while increasing read throughput and avoiding load pathologies.

5
End-to-end resource allocation and scheduling for micro-services

In Chapter 4, we focused on a two-tier setting comprising storage clients and servers. In this chapter, we zoom out and focus on a more general setting, namely, multi-tenant distributed systems composed of small services: Service-oriented Architectures (SOA) and Microservices. For simplicity, we henceforth refer to such systems as SOAs.

In recent years, many organizations like Netflix, Amazon, Uber, SoundCloud, Google and Spotify have adopted Service-oriented Architectures (SOAs) [70] – and their refinement, Micro-services [112] – to build large-scale Web applications [32, 105, 109, 128, 138] as well as infrastructure systems [20, 142]. SOA systems consist of fine-grained, loosely coupled *services* that communicate via lightweight API calls over the network. Every service comprises multiple *service instances* or *processes*, each running inside a physical server or virtual machine. For instance, Netflix has separate services for managing movie data, user data, authentication, and recommendations [107]. Typically, these divisions align with developer team structures [113]. These systems generally serve multiple *tenants*, where a tenant may represent different external customers or consumers of the service, but also internal product groups, applications, or differentiated background tasks.

Requests from different tenants compete for *shared* resources at service instances. Tenants and operators in this setting care about diverse performance objectives, including guaranteed throughput, fairness, meeting deadlines, ensuring priorities, and achieving low latencies. As we elaborate below, SOAs have characteristics that lead to unique challenges in satisfying these objectives.

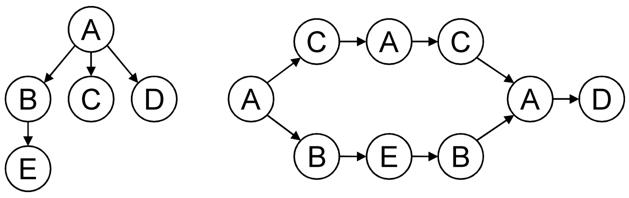


Figure 5.1: ***(Left)*** Example illustrating a set of five services, and ***(Right)*** an execution path of a request across these services. The requests execute in two parallel branches, followed by a call to service D. Service A calls C twice.

Complex request-response characteristics. Request execution in SOAs spans tens to hundreds of services, each comprising tens to thousands of instances with processing times at each tier ranging from sub-milliseconds to minutes [88]. Consider the example of Figure 5.1. The sequence of service instances traversed by a request (i.e., its *execution graph*), and its resource demands on different services are generally unknown when the request first enters the system (at node A in the figure). These depend on multiple factors like the APIs invoked at each hop, the supplied arguments, the content of caches, as well as the use of load balancing along the graph.

Limited end-to-end visibility. By design, dependent services in SOAs have little visibility into each other. This makes it difficult for a given service to infer issues when they arise in its immediate as well as *transitive* dependencies. Without end-to-end mechanisms, services can at best only observe and react to overload at their dependencies by observing characteristics like high latencies or the number of request timeouts [57, 108]. The loss of information as requests cross process boundaries makes it challenging to schedule and manage resources end-to-end to meet different performance objectives.

These two characteristics intersect to complicate resource management in SOAs. Several case studies highlight how complex interactions in SOAs not only lead to sharp degradation in performance (e.g., lower throughput and higher latencies), but also trigger cascading behaviors that ultimately result in wide-spread application outages [31, 81, 85, 136]. To quote Steve Yegge, *"Every one of your peer teams suddenly becomes a potential DoS attacker"* [151]. Furthermore, existing libraries [108, 137] for building SOAs require extensive tuning of static thresholds for rate limiters and circuit breakers [57]. Setting these thresholds manually in a complex distributed system is fragile and becomes out of date quickly as systems evolve [90, 110]. These challenges are but symptoms of a more pressing problem: that is, the lack of adaptive, *end-to-end* resource management and scheduling for SOAs. While resource management and scheduling have been widely studied in other system domains, such as network packet scheduling [65, 102, 125], task scheduling in big-data systems [76, 152], and process and IO scheduling [38, 79, 133, 135], the SOA domain has received surprisingly little attention.

Our objective is to achieve *end-to-end performance objectives across communicating services in SOA systems on metrics of concern to tenants as well as operators.* As an example, consider an SOA application with two classes of APIs: C_1, for interactive user-facing requests, and C_2, for submitting logs and performance metrics. The provider might want to execute C_1 requests with latency deadline of 500ms and C_2 requests with aggregate throughput of 1000 requests per second. Our goal is to provide simple abstractions to enforce such end-to-end policies through local scheduling, rate limiting, and back pressure mechanisms at individual processes.

In this chapter, we illustrate the fundamental challenges to flexibly achieving diverse performance objectives in SOAs such as effective overload management, performance isolation and fairness across different tenants as well as meeting end-to-end request deadlines. We address these challenges with the design and implementation of Cicero, a framework for building multi-tenant SOA systems with end-to-end resource management and scheduling capabilities. Towards this end, we make the following design choices:

- avoid centralized coordination across all services to maintain autonomy of each service and only exchange minimal information across services,
- operate without a-priori knowledge of request execution paths and resource demands in each service,
- support a wide range of scheduling goals and policies such as achieving high resource utilization, performance isolation, and meeting deadlines.

Solution overview. We achieve our goals via a novel composition of multiple components that complement each other. First, each instance of a service monitors the rates of request arrival, utilization of local resources, and the number of calls to downstream services for each tenant. Second, requests are tagged with metadata such as the tenant ID, deadline, or work executed so far.

The above two components of our approach give us enough visibility into request execution across the service DAG. We then build upon this information to perform fine-grained, policy-based resource management. Every instance of a service in Cicero locally determines rate limits for each tenant according to a resource management policy. Cicero then uses a novel distributed rate adaptation protocol to automatically set rate limits at every upstream service that respects rates expressed via local policies at downstream services, allowing the system to quickly react to bottlenecks that manifest at any point along request execution paths. Furthermore, the metadata propagation enables request scheduling inside each service instance that can execute a wide range of common resource scheduling policies such as fair queuing across tenants and end-to-end "aware" variants of shortest remaining time first (SRTF) [60] and least slack time first (LSTF) [86].

Importantly, Cicero separates policy from mechanism: we provide APIs to express policies that manage rate limiters and request schedulers at each process to meet end-to-end objectives. In our evaluations, we demonstrate meeting several performance objectives using Cicero, such as avoiding cascading failures, meeting end-to-end deadlines (10x improvement in the 99th percentile latency/deadline ratio), and isolating low latency clients from high throughput tenants (2x improved average latencies).

5.1 Context, objectives and challenges

In this section, we provide relevant background on Service-oriented Architectures (SOAs) and explain the resource management challenges in them.

5.1.1 SOA characteristics

Services and processes. In an SOA, a single application such as Amazon's e-commerce site or Netflix' streaming service is composed of multiple *services*. These services are typically maintained by separate teams (or even third parties) and communicate exclusively over well-specified APIs [112]. See an illustration in Figure 5.1. Each service typically runs multiple instances (essentially, OS processes) that are distributed across multiple servers or virtual machines. Requests are dispatched to instances based on the type of service; for example, requests can be load-balanced among processes of a stateless service, whereas routing in stateful services is typically based on some form of hashing. While a request typically enters the system through an *entry point* service such as a set of front-end web servers (e.g., A in the figure), requests may also originate from internal systems that access shared infrastructure services.

Workflows. We define a *workflow* as a set of requests that belong to the same class or tenant and are bound by the same resource management criteria. We reason about multi-tenancy at the granularity of different workflows. For instance, Netflix's streaming devices interact with their backend via different APIs pertaining to user data, recommendations, movie data as well as ratings [107]. The activity triggered by each of these API calls may be categorized as a workflow each (therefore enabling each API call to have different end-to-end performance objectives such as deadlines). Similarly, background system activity (such as geo-replication) may be classified as a separate workflow as well.

Workflow execution DAGs are complex and opaque. Request execution in SOAs are best represented as a DAG of visited services. For instance, consider the Bing SOA, which is made up of a hundred services and wherein workflows have different execution DAGs [88]. The DAGs are both *deep* – 20% of the workflows contain sequences of at least 10 services – and *many-way parallel* – 10% of services aggregate responses from 9 other services. A crucial aspect of request execution in

SOAs is the *opacity* of this execution graph and its corresponding resource consumption to each service. That is, a service is typically oblivious to *(i)* the end-to-end execution graph of the request, which depends on load balancing, multiple levels of caching, number of instances per service, and API parameters used to invoke different services, *(ii) request amplification*, wherein a single request at an upstream service might correspond to thousands of requests at a downstream service, *(iii) request cost*, where different requests at an upstream service may have varying costs further downstream; for instance, the cost of loading an object at a storage service is proportional to the object size and is potentially unknown when an *application-level request* first enters the system at an entry point. Lastly, workflow characteristics may change as the codebase for individual services evolve, further aggravating to the opacity of request execution graphs [90].

5.1.2 Goals of Cicero

Our objective is to flexibly enforce diverse multi-tenant resource management policies in the SOA setting. Policies and objectives we consider within scope include:

High utilization and avoiding overload. Avoiding wasteful over provisioning is a desirable goal for data center operators [62]. At the same time, it is important to avoid the dangers of overload, which can result in cascading failures due to complex interactions [81, 85].

Fairness and performance isolation. Fair sharing of resources at every instance of a service is a desirable property in large systems. Multiple notions of fairness are of interest here, including bottleneck fairness [100], DRF [74], and weighted fair sharing [125].

Minimum throughput guarantees. A shared service may be provisioned to offer a subset of users minimum throughput guarantees [63], with other users receiving best effort service.

Differentiated services. Requests from different workflows may need to be scheduled at each hop according to pre-defined priorities [62]. For instance, traffic from paying users may receive higher scheduling priority at every service. Alternatively, activity triggered by interactive, user-facing workflows often require higher priorities over background workloads [62].

End-to-end deadlines and latency SLOs. Different workflows may have varying end-to-end deadlines, typically determined by SLOs [63]. As an example, user facing web-sites often need to load a page in under 100-400ms for a fluid user experience [127].

5.1.3 Challenges of regulating load and latencies

We now discuss how (*i*) the lack of end-to-end visibility and (*ii*) the complex workflow DAG structures in SOAs make it challenging to realize the above policies. We first describe several production outages that highlight the challenges we focus on in this work.

Outages in production. Visual Studio Online experienced an outage [85] caused by an interaction of two different request types in a hierarchy of services. A single workflow was accessing a slow database deep in the service hierarchy. The blocking RPC calls from the upstream service eventually exhausted the service's thread pool. This subsequently starved *other unrelated* requests contending for the thread pool to initiate connections towards an authentication service, causing widespread application unavailability. A similar interaction across tiers led to an Amazon AWS outage [81]. Services therefore have to be aware of potential bottlenecks among their downstream services. In another episode, a performance degradation of some Amazon EBS instances triggered a sequence of bottlenecks in related services. During the firefighting effort, operators *manually intervened* to aggressively throttle upstream EC2 service APIs to reduce load on the downstream EBS service, which affected more customers than necessary [31]. Such manual intervention is error prone and challenging to reason about.

Regulating system load. Based on the issues observed in the above production systems, we now illustrate the fundamental challenges of regulating system load through a concrete example. Consider the set of services from Figure 5.1 (left) and two workflows w_1 and w_2; w_1 executes in A, followed by B and then E, and returns, while w_2 only executes in A. When w_1 and w_2 execute together and service E becomes congested (EBS slowdown), service A (upstream EC2 front end) is not aware of the downstream bottleneck and also does not know if w_1 (specific EC2 API call) uses that bottleneck or not. Also, when w_1 requests execute in A but timeout in congested E, the work they executed in A and B is wasted. Rate limiting w_1 at the entry point to A would reduce wasted resources *and* increase the throughput of w_2 since more resources in A would be available. However, reasoning about wasted resources, goodput, and fairness across tenants is difficult because of the limited visibility and complex execution patterns. Since workflows contend for shared resources in the system, their respective rate limits at every hop have *mutual dependencies* and therefore cannot be tuned independently of each other. Furthermore, the exact values of these rate limits depend on characteristics such as how requests are routed internally in the system, how requests amplify or fan-out between tiers and how costly these requests are at different components. Identifying these rate limits in a dynamic distributed system with limited global information is thus challenging.

Achieving low end-to-end latency and deadlines. Scheduling request processing at every hop in the system to meet *end-to-end* latency guarantees is challenging due to the complex structure of execution DAGs and the inherently stochastic nature of the problem. More specifically, achieving latency goals depends on the processing times for each workflow at every service and stages of their execution DAGs [88]. In the above example, if w_1 has a 300ms end-to-end deadline and requires 250ms of processing time at E, it only has a budget of 50ms to complete A and B. Despite myriad existing scheduling algorithms to prioritize requests with different performance objectives (such as shortest remaining time first (SRTF) [60] and least slack time first (LSTF) [86]), *realizing* these policies in a fully distributed setting remains non-trivial. These algorithms rely on information such as the remaining processing time and slack to deadlines, which need to be dynamically estimated across diverse workflows. While literature exists on achieving end-to-end latency guarantees in communication networks [117, 129, 130, 134], their underlying models are simpler (e.g., single path vs. DAG, single resource vs. multiple resources), and cannot directly apply to our context.

5.2 The Cicero system

Cicero is a framework for building SOAs that supports end-to-end resource management and scheduling for workflows. It transparently monitors workflow characteristics and resource utilization per-workflow at each process to: *i)* enforce rate-limits *end-to-end*: ensuring that rate-limits at upstream services reflect bottlenecks and workflow behavior further downstream, and *ii)* prioritize request execution at each service based on end-to-end characteristics of the request, such as deadlines and the amount of remaining processing time.

5.2.1 Design space

Intuitively, we can view this problem through the lens of network congestion control. However, the workflow rate-limiting problem in SOAs differs from the network congestion control problem in important ways. In TCP, sources perform rate-limiting and coordinate with the endpoints directly for flow-control. They then infer congestion in the network either indirectly through congestion signals (such as packet loss and latency) or through explicit feedback [2, 67] to tune their sending rates. Endpoints of a network flow are fixed (this is true even for multicast congestion control [26]). These assumptions do not hold in an SOA. Upstream services do not know about their transitive dependencies, and due to effects such as request amplification, caching and routing, subsequent requests of the same workflow may be processed by entirely different downstream services (and instances). Furthermore, different workflows may have end-to-end completion times that vary by orders of magnitude, unlike end-to-end RTTs in networks. These properties make it challenging to reason about system stability and convergence when designing source-based congestion control algorithms. Another alternative is to use centralized coordination [100], which, however, makes it challenging to make *per-request* (as opposed to per-workflow) scheduling decisions while accounting for complex workflow DAG structures. As we demonstrate, a fully distributed design overcomes this limitation.

5.2.2 Cicero components

Our design has three core requirements: *(i)* avoid centralized coordination, *(ii)* exchange minimal information between services, *(iii)* operate without prior knowledge of a workflow's costs and graph structure. At the same time, we want Cicero to provide building blocks that enable operators to enforce flexible system-wide policies depending on their requirements with minimal tuning.

Per-process local model per-workflow. Each instance of a service in Cicero maintains a *minimal, local, per-workflow model* of request execution, which automatically adapts to workload changes. In our implementation, processes maintain *average* rates of arrival, load on local resources, and the number of calls to downstream services (request amplification) by workflow (Table 5.2). Our approach can also be extended to use more sophisticated prediction techniques [38].

Metadata propagation. Requests are tagged with metadata that enable distributed scheduling to achieve different performance objectives. The propagated metadata includes the workflow ID, the elapsed time so far, the deadline (if any), as well as per-workflow statistics we estimate on-the-fly such as the total service time and cost associated with each request, and per-request estimations of the remaining processing time.

Distributed rate- and admission- control. Every process uses a local rate-limiting policy in order to identify admission rates for requests of different workflows. To ensure that workflows are rate-limited as *early as possible* instead of only being throttled at the point of congestion (§5.1.3), we use a distributed rate adaptation algorithm (§5.2.4) wherein each service instance announces per-workflow rate limits to its upstream services. With every service executing this algorithm, upstream services are informed about per-workflow rate-limits that match bottlenecks further downstream. Furthermore, we perform admission control in order to avoid wasting resources on requests that will not complete within their deadlines.

Request scheduling and sub-tasks. At each process, an admitted request from a workflow triggers one or more *sub-tasks* (similar to Hystrix's *commands* [108]). Developers build their micro-services as a composition of sub-tasks. Sub-tasks at a service instance may in turn call further downstream services. These sub-tasks invoke a scheduler that enforces diverse scheduling policies. For example, the scheduler may enforce fair queuing across workflows on local resources such as disks, CPU, or connection pools, to guarantee performance isolation. Alternatively, sub-tasks may be ordered based on scheduling policies such as shortest job first (SJF), earliest deadline first (EDF), or least slack time first (LSTF), which are used to optimize for a range of end-to-end performance goals.

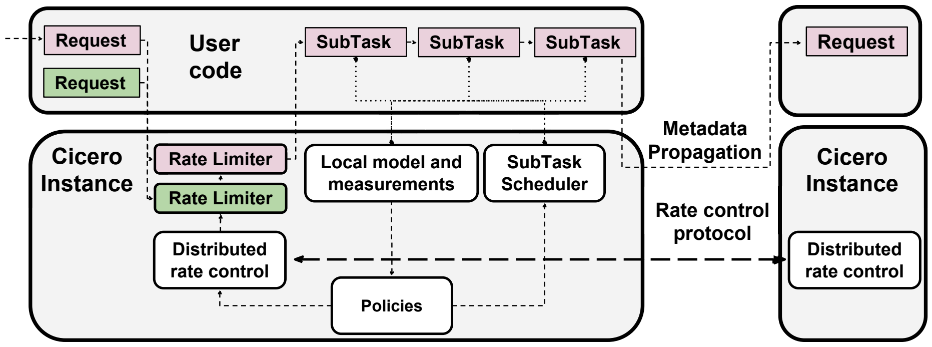


Figure 5.2: Cicero architecture. Policies examine resource utilization by different workflows locally and determine rate limiting and sub-task scheduler behavior. Distributed rate control automatically ensures that upstream rate limits reflect downstream bottlenecks. Metadata propagation enables end-to-end scheduling policies.

5.2.3 Cicero architecture

Figure 5.2 depicts Cicero's architecture, which includes user code, operator policies, and the Cicero core.

User code. Developers building micro-services express APIs and business logic as a *composition* of sub-tasks. As an example, a payment gateway service might execute an external API for managing a credit card payment using three sub-tasks in sequence: fetching the relevant user object based on the API call, processing the payment order, and then finalizing a credit card payment. Developers use Cicero's APIs to assign workflow IDs to requests and sub-tasks, register resources (and their capacities) to be monitored by Cicero, and provide hints to the scheduler such as request deadlines. We omit showing example user code and APIs due to space restrictions.

Operator policies. Operators in Cicero specify rate limiting policies that execute at each hop. These policies run periodically and output rate limits per-workflow after observing resource utilization locally. The policies set rate limits to match resource sharing objectives such as fairness or minimum throughput guarantees. Furthermore, operators also specify local scheduling policies that prioritize sub-task execution according to objectives such as meeting deadlines. Table 5.1 shows example APIs for writing rate limiting and sub-task scheduler policies.

Cicero core. The core bridges user code and operator policies. As a user's sub-tasks execute, they acquire and release resources such as connections from a connection pool and threads from a threadpool. Each request in Cicero has a context object propagated with it, which holds necessary metadata required for the operator policies

`CreateRateLimiter(workflow, < properties >)` Create a new rate limiter for a workflow
`Get/SetRateLimiterProperty(workflow, < properties >)` Get/Set admission rate, max burst size or max queue size for a rate limiter
`GetAdmissionRatesFromRemote(workflow, process)` Retrieve admission rates from a downstream process
`GetResourceUtilization(workflow, resource)` Get the resource utilization for a workflow against a resource
`GetWorkflowArrivalRate(workflow)` Get the arrival rate for requests of a workflow
`GetSubtaskCost(workflowId, subtaskId)` Get the costs of a sub-task type by workflow
`Get/SetRequestMetadata(reqCtx, < metadata >)` Get/Set metadata for a request (examples in Table 5.3)
`DropRequest/Subtask(reqCtx)` Drop request or sub-task processing (to be handled by user)
`ConfigureResourceSemaphore(resource, < properties >)` Configure concurrency and queue size thresholds for semaphores

Table 5.1: Example APIs used by resource sharing and scheduling policy developers

such as the workflow ID, elapsed time, deadline and metrics that estimate request progress (§ 5.2.4). Furthermore, Cicero monitors utilization of the local resources and infers properties of the workflows (§5.2.4). The metadata propagation and local model inform resource management decisions by the operator policies (§5.2.5). The distributed rate adaptation then automatically adapt rate limits at upstream processes to match rate limits at downstream services, while factoring in workflow characteristics (§5.2.4). The sub-task scheduler is invoked between each execution of a sub-task, wherein it prioritizes sub-tasks based on the scheduling policies specified by the operator (§5.2.4).

5.2.4 Cicero core

Measuring workflow and resource properties

Resources in Cicero are protected by semaphores to limit concurrency [108]. As sub-tasks from different workflows execute, they acquire and release these semaphores when using different resources. The duration for which a semaphore is held is the service time. We measure and maintain exponentially weighted moving averages (EWMA) of the service times st against each resource by workflow. The total service time consumed by a workflow in a measurement interval gives us the demand of

w_j	Workflow j
s_n	Service n, defined as a set of processes
p_i	Process i
$\sigma_{i,j}$	Admission rate of w_j at p_i
$\mathcal{U}_i$, $\mathcal{D}_i$	Set of upstream and downstream services for p_i
$\alpha_{i,d,j}$	Amplification factor at p_i to downstream service d for w_j
$\mathcal{R}_i$	Set of resources in process p_i
$c_{i,k}$	Capacity of resource k in p_i
$l_{i,j,k}$	Average load on resource k in p_i by w_j

Table 5.2: Notation used for algorithm description

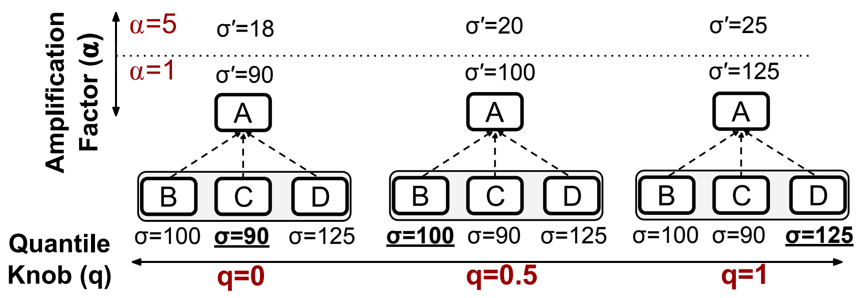


Figure 5.3: Rate aggregation: influence of quantile knob (q) and amplification factor (α) in how upstream services aggregate downstream admission rates (σ). α for a workflow is measured automatically by the system, whereas q is adjusted by an operator (see §5.3.5 for a sensitivity study).

that workflow against a resource. Cicero maintains EWMAs of measurements such as, *(i)* arrival rates of requests per-workflow, *(ii)* the number of further calls per-request to downstream services or amplification factor (α), *(iii)* the local costs per workflow (l), and *(iv)* the average completion time of a workflow once admitted (L). Importantly, we track *averages* of these measurements over a control interval. These measurements inform resource allocation decisions by the rate control component and the sub-task scheduler (Figure 5.2). Our approach does not preclude using more sophisticated prediction techniques or exogenous models of workflow behavior [38].

Distributed rate adaptation

We now describe the distributed algorithm in Cicero to adapt per-workflow rate limits system-wide. We compute the rate of workflow w_j on each process p_i as a minimum of the local bottleneck rate at the process and bottleneck rate across all downstream services. In our approach, every process p_i periodically (by default 100ms) executes two steps in sequence (**Algorithm 3**). First, p_i queries processes of its downstream services ($d \in D_i$) for their admission rates per workflow w_j (that is, a vector of rates per service d, $\{\sigma_{d,j}\}$). The rates announced by individual processes of a service are aggregated into a single rate *per service*. From these rates per service, the minimum is chosen for each workflow, yielding the downstream bottleneck rate per-workflow σ'_j. Next, p_i computes rate limits according to its local resources based on a policy such as bottleneck fairness or minimum throughput guarantees (§5.2.5). p_i then sets its announced σ_j to the minimum of its local and downstream rate limits. Every iteration of these steps *bubble up* admission rates through the service topology, with processes of upstream services enforcing admission rates that match downstream bottlenecks. We now discuss the rate aggregation in detail.

Rate aggregation. The aggregation phase is responsible for bubbling up rate limits, and is key to Cicero maintaining efficient resource utilization. In lines 1-2 of Algorithm 3, each process polls all processes of its downstream services for their admission rates (σ). Recall that requests in an SOA can amplify (§5.1), wherein a single admitted request at a process can trigger multiple outgoing requests to downstream services. Line 1 corrects for this effect by scaling the downstream rate in proportion to their *average* amplification factors (α). Lastly, line 3 reduces all the admission rates per-workflow from different processes of a service, into a single per-workflow rate limit, σ'_j. The calculation of a single rate for an entire service is parameterized by a quantile knob $q = [0, 1]$. Given that processes compute their local admission rates based on observed demand, load and system capacity, different processes of a downstream service may not announce identical admission rates (this can happen due to differences in capacity across services because of skewed workloads, failures or throughput reduction caused by background activities). The q parameter allows an upstream process to be pessimistic or optimistic in the face of such uncertainty. Figure 5.3 shows the influence of q and α in the rate-aggregation process. The extreme values of q regulate the system load to either minimize wasted work and "match the slowest link" ($q = 0$, Figure 5.3 (left)) or maximize utilization by setting rates according to the highest admission rate from the downstream service's processes ($q = 1$, Figure 5.3 (right)). $q = 0$ leads to Cicero executing a *min* aggregation of the admission rates from the downstream services up to the entry points, and $q = 1$ performs a *max* aggregation. In our experiments, we use $q = 0.5$ (Figure 5.3 (center)) to perform a median aggregation, and we also demonstrate the sensitivity to this parameter (§5.3). For scenarios where the number of processes in a service is small, the quantile selection may use a linear interpolation between the values of ranks closest to the quantile. Once the downstream bottleneck rate σ'_j is calculated, the process executes a local policy (§5.2.5) to determine local bottleneck rates per workflow, $\sigma^{local}_{i,j}$. Lastly, for each workflow w_j, the algorithm finalizes its announced admission rates $\sigma_{i,j}$ to be the minimum of $\sigma^{local}_{i,j}$ and σ'_j. When polled, downstream processes return a $\sigma_{i,j}$ to a parent in proportion to the demand arriving from the parent.

Algorithm 3 Rate adaptation at p_i (Every 100ms)

```
   Constants: $q$: quantile parameter | Initialize $\{\sigma'_j\} \leftarrow \infty$
1: for all $d \mid \mathcal{D}_i$ do                                  ▷ In parallel
2:     $\{\sigma_{d,j}\} \leftarrow \text{query}(d)/\alpha_{i,d,j}$          ▷ Scale by amplification factor
   // Admission rate of downstream service is $q^{th}$ quantile of $\{\sigma_{d,j}\}$
3:     for all $w_j$ do
4:         $\sigma'_j \leftarrow min(\sigma'_j, \text{quantile}(\{\sigma_{d,j}\}, q))$
5: $\{\sigma^{local}_{i,j}\} \leftarrow LocalResourceSharingPolicy()$
6: for all $w_j$ do
7:     $\sigma_{i,j} \leftarrow \min(\sigma^{local}_{i,j}, \sigma'_j)$
```

Metadata	**Used by**
Workflow ID	Rate limiters, fair queueing, resource accounting
Elapsed service time ($st^{elapsed}$) *	LSTF, SRTF
Total service time (st^{total}) *	LSTF
Work so far ($\theta^{elapsed}$) *	LASF
Total work (θ^{total})*	SJF
Deadline, δ	EDF, LSTF, drop logic
Start time	LSTF, drop logic

Table 5.3: Metadata usage (*estimated using progress metrics)).

Metadata propagation

Cicero not only taints requests with metadata to supply relevant execution context but also to estimate request progress. With progress estimation, we infer metrics such as the total service time or total consumed resources by a request *as it executes* in a distributed environment. All requests are tagged with a workflow ID that is used to route requests to their respective rate-limiters, an end-to-end deadline δ and the start timestamp.

Cicero by design operates without prior knowledge of a workflow's costs and execution DAG structure. However, as discussed in §5.1.3, to use scheduling algorithms such as SRTF [60] and LSTF [86] to meet end-to-end latency objectives, Cicero needs to track metrics such as the total and the remaining service time for each request *as they execute*. Note, the total service time is different from the average *end-to-end latency* experienced by a request; the service time does not include queuing delays at each hop. This is important, because if a request needs only 1ms of processing at an instance, but the same type of request experienced 100ms of queuing delay in the past, it mischaracterizes the request's priority for algorithms such as SRTF or LSTF. We therefore need to estimate metrics per-request that reflect its true execution progress (*progress metrics*). A progress metric can be queried for the total end-to-end estimate, elapsed and remaining values at any point in the request execution. We track two progress metrics: the service time *st* and the total load θ (demand divided by capacity), which enable multiple scheduling algorithms (Table 5.3).

Progress metrics. Our solution (details forthcoming) to track the elapsed *st* and θ emulates the standard recursive algorithm to compute the sum of values of all vertices in a DAG. However, request execution DAGs can have both parallel and sequential stages (§5.1.1). For metrics such as the *remaining service time st*, the *sum* of service times measured at every branch of a parallel execution across processes over-estimates the true service time left in the system. To correct for this, we sum metrics across sequential stages, and in parallel stages, we divide the sub-tree sums by the degree of parallelism. On the other hand, to track the total cost of a request (in terms of θ), a sum gives the expected result.

Progress metric update algorithm. Consider a metric m that we want to estimate end-to-end (where m is the service time or load). We propagate two counters per metric, a_m and b_m along with the request. a_m holds the total value of the metric up to the time the request was received at a process. b_m tracks the total value of the metric in the currently executing process and its downstream processes. At any point in time, the *elapsed* estimate of m is equal to $a_m + b_m$. The intuition is that we keep updating b_m at each hop and when execution inside a downstream process completes, we update b_m at the parent. The counters start as $(a_m + b_m, 0)$ at each hop, with the values of $a_m = 0$ and $b_m = 0$ corresponding to the initial state at the entry points. When a sub-task of a request is successfully serviced at a resource, we update b_m by the relevant metric. For instance, if we are tracking the elapsed service time (st) and we service a request for 6ms, we update (a_{st}, b_{st}) to $(a_{st}, b_{st} + 6)$. When an execution of p parallel downstream executions complete and control is resumed at the parent process, the parent updates its local values of (a_m, b_m) to $(a_m, b_m + \sum b_i)$ if the metric is a sum type metric (like when computing the total load θ for the request), or to $(a_m, b_m + \sum b_i/p)$ for metrics such as the service time. This recursive approach works consistently across both sequential and parallel DAGs. When a request's execution completes end-to-end at the entry point, the resulting values for (a_m, b_m) for every metric m yields the required total T_m, which is maintained at the entry points as an EWMA. For future invocations of the workflow, the total value T_m is propagated along with the request. Scheduling algorithms that use the remaining value of a metric m (such as SRTF and LSTF) estimate it as $T_m - (a_m + b_m)$ at any instant.

Sub-task scheduling

While rate limiting regulates the average load on each process, sub-tasks triggered by the arrival of a request at a process are executed by local schedulers. These schedulers may enforce performance isolation between sub- tasks of different workflows (to say, protect low latency tasks from head-of-line blocking in the presence of throughput heavy workloads [62]) or prioritize their execution based on end-to-end performance objectives (differentiated services or deadlines). Local scheduling interleaves sub-tasks of different workflows in *time*, and are essential to Cicero's ability to meet end-to-end latency and deadline goals (§5.3.2).

5.2.5 Operator policies

Policies to compute rate limits. Operators may choose to provide static throughput guarantees, calculate rates based on bottleneck fairness, or receive feedback from the local queue schedulers and resources. As long as every process exposes its per-workflow admission rates to its upstream neighbors, *the rate aggregation mechanism transparently ensures that upstream processes converge to rate limits that factor in downstream restrictions* (§5.2.4). We implemented a bottleneck fairness policy (**Algorithm** 4) similar to [100]. With this policy, each process p_i checks if a local resource is bottlenecked. If not, it ramps up the announced admission rates σ for every workflow for which p_i is a leaf service (no further downstream services), by an additive probe factor β, scaled according to the amount of spare capacity available (this increases the rate faster when there is spare capacity available and is more conservative otherwise) (line 5). For non-leaf workflows, p_i inherits the aggregated downstream rate σ'_j (line 7). If instead a local resource is bottlenecked, the system calculates max-min fair shares for the contending workflows (line 9).

Local scheduling policies. We now discuss multiple scheduling policies realized using our framework. We implemented a multi-resource fair queuing scheme similar to [126]. Fair queuing across workflows is particularly useful at protecting short and bursty workflows that do not benefit from rate limiting (§5.3.1). The scheduler uses the deficit round-robin algorithm [125], wherein every workflow gets a number of credits per-round and credits are consumed based on the expected cost of the sub-task. A fixed number of credits per-round are budgeted across each workflow in proportion to the shares per-workflow (computed via a bottleneck fairness allocation or via DRF [74]). To meet end-to-end deadlines, our LSTF scheduling policy strives to favor sub-tasks with the least remaining slack (§5.3.2). We also realized scheduling algorithms such as LASF [115], which favors short-lived workflows when the system has bursty arrivals. All scheduling policies are enabled by progress metrics and other metadata propagated via the requests. Algorithm 5 illustrates the priority assignment procedure for different scheduling algorithms.

Admission control and drop policies. To regulate lengths of different queues in the system (rate limiters, scheduler, and resource queues), requests need to be dropped on time according to different policies. For instance, the drop policy we use across all our experiments ensures that when a request from a workflow w_j arrives at a rate-limiter in p_i (shaping at rate $\sigma_{i,j}$), the rate-limiter only queues

Algorithm 4 Bottleneck Fairness Policy at p_i

```
    Constants: $\beta$: Probe factor for increasing rate
 1: $k' \leftarrow \text{argmin}\{\phi_k = \frac{c_{i,k} - \sum_{j=1}^{|\mathcal{W}|} l_{i,j,k}}{c_{i,k}}\}$        ▷ Spare capacity fraction
 2: if $\phi_{k'} \geq 0$ then
 3:     for all $w_j$ do
 4:         if leaf node for $w_j$ then
 5:             $\sigma_{i,j}^{local} \leftarrow \sigma_{i,j} + \beta(1 + \frac{\phi_{k'}}{|w_j|l_{i,j,k'}>0|})$        ▷ Ramp-up
 6:         else
 7:             $\sigma_{i,j}^{local} \leftarrow \sigma'_j$        ▷ Inherit rates from downstream (Alg 3)
 8: else
 9:     $\{shares_j\} \leftarrow \text{MaxMinFairness}(\{l_{i,j,k'}\}, r_{k'})$
10:     for all $w_j \in l_{i,j,k'} > 0$ do
11:         ratio $\leftarrow \{shares_j\}\frac{c_{i,k}}{l_{i,j,k'}}$
12:         if ratio $\geq 1$ then
13:             $\sigma_{i,j}^{local} \leftarrow \sigma_{i,j} + \beta$        ▷ Under-utilizing fair share
14:         else
15:             $\sigma_{i,j}^{local} \leftarrow \sigma_{i,j} \cdot$ ratio        ▷ Exceeding fair share
```

a request enough such that it remains feasible to meet the deadline. The policy computes the maximum tolerable queuing delay for a request as $delay^{max} = \delta_{req_j} - (ElapsedTime(req_j) + L_{i,j})$. If the current backlog on w_j's rate-limiter exceeds $delay^{max} * \sigma_{i,j}$, it drops the request. That is, each service locally uses the estimated completion time ($L_{i,j}$) for a request of the incoming workflow, the elapsed time of the request, as well as the request's deadline to calculate a maximum tolerable queuing delay. If this delay exceeds the expected queuing delay on the rate-limiter, the request is dropped. Cicero thus drops requests that have little chance of completing within their deadlines, freeing up resources for other requests. Note, when we drop a request or sub-task, Cicero simply returns control back to user code (which may then gracefully degrade service [49]).

5.2.6 Implementation

Our prototype is implemented as a C# library and makes heavy use of C#'s primitives for asynchronous programming. For now, our prototype passes request metadata explicitly as a context object parameter throughout the system. However, aspect instrumentation [100] will allow this metadata to be propagated transparently. As in [108], Cicero uses the notion of semaphore protected calls to monitor service times against different resources, and uses these measurements to inform resource management policies. Cicero however can also be extended to accommodate

Algorithm 5 Priority Scheduling at p_i

```
1: function ENQUEUE(Workflow $w_j$, SubTask $t$)
2:     $Heap_{w_j}$.add($t$, Priority($t$) )
3: function LSTF-PRIORITY($req$)
4:     $\{st_{req}^{remaining} \leftarrow st_{req}^{total} - st_{req}^{elapsed}$
5:     return $\delta_{req} - (time_{req}^{elapsed} + st_{req}^{remaining})\}$
6: function EDF-PRIORITY($req$)$\{$return $\delta_{req} - time_{req}^{elapsed}\}$
7: function SRTF-PRIORITY($req$)$\{$return $st_{req}^{total} - st_{req}^{elapsed}\}$
8: function LASF-PRIORITY($req$)$\{$return $st_{req}^{elapsed}\}$
```

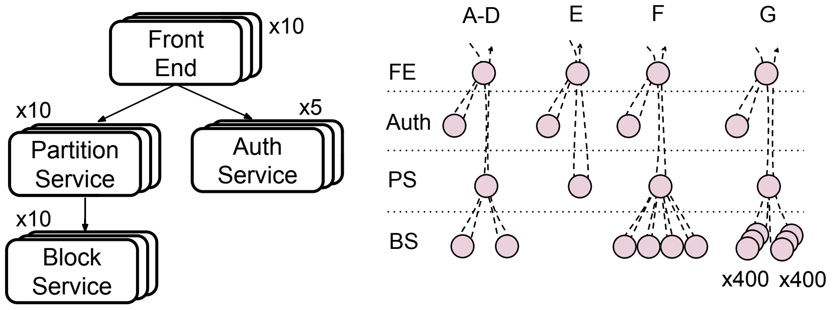


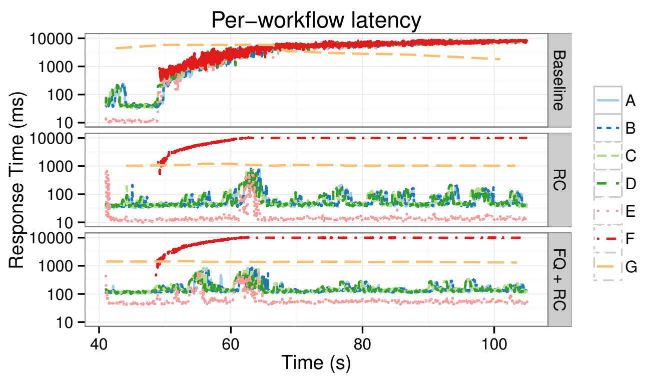
Figure 5.4: *(Left)* Cloud data store topology used in the evaluation. Workflows traverse four services (FE, Auth, PS and BS). Every request to a PS triggers multiple read/write sub-tasks to the BSs as well as local computations. *(Right)* Workflow DAGs used in performance isolation experiment.

diverse resource monitoring and accounting techniques as used by [38,100,126,135], which we consider out of scope for this chapter.

5.3 System evaluation

In this section, we demonstrate how Cicero enforces different resource management policies: *(i)* Avoid overload and provide isolation in the presence of aggressive workflows (§5.3.1), *(ii)* Meet end-to-end deadlines (§5.3.2), *(iii)* Isolate low-latency traffic from high throughput traffic (§5.3.3). We further stress Cicero's distributed rate adaptation by showing *(iv)* how it reacts to hotspots (§5.3.4), and demonstrate *(iv)* how operators can navigate the goodput vs utilization trade-off using the quantile knob q (§5.3.5).

Experimental Setup. We run our experiments on a testbed comprising forty virtual machines. Each VM has a single 2.40 GHz CPU core, 2GB of RAM and runs Windows Server 2012 R2. All services make use of the .NET CLR version 4.5. Each instance of a service in our experiments runs as a process inside a VM.



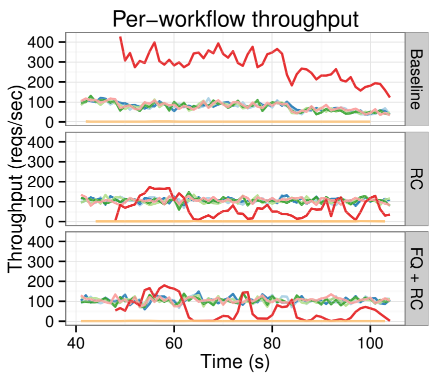


Figure 5.5: Performance isolation experiment. (*Top*) The latency timeseries of the experiment smoothened with a moving median shows how the aggressive workflow (F, red) drives the system into overload. End-to-end rate-control throttles F at the ingress and protects other workflows. (*Bottom*) Throughput obtained by all workflows. In the baseline, throughputs of all workflows gradually degrade as all requests time out.

We demonstrate Cicero's effectiveness using the service topology of a large production cloud storage system that exhibits complex DAGs of operations (Figure 5.4). We emulate the characteristics of the production system (request routing, execution DAGs, and sub-task costs), as opposed to running Cicero within the original system. The system comprises four tiers: front-end (FE), authentication (AUTH), partition service (PS), and block storage (BS) services. FEs are the entry points that accept client requests. FEs first verify client requests against an AUTH server. They then route the request to a PS that holds the table for a tenant, determined via consistent hashing. The PS process then issues a sequence of reads and writes to the BS service before executing compute work locally and returning results. The workflow DAG thus exhibits multiple stages. Our setup comprises ten FEs, five AUTH servers, ten PS instances and ten BS servers. Cicero monitors resource utilization by workflow across all thread pools and connection pools in the system. In addition, we emulate

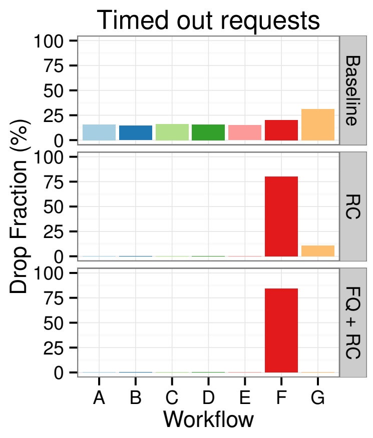


Figure 5.6: When running rate-control only, the bursty tenant is not guaranteed performance isolation as the steady workflows dominate system queues. A fair-queuing scheduler resolves this.

a disk resource at the BS servers as well as a local compute stage at the PS. For these two emulated resources, we vary the service times for different workflows to study different bottleneck scenarios (service times are drawn from exponential distributions). We drive client workloads from five VMs. Every workflow has a fixed number of PS partitions. Each PS partition corresponds to a fixed number of blocks on the BS tier, uniformly distributed across all available BS processes. Open-loop clients generate requests according to a poisson process [114] and are bursty.

5.3.1 Can Cicero enforce performance isolation?

The workflow DAGs for this experiment are indicated in Figure 5.4 (right). Workflows *A*-*D* are read-write workloads with an arrival rate of 100 reqs/sec each at the FEs. Every request from these workflows at a PS triggers a read and write request to the BS in sequence followed by some compute work at the PS. Workflow *E* issues metadata queries that are serviced locally by the PS without any interactions with the BS layer. Workflow *F* is an aggressive tenant's workload generated by four clients that exceeds their fair share at the BS tier. Workflow *G* is bursty traffic with an arrival rate of one request every two seconds. Each request from workflow *G* triggers 400 reads in parallel, followed by 400 writes to the BSs, each of which consumes a disk resource for 200ms on average.

We compare performance across three scenarios: *(i)* Timeouts only, where the system only makes use of deadline based timeouts and does not use fair-queuing or rate-control (baseline). *(ii)* end-to-end rate control only (RC), and *(iii)* end-to-end rate control and local fair scheduling with per-workflow FIFO queues (FQ+RC).

Figures 5.5 demonstrates system behavior across the three scenarios. When workflow F, the aggressive tenant, activates at time t=50s, the resulting overload drives all workflows to throughput collapse. Given the C# RPC library's request timeout of ten seconds[1], requests queue up internally in the system, blocking different resources and thus inflating latencies for all workflows. Instead, with Cicero's rate-adaptation (Figure 5.5, top) this is not the case. The rate-limiting throttles workflow F whereas other workflows retain their expected throughput. Since F is an open-loop workload that does not lower its sending rate despite being throttled, its throughput exhibits the instability seen in the oscillations in Figure 5.5 (bottom). However, rate-control alone does not guarantee fair access to local resources at each service. This means that workflows with a stable rate have a higher degree of presence across system-wide resource queues, which cause bursty workloads to suffer from head-of-line blocking. Workflow G thus suffers because of the higher queue occupancies from the other workflows, indicated by a timeout fraction of 12% (Figure 5.6). Instead, with the combination of per-service fair-scheduling and rate-control, G is guaranteed progress at each stage. The key take-away here is that thresholds such as timeouts are challenging to set correctly system-wide with an end-to-end performance objective in mind. In practice, such thresholds are often hard-coded [100] and therefore fragile. On the other hand, Cicero automatically adapts rate limits based on dynamic system conditions.

5.3.2 Can Cicero meet end-to-end deadline targets?

We replay a trace of 30K requests from a production instance of the large cloud storage system, which involves a mix of different APIs. Given that the traces do not indicate deadlines per request, we measure the completion times for each request in isolation. We then correct for the higher system loads in the experiment by setting the deadline to four times the base completion time when measured in isolation. The workload includes APIs that trigger scans involving *thousands* of sub-tasks and multiple API calls that only contend for the local resource at the PS. We vary the number of client threads that generate these requests in a closed loop from 100 to 250 to increase system load.

We compare results across a baseline, FIFO+RC, EDF+RC and LSTF+RC. Table 5.4 indicates different percentiles for the latencies normalized by the deadline (LND). An LND of 1.0 implies that the latency was equal to the deadline. At higher loads, the baseline incurs increased latencies and thus misses deadlines by large factors. At the highest tested load of 250 clients generating requests, the baseline's average LND is 1.33, while the 95th percentile reaches as high as 4.83. The improvement in LND is evident as soon as we switch on rate-limiting (FIFO+RC), since the

[1]Request timeouts in some public cloud service APIs [13, 23] are often tens of seconds, if not minutes.

# Clients	Algorithm	LND (Mean)	LND (p95)	LND (p99)
100	Baseline	0.39	0.75	0.98
	FIFO+RC	0.34	0.64	0.95
	EDF+RC	0.30	0.52	0.71
	LSTF+RC	0.32	0.71	0.81
150	Baseline	0.61	**1.32**	**5.68**
	FIFO+RC	0.33	0.60	0.84
	EDF+RC	0.31	0.52	0.69
	LSTF+RC	0.3	0.51	0.71
200	Baseline	**1.09**	**2.75**	**18.47**
	FIFO+RC	0.71	**1.63**	**2.95**
	EDF+RC	0.38	0.79	**1.17**
	LSTF+RC	0.34	0.61	0.87
250	Baseline	**1.33**	**4.83**	**19.71**
	FIFO+RC	0.98	**2.58**	**10.5**
	EDF+RC	0.49	**1.25**	**2.15**
	LSTF+RC	0.46	**1.12**	**1.82**

Table 5.4: Mean, 95th percentile and 99th percentile values of latencies normalized by the deadline (LND) for requests from the production workload. An LND of 1.0 means the end-to-end latency equaled the deadline.

heavier workflows are throttled and dropped before they cause downstream congestion. With 250 clients, all algorithms with rate-limiting drop close to 21% of requests since it is infeasible to meet their deadlines (§5.2.5), freeing up resources for other requests. We also note that LSTF outperforms EDF across all runs. Recall that EDF only prioritizes requests based on the proximity to the deadline. Therefore, a request may not make enough progress until it is too late [53]. On the other hand, LSTF *also* factors in the remaining service time which can be estimated because of Cicero's progress metrics. LSTF here highlights the benefits of scheduling based on end-to-end characteristics of requests using Cicero.

5.3.3 Can Cicero isolate low-latency workflows from high-throughput workflows?

A common scenario in cloud storage systems is the co-existence of throughput intensive workflows which involve bulk reads/writes as well as low-latency workflows that have soft deadlines. Here, we run Cicero with the bottleneck fairness policy in conjunction with the fair scheduler. We vary the number of throughput intensive clients from 40 to 200, each of which runs in a closed loop. Every request from this workflow arriving at a PS triggers twenty sub-tasks to the BS. Six low latency clients (one workflow each) submit requests at a rate of 10 requests per-second (60 rps in total),

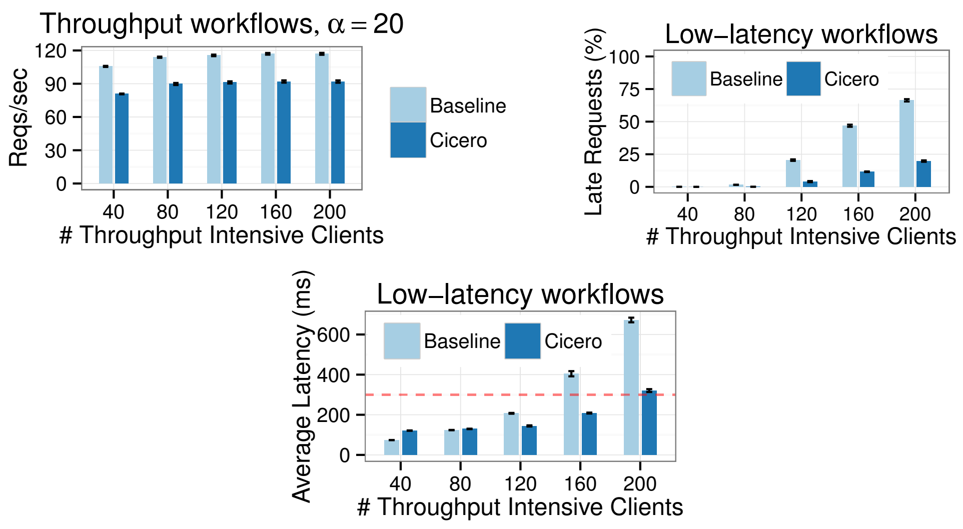


Figure 5.7: Average latencies and fraction of late requests for latency sensitive workflows in the presence of throughput focused clients. The low latency clients with 300ms deadlines are able to meet a high fraction of deadlines (˜80% hit rate in the presence of 200 throughput intensive clients).

with every request having a 300ms deadline. Figure 5.7 indicates our results. When only 40 high-throughput clients are active, Cicero and the baseline successfully meet all deadlines. The baseline presents an improved average latency for the low-latency clients because it does not incur the added overhead of our DRR-based fair scheduler (also observed by [126]). However, at higher loads, Cicero's performance degrades gracefully, with a high fraction of admitted requests meeting their deadlines (˜80% hit rate in the presence of 200 high throughput clients). Our implementation cannot guarantee latency for a request unless it *(i)* compromises on being work-conserving or *(ii)* preempts on-going work at any resource (which, for resources such as locks or connection objects is not practical, and has non-trivial ramifications on application layer logic). Thus, a request arriving at a service can get unlucky due to bad timing: a sub-task from a low-latency workflow may arrive *right after* a burst of requests from other workflows were scheduled (and thus suffer from head-of-line blocking at the local resources). Cicero shapes the high throughput clients to their fair share of resources alongside the low-latency clients.

5.3.4 Can the rate adaptation react to hotspots from skewed access patterns?

We now evaluate a scenario where we create a skewed access pattern. Four workflows activate at different times. A and B are consistently routed to the same PS, causing

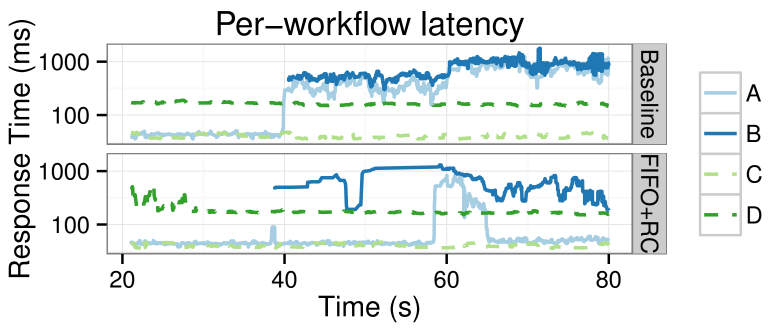


Figure 5.8: Skewed workload experiment, with a median smoothened timeseries of the latencies. Workfow B starts at t=40s, contends at at the same PS workflow A is being routed through, and doubles its sending rate at t=60s. Cicero's rate-limiting shields A from B in both instances. The background workflows (*dashed*) are within their fair share of resources in the system and are thus unaffected.

a hotspot on the local semaphore protected resource. A is an open loop workflow generated by a single client. B starts at $t = 40s$ with thirty clients, and at $t = 60s$, doubles its sending rate with an additional thirty clients joining the system. We run only the rate-limiter component with FIFO scheduling to isolate the effects of the former (that is, we only use a FIFO sub-task scheduler at each node). C and D are background workflows that exert pressure on the BSs. Figure 5.8 shows a rolling median of the latency timeseries with and without Cicero. In the baseline, when A enters the system at $t = 40s$, the surge of client requests immediately cause contention at all queues at the PS, including the shared sub-task scheduler queue as well as a semaphore being contended for. The resulting head-of-line blocking inflates latencies for B. When using Cicero, A experiences a spike in latencies at $t = 40s$ when B enters the system first. However, twenty seconds later, when B doubles its sending rate, it forces head-of-line blocking and higher latencies for A as with the baseline (recall that we are not using local fair-queuing here). However, Cicero soon computes rates based on the observed costs of B and begins to adapt σ_B. When the rate-limiters and system queues stabilize from the unexpected surge, A retains its expected latencies, whereas B is throttled at the entry point. Cicero also (correctly) avoids throttling the background workflows which are not contributing to congestion (and are not exceeding their fair shares at any node).

5.3.5 Quantile knob sensitivity analysis

Lastly, we demonstrate the impact of the quantile knob in the rate adaptation mechanism. We consider a scan workflow generated by 180 client threads, which

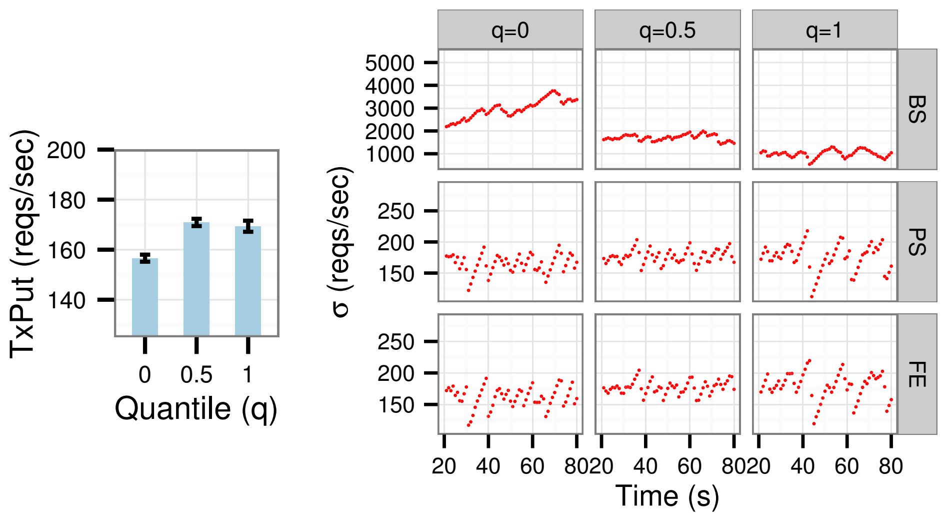


Figure 5.9: σ adaptation from the BSs up to the FEs for a workflow, for different values of the q parameter. BSs adapt their rates independently based on their demands, PS's aggregate the rates to calculate their local σ using a quantile, and the FE's repeat the same procedure against the PS'.

triggers ten back-to-back requests between the PS and BSs. This workflow competes with a lighter open-loop workflow for system resources, triggering our max-min fairness policy. We study the impact of the quantile knob on the scan workflow's throughput. Figure 5.9 shows a timeseries where each data point represents $\sum \sigma$ per-tier in a second for the scan workflow, for different values of q. With $q = 0$, each PS selects the minimum advertised σ from the individual BS processes, and the FEs repeat the procedure with the PS processes. This leads to conservative σ values at the FEs, leading to low utilization at the BSs. This is evident in that *(i)* the BS tier probes for more demand by advertising higher rates (Figure 5.9 (top row, left)), and *(ii)* lower end-to-end throughput for the workflow (Figure 5.9 (left)). With $q = 1$, the rates are *max* aggregated, and upstream services thus push more demand to the BSs. This results in increased congestion at the BSs, which therefore exert more backpressure by announcing lower rates. With $q = 0.5$, we strike a balance by upstream services matching the true capacity of the BSs. With $q = 0.5$, the σ values at the FE do not oscillate as much as with $q = 0$ and $q = 1$, leading to a stable load at the BS. Also, note that by scaling the rates between each tier according to the amplification factor α, Cicero correctly sets σ at the FEs which match the capacity of the BS tier.

5.4 Discussion

Combining Cicero with smarter load-balancing: Cicero's mechanisms work orthogonally to the choice of load-balancing algorithms that a process uses to select a replica server from another service. As established in Chapter 4, the load-balancing technique used affects not only the latency distribution but also the overall system throughput. We believe opportunities exist for combining Cicero's end-to-end scheduling policies with various load-balancing mechanisms between tiers. For instance, requests with tighter slacks may be forwarded to faster replicas. Furthermore, a replica selection scheme that uses rate control as in C3 can inform a process about the total available throughput to a cluster of replica servers (rather than the process inferring the total throughput exclusively based on reports from downstream processes). We believe combining replica selection with the mechanisms in Cicero is an interesting avenue for future work.

Rate adaptation and service topology: While Cicero's design has been motivated by real-world SOAs, there are service topologies for which our rate aggregation scheme needs to be extended to handle corner cases correctly. For instance, all nodes in a Cassandra cluster are arranged in a ring, wherein each node can receive a request and then route it internally within the cluster. This makes a single node both an upstream and a downstream service for the same workflow; therefore, our aggregation scheme needs a mechanism to break such cycles. Another option here is for the services communicating with the Cassandra cluster to infer rates to the cluster on their own (as C3's rate controller might do). Processes can then use these measured rates for the rest of the rate adaptation protocol.

Recurring workflows: Currently, Cicero assumes no prior knowledge about a workflow's characteristics, and instead, assumes that a workflow exhibits stable average case behaviour (or one that changes slowly with respect to Cicero's control interval). Akin to literature in cluster scheduling [76], Cicero's design can benefit from an accurate model of a workflow's characteristics (to capture properties such as a periodicity of bursty arrivals).

5.5 Summary

We highlighted the unique challenges with resource management in the context of SOAs and Microservices. We presented Cicero, which uses a composition of multiple techniques, including distributed rate control and metadata propagation to enable scheduling requests with their end-to-end objectives in mind. Our design does not involve centralized coordination and does not assume prior knowledge of a request's characteristics. Through performance evaluations, we demonstrated Cicero's effectiveness at enforcing diverse policies including providing performance isolation in extreme settings, reducing end-to-end latencies and meeting deadlines.

6 Conclusions and outlook

As scale-out distributed systems grow larger than ever in response to the pressures of scale, they face several challenges on the way to achieving predictable performance. These include complex inter-server request-response patterns, and server-side performance fluctuations arising from skews and hot-spots in data access patterns, background activities, as well as multi-tenancy. Many of these effects manifest at very short timescales (10s to 100s of milliseconds), and compromise end-to-end system performance. This motivates the need for online, adaptive mechanisms for distributed systems to remain robust to these dynamic conditions. In this thesis, we thus present simple-yet-practical adaptive techniques for achieving predictable performance in the context of two popular classes of large-scale distributed systems: cloud data-stores as well as micro-services.

This thesis advances the state of the art in the context of distributed replica selection algorithms for cloud data-stores. The key insight of our solution is the composition of two mechanisms: one, a replica ranking heuristic that aims to balance queue-lengths at servers in proportion to their service-time differences; and second, distributed rate-adaptation to ensure that the collective demands of all clients match the capacities of individual servers. This combination of load "balancing" as well as "regulation" is a key departure from prior literature in replica selection, wherein the load regulation angle has been ignored.

Furthermore, this thesis presents one of the first solutions for end-to-end resource management in the context of micro-services and service-oriented architectures. Despite the promise of micro-services as a way to build scalable, loosely coupled, and modular systems, these architectures lead to complex inter-server interactions that make resource management and scheduling challenging. This thesis proposes simple adaptive algorithms, architecturally compatible with today's popular libraries for building SOAs, that enable enforcement of end-to-end resource management policies in an SOA setting.

To wrap up, we summarize the core contributions of this thesis, and highlight directions for future work.

6.1 Summary

The contributions of this thesis are as follows:

- **Adaptive replica selection for cloud data-stores:** Chapter 4 makes the key observation that adaptive replica selection can be used to overcome many sources of performance variability that plague cloud data-stores. At the same time, it highlights the challenges of designing effective and practical replica selection mechanisms using extensive measurements of the Cassandra distributed database. We propose C3, an adaptive replica selection mechanism that is robust to performance variability in the environment. System evaluations conducted across a wide-band of operational settings, from using industry-standard benchmarks on Amazon EC2, to production experiments at Spotify and SoundCloud, confirm the effectiveness of the C3 scheme in reducing tail-latencies without trading off (and even significantly improving) system throughput.
- **End-to-end resource management and scheduling for micro-services:** Chapter 5 brings to attention the challenge of achieving end-to-end performance predictability for micro-services and service-oriented architectures, an emerging class of distributed systems that have received little attention from the research community. We propose Cicero, a system that enforces a diverse range of end-to-end resource management policies in a micro-services setting. We demonstrate the effectiveness of end-to-end, policy-based, adaptive back-pressure and scheduling, enabled through minimal information exchange between neighboring services. Our system evaluation validates Cicero's design; we demonstrate how Cicero achieves highly desirable performance objectives such as meeting end-to-end deadlines, preventing cascading failures, as well as ensuring performance isolation between different tenants.

6.2 Future directions

The challenges of scale continue to push the boundaries of distributed system design. Yet, several "design relics" from the past lead to lost opportunities for end-to-end scheduling and resource management in distributed systems. First, individual servers in WSCs today still run single host operating systems, wherein several layers of abstraction separate userspace programs from hardware resources. Second, the network in data centers is still seen as a black box by the applications that rely on them. Third, runtimes and programming languages continue to expose developers to many distributed system complexities because of the low-level abstractions used. We now discuss future directions that are tied to each of these three obstacles.

Full stack end-to-end scheduling: Our work on Cicero explored how application-layer processes, with minimal cooperation, can achieve various end-to-end performance objectives and resource management policies. However, Cicero cannot influence scheduling decisions for requests at lower layers of the operating system stack (such as at the storage sub-system or network). A first step would be to instrument the stack as done by IOFlow [135], and have mechanisms to maintain a mapping of application-layer workflows to sub-system activity. Such a mapping will allow us to reap opportunities for improved scheduling and performance isolation. However, more generally, we need to fundamentally re-think the operating system stack for data center environments [124]. Today, the pooling of hardware resources is often done above the operating system in userspace using distributed file-systems, data stores as well as cluster schedulers. Some of this functionality may be best served instead by a distributed exokernel operating system. This opens avenues for distributed applications and runtimes to tightly control how they make use of hardware resources.

Application and network co-design: Several recent hardware trends make for exciting opportunities in distributed system and network co-design. Recent work [61,120] has demonstrated the performance benefits that can be reaped if distributed systems avoid having to make worst-case assumptions about the underlying network. On this note, we believe there is ample opportunity for building distributed systems that take advantage of lossless fabrics such as RDMA over converged ethernet as well as NIC offloading to speed up their critical paths. Furthermore, such co-designs also lead naturally to the question of *cross-layer scheduling*. For instance, in our work on Cicero, we see how micro-services exhibit complex inter-process communication patterns, which results in network traffic patterns that differ from those of cluster computing applications. We believe this creates untapped opportunities for optimizations that involve application-aware scheduling of compute and network resources in unison.

Execution runtimes for distributed systems: Several classes of distributed applications can typically be modeled as complex DAGs of compute activity and data flows across several processes. In data-analytics frameworks such as Apache Tez, this DAG is often explicit in the programming model itself. An important advantage of such a programming model is that it lends itself easily for further optimization by the underlying runtime. However, the information about an application's DAG structure often does not percolate throughout all the individual sub-systems in the stack (for instance, down to the underlying distributed filesystem). Furthermore, there are several classes of distributed systems where the DAG structure is not even latent in the programming model (such as low-latency interactive web-services). Runtimes for distributed systems that exploit these DAG structures will open opportunities for *end-to-end* performance optimization. Such runtimes are especially relevant considering the growing popularity of the distributed actors approach to building distributed systems [46], and the optimization opportunities it provides [111].

List of Figures

List of Tables

Bibliography

[1] 2013 Founders' Letter. https://abc.xyz/investor/founders-letters/2013/. Last accessed: Mar 31, 2016.

[2] 802.1Qbb - Priority-based Flow Control. http://www.ieee802.org/1/pages/802.1bb.html. 2016.

[3] Akka. http://akka.io/. Last accessed: Sept 25, 2014.

[4] Amazon elb. http://goo.gl/F5A1Em. Last accessed: Sept 24, 2014.

[5] Apache Cassandra. http://cassandra.apache.org/. Last accessed: June 10, 2013.

[6] Apache Cassandra Use Cases. http://planetcassandra.org/apache-cassandra-use-cases/. Last accessed: Sept 25, 2014.

[7] Apache HBase. https://hbase.apache.org//. Last accessed: Mar 23, 2016.

[8] Astyanax. https://github.com/Netflix/astyanax. Last accessed: Jan 5, 2015.

[9] Auto Scaling in the Amazon Cloud. http://techblog.netflix.com/2012/01/auto-scaling-in-amazon-cloud.html. Last accessed: Mar 30, 2016.

[10] Cassandra Documentation. http://www.datastax.com/documentation/cassandra/2.0. Last accessed: Sept 25, 2014.

[11] Cloud Computing Trends: 2016 State of the Cloud Survey. http://www.rightscale.com/blog/cloud-industry-insights/cloud-computing-trends-2016-state-cloud-survey. Last accessed: Apr 1, 2016.

[12] DB-Engines Ranking of Wide Column Stores. http://db-engines.com/en/ranking/wide+column+store. Last accessed: Sept 25, 2014.

[13] Dealing with DeadlineExceededErrors. https://cloud.google.com/appengine/articles/deadlineexceedederrors. 2012.

[14] ELS: a latency-based load balancer. https://labs.spotify.com/2015/12/09/els-part-2/. Last accessed: Apr 1, 2016.

[15] Kubernetes. http://github.com/kubernetes/kubernetes. Last accessed: Apr 5, 2016.

[16] Load Balancing and Proxy Configuration. http://docs.basho.com/riak/1.4.0/cookbooks/Load-Balancing-and-Proxy-Configuration/. Last accessed: Sept 24, 2014.

[17] Memcached. https://memcached.org/. Last accessed: Mar 23, 2016.

[18] MongoDB. https://www.mongodb.org/. Last accessed: Mar 23, 2016.

[19] Nginx. http://nginx.org/en/docs/http/load_balancing.html. Last accessed: Sept 24, 2014.

[20] Openstack. https://www.openstack.org/.

[21] Redis. http://redis.io/. Last accessed: Mar 23, 2016.

[22] Riak: AWS Performance Tuning. http://docs.basho.com/riak/latest/ops/tuning/aws/. Last accessed: Sept 24, 2014.

[23] Setting Timeouts for Blob Service Operations. https://msdn.microsoft.com/en-us/library/azure/dd179431.aspx. 2016.

[24] Shipment forecast of tablets, laptops and desktop PCs worldwide from 2010 to 2019 (in million units). http://www.statista.com/statistics/272595/global-shipments-forecast-for-tablets-laptops-and-desktop-pcs/. Last accessed: Mar 31, 2016.

[25] The Top 20 Valuable Facebook Statistics - Updated December 2015. https://zephoria.com/top-15-valuable-facebook-statistics/. Last accessed: Mar 31, 2016.

[26] TCP-Friendly Multicast Congestion Control (TFMCC): Protocol Specification. https://tools.ietf.org/html/rfc4654, 2006.

[27] Erlang at Facebook. https://www.erlang-factory.com/upload/presentations/31/EugeneLetuchy-ErlangatFacebook.pdf, 2009. Last accessed: Apr 1, 2016.

[28] MySQL project website. http://www.mysql.de, Dec. 2009. Last accessed: Apr 1, 2016.

[29] The Life of a Typeahead Query. "https://www.facebook.com/notes/facebook-engineering/the-life-of-a-typeahead-query/389105248919/", 2010. Last accessed: Apr 1, 2016.

[30] Netflix: we spend money on movies, not on servers. http://goo.gl/oOAiyf, 2012. Last accessed: Apr 1, 2016.

[31] Summary of the October 22, 2012 AWS Service Event in the US-East Region, 2012. https://aws.amazon.com/message/680342/.

[32] Docker at Spotify. http://goo.gl/53t3XN, 2013. Last accessed: Apr 1, 2016.

[33] Bug 727708: Spotify position in support of systemd in the default init debate. https://lists.debian.org/debian-ctte/2014/01/msg00287.html, 2014. Last accessed: Apr 1, 2016.

[34] Airbnb Shares The Keys To Its Infrastructure. http://www.nextplatform.com/2015/09/10/airbnb-shares-the-keys-to-its-infrastructure/, 2015. Last accessed: Apr 1, 2016.

[35] Alizadeh, M., Greenberg, A., Maltz, D. A., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., and Sridharan, M. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)* (New York, NY, USA, 2010), ACM, pp. 63–74.

[36] Alizadeh, M., Yang, S., Sharif, M., Katti, S., McKeown, N., Prabhakar, B., and Shenker, S. pfabric: Minimal near-optimal datacenter transport. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)* (New York, NY, USA, 2013), ACM, pp. 435–446.

[37] Ananthanarayanan, G., Kandula, S., Greenberg, A., Stoica, I., Lu, Y., Saha, B., and Harris, E. Reining in the Outliers in Map-reduce Clusters Using Mantri. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Berkeley, CA, USA, 2010), USENIX Association, pp. 265–278.

[38] Angel, S., Ballani, H., Karagiannis, T., O'Shea, G., and Thereska, E. End-to-end performance isolation through virtual datacenters. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Berkeley, CA, USA, 2014), USENIX Association, pp. 233–248.

[39] Anthony Fisk. Why not to multi-tenant cassandra, http://anthonyfisk.blogspot.de/2015/06/why-not-to-multi-tenant-cassandra.html. Last accessed: Mar 31, 2016.

[40] Arce, G. R. *Nonlinear Signal Processing: A Statistical Approach.* Wiley, 2004.

[41] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2012), ACM, pp. 53–64.

[42] B. Sigelman et al. Dapper, a large-scale distributed systems tracing infrastructure. Tech. rep., Google, Inc., 2010.

[43] Barroso, L. A., Clidaras, J., and Hölzle, U. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*, 2nd ed. Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, 2013.

[44] Barroso, L. A., Dean, J., and Hölzle, U. Web search for a planet: The google cluster architecture. *IEEE Micro 23*, 2 (Mar. 2003), 22–28.

[45] Beaver, D., Kumar, S., Li, H. C., Sobel, J., and Vajgel, P. Finding a needle in haystack: Facebook's photo storage. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Berkeley, CA, USA, 2010), USENIX Association, pp. 1–8.

[46] Bernstein, P. A., Bykov, S., Geller, A., Kliot, G., and Thelin, J. Orleans: Distributed virtual actors for programmability and scalability. Tech. Rep. MSR-TR-2014-41, March 2014.

[47] Bispo, C. F. The single-server scheduling problem with convex costs. *Queueing Systems 73*, 3 (2013).

[48] Boutin, E., Ekanayake, J., Lin, W., Shi, B., Zhou, J., Qian, Z., Wu, M., and Zhou, L. Apollo: scalable and coordinated scheduling for cloud-scale computing. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Broomfield, CO, Oct. 2014), USENIX Association, pp. 285–300.

[49] Brewer, E. A. Lessons from giant-scale services. *Internet Computing, IEEE 5*, 4 (2001), 46–55.

[50] Bronson, N., Amsden, Z., Cabrera, G., Chakka, P., Dimov, P., Ding, H., Ferris, J., Giardullo, A., Kulkarni, S., Li, H., Marchukov, M., Petrov, D., Puzar, L., Song, Y. J., and Venkataramani, V. Tao: Facebook's distributed data store for the social graph. In *Proceedings of USENIX Annual Technical Conference (ATC)* (San Jose, CA, 2013), USENIX, pp. 49–60.

[51] Brutlag, J. Speed Matters. http://googleresearch.blogspot.com/2009/06/speed-matters.html. Last accessed: Sept 24, 2014.

[52] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst. 26*, 2 (June 2008), 4:1–4:26.

[53] Chen, S., Stankovic, J. A., Kurose, J. F., and Towsley, D. Performance evaluation of two new disk scheduling algorithms for real-time systems. *Real-Time Systems 3*, 3 (1991), 307–336.

[54] Chowdhury, M., and Stoica, I. Efficient coflow scheduling without prior knowledge. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)* (New York, NY, USA, 2015), ACM, pp. 393–406.

[55] Chowdhury, M., Zaharia, M., Ma, J., Jordan, M. I., and Stoica, I. Managing data transfers in computer clusters with orchestra. *SIGCOMM Comput. Commun. Rev. 41*, 4 (Aug. 2011), 98–109.

[56] Chowdhury, M., Zhong, Y., and Stoica, I. Efficient coflow scheduling with varys. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)* (New York, NY, USA, 2014), ACM, pp. 443–454.

[57] Christensen, B. Application Resilience in a Service-oriented Architecture. http://goo.gl/OTKDmQ, 2013. Last accessed: Apr 1, 2016.

[58] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)* (New York, NY, USA, 2010), ACM, pp. 143–154.

[59] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., and Woodford, D. Spanner: Google's globally-distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (Hollywood, CA, Oct. 2012), USENIX Association, pp. 261–264.

[60] Crovella, M., Frangioso, R., and Harchol-Balter, M. Connection scheduling in web servers. In *USENIX Symposium on Internet Technologies and Systems* (1999), vol. 10, pp. 243–254.

[61] Dang, H. T., Sciascia, D., Canini, M., Pedone, F., and Soulé, R. Netpaxos: Consensus at network speed. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research* (New York, NY, USA, 2015), ACM, pp. 5:1–5:7.

[62] Dean, J., and Barroso, L. A. The Tail At Scale. *Communications of the ACM 56* (2013), 74–80.

[63] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's Highly Available Key-value Store. In *ACM Symposium on Operating systems principles (SOSP)* (New York, NY, USA, 2007), ACM, pp. 205–220.

[64] Delgado, P., Dinu, F., Kermarrec, A.-M., and Zwaenepoel, W. Hawk: Hybrid datacenter scheduling. In *Proceedings of USENIX Annual Technical Conference (ATC)* (Santa Clara, CA, July 2015), USENIX Association, pp. 499–510.

[65] Demers, A., Keshav, S., and Shenker, S. Analysis and simulation of a fair queueing algorithm. In *Symposium Proceedings on Communications Architectures and Protocols* (New York, NY, USA, 1989), SIGCOMM '89, ACM, pp. 1–12.

[66] Dogar, F. R., Karagiannis, T., Ballani, H., and Rowstron, A. Decentralized task-aware scheduling for data center networks. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)* (2014), ACM, pp. 431–442.

[67] Dukkipati, N., and McKeown, N. Why flow-completion time is the right metric for congestion control. *SIGCOMM Comput. Commun. Rev. 36*, 1 (Jan. 2006), 59–62.

[68] Ellis, J. How not to benchmark Cassandra: a case study. http://goo.gl/SS9YCN, 2014. Last accessed: Apr 1 2016.

[69] Engineering, F. Under the hood: MySQL Pool Scanner (MPS). https://goo.gl/qJw8ZI, 2013. Last accessed: Apr 1, 2016.

[70] Erl, T. *Service-Oriented Architecture (SOA): Concepts, Technology, and Design* . Prentice Hall, 2005.

[71] Ferguson, A. D., Bodik, P., Kandula, S., Boutin, E., and Fonseca, R. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *Proceedings of the European Conference on Computer Systems (EuroSys)* (New York, NY, USA, 2012), ACM, pp. 99–112.

[72] Ghemawat, S., Gobioff, H., and Leung, S.-T. The google file system. In *ACM SIGOPS operating systems review* (2003), vol. 37, ACM, pp. 29–43.

[73] Ghemawat, S., Gobioff, H., and Leung, S.-T. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 29–43.

[74] Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., and Stoica, I. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2011), Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI), USENIX Association, pp. 323–336.

[75] Gog, I., Giceva, J., Schwarzkopf, M., Vaswani, K., Vytiniotis, D., Ramalingam, G., Costa, M., Murray, D. G., Hand, S., and Isard, M. Broom: Sweeping out garbage collection from big data systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)* (Kartause Ittingen, Switzerland, May 2015), USENIX Association.

[76] Grandl, R., Ananthanarayanan, G., Kandula, S., Rao, S., and Akella, A. Multi-resource packing for cluster schedulers. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)* (2014), ACM, pp. 455–466.

[77] Gray, W. D., and Boehm-Davis, D. Milliseconds matter: An introduction to microstrategies and to their use in describing and predicting interactive behavior. *Journal of Experimental Psychology: Applied 6* (2000).

[78] Grosvenor, M. P., Schwarzkopf, M., Gog, I., Watson, R. N., Moore, A. W., Hand, S., and Crowcroft, J. Queues Don't Matter When You Can JUMP Them! In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Oakland, CA, May 2015), USENIX Association, pp. 1–14.

[79] Gulati, A., Ahmad, I., Waldspurger, C. A., et al. Parda: Proportional allocation of resources for distributed storage access. In *Proccedings of the International Conference on File and Storage Technologies (FAST)* (Berkeley, CA, USA, 2009), USENIX Association, pp. 85–98.

[80] Gulati, A., Merchant, A., and Varman, P. J. mclock: handling throughput variability for hypervisor io scheduling. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Berkeley, CA, USA, 2010), USENIX Association, pp. 437–450.

[81] Guo, Z., McDirmid, S., Yang, M., Zhuang, L., Zhang, P., Luo, Y., Bergan, T., Musuvathi, M., Zhang, Z., and Zhou, L. Failure recovery: When the cure is worse than the disease. In *Presented as part of the 14th Workshop on Hot Topics in Operating Systems* (Berkeley, CA, 2013), USENIX.

[82] Ha, S., Rhee, I., and Xu, L. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *SIGOPS Oper. Syst. Rev. 42*, 5 (2008).

[83] Hadoop, A. http://hadoop.apache.org/. Last accessed: June 10, 2013.

[84] Hao, M., Soundararajan, G., Kenchammana-Hosekote, D., Chien, A. A., and Gunawi, H. S. The tail at store: A revelation from millions of hours of disk and ssd deployments. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (Santa Clara, CA, Feb. 2016), USENIX Association, pp. 263–276.

[85] Harry, B. Explanation of July 18th outage. http://goo.gl/DPuJBm, 2014. Last accessed: Apr 1, 2016.

[86] Herrtwich, R. G. *An introduction to real-time scheduling.* International Computer Science Institute, 1990.

[87] Hong, C.-Y., Caesar, M., and Godfrey, P. B. Finishing flows quickly with preemptive scheduling. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)* (New York, NY, USA, 2012), SIGCOMM '12, ACM, pp. 127–138.

[88] Jalaparti, V., Bodik, P., Kandula, S., Menache, I., Rybalkin, M., and Yan, C. Speeding up Distributed Request-Response Workflows. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)* (New York, NY, USA, 2013), ACM, pp. 219–230.

[89] Jang, K., Sherry, J., Ballani, H., and Moncaster, T. Silo: Predictable message latency in the cloud. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)* (New York, NY, USA, 2015), ACM, pp. 435–448.

[90] Jones, E. Retries considered harmful. http://www.evanjones.ca/retries-considered-harmful.html, 2015. Last accessed: Apr 1, 2016.

[91] Kalantzis, C. Eventual Consistency != Hopeful Consistency, talk at Cassandra Summit. https://www.youtube.com/watch?v=A6qzx_HE3EU, 2013. Last accessed: Apr 1, 2016.

[92] Kalantzis, C. Revisiting 1 Million Writes per second. http://goo.gl/Y4Fr7Y, 2014. Last accessed: Apr 1, 2016.

[93] Kambadur, M., Moseley, T., Hank, R., and Kim, M. A. Measuring Interference Between Live Datacenter Applications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Los Alamitos, CA, USA, 2012), SC '12, IEEE Computer Society Press, pp. 51:1–51:12.

[94] Kapoor, R., Porter, G., Tewari, M., Voelker, G. M., and Vahdat, A. Chronos: Predictable Low Latency for Data Center Applications. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)* (New York, NY, USA, 2012), SoCC '12, ACM, pp. 9:1–9:14.

[95] Li, J., Sharma, N. K., Ports, D. R. K., and Gribble, S. D. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)* (New York, NY, USA, 2014), ACM, pp. 9:1–9:14.

[96] Liljencrantz, A. How Not to Use Cassandra, talk at Cassandra Summit. https://www.youtube.com/watch?v=0u-EKJBPrj8, 2013. Last accessed: Apr 1, 2016.

[97] Liu, Y., Lo Presti, F., Misra, V., Towsley, D., and Gu, Y. Fluid models and solutions for large-scale ip networks. In *ACM SIGMETRICS Performance Evaluation Review* (2003), vol. 31, ACM, pp. 91–101.

[98] Lumb, C. R., and Golding, R. D-SPTF: Decentralized Request Distribution in Brick-based Storage Systems. *SIGOPS Oper. Syst. Rev. 38*, 5 (Oct. 2004), 37–47.

[99] Maas, M., Harris, T., Asanović, K., and Kubiatowicz, J. Trash day: Coordinating garbage collection in distributed systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)* (Berkeley, CA, USA, 2015), USENIX Association, pp. 1–1.

[100] Mace, J., Bodik, P., Fonseca, R., and Musuvathi, M. Retro: Targeted resource management in multi-tenant distributed systems. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (Oakland, CA, May 2015), USENIX Association, pp. 589–603.

[101] McCullough, J. C., Dunagan, J., Wolman, A., and Snoeren, A. C. Stout: An Adaptive Interface to Scalable Cloud Storage. In *USENIX ATC* (Berkeley, CA, USA, 2010), USENIX Association, pp. 4–4.

[102] McKenney, P. E. Stochastic Fair Queing. In *INFOCOM* (1990).

[103] Mitzenmacher, M. How Useful Is Old Information? *IEEE Trans. Parallel Distrib. Syst. 11*, 1 (Jan. 2000).

[104] Mitzenmacher, M. The power of two choices in randomized load balancing. *IEEE Trans. Parallel Distrib. Syst. 12*, 10 (Oct. 2001).

[105] Munns, C. I Love APIs 2015: Microservices at Amazon. http://goo.gl/aVWlpY, 2015. Last accessed: Apr 1, 2016.

[106] Muralidhar, S., Lloyd, W., Roy, S., Hill, C., Lin, E., Liu, W., Pan, S., Shankar, S., Sivakumar, V., Tang, L., and Kumar, S. f4: Facebook's warm blob storage system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), USENIX Association, pp. 383–398.

[107] Netflix. Embracing the Differences : Inside the Netflix API Redesign. http://techblog.netflix.com/2012/07/embracing-differences-inside-netflix.html, 2012.

[108] Netflix. Introducing Hystrix for Resilience Engineering. http://goo.gl/h9brP0, 2012. Last accessed: Apr 1, 2016.

[109] Netflix. Netflix Operations: Part I, Going Distributed. http://goo.gl/rElQem, 2012. Last accessed: Apr 1, 2016.

[110] Netflix. Strategy for tuning the hystrix configuratio. https://github.com/Netflix/Hystrix/issues/866, 2015.

[111] Newell, A., Kliot, G., Menache, I., Gopalan, A., Akiyama, S., and Silberstein, M. Optimizing distributed actor systems for dynamic interactive services. In *Proceedings of the European Conference on Computer Systems (EuroSys)* (April 2016), ACM – Association for Computing Machinery.

[112] Newman, S. *Building Microservices.* O'Reilly Media, 2015.

[113] Nginx. Adopting Microservices at Netflix: Lessons for Team and Process Design. https://goo.gl/KOrUfT, 2015. Last accessed: Apr 1, 2016.

[114] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., and Venkataramani, V. Scaling Memcache at Facebook. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Berkeley, CA, USA, 2013), USENIX Association, pp. 385–398.

[115] Nuyens, M., and Wierman, A. The foreground–background queue: a survey. *Performance evaluation 65*, 3 (2008), 286–307.

[116] Ousterhout, K., Wendell, P., Zaharia, M., and Stoica, I. Sparrow: Distributed, Low Latency Scheduling. In *ACM Symposium on Operating systems principles (SOSP)* (New York, NY, USA, 2013), ACM, pp. 69–84.

[117] Parekh, A. K., and Gallagher, R. G. A generalized processor sharing approach to flow control in integrated services networks: the multiple node case. *IEEE/ACM Transactions on Networking (ToN) 2*, 2 (1994), 137–150.

[118] Popa, L., Kumar, G., Chowdhury, M., Krishnamurthy, A., Ratnasamy, S., and Stoica, I. FairCloud: Sharing the Network in Cloud Computing. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)* (New York, NY, USA, 2012), ACM, pp. 187–198.

[119] Popa, L., Yalagandula, P., Banerjee, S., Mogul, J. C., Turner, Y., and Santos, J. R. ElasticSwitch: Practical Work-Conserving Bandwidth Guarantees for Cloud Computing. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)* (2013).

[120] Ports, D. R. K., Li, J., Liu, V., Sharma, N. K., and Krishnamurthy, A. Designing distributed systems using approximate synchrony in data center networks. In *Proc. USENIX NSDI '15* (Berkeley, CA, USA, 2015), pp. 43–57.

[121] Roussopoulos, M., and Baker, M. Practical Load Balancing for Content Requests in Peer-to-Peer Networks. *Distributed Computing 18*, 6 (2006).

[122] Sanfilippo, S. Redis latency spikes and the 99th percentile. http://antirez.com/news/83, 2014. Last accessed: Apr 1, 2016.

[123] Schad, J., Dittrich, J., and Quiané-Ruiz, J.-A. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *VLDB Endowment 3*, 1-2 (Sept. 2010), 460–471.

[124] Schwarzkopf, M., Grosvenor, M. P., and Hand, S. New wine in old skins: The case for distributed operating systems in the data center. In *Proceedings of the 4th Asia-Pacific Workshop on Systems* (New York, NY, USA, 2013), APSys '13, ACM, pp. 9:1–9:7.

[125] Shreedhar, M., and Varghese, G. Efficient fair queueing using deficit round robin. *SIGCOMM Comput. Commun. Rev. 25*, 4 (Oct. 1995), 231–242.

[126] Shue, D., Freedman, M. J., and Shaikh, A. Performance isolation and fairness for multi-tenant cloud storage. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 349–362.

[127] Souders, S. Velocity and the bottom line. http://radar.oreilly.com/2009/07/velocity-making-your-site-fast.html, 2009. Last accessed: Apr 1, 2016.

[128] SoundCloud. Building products at SoundCloud - part I: Dealing with the monolith. https://goo.gl/Qra2tA, 2014. Last accessed: Apr 1, 2016.

[129] Srikant, R. *The mathematics of Internet congestion control.* Springer Science & Business Media, 2012.

[130] Starobinski, D., and Sidi, M. Stochastically bounded burstiness for communication networks. In *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE* (1999), vol. 1, IEEE, pp. 36–42.

[131] Stewart, C., Chakrabarti, A., and Griffith, R. Zoolander: Efficiently Meeting Very Strict, Low-Latency SLOs. In *Proceedings of the International Conference on Autonomic Computing (ICAC)* (2013).

[132] Sumbaly, R., Kreps, J., Gao, L., Feinberg, A., Soman, C., and Shah, S. Serving Large-scale Batch Computed Data with Project Voldemort. In *Proccedings of the International Conference on File and Storage Technologies (FAST)* (Berkeley, CA, USA, 2012), USENIX Association, pp. 18–18.

[133] Suresh, L., Canini, M., Schmid, S., and Feldmann, A. C3: Cutting tail latency in cloud data stores via adaptive replica selection. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)* (Oakland, CA, May 2015), USENIX Association, pp. 513–527.

[134] Tassiulas, L. Adaptive back-pressure congestion control based on local information. *Automatic Control, IEEE Transactions on 40*, 2 (1995), 236–250.

[135] Thereska, E., Ballani, H., O'Shea, G., Karagiannis, T., Rowstron, A., Talpey, T., Black, R., and Zhu, T. IOFlow: A Software-Defined Storage Architecture. In *ACM Symposium on Operating systems principles (SOSP)* (New York, NY, USA, 2013), ACM, pp. 182–196.

[136] Tonse, S. MicroServices at Netflix - challenges of scale. http://goo.gl/9j5wSv, 2014. Last accessed: Apr 1, 2016.

[137] Twitter. Finagle: A Protocol-Agnostic RPC System. https://goo.gl/ITebZs, 2011. Last accessed: Apr 1, 2016.

[138] Uber. Service-oriented Architecture: Scaling the Uber Codebase as We Grow. https://eng.uber.com/soa/, 2015. Last accessed: Apr 1, 2016.

[139] van Mieghem, J. A. Dynamic Scheduling with Convex Delay Costs: The Generalized $c|mu$ Rule. *The Annals of Applied Probability 5* (1995).

[140] Vasić, N., Novaković, D., Miučin, S., Kostić, D., and Bianchini, R. Dejavu: accelerating resource allocation in virtualized environments. In *ACM SIGARCH computer architecture news* (2012), vol. 40, ACM, pp. 423–436.

[141] Venkataramani, V., Amsden, Z., Bronson, N., Cabrera III, G., Chakka, P., Dimov, P., Ding, H., Ferris, J., Giardullo, A., Hoon, J., Kulkarni, S., Lawrence, N., Marchukov, M., Petrov, D., and Puzar, L. TAO: How Facebook Serves the Social Graph. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2012).

[142] Verma, A., Pedrosa, L., Korupolu, M. R., Oppenheimer, D., Tune, E., and Wilkes, J. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)* (Bordeaux, France, 2015), ACM, pp. 18:1–18:17.

[143] Vulimiri, A., Godfrey, P. B., Mittal, R., Sherry, J., Ratnasamy, S., and Shenker, S. Low Latency via Redundancy. In *CoNEXT* (New York, NY, USA, 2013), ACM, pp. 283–294.

[144] Wang, A., Venkataraman, S., Alspaugh, S., Katz, R., and Stoica, I. Cake: Enabling High-level SLOs on Shared Storage Systems. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)* (New York, NY, USA, 2012), SoCC '12, ACM, pp. 14:1–14:14.

[145] Wang, H., and Varman, P. Balancing Fairness and Efficiency in Tiered Storage Systems with Bottleneck-aware Allocation. In *FAST* (Berkeley, CA, USA, 2014), USENIX Association, pp. 229–242.

[146] Wei, D. X., Jin, C., Low, S. H., and Hegde, S. Fast tcp: motivation, architecture, algorithms, performance. *IEEE/ACM Transactions on Networking (ToN) 14*, 6 (2006), 1246–1259.

[147] Wilson, C., Ballani, H., Karagiannis, T., and Rowtron, A. Better never than late: meeting deadlines in datacenter networks. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)* (New York, NY, USA, 2011), ACM, pp. 50–61.

[148] Wu, Z., Yu, C., and V. Madhyastha, H. CosTLO: Cost-Effective Redundancy for Lower Latency Variance on Cloud Storage Services. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Berkeley, CA, USA, 2015), USENIX Association, pp. 543–557.

[149] Xu, L., Harfoush, K., and Rhee, I. Binary increase congestion control (bic) for fast long-distance networks. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies* (2004), vol. 4, IEEE, pp. 2514–2524.

[150] Xu, Y., Musgrave, Z., Noble, B., and Bailey, M. Bobtail: Avoiding Long Tails in the Cloud. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Berkeley, CA, USA, 2013), USENIX Association, pp. 329–342.

[151] Yegge, S. Google+ post. https://plus.google.com/+RipRowan/posts/eVeouesvaVX, 2011. Last accessed: Apr 1, 2016.

[152] Zaharia, M., Borthakur, D., Sen Sarma, J., Elmeleegy, K., Shenker, S., and Stoica, I. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the European Conference on Computer Systems (EuroSys)* (New York, NY, USA, 2010), ACM, pp. 265–278.

[153] Zaharia, M., Konwinski, A., Joseph, A. D., Katz, R., and Stoica, I. Improving MapReduce Performance in Heterogeneous Environments. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Berkeley, CA, USA, 2008), USENIX Association, pp. 29–42.

[154] Zhu, T., Tumanov, A., Kozuch, M. A., Harchol-Balter, M., and Ganger, G. R. PriorityMeister: Tail Latency QoS for Shared Networked Storage. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)* (New York, NY, USA, 2014), ACM, pp. 29:1–29:14.